

TypeScript: Modern JavaScript Development

Leverage the features of TypeScript to boost
your development skills and create captivating
web applications

A course in three modules

Packt

BIRMINGHAM - MUMBAI

TypeScript: Modern JavaScript Development

Copyright © 2016 Packt Publishing

Published on: December 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78728-908-6

www.packtpub.com

Preface

It wasn't a long time ago that many JavaScript engineers or, most of the time, web frontend engineers, were still focusing on solving detailed technical issues, such as how to lay out specific content cross-browsers and how to send requests cross-domains.

At that time, a good web frontend engineer was usually expected to have notable experience on how detailed features can be implemented with existing APIs. Only a few people cared about how to write application-scale JavaScript because the interaction on a web page was really simple and no one wrote ASP in JavaScript.

However, the situation has changed tremendously. JavaScript has become the only language that runs everywhere, cross-platform and cross-device. In the main battlefield, interactions on the Web become more and more complex, and people are moving business logic from the backend to the frontend. With the growth of the Node.js community, JavaScript is playing a more and more important roles in our life.

TypeScript is indeed an awesome tool for JavaScript. Unfortunately, intelligence is still required to write actually robust, maintainable, and reusable code. TypeScript allows developers to write readable and maintainable web applications. Editors can provide several tools to the developer, based on types and static analysis of the code.

What this learning path covers

Module 1, Learning TypeScript, introduces many of the TypeScript features in a simple and easy-to-understand format. This book will teach you everything you need to know in order to implement large-scale JavaScript applications using TypeScript. Not only does it teach TypeScript's core features, which are essential to implement a web application, but it also explores the power of some tools, design principles, best practices, and it also demonstrates how to apply them in a real-life application.

Module 2, TypeScript Design Patterns, is collection of the most important patterns you need to improve your applications' performance and your productivity. Each pattern is accompanied with rich examples that demonstrate the power of patterns for a range of tasks, from building an application to code testing.

Module 3, TypeScript Blueprints, shows you how to use TypeScript to build clean web applications. You will learn how to use Angular 2 and React. You will also learn how you can use TypeScript for servers, mobile apps, command-line tools, and games. You will also learn functional programming. This style of programming will improve your general code skills. You will see how this style can be used in TypeScript.

What you need for this learning path

You will need the TypeScript compiler and a text editor. This learning path explains how to use Atom, but it is also possible to use other editors, such as Visual Studio 2015, Visual Studio Code, or Sublime Text.

You also need an Internet connection to download the required references and online packages and libraries, such as jQuery, Mocha, and Gulp. Depending on your operating system, you will need a user account with administrative privileges in order to install some of the tools used in this learning path. Also to compile TypeScript, you need NodeJS. You can find details on how you can install it in the first chapter of the third module.

Who this learning path is for

This learning path is for the intermediate-level JavaScript developers aiming to learn TypeScript to build beautiful web applications and fun projects. No prior knowledge of TypeScript is required but a basic understanding of jQuery is expected. This learning path is also for experienced TypeScript developer wanting to take their skills to the next level, and also for web developers who wish to make the most of TypeScript.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – 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 course'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 course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** to the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Ziipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/TypeScript-Modern-JavaScript-Development>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

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 courses – 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 course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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 course in the search field. The required information will appear under the **Errata** section.

Module 1: Learning TypeScript

Chapter 1: Introducing TypeScript	1
The TypeScript architecture	2
TypeScript language features	4
Putting everything together	26
Summary	27
Chapter 2: Automating Your Development Workflow	29
A modern development workflow	29
Prerequisites	30
Source control tools	33
Package management tools	38
Task runners	43
Test runners	53
Synchronized cross-device testing	55
Continuous Integration tools	58
Scaffolding tools	59
Summary	61
Chapter 3: Working with Functions	63
Working with functions in TypeScript	64
Asynchronous programming in TypeScript	83
Summary	98
Chapter 4: Object-Oriented Programming with TypeScript	99
SOLID principles	100
Classes	101
Interfaces	104
Association, aggregation, and composition	105
Inheritance	107
Generic classes	115

Generic constraints	118
Applying the SOLID principles	123
Namespaces	127
Modules	129
Circular dependencies	136
Summary	138
Chapter 5: Runtime	139
The environment	140
The runtime	141
The this operator	144
Prototypes	148
Closures	158
Summary	164
Chapter 6: Application Performance	165
Prerequisites	166
Performance and resources	166
Performance metrics	167
Performance analysis	169
Performance automation	186
Exception handling	189
Summary	191
Chapter 7: Application Testing	193
Software testing glossary	194
Prerequisites	196
Testing planning and methodologies	200
Setting up a test infrastructure	203
Creating test assertions, specs, and suites with Mocha and Chai	213
Test spies and stubs with Sinon.JS	220
Creating end-to-end tests with Nightwatch.js	227
Generating test coverage reports	228
Summary	230
Chapter 8: Decorators	231
Prerequisites	231
Annotations and decorators	232
Summary	249
Chapter 9: Application Architecture	251
The single-page application architecture	252
The MV* architecture	258
Common components and features in the MV* frameworks	259

Choosing an application framework	273
Writing an MVC framework from scratch	274
Summary	299
Chapter 10: Putting Everything Together	301
Prerequisites	302
The application's requirements	302
The application's data	303
The application's architecture	304
The application's file structure	305
Configuring the automated build	307
The application's layout	310
Implementing the root component	310
Implementing the market controller	312
Implementing the NASDAQ model	314
Implementing the NYSE model	316
Implementing the market view	316
Implementing the market template	319
Implementing the symbol controller	320
Implementing the symbol view	323
Implementing the chart model	325
Implementing the chart view	327
Testing the application	330
Preparing the application for a production release	330
Summary	332

Module 1

Learning TypeScript

Exploit the features of TypeScript to develop and maintain captivating web applications with ease

1

Introducing TypeScript

This book focuses on TypeScript's object-oriented nature and how it can help you to write better code. Before diving into the object-oriented programming features of TypeScript, this chapter will give you an overview of the history behind TypeScript and introduce you to some of the basics.

In this chapter, you will learn about the following concepts:

- The TypeScript architecture
- Type annotations
- Variables and primitive data types
- Operators
- Flow control statements
- Functions
- Classes
- Interfaces
- Modules

The TypeScript architecture

In this section, we will focus on the TypeScript's internal architecture and its original design goals.

Design goals

In the following points, you will find the main design goals and architectural decisions that shaped the way the TypeScript programming language looks like today:

- Statically identify JavaScript constructs that are likely to be errors. The engineers at Microsoft decided that the best way to identify and prevent potential runtime issues was to create a strongly typed programming language and perform static type checking at compilation time. The engineers also designed a language services layer to provide developers with better tools.
- High compatibility with the existing JavaScript code. TypeScript is a superset of JavaScript; this means that any valid JavaScript program is also a valid TypeScript program (with a few small exceptions).
- Provide a structuring mechanism for larger pieces of code. TypeScript adds class-based object orientation, interfaces, and modules. These features will help us structure our code in a much better way. We will also reduce potential integration issues within our development team and our code will become more maintainable and scalable by adhering to the best object-oriented principles and practices.
- Impose no runtime overhead on emitted programs. It is common to differentiate between design time and execution time when working with TypeScript. We use the term *design time code* to refer to the TypeScript code that we write while designing an application; we use the terms *execution time code* or *runtime code* to refer to the JavaScript code that is executed after compiling some TypeScript code.

TypeScript adds features to JavaScript but those features are only available at design time. For example, we can declare interfaces in TypeScript but since JavaScript doesn't support interfaces, the TypeScript compiler will not declare or try to emulate this feature in the output JavaScript code.

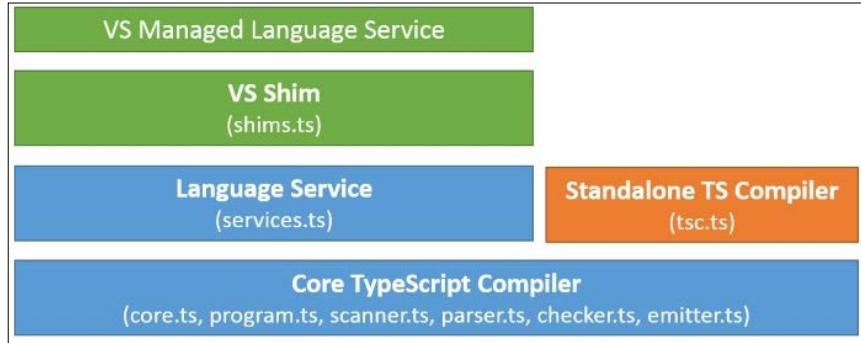
The Microsoft engineers provided the TypeScript compiler with mechanisms such as code transformations (converting TypeScript features into plain JavaScript implementations) and type erasure (removing static type notation) to generate really clean JavaScript code. Type erasure removes not only the type annotations but also all the TypeScript exclusive language features such as interfaces.

Furthermore, the generated code is highly compatible with web browsers as it targets the ECMAScript 3 specification by default but it also supports ECMAScript 5 and ECMAScript 6. In general, we can use the TypeScript features when compiling to any of the available compilation targets, but there are some features that will require ECMAScript 5 or higher as the compilation target.

- Align with the current and future ECMAScript proposals. TypeScript is not just compatible with the existing JavaScript code; it will also potentially be compatible with future versions of JavaScript. The majority of Typescript's additional features are based on the future ECMAScript proposals; this means many TypeScript files will eventually become valid JavaScript files.
- Be a cross-platform development tool. Microsoft released TypeScript under the open source Apache license and it can be installed and executed in all major operating systems.

TypeScript components

The TypeScript language is internally divided into three main layers. Each of these layers is, in turn, divided into sublayers or components. In the following diagram, we can see the three layers (green, blue, and orange) and each of their internal components (boxes):



In the preceding diagram, the acronym **VS** refers to Microsoft's Visual Studio, which is the official integrated development environment for all the Microsoft products (including TypeScript). We will learn more about this and the other IDEs in the next chapter.

Each of these main layers has a different purpose:

- The language: It features the TypeScript language elements.
- The compiler: It performs the parsing, type checking, and transformation of your TypeScript code to JavaScript code.
- The language services: It generates information that helps editors and other tools provide better assistance features such as IntelliSense or automated refactoring.
- IDE integration: In order to take advantages of the TypeScript features, some integration work is required to be done by the developers of the IDEs. TypeScript was designed to facilitate the development of tools that help to increase the productivity of JavaScript developers. As a result of these efforts, integrating TypeScript with an IDE is not a complicated task. A proof of this is that the most popular IDEs these days include a good TypeScript support.



In other books and online resources, you may find references to the term transpiler instead of compiler. A **transpiler** is a type of compiler that takes the source code of a programming language as its input and outputs the source code into another programming language with more or less the same level of abstraction.

We don't need to go into any more detail as understanding how the TypeScript compiler works is out of the scope of this book; however, if you wish to learn more about this topic, refer to the TypeScript language specification, which can be found online at <http://www.typescriptlang.org/>.

TypeScript language features

Now that you have learned about the purpose of TypeScript, it's time to get our hands dirty and start writing some code.

Before you can start learning how to use some of the basic TypeScript building blocks, you will need to set up your development environment. The easiest and fastest way to start writing some TypeScript code is to use the online editor available on the official TypeScript website at <http://www.typescriptlang.org/> Playground, as you can see in the following screenshot:



learn

play

download

interact

TypeScript

TypeScript

Select...

Share

Run

JavaScript

```
1 class Greeter {  
2     greeting: string;  
3     constructor(message: string) {  
4         this.greeting = message;  
5     }  
6     greet() {  
7         return "Hello, " + this.greeting;  
8     }  
9 }  
10 var greeter = new Greeter("world");
```

```
1 var Greeter = (function () {  
2     function Greeter(message) {  
3         this.greeting = message;  
4     }  
5     Greeter.prototype.greet = function () {  
6         return "Hello, " + this.greeting;  
7     };  
8     return Greeter;  
9 })();  
10 var greeter = new Greeter("world");  
11
```

[Privacy Statement](#) | [Terms of Use](#) | [Trademarks](#) © 2012 - 2014

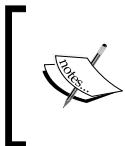
Microsoft

The code you enter in the TypeScript playground runs entirely in your browser and is not sent to Microsoft.

In the preceding screenshot, you will be able to use the text editor on the left-hand side to write your TypeScript code. The code is automatically compiled to JavaScript and the output code will be inserted in the text editor located on the right-hand side of the screen. If your TypeScript code is invalid, the JavaScript code on the right-hand side will not be refreshed.

Alternatively, if you prefer to be able to work offline, you can download and install the TypeScript compiler. If you work with Visual Studio, you can download the official TypeScript extension (version 1.5 beta) from <https://visualstudiogallery.msdn.microsoft.com/107f89a0-a542-4264-b0a9-eb91037cf7af>. If you are working with Visual Studio 2015, you don't need to install the extension as Visual Studio 2015 includes TypeScript support by default.

If you use a different code editor or you use the OS X or Linux operating systems, you can download an npm module instead. Don't worry if you are not familiar with npm. For now, you just need to know that it stands for Node Package Manager and is the default Node.js package manager.



There are TypeScript plugins available for many popular editors such as Sublime <https://github.com/Microsoft/TypeScript-Sublime-Plugin> and Atom <https://atom.io/packages/atom-typescript>.

In order to be able to use npm, you will need to first install Node.js in your development environment. You will be able to find the Node.js installation files on the official website at <https://nodejs.org/>.

Once you have installed Node.js in your development environment, you will be able to run the following command in a console or terminal:

```
npm install -g typescript
```

OS X users need to use the sudo command when installing global (-g) npm packages. The sudo command will prompt for user credentials and install the package using administrative privileges:

```
sudo npm install -g typescript
```

Create a new file named test.ts and add the following code to it:

```
var t : number = 1;
```

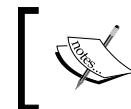
Save the file into a directory of your choice and once you have saved the file open the console, select the directory where you saved the file, and execute the following command:

```
tsc test.ts
```

The tsc command is a console interface for the TypeScript compiler. This command allows you to compile your TypeScript files into JavaScript files. The compiler features many options that will be explored in the upcoming chapters of this book.

In the preceding example, we used the tsc command to transform the test.ts file into a JavaScript file.

If everything goes right, you will find a file named test.js in the same directory in which the test.ts file was located. Now, you know how to compile your TypeScript code into JavaScript and we can start learning about the TypeScript features.



You will be able to learn more about editors and other tools in *Chapter 2, Automating Your Development Workflow*.

Types

As we have already learned, TypeScript is a typed superset of JavaScript. TypeScript added optional static type annotations to JavaScript in order to transform it into a strongly typed programming language. The optional static type annotations are used as constraints on program entities such as functions, variables, and properties so that compilers and development tools can offer better verification and assistance (such as IntelliSense) during software development.

Strong typing allows the programmer to express his intentions in his code, both to himself and to others in the development team.

TypeScript's type analysis occurs entirely at compile time and adds no runtime overhead to program execution.

Optional static type notation

The TypeScript language service is really good at inferring types, but there are certain cases where it is not able to automatically detect the type of an object or variable. For these cases, TypeScript allows us to explicitly declare the type of a variable. The language element that allows us to declare the type of a variable is known as **optional static type notation**. For a variable, the type notation comes after the variable name and is preceded by a colon:

```
var counter;           // unknown (any) type
var counter = 0;       // number (inferred)
var counter : number; // number
var counter : number = 0; // number
```

As you can see, the type of the variable is declared after the name, this style of type notation is based on type theory and helps to reinforce the idea of types being optional. When no type annotations are available, TypeScript will try to guess the type of the variable by examining the assigned values. For example, in the second line in the preceding code snippet, we can see that the variable counter has been identified as a numeric variable because the numeric value 0 was assigned as its value. This process in which types are automatically detected is known as **Type inference**, when a type cannot be inferred the special type any is used as the type of the variable.

Variables, basic types, and operators

The basic types are the Boolean, number, string, array, void types, and all user defined Enum types. All types in TypeScript are subtypes of a single top type called the **Any type**. The any keyword references this type. Let's take a look at each of these primitive types:

Data Type	Description
Boolean	Whereas the string and number data types can have a virtually unlimited number of different values, the Boolean data type can only have two. They are the literals <code>true</code> and <code>false</code> . A Boolean value is a truth value; it specifies whether the condition is true or not. <code>var isDone: boolean = false;</code>
Number	As in JavaScript, all numbers in TypeScript are floating point values. These floating-point numbers get the type <code>number</code> . <code>var height: number = 6;</code>
String	You use the string data type to represent text in TypeScript. You include string literals in your scripts by enclosing them in single or double quotation marks. Double quotation marks can be contained in strings surrounded by single quotation marks, and single quotation marks can be contained in strings surrounded by double quotation marks. <code>var name: string = "bob"; name = 'smith';</code>
Array	TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of the two ways. In the first, you use the type of the elements followed by <code>[]</code> to denote an array of that element type: <code>var list:number[] = [1, 2, 3];</code> The second way uses a generic array type, <code>Array</code> : <code>var list:Array<number> = [1, 2, 3];</code>
Enum	An enum is a way of giving more friendly names to sets of numeric values. By default, enums begin numbering their members starting at 0, but you can change this by manually setting the value of one to its members. <code>enum Color {Red, Green, Blue}; var c: Color = Color.Green;</code>

Data Type	Description
Any	<p>The any type is used to represent any JavaScript value. A value of the any type supports the same operations as a value in JavaScript and minimal static type checking is performed for operations on any values.</p> <pre>var notSure: any = 4; notSure = "maybe a string instead"; notSure = false; // okay, definitely a boolean</pre> <p>The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt in and opt out of type checking during compilation. The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:</p> <pre>var list:any[] = [1, true, "free"]; list[1] = 100;</pre>
Void	<p>The opposite in some ways to any is void, the absence of having any type at all. You will see this as the return type of functions that do not return a value.</p> <pre>function warnUser(): void { alert("This is my warning message"); }</pre>

JavaScript's primitive types also include undefined and null. In JavaScript, undefined is a property in the global scope that is assigned as a value to variables that have been declared but have not yet been initialized. The value null is a literal (not a property of the global object). It can be assigned to a variable as a representation of no value.

```
var TestVar;           // variable is declared but not initialized
alert(TestVar);       // shows undefined
alert(typeof TestVar); // shows undefined

var TestVar = null;    // variable is declared and value null is
                      // assigned as value
alert(TestVar);       // shows null
alert(typeof TestVar); // shows object
```

In TypeScript, we will not be able to use null or undefined as types:

```
var TestVar : null;      // Error, Type expected
var TestVar : undefined; // Error, cannot find name undefined
```

Since null or undefined cannot be used as types, both the variable declarations in the preceding code snippet are invalid.

Var, let, and const

When we declare a variable in TypeScript, we can use the `var`, `let`, or `const` keywords:

```
var mynum : number = 1;  
let isValid : boolean = true;  
const apiKey : string = "0E5CE8BD-6341-4CC2-904D-C4A94ACD276E";
```

Variables declared with `var` are scoped to the nearest function block (or global, if outside a function block).

Variables declared with `let` are scoped to the nearest enclosing block (or global if outside any block), which can be smaller than a function block.

The `const` keyword creates a constant that can be global or local to the block in which it is declared. This means that constants are block scoped. You will learn more about scopes in *Chapter 5, Runtime*.



The `let` and `const` keywords have been available since the release of TypeScript 1.4 but only when the compilation target is ECMAScript 6. However, they will also work when targeting ECMAScript 3 and ECMAScript 5 once TypeScript 1.5 is released.

Union types

TypeScript allows you to declare union types:

```
var path : string[]|string;  
path = '/temp/log.xml';  
path = ['/temp/log.xml', '/temp/errors.xml'];  
path = 1; // Error
```

Union types are used to declare a variable that is able to store a value of two or more types. In the preceding example, we have declared a variable named `path` that can contain a single path (`string`), or a collection of paths (`array of string`). In the example, we have also set the value of the variable. We assigned a `string` and an `array of strings` without errors; however, when we attempted to assign a numeric value, we got a compilation error because the union type didn't declare a `number` as one of the valid types of the variable.

Type guards

We can examine the type of an expression at runtime by using the `typeof` or `instanceof` operators. The TypeScript language service looks for these operators and will change type inference accordingly when used in an `if` block:

```
var x: any = { /* ... */ };
if(typeof x === 'string') {
    console.log(x.splice(3, 1)); // Error, 'splice' does not exist
    on 'string'
}
// x is still any
x.foo(); // OK
```

In the preceding code snippet, we have declared an `x` variable of type `any`. Later, we check the type of `x` at runtime by using the `typeof` operator. If the type of `x` results to be `string`, we will try to invoke the method `splice`, which is supposed to a member of the `x` variable. The TypeScript language service is able to understand the usage of `typeof` in a conditional statement. TypeScript will automatically assume that `x` must be a `string` and let us know that the `splice` method does not exist on the type `string`. This feature is known as **type guards**.

Type aliases

TypeScript allows us to declare type aliases by using the `type` keyword:

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

Type aliases are exactly the same as their original types; they are simply alternative names. Type aliases can help us to make our code more readable but it can also lead to some problems.

If you work as part of a large team, the indiscriminate creation of aliases can lead to maintainability problems. In the book, *Maintainable JavaScript*, Nicholas C. Zakas, the author recommends to avoid modifying objects you don't own. Nicholas was talking about adding, removing, or overriding methods in objects that have not been declared by you (DOM objects, BOM objects, primitive types, and third-party libraries) but we can apply this rule to the usage of aliases as well.

Ambient declarations

Ambient declaration allows you to create a variable in your TypeScript code that will not be translated into JavaScript at compilation time. This feature was designed to facilitate integration with the existing JavaScript code, the **DOM (Document Object Model)**, and **BOM (Browser Object Model)**. Let's take a look at an example:

```
customConsole.log("A log entry!"); // error
```

If you try to call the member `log` of an object named `customConsole`, TypeScript will let us know that the `customConsole` object has not been declared:

```
// Cannot find name 'customConsole'
```

This is not a surprise. However, sometimes we want to invoke an object that has not been defined, for example, the `console` or `window` objects.

```
console.log("Log Entry!");
var host = window.location.hostname;
```

When we access DOM or BOM objects, we don't get an error because these objects have already been declared in a special TypeScript file known as **declaration files**. You can use the `declare` operator to create an ambient declaration.

In the following code snippet, we will declare an interface that is implemented by the `customConsole` object. We then use the `declare` operator to add the `customConsole` object to the scope:

```
interface ICustomConsole {
    log(arg : string) : void;
}
declare var customConsole : ICustomConsole;
```



Interfaces are explained in greater detail later in the chapter.

We can then use the `customConsole` object without compilation errors:

```
customConsole.log("A log entry!"); // ok
```

TypeScript includes, by default, a file named `lib.d.ts` that provides interface declarations for the built-in JavaScript library as well as the DOM.

Declaration files use the file extension `.d.ts` and are used to increase the TypeScript compatibility with third-party libraries and run-time environments such as Node.js or a web browser.



We will learn how to work with declaration files in *Chapter 2, Automating Your Development Workflow*.

Arithmetic operators

The following arithmetic operators are supported by the TypeScript programming language. In order to understand the examples, you must assume that variable A holds 10 and variable B holds 20.

Operator	Description	Example
+	This adds two operands	A + B will give 30
-	This subtracts the second operand from the first	A - B will give -10
*	This multiplies both the operands	A * B will give 200
/	This divides the numerator by the denominator	B / A will give 2
%	This is the modulus operator and remainder after an integer division	B % A will give 0
++	This is the increment operator that increases the integer value by 1	A++ will give 11
--	This is the decrement operator that decreases the integer value by 1	A-- will give 9

Comparison operators

The following comparison operators are supported by the TypeScript language. In order to understand the examples, you must assume that variable A holds 10 and variable B holds 20.

Operator	Description	Example
==	This checks whether the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is false. A == "10" is true.
====	This checks whether the value and type of two operands are equal or not. If yes, then the condition becomes true.	A === B is false. A === "10" is false.
!=	This checks whether the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.

Operator	Description	Example
>	This checks whether the value of the left operand is greater than the value of the right operand; if yes, then the condition becomes true.	(A > B) is false.
<	This checks whether the value of the left operand is less than the value of the right operand; if yes, then the condition becomes true.	(A < B) is true.
>=	This checks whether the value of the left operand is greater than or equal to the value of the right operand; if yes, then the condition becomes true.	(A >= B) is false.
<=	This checks whether the value of the left operand is less than or equal to the value of the right operand; if yes, then the condition becomes true.	(A <= B) is true.

Logical operators

The following logical operators are supported by the TypeScript language. In order to understand the examples, you must assume that variable A holds 10 and variable B holds 20.

Operator	Description	Example
&&	This is called the logical AND operator. If both the operands are nonzero, then the condition becomes true.	(A && B) is true.
	This is called logical OR operator. If any of the two operands are nonzero, then the condition becomes true.	(A B) is true.
!	This is called the logical NOT operator. It is used to reverse the logical state of its operand. If a condition is true, then the logical NOT operator will make it false.	!(A && B) is false.

Bitwise operators

The following bitwise operators are supported by the TypeScript language. In order to understand the examples, you must assume that variable A holds 2 and variable B holds 3.

Operator	Description	Example
&	This is called the Bitwise AND operator. It performs a Boolean AND operation on each bit of its integer arguments.	(A & B) is 2
	This is called the Bitwise OR operator. It performs a Boolean OR operation on each bit of its integer arguments.	(A B) is 3.

Operator	Description	Example
<code>^</code>	This is called the Bitwise XOR operator. It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.	$(A \wedge B)$ is 1.
<code>~</code>	This is called the Bitwise NOT operator. It is a unary operator and operates by reversing all bits in the operand.	$(\sim B)$ is -4
<code><<</code>	This is called the Bitwise Shift Left operator. It moves all bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, and so on.	$(A << 1)$ is 4
<code>>></code>	This is called the Bitwise Shift Right with sign operator. It moves all bits in its first operand to the right by the number of places specified in the second operand.	$(A >> 1)$ is 1
<code>>>></code>	This is called the Bitwise Shift Right with zero operators. This operator is just like the <code>>></code> operator, except that the bits shifted in on the left are always zero,	$(A >>> 1)$ is 1

[ One of the main reasons to use bitwise operators in languages such as C++, Java, or C# is that they're extremely fast. However, bitwise operators are often considered not that efficient in TypeScript and JavaScript. Bitwise operators are less efficient in JavaScript because it is necessary to cast from floating point representation (how JavaScript stores all of its numbers) to a 32-bit integer to perform the bit manipulation and back.]

Assignment operators

The following assignment operators are supported by the TypeScript language.

Operator	Description	Example
<code>=</code>	This is a simple assignment operator that assigns values from the right-side operands to the left-side operand.	$C = A + B$ will assign the value of $A + B$ into C
<code>+=</code>	This adds the AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.	$C += A$ is equivalent to $C = C + A$

Operator	Description	Example
<code>-=</code>	This subtracts the AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	<code>C -= A</code> is equivalent to <code>C = C - A</code>
<code>*=</code>	This multiplies the AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	<code>C *= A</code> is equivalent to <code>C = C * A</code>
<code>/=</code>	This divides the AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	<code>C /= A</code> is equivalent to <code>C = C / A</code>
<code>%=</code>	This is the modulus AND assignment operator. It takes the modulus using two operands and assigns the result to the left operand.	<code>C %= A</code> is equivalent to <code>C = C % A</code>

Flow control statements

This section describes the decision-making statements, the looping statements, and the branching statements supported by the TypeScript programming language.

The single-selection structure (if)

The following code snippet declares a variable of type Boolean and name `isValid`. Then, an `if` statement will check whether the value of `isValid` is equal to `true`. If the statement turns out to be true, the `Is valid!` message will be displayed on the screen.

```
var isValid : boolean = true;

if(isValid) {
    alert("is valid!");
}
```

The double-selection structure (if...else)

The following code snippet declares a variable of type Boolean and name `isValid`. Then, an `if` statement will check whether the value of `isValid` is equal to `true`. If the statement turns out to be true, the message `Is valid!` will be displayed on the screen. On the other side, if the statement turns out to be false, the message `Is NOT valid!` will be displayed on the screen.

```
var isValid : boolean = true;

if(isValid) {
    alert("Is valid!");
}
```

```
    }
} else {
    alert("Is NOT valid!");
}
```

The inline ternary operator (?)

The inline ternary operator is just an alternative way of declaring a double-selection structure.

```
var isValid : boolean = true;
var message = isValid ? "Is valid!" : "Is NOT valid!";
alert(message);
```

The preceding code snippet declares a variable of type Boolean and name `isValid`. Then it checks whether the variable or expression on the left-hand side of the operator `?` is equal to true.

If the statement turns out to be true, the expression on the left-hand side of the character will be executed and the message `Is valid!` will be assigned to the message variable.

On the other hand, if the statement turns out to be false, the expression on the right-hand side of the operator will be executed and the message, `Is NOT valid!` will be assigned to the message variable.

Finally, the value of the message variable is displayed on the screen.

The multiple-selection structure (switch)

The `switch` statement evaluates an expression, matches the expression's value to a case clause, and executes statements associated with that case. A `switch` statement and enumerations are often used together to improve the readability of the code.

In the following example, we will declare a function that takes an enumeration `AlertLevel`. Inside the function, we will generate an array of strings to store e-mail addresses and execute a `switch` structure. Each of the options of the enumeration is a case in the `switch` structure:

```
enum AlertLevel{
    info,
    warning,
    error
}
```

```

function getAlertSubscribers(level : AlertLevel){
    var emails = new Array<string>();
    switch(level) {
        case AlertLevel.info:
            emails.push("cst@domain.com");
            break;
        case AlertLevel.warning:
            emails.push("development@domain.com");
            emails.push("sysadmin@domain.com");
            break;
        case AlertLevel.error:
            emails.push("development@domain.com");
            emails.push("sysadmin@domain.com");
            emails.push("management@domain.com");
            break;
        default:
            throw new Error("Invalid argument!");
    }
    return emails;
}

getAlertSubscribers(AlertLevel.info); // ["cst@domain.com"]
getAlertSubscribers(AlertLevel.warning); // ["development@domain.com", "sysadmin@domain.com"]

```

The value of the level variable is tested against all the cases in the switch. If the variable matches one of the cases, the statement associated with that case is executed. Once the case statement has been executed, the variable is tested against the next case.

Once the execution of the statement associated to a matching case is finalized, the next case will be evaluated. If the `break` keyword is present, the program will not continue the execution of the following case statement.

If no matching case clause is found, the program looks for the optional `default` clause, and if found, it transfers control to that clause and executes the associated statements.

If no `default` clause is found, the program continues execution at the statement following the end of switch. By convention, the `default` clause is the last clause, but it does not have to be so.

The expression is tested at the top of the loop (while)

The `while` expression is used to repeat an operation while a certain requirement is satisfied. For example, the following code snippet, declares a numeric variable `i`. If the requirement (the value of `i` is less than 5) is satisfied, an operation takes place (increase the value of `i` by 1 and display its value in the browser console). Once the operation has completed, the accomplishment of the requirement will be checked again.

```
var i : number = 0;
while (i < 5) {
    i += 1;
    console.log(i);
}
```

In a `while` expression, the operation will take place only if the requirement is satisfied.

The expression is tested at the bottom of the loop (do...while)

The `do-while` expression is used to repeat an operation until a certain requirement is not satisfied. For example, the following code snippet declares a numeric variable `i` and repeats an operation (increase the value of `i` by 1 and display its value in the browser console) for as long as the requirement (the value of `i` is less than 5) is satisfied.

```
var i : number = 0;
do {
    i += 1;
    console.log(i);
} while (i < 5);
```

Unlike the `while` loop, the `do-while` expression will execute at least once regardless of the requirement value as the operation will take place before checking if a certain requirement is satisfied or not.

Iterate on each object's properties (for...in)

The `for-in` statement by itself is not a *bad practice*; however, it can be misused, for example, to iterate over arrays or array-like objects. The purpose of the `for-in` statement is to enumerate over object properties.

```
var obj : any = { a:1, b:2, c:3 };  
for (var key in obj) {  
    console.log(key + " = " + obj[key]);  
}  
  
// Output:  
// "a = 1"  
// "b = 2"  
// "c = 3"
```

The following code snippet will go up in the prototype chain, also enumerating the inherited properties. The `for-in` statement iterates the entire prototype chain, also enumerating the inherited properties. When you want to enumerate only the object's own properties (the ones that aren't inherited), you can use the `hasOwnProperty` method:

```
for (var key in obj) {  
    if (obj.hasOwnProperty(prop)) {  
        // prop is not inherited  
    }  
}
```

Counter controlled repetition (for)

The `for` statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or a set of statements executed in the loop.

```
for (var i: number = 0; i < 9; i++) {  
    console.log(i);  
}
```

The preceding code snippet contains a `for` statement, it starts by declaring the variable `i` and initializing it to 0. It checks whether `i` is less than 9, performs the two succeeding statements, and increments `i` by 1 after each pass through the loop.

Functions

Just as in JavaScript, TypeScript functions can be created either as a named function or as an anonymous function. This allows us to choose the most appropriate approach for an application, whether we are building a list of functions in an API or a one-off function to hand over to another function.

```
// named function
function greet(name? : string) : string {
    if(name){
        return "Hi! " + name;
    }
    else
    {
        return "Hi!";
    }
}

// anonymous function
var greet = function(name? : string) : string {
    if(name){
        return "Hi! " + name;
    }
    else
    {
        return "Hi!";
    }
}
```

As we can see in the preceding code snippet, in TypeScript we can add types to each of the parameters and then to the function itself to add a return type. TypeScript can infer the return type by looking at the return statements, so we can also optionally leave this off in many cases.

There is an alternative function syntax, which uses the arrow (`=>`) operator after the function's return type and skips the usage of the `function` keyword.

```
var greet = (name : string) : string => {
    if(name){
        return "Hi! " + name;
    }
    else
    {
        return "Hi! my name is " + this.fullname;
    }
};
```

The functions declared using this syntax are commonly known as **arrow functions**. Let's return to the previous example in which we were assigning an anonymous function to the `greet` variable. We can now add the type annotations to the `greet` variable to match the anonymous function signature.

```
var greet : (name : string) => string = function(name : string) :  
string {  
    if(name){  
        return "Hi! " + name;  
    }  
    else  
    {  
        return "Hi!";  
    }  
};
```

[ Keep in mind that the arrow function (`=>`) syntax changes the way the `this` operator works when working with classes. We will learn more about this in the upcoming chapters.]

Now you know how to add type annotations to force a variable to be a function with a specific signature. The usage of this kind of annotations is really common when we use a call back (functions used as an argument of another function).

```
function sume(a : number, b : number, callback : (result:number)  
=> void){  
    callback(a+b);  
}
```

In the preceding example, we are declaring a function named `sume` that takes two numbers and a `callback` as a function. The type annotations will force the `callback` to return `void` and take a `number` as its only argument.

[ We will focus on functions in *Chapter 3, Working with Functions*.]

Classes

ECMAScript 6, the next version of JavaScript, adds class-based object orientation to JavaScript and, since TypeScript is based on ES6, developers are allowed to use class-based object orientation today, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

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

```
class Character {  
    fullname : string;  
    constructor(firstname : string, lastname : string) {  
        this.fullname = firstname + " " + lastname;  
    }  
    greet(name? : string) {  
        if(name)  
        {  
            return "Hi! " + name + "! my name is " + this.fullname;  
        }  
        else  
        {  
            return "Hi! my name is " + this.fullname;  
        }  
    }  
}  
  
var spark = new Character("Jacob", "Keyes");  
var msg = spark.greet();  
alert(msg); // "Hi! my name is Jacob Keyes"  
var msg1 = spark.greet("Dr. Halsey");  
alert(msg1); // "Hi! Dr. Halsey! my name is Jacob Keyes"
```

In the preceding example, we have declared a new class `Character`. This class has three members: a property called `fullname`, a constructor, and a method `greet`. When we declare a class in TypeScript, all the methods and properties are public by default.

You'll notice that when we refer to one of the members of the class (from within itself) we prepend the `this` operator. The `this` operator denotes that it's a member access. In the last lines, we construct an instance of the `Character` class using a `new` operator. This calls into the constructor we defined earlier, creating a new object with the `Character` shape, and running the constructor to initialize it.

TypeScript classes are compiled into JavaScript functions in order to achieve compatibility with ECMAScript 3 and ECMAScript 5.



We will learn more about classes and other object-oriented programming concepts in *Chapter 4, Object-Oriented Programming with TypeScript*.

Interfaces

In TypeScript, we can use interfaces to enforce that a class follow the specification in a particular contract.

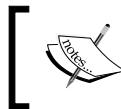
```
interface LoggerInterface{
    log(arg : any) : void;
}

class Logger implements LoggerInterface{
    log(arg) {
        if(typeof console.log === "function"){
            console.log(arg);
        }
        else
        {
            alert(arg);
        }
    }
}
```

In the preceding example, we have defined an interface `loggerInterface` and a class `Logger`, which implements it. TypeScript will also allow you to use interfaces to declare the type of an object. This can help us to prevent many potential issues, especially when working with object literals:

```
interface UserInterface{
    name : string;
    password : string;
}

var user : UserInterface = {
    name : "",
    password : "" // error property password is missing
};
```



We will learn more about interfaces and other object-oriented programming concepts in *Chapter 4, Object-Oriented Programming with TypeScript*.

Namespaces

Namespaces, also known as internal modules, are used to encapsulate features and objects that share a certain relationship. Namespaces will help you to organize your code in a much clearer way. To declare a namespace in TypeScript, you will use the `namespace` and `export` keywords.

```
namespace Geometry{
    interface VectorInterface {
        /* ... */
    }
    export interface Vector2dInterface {
        /* ... */
    }
    export interface Vector3dInterface {
        /* ... */
    }
    export class Vector2d implements VectorInterface,
Vector2dInterface {
        /* ... */
    }
    export class Vector3d implements VectorInterface,
Vector3dInterface {
        /* ... */
    }
}

var vector2dInstance : Geometry.Vector2dInterface = new
Geometry.Vector2d();
var vector3dInstance : Geometry.Vector3dInterface = new
Geometry.Vector3d();
```

In the preceding code snippet, we have declared a namespace that contains the classes `vector2d` and `vector3d` and the interfaces `VectorInterface`, `Vector2dInterface`, and `Vector3dInterface`. Note that the first interface is missing the keyword `export`. As a result, the interface `VectorInterface` will not be accessible from outside the namespace's scope.



In *Chapter 4, Object-Oriented Programming with TypeScript*, we'll be covering namespaces (internal modules) and external modules and we'll discuss when each is appropriate and how to use them.

Putting everything together

Now that we have learned how to use the basic TypeScript building blocks individually, let's take a look at a final example in which we will use modules, classes, functions, and type annotations for each of these elements:

```
module Geometry{  
    export interface Vector2dInterface {  
        toArray(callback : (x : number[]) => void) : void;  
        length() : number;  
        normalize();  
    }  
    export class Vector2d implements Vector2dInterface {  
        private _x: number;  
        private _y : number;  
        constructor(x : number, y : number){  
            this._x = x;  
            this._y = y;  
        }  
        toArray(callback : (x : number[]) => void) : void{  
            callback([this._x, this._y]);  
        }  
        length() : number{  
            return Math.sqrt(this._x * this._x + this._y * this._y);  
        }  
        normalize(){  
            var len = 1 / this.length();  
            this._x *= len;  
            this._y *= len;  
        }  
    }  
}
```

The preceding example is just a small portion of a basic 3D engine written in JavaScript. In 3D engines, there are a lot of mathematical calculations involving matrices and vectors. As you can see, we have defined a module `Geometry` that will contain some entities; to keep the example simple, we have only added the class `Vector2d`. This class stores two coordinates (`x` and `y`) in 2d space and performs some operations on the coordinates. One of the most used operations on vectors is normalization, which is one of the methods in our `Vector2d` class.

3D engines are complex software solutions, and as a developer, you are much more likely to use a third-party 3D engine than create your own. For this reason, it is important to understand that TypeScript will not only help you to develop large-scale applications, but also to work with large-scale applications. In the following code snippet, we will use the module declared earlier to create a `Vector2d` instance:

```
var vector : Geometry.Vector2dInterface = new
Geometry.Vector2d(2,3);
vector.normalize();
vector.toArray(function(vectorAsArray : number[]) {
    alert(' x :' + vectorAsArray[0] + ' y : ' + vectorAsArray[1]);
});
```

The type checking and IntelliSense features will help us create a `Vector2d` instance, normalize its value, and convert it into an array to finally show its value on screen with ease.



```
28 var vector : Geometry.Vector2dInterface = new Geometry.Vector2d(2,3);
29 vector.|
```

30 ↳ length (method) Geometry.Vector2dInterface.length(): number
↳ normalize
↳ toArray

Summary

In this chapter, you have learned about the purposes of TypeScript. You have also learned about some of the design decisions made by the TypeScript engineers at Microsoft.

Towards the end of this chapter, you learned a lot about the basic building blocks of a TypeScript application. You started to write some TypeScript code for the first time and you can now work with type annotations, variables and primitive data types, operators, flow control statements, functions, and classes.

In the next chapter, you will learn how to automate your development workflow.

2

Automating Your Development Workflow

After taking a first look at the main TypeScript language features, we will now learn how to use some tools to automate our development workflow. These tools will help us to reduce the amount of time that we usually spend on simple and repetitive tasks.

In this chapter, we will learn about the following topics:

- An overview of the development workflow
- Source control tools
- Package management tools
- Task runners
- Test runners
- Integration tools
- Scaffolding tools

A modern development workflow

Developing a web application with high quality standards has become a time-consuming activity. If we want to achieve a great user experience, we will need to ensure that our applications can run as smoothly as possible on many different web browsers, devices, Internet connection speeds, and screen resolutions. Furthermore, we will need to spend a lot of our time working on quality assurance and performance optimization tasks.

As developers, we should try to minimize the time spent on simple and repetitive tasks. This might sound familiar as we have been doing this for years. We started by writing build scripts (such as makefiles) or automated tests and today, in a modern web development workflow, we use many tools to try to automate as many tasks as we can. These tools can be categorized into the following groups:

- Source control tools
- Package management tools
- Task runners
- Test runners
- Continuous integration tools
- Scaffolding tools

Prerequisites

You are about to learn how to write a script, which will automate many tasks in your development workflow; however, before that, we need to install a few tools in our development environment.

Node.js

Node.js is a platform built on V8 (Google's open source JavaScript engine). Node.js allows us to run JavaScript outside a web browser. We can write backend and desktop applications using JavaScript with Node.js.

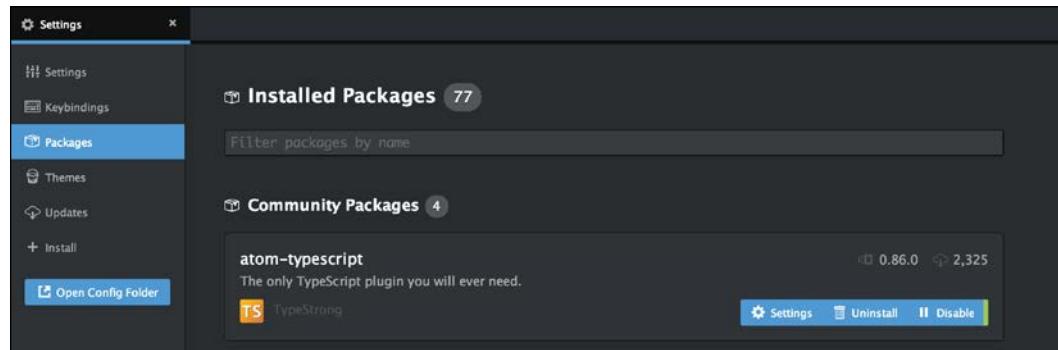
We are not going to write server-side JavaScript applications but we are going to need Node.js because many of the tools used in this chapter are Node.js applications.

If you didn't install Node.js in the previous chapter, you can visit <https://nodejs.org> to download the installer for your operating system.

Atom

Atom is an open source editor developed by the GitHub team. The open source community around this editor is really active and has developed many plugins and themes. You can download Atom from <https://atom.io/>.

Once you have completed the installation, open the editor and go to the preferences window. You should be able to find a section within the preferences window to manage packages and another to manage themes just like the ones that we can see in the following screenshot:



The Atom user interface is slightly different from the other operating systems. Refer to the Atom documentation at <https://atom.io/docs> if you need additional help to manage packages and themes.

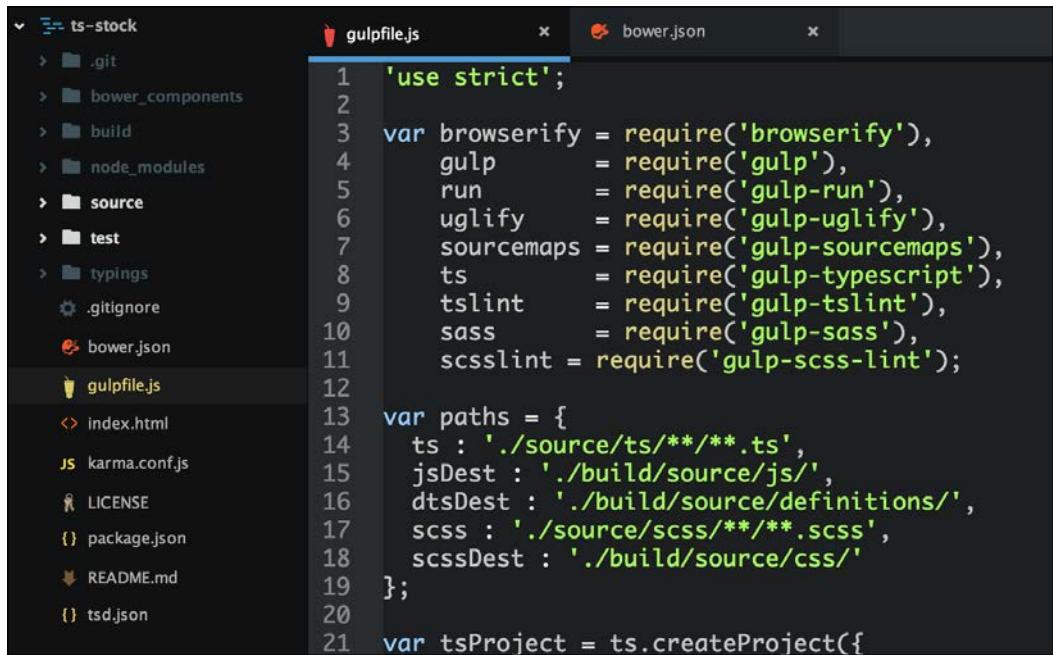
We need to search for the `atom-typescript` package in the package management section and install it. We can additionally visit the themes section and install a theme that makes us feel more comfortable with the editor.

We will use Atom instead of Visual Studio because Atom is available for Linux, OS X, and Windows, so it will suit most readers.

Unfortunately, we will not cover Visual Studio Code because it was announced when this book was about to be published. Visual Studio Code is a lightweight IDE developed by Microsoft and available for free for Windows, OS X, and Linux. You can visit <https://code.visualstudio.com/> if you wish to learn more about it.

If you want to work with Visual Studio, you will be able to find the extension to enable Typescript support in Visual Studio at <https://visualstudiogallery.msdn.microsoft.com/2d42d8dc-e085-45eb-a30b-3f7d50d55304>.

One of the highest rated themes is called `seti-ui` and is particularly useful because it uses a really good set of icons to help us to identify each file in our application. For example, the `gulpfile.js` or `bower.json` files (we will learn about these files later) are just JavaScript and JSON files but the `seti-ui` theme is able to identify that they are the Gulp and Bower configuration files respectively and will display their icons accordingly.



```
1  'use strict';
2
3  var browserify = require('browserify'),
4      gulp       = require('gulp'),
5      run        = require('gulp-run'),
6      uglify     = require('gulp-uglify'),
7      sourcemaps = require('gulp-sourcemaps'),
8      ts          = require('gulp-typescript'),
9      tslint      = require('gulp-tslint'),
10     sass        = require('gulp-sass'),
11     scsslint   = require('gulp-scss-lint');
12
13 var paths = {
14     ts : './source/ts/**/*.*ts',
15     jsDest : './build/source/js/',
16     dtsDest : './build/source/definitions/',
17     scss : './source/scss/**/*.*scss',
18     scssDest : './build/source/css/'
19 };
20
21 var tsProject = ts.createProject({
```

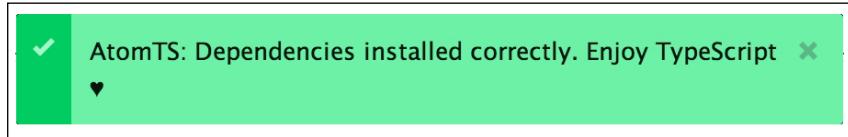
We can install this theme by opening the console of our operating system and running the following commands:

```
cd ~/.atom/packages
git clone https://github.com/jesseweed/seti-ui --depth=1
```



You need to install Git to be able to run the preceding command. You will find some information about the Git installation later on in this chapter.

Once we have installed the theme and TypeScript plugin, we will need to close the Atom editor and open it again to make the changes effective. If everything goes well, we will get a confirmation message in the top-right corner of the editor window.



Git and GitHub

Towards the end of this chapter, we will learn how to configure a continuous integration build server. The build server will observe changes in our application's code and ensure that the changes don't break the application.

In order to be able to observe the changes in the code, we will need to use a source control system. There are a few source control systems available. Some of the most widely used ones are Subversion, Mercurial and Git.

Source control systems have many benefits. First, they enable multiple developers to work on a source file without any work being overridden.

Second, source control systems are also a good way of keeping previous copies of a file or auditing its changes. These features can be really useful, for example, when trying to find out when a new bug was introduced for the first time.

While working through the examples, we will perform some changes to the source code. We will use Git and GitHub to manage these changes. To install Git, go to <http://git-scm.com/downloads> and download the executable for your operating system. Then, go to <https://github.com/> to create a GitHub account. While creating the GitHub account, you will be offered a few different subscription plans, the free plan offers everything we need to follow the examples in this chapter.

Source control tools

Now that we have installed Git and created a GitHub account, we will use GitHub to create a new code repository. A repository is a central file storage location. It is used by the source control systems to store multiple versions of files. While a repository can be configured on a local machine for a single user, it is often stored on a server, which can be accessed by multiple users.

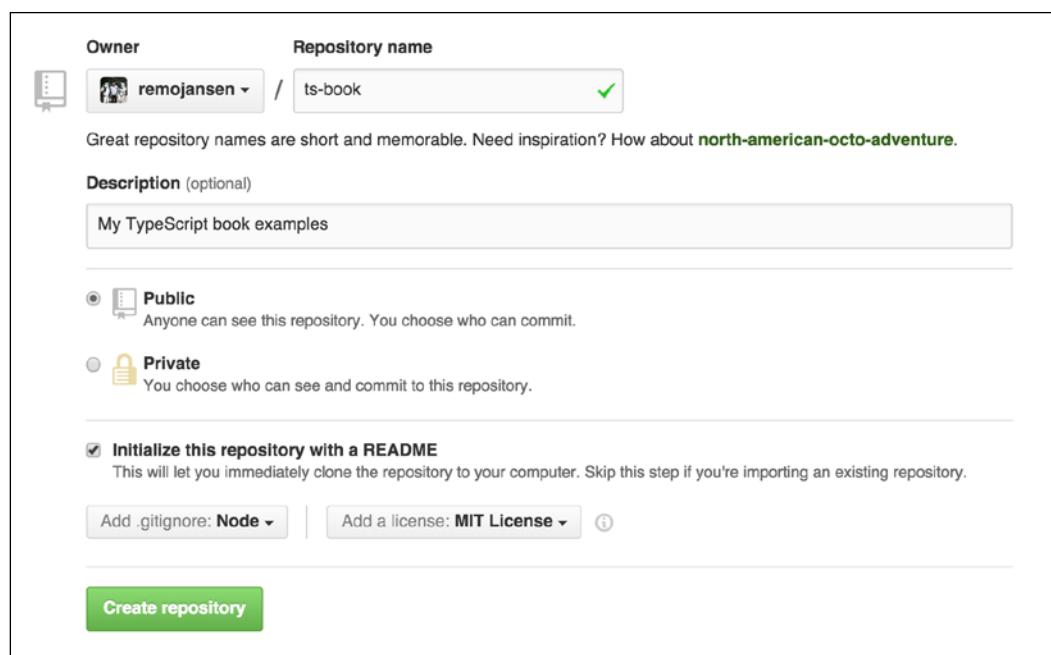


GitHub offers free source control repositories for open source projects. GitHub is really popular within the open source community and many popular projects are hosted on GitHub (including TypeScript). However, GitHub is not the only option available and you can use a local Git repository or another source control service provider such as Bitbucket. If you wish to learn more about these alternatives, refer to the official Git documentation at <https://git-scm.com/doc> or the BitBucket website at <https://bitbucket.org/>.

To create a new repository on GitHub, log in to your GitHub account and click on the link to create a new repository, which we can find in the top-right corner of the screen.



A web form similar to the one in the following screenshot will then appear. This form contains some fields, which allow us to set the repository's name, description, and privacy settings.



Owner **remojansen** Repository name **ts-book**

Great repository names are short and memorable. Need inspiration? How about [north-american-octo-adventure](#).

Description (optional)
My TypeScript book examples

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

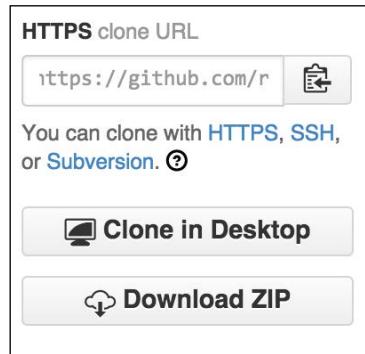
Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **Node** | Add a license: **MIT License** ⓘ

Create repository

We can also add a `README.md` file, which uses markdown syntax and is used to add whatever text we want to the repository's home page on GitHub. Furthermore, we can add a default `.gitignore` file, which is used to specify files that we would like to be ignored by Git and therefore not saved into the repository.

Last but not least, we can also select a software license to cover our source code. Once we have created the repository, we will navigate to our profile page on GitHub, find the repository that we have just created, and go to the repository's page. On the repository's page, we will be able to find the clone URL at the bottom-right corner of the page.



We need to copy the repository's clone URL, open a console, and use the URL as an argument of the `git clone` command:

```
git clone https://github.com/user-name/repository-name.git
```

Sometimes the Windows command-line interface is not able to find the Git and Node.js commands.

The easiest way to get around this issue is to use the Git console (installed with Git) rather than using the Windows command line.

If you want to use the Windows console, you will need to manually add the Git and Node installation paths to the Windows PATH environment variable.

Also, note that we will use the UNIX path syntax in all the examples.

If you are working with OS X or Linux, the default command-line interface should work fine.



The command output should look similar to this:

```
Cloning into 'repository-name'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

We can then move inside the repository by using the change directory command (cd) and use the git status command to check the local repository's status:

```
cd repository-name
git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

We will use GitHub throughout this book. However, if you want to use a local repository, you can use the Git init command to create an empty repository.



Refer to the Git documentation at <http://git-scm.com/docs/git-init> to learn more about the git init command and working with a local repository.

The git status command is telling us that there are no changes in our working directory. Let's open the repository folder in Atom and create a new file called gulpfile.js. Now, run the git status command again, and we will see that there are some new untracked files:

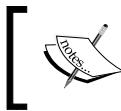
```
On branch master
Your branch is up-to-date with 'origin/master'.
```

Untracked files:

(use "git add <file>..." to include in what will be committed)

gulpfile.js

```
nothing added to commit but untracked files present (use "git add" to track)
```



The files in the Atom project explorer are displayed using a color code, which will help us to identify whether a file is new, or has changed since we cloned the repository.

When we make some changes, such as adding a new file or changing an existing file, we need to execute the `git add` command to indicate that we want to add that change to a snapshot:

```
git add gulpfile.js  
git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)
```

`new file: gulpfile.js`

Now that we have staged the content we want to snapshot, we have to run the `git commit` command to actually record the snapshot. Recording a snapshot requires a commentary field, which can be provided using the `git commit` command together with its `-m` argument:

```
git commit -m "added the new gulpfile.js"
```

If everything has gone well, the command output should be similar to the following:

```
[master 2a62321] added the new file gulpfile.js  
1 file changed, 1 insertions(+)  
create mode 100644 gulpfile.js
```

To share the commit with other developers, we need to push our changes to the remote repository. We can do this by executing the `git push` command:

```
git push
```

The `git push` command will ask for our GitHub username and password and then send the changes to the remote repository. If we visit the repository's page on GitHub, we will be able to find the recently created file. We will return to GitHub later in this chapter to configure our continuous integration server.



If you are working in a large team, you might encounter some file conflicts when attempting to push some changes to the remote repository. Resolving a file conflict is out of the scope of this book; however, if you need further information about Git, you will find an extensive user manual at <https://www.kernel.org/pub/software/scm/git/docs/user-manual.html>.

Package management tools

Package management tools are used for dependency management, so that we no longer have to manually download and manage our application's dependencies. We will learn how to work with three different package management tools: Bower, npm, and tsd.

npm

The npm package manager was originally developed as the default Node.js package management tool, but today it is used by many tools. Npm uses a configuration file, called `package.json`, to store references to all the dependencies in our application. It is important to remember that we will normally use npm to install dependencies that we will use on the server side, in a desktop application, or with development tools.

Before we install any packages, we should add a `package.json` file to our project. We can do it by executing the following command:

```
npm init
```

The `npm init` command will ask for some basic information about our project, including its name, version, description, entry point, test command, Git repository, keywords, author and license.



Refer to the official npm documentation at <https://docs.npmjs.com/files/package.json> if you are unsure about the purposes of some of the `package.json` fields mentioned earlier.

The `npm` command will then show us a preview of the `package.json` file that is about to be generated and ask for our final confirmation.



Remember that you need to have Node.js installed to be able to use the `npm` command tool.

After creating the project's package.json file, run the `npm install` command to install our first dependency. The `npm install` command takes the name of one or multiple dependencies separated by a single space as an argument and a second argument to indicate the scope of the installation.

The scope can be:

- A dependency at development time (testing frameworks, compilers, and so on)
- A dependency at runtime (a web framework, database ORMs, and so on)

We will use the `gulp-typescript` npm package to compile our TypeScript code; so, let's install it as a development dependency (using the `--save-dev` argument):

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

To install a global dependency, we will use the `-g` argument:

```
npm install typescript -g
```

We might need administrative privileges to install packages with global scope in our development environment, as we already learned in the previous chapter.

Also, note that npm will not add any entries to our `package.json` file when installing packages with global scope but it is important that we manually add the right dependencies to the `devDependencies` and `peerDependencies` sections in the `package.json` file to guarantee that the continuous integration build server will resolve all our project's dependencies correctly. We will learn about the continuous integration build server in detail later in this chapter.

To install a runtime dependency, use the `--save` argument:

```
npm install jquery --save
```

JQuery is probably the most popular JavaScript framework or library ever created. It is used to facilitate the usage of some browser APIs without having to worry about some vendor-specific differences in the APIs. JQuery also provides us with many helpers that will help us reduce the amount of code necessary to perform tasks such as selecting an HTML node within the tree of nodes in an HTML document.

It is assumed that the readers of this book have a good understanding of JQuery. If you need to learn more about JQuery, refer to the official documentation at <https://api.jquery.com/>.

Once we have installed some dependencies in the package.json file, the contents should look similar to this:

```
{  
  "name": "repository-name",  
  "version": "1.0.0",  
  "description": "example",  
  "main": "index.html",  
  "scripts": {  
    "test": "test"  
  },  
  "repository": {  
    "type": "git",  
    "url": "https://github.com/username/repository-name.git"  
  },  
  "keywords": [  
    "typescript",  
    "demo",  
    "example"  
  ],  
  "author": "Name Surname",  
  "contributors": [],  
  "license": "MIT",  
  "bugs": {  
    "url": "https://github.com/username/repository-name/issues"  
  },  
  "homepage": "https://github.com/username/repository-name",  
  "engines": {},  
  "dependencies": {  
    "jquery": "^2.1.4"  
  },  
  "devDependencies": {  
    "gulp-typescript": "^2.8.0"  
  }  
}
```

Some fields in the package.json file must be configured manually. To learn more about the available package.json configuration fields, visit <https://docs.npmjs.com/files/package.json>.



The versions of the npm packages used throughout this book may have been updated since the publication of this book. Refer to the packages documentation at <https://npmjs.com> to find out potential incompatibilities and learn about new features.

All the npm packages will be saved under the `node_modules` directory. We should add the `node_modules` directory to our `.gitignore` file as it is recommended to avoid saving the application's dependencies into source control. We can do this by opening the `.gitignore` file and adding a new line that contains the name of the folder (`node_modules`).

The next time we clone our repository, we will need to download all our dependencies again, but to do so, we will only need to execute the `npm install` command without any additional parameters:

```
npm install
```

The package manager will then search for the package `.json` file and install all the declared dependencies.



If, in the future, we need to find an npm package name, we will be able to use the npm search engines at <https://www.npmjs.com> in order to find it.

Bower

Bower is another package management tool. It is really similar to npm but it was designed specifically to manage frontend dependencies. As a result, many of the packages are optimized for its usage in a web browser.

We can install Bower by using npm:

```
npm install -g bower
```

Instead of the package `.json` file, Bower uses a configuration file named `bower.json`. We can use the majority of the npm commands and arguments in Bower. For example, we can use the `bower init` command to create the initial bower configuration file:

```
bower init
```



The initial configuration file is quite similar to the package `.json` file. Refer to the official documentation at <http://bower.io/docs/config/> if you want to learn more about the `bower.json` configuration fields.

We can also use the `bower install` command to install a package:

```
bower install jquery
```

Furthermore, we can also use the install scope arguments:

```
bower install jquery --save  
bower install jasmine --save-dev
```

All the Bower packages will be saved under the `bower_components` directory. As you have already learned, it is recommended to avoid saving your application's dependencies in your remote repository, so you should also add the `bower_components` directory to your `.gitignore` file.

tsd

In the previous chapter, we learned that TypeScript by default includes a file `lib.d.ts` that provides interface declarations for the built-in JavaScript objects as well as the **Document Object Model (DOM)** and **Browser Object Model (BOM)** APIs. The TypeScript files with the extension `.d.ts` are a special kind of TypeScript file known as **type definition files** or **declaration files**.

The type definition files usually contain the type declarations of third-party libraries. These files facilitate the integration between the existing JavaScript libraries and TypeScript. If, for example, we try to invoke the JQuery in a TypeScript file, we will get an error:

```
$ .ajax({ / **/ }); // cannot find name '$'
```

To resolve this issue, we need to add a reference to the JQuery type definition file in our TypeScript code, as shown in the following line of code:

```
///
```

Fortunately, we don't need to create the type definition files because there is an open source project known as **DefinitelyTyped** that already contains type definition files for many JavaScript libraries. In the early days of TypeScript development, developers had to manually download and install the type definition files from the DefinitelyTyped project website, but those days are gone, and today we can use a much better solution known as `tsd`.

The `tsd` acronym stands for **TypeScript Definitions** and it is a package manager that will help us to manage the type definition files required by our TypeScript application. Just like `npm` and `bower`, `tsd` uses a configuration file named `tsd.json` and stores all the downloaded packages under a directory named `typings`.

Run the following command to install `tsd`:

```
npm install tsd -g
```

We can use the `tsd init` command to generate the initial `tsd.json` file and the `tsd install` command to download and install dependencies:

```
tsd init // generate tsd.json  
tsd install jquery --save // install jquery type definitions
```

You can visit the DefinitelyTyped project website at <https://github.com/borisyankov/DefinitelyTyped> to search for `tsd` packages.

Task runners

A task runner is a tool used to automate tasks in the development process. The task can be used to perform a wide variety of operations such as the compilation of TypeScript files or the compression of JavaScript files. The two most popular JavaScript task runners these days are Grunt and Gulp.

Grunt started to become popular in early 2012 and since then the open source community has developed a large number of Grunt-compatible plugins.

On the other hand, Gulp started to become popular in late 2013; therefore, there are less plugins available for Gulp, but it is quickly catching up with Grunt.

Besides the number of plugins available, the main difference between Gulp and Grunt is that while in Grunt we will work using files as the input and output of our tasks, in Gulp we will work with streams and pipes instead. Grunt is configured using some configuration fields and values. However, Gulp prefers code over configuration. This approach makes the Gulp configuration somehow more minimalistic and easier to read.



In this book, we will work with Gulp; however, if you want to learn more about Grunt, you can do so at <http://gruntjs.com/>.

In order to gain a good understanding of Gulp, we can use the project that we have already created and add some extra folders and files to it. Alternatively, we can start a new project from scratch. We will configure some tasks, which will reference paths, folders, and files numerous times, so the following directory tree structure should help us understand each of these tasks:

```
|── LICENSE  
|── README.md  
|── index.html  
|── gulpfile.js  
|── karma.conf.js
```

```
└── tsd.json
└── package.json
└── bower.json
└── source
    └── ts
        └── *.ts
└── test
    └── main.test.ts
└── data
    └── *.json
└── node_modules
    └── ...
└── bower_components
    └── ...
└── typings
    └── ...
```

A copy of a finished example project is provided in the companion source code. The code is provided to help you follow the content. You can use the finished project to help improve the understanding of the concepts discussed in the rest of this chapter.



Let's start by installing gulp globally with npm:

```
npm install -g gulp
```

Then install gulp in our package.json devDependencies:

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

Create a JavaScript file named `gulpfile.js` inside the root folder of our project, which should contain the following piece of code:

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

gulp.task('default', function() {
  console.log('Hello Gulp!');
});
```

And, finally, run `gulp` (we must execute this command from where the `gulpfile.js` file is located):

```
gulp
```

We have created our first Gulp task, which is named `default`. When we run the `gulp` command, it will automatically try to search for the `gulpfile.js` file in the current directory, and once found, it will try to find the default task.

Checking the quality of the TypeScript code

The `default` task is not performing any operations in the preceding example, but we will normally use a Gulp plugin in each task. We will now add a second task, which will use the `gulp-tslint` plugin to check whether our TypeScript code follows a series of recommended practices.

We need to install the plugin with `npm`:

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

We can then load the plugin in to our `gulpfile.js` file and add a new task:

```
var tslint = require('gulp-tslint');
gulp.task('lint', function() {
    return gulp.src([
        './source/ts/**/*.*ts', './test/**/*.*test.ts'
    ]).pipe(tslint())
        .pipe(tslint.report('verbose'));
});
```

We have named the new task `lint`. Let's take a look at the operations performed by the `lint` task, step by step:

1. The `gulp src` function will fetch the files in the directory located at `./source/ts` and its subdirectories with the file extension `.ts`. We will also fetch all the files in the directory located at `./test` and its subdirectories with the file extension `.test.ts`.
2. The output stream of the `src` function will be then redirected using the `pipe` function to be used as the `tslint` function input.
3. Finally, we will use the output of the `tslint` function as the input of the `tslint.report` function.

Now that we have added the `lint` task, we will modify the `gulpfile.js` file to indicate that we want to run `lint` as a subtask of the `default` task:

```
gulp.task('default', ['lint']);
```



Many plugins allow us to indicate that some files should be ignored by adding the exclamation symbol (!) before a path. For example, the path `!path/*.d.ts` will ignore all files with the extension `.d.ts`; this is useful when the declaration files and source code files are located in the same directory.

Compiling the TypeScript code

We will now add two new tasks to compile our TypeScript code (one for the application's logic and one for the application's unit tests).

We will use the `gulp-typescript` plugin, so remember to install it as a development dependency using the npm package manager, just as we did previously in this chapter:

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

We can then create a new `gulp-typescript` project object:

```
var ts = require('gulp-typescript');
var tsProject = ts.createProject({
    removeComments : true,
    noImplicitAny : true,
    target : 'ES3',
    module : 'commonjs',
    declarationFiles : false
});
```

It has been announced that the `gulp-typescript` plugin will soon support the usage of a special JSON file named `tsconfig.json`. This file is used to store the TypeScript compiler configuration. When the file is available, it is used by the compiler during the compilation process.



The `tsconfig.json` file is useful because it prevents us from having to write all the desired compiler parameters when using its console interface. Refer to the `gulp-typescript` documentation, which can be found at <https://www.npmjs.com/package/gulp-typescript>, to learn more about this feature.

In the preceding code snippet, we have loaded the TypeScript compiler as a dependency and then created an object named `tsProject`, which contains the settings to be used by the TypeScript compiler during the compilation of our code. We are now ready to compile our application's source code:

```
gulp.task('tsc', function() {
  return gulp.src('./source/ts/**/*.{ts,tsd}')
    .pipe(ts(tsProject))
    .js.pipe(gulp.dest('./temp/source/js'));
});
```

The `tsc` task will fetch all the `.ts` files in the directory located at `./source/ts` and its subdirectories and pass them as a stream to the TypeScript compiler. The compiler will use the compilation settings passed as the `tsProject` argument and then save the output JavaScript files into the path `./temp/sources/js`.

We also need to compile some unit tests written in TypeScript. The tests are located in the `test` folder and we want the output JavaScript files to be stored under `temp/test`. Using the same project configuration object in a different task and with different input files can result in bad performance and unexpected behavior; so we need to initialize another `gulp-typescript` project object. This time we will name the object `tsTestProject`:

```
var tsTestProject = ts.createProject({
  removeComments : true,
  noImplicitAny : true,
  target : 'ES3',
  module : 'commonjs',
  declarationFiles : false
});
```

The `tsc-test` task is almost identical to the `tsc` task, but instead of compiling the application's code, it will compile the application's tests. Since the source and test are located in different directories, we have used different paths in this task:

```
gulp.task('tsc-tests', function() {
  return gulp.src('./test/**/*.{test,ts}')
    .pipe(ts(tsTestProject ))
    .js.pipe(gulp.dest('./temp/test/'));
});
```

We will update the default task once more in order to perform the new tasks:

```
gulp.task('default', ['lint', 'tsc', 'tsc-tests']);
```

Optimizing a TypeScript application

When we compile our Typescript code, the compiler will generate a JavaScript file for each compiled TypeScript file. If we run the application in a web browser, these files won't really be useful on their own because the only way to use them would be to create an individual HTML script tag for each one of them.

Alternatively, we could follow two different approaches:

- We could use a tool, such as the RequireJS library, to load each of those files on demand using AJAX. This approach is known as asynchronous module loading. To follow this approach, we will need to change the configuration of the TypeScript compiler to use the **asynchronous module definition (AMD)** notation.
- We could configure the TypeScript compiler to use the CommonJS module notation and use a tool, such as Browserify, to trace the application's modules and dependencies and generate a highly optimized single file, which will contain all the application's modules.

In this book, we will use the CommonJS method because it is highly integrated with Browserify and Gulp.



If you have never worked with AMD or CommonJS modules before, don't worry too much about it for now. We will focus on modules in *Chapter 4, Object-Oriented Programming with TypeScript*.

We can find the application's root module (named `main.ts` in our example) in the companion code. This file contains the following code:

```
///<reference path=".//references.d.ts" />

import { headerView } from './header_view';
import { footerView } from './footer_view';
import { loadingView } from './loading_view';

headerView.render();
footerView.render();
loadingView.render();
```



The preceding `import` statements are used to access the contents of some external modules. We will learn more about external modules in *Chapter 4, Object-Oriented Programming with TypeScript*.

When compiled (using the CommonJS module notation), the output JavaScript code will look like this:

```
var headerView = require('./header_view');
var footerView = require('./footer_view');
var loadingView = require('./loading_view');
headerView.render();
footerView.render();
loadingView.render();
```

As we can see in the first three lines, the `main.js` file depends on the other three JavaScript files: `header_view.js`, `footer_view.js`, and `loading_view.js`. If we check the companion code, we will see that these files also have some dependencies.

We will normally refer to these dependencies as modules. Importing a module allows us to use the public parts (also known as the exported parts) of a module from another module.

Browserify is able to trace the full tree of dependencies and generate a highly optimized single file, which will contain all the application's modules and dependencies.

We will now add two new tasks to our automated build (`gulpfile.js`). In the first one, we will configure Browserify to trace the dependencies of our application's modules. In the second one, we will configure Browserify to trace the dependencies of our application's unit tests.

We need to install some packages before implementing the new task:

```
npm install browserify vinyl-transform gulp-uglify gulp-sourcemaps
```

We can then import the modules and write some initialization code:

```
var browserify = require('browserify'),
    transform = require('vinyl-transform'),
    uglify = require('gulp-uglify'),
    sourcemaps = require('gulp-sourcemaps');

var browserified = transform(function(filename) {
  var b = browserify({ entries: filename, debug: true });
  return b.bundle();
});
```

In the preceding code snippet, we have loaded the required plugins and declared a function named `browserified`, which is required for compatibility reasons. The `browserified` function will transform a regular Node.js stream into a Gulp (buffered vinyl) stream.

Let's proceed to implement the actual task:

```
gulp.task('bundle-js', function () {
  return gulp.src('./temp/source/js/main.js')
    .pipe(browserified)
    .pipe(sourcemaps.init({ loadMaps: true }))
    .pipe(uglify())
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('./dist/source/js/'));
});
```

The task we just defined will take the file `main.js` as the entry point of our application and trace all the application's modules and dependencies from this point. It will then generate one single stream containing a highly optimized JavaScript.

We will then use the `uglify` plugin to minimize the output size. The reduced file size will reduce the application's loading time, but will make it harder to debug. We will also generate a source map file to facilitate the debugging process.

 Uglify removes all line breaks and whitespaces and reduces the length of some variable names. The source map files allow us to map the reduced file to its original code while debugging.

A source map provides a way of mapping code within a compressed file back to its original position in a source file. This means we can easily debug an application even after its assets have been optimized. The Chrome and Firefox developer tools both ship with built-in support for source maps.

The `bundle-test` task is really similar to the previous task. This time, we will avoid using uglify and source maps because usually we won't need to optimize the download times of our unit tests. As you can see, we don't have a single entry point because we will allow the existence of multiple entry points (each entry point will be linked to a collection of automated tests known as test suite. Don't worry if you are not familiar with this term, as we will learn more about it in *Chapter 7, Application Testing*):

```
gulp.task('bundle-test', function () {
  return gulp.src('./temp/test/**/**.test.js')
    .pipe(browserified)
    .pipe(gulp.dest('./dist/test/'));
});
```

Finally, we have to update the default task to also perform the new tasks:

```
gulp.task('default', ['lint', 'tsc', 'tsc-tests', 'bundle-js',
  'bundle-test']);
```



We have created a task to compile the TypeScript files into JavaScript files. The JavaScript files are stored in a temporary folder and a second task bundles all the JavaScript files into a single file. In a real corporate environment, it is not recommended to store files temporarily when working with Gulp. We can perform all these operations with one single task by passing the output stream of an operation as the input of the following operation. However, in this book, we will try to split the tasks to facilitate the understanding of each task.

If we try to execute the default task after adding these changes, we will probably experience some issues because the tasks are executed in parallel by default. We will now learn how to control the task's execution order to avoid this kind of issue.

Managing the Gulp tasks' execution order

Sometimes we will need to run our tasks in a certain order (for example, we need to compile our TypeScript into JavaScript before we can execute our unit tests). Controlling the tasks' execution order can be challenging since in Gulp all the tasks are asynchronous by default.

There are three ways to make a task synchronous:

- Passing in a callback
- Returning a stream
- Returning a promise



Refer to *Chapter 3, Working with Functions* to learn more about the usage callbacks and promises.

Let's take a look at the first two ways (we will not cover the usage of promises in this chapter):

```
// Passing a callback (cb)
gulp.task('sync', function (cb) { // note the cb argument
    // setTimeout could be any async task
    setTimeout(function () {
        cb(); // note the cb usage here
    }, 1000);
});

// Returning a stream
gulp.task('sync', function () {
```

```
        return gulp.src('js/*.js') // note the return keyword here
            .pipe(concat('script.min.js'))
            .pipe(uglify())
            .pipe(gulp.dest('../dist/js'));
    });
}
```

Now that we have a synchronous task, we can combine it with the task dependency notation to manage the execution order:

```
gulp.task('secondTask', ['sync'], function () {
    // this task will not start until
    // the sync task is all done!
});
```

In the preceding code snippet, the `secondTask` task will not start until the `sync` task is done. Now, let's imagine that there is a third task named `thirdTask`. We will write the following code snippet hoping that it will execute the `sync` task before the `thirdTask` task and finally the `default` task, but it will in fact run the `sync` task and `thirdTask` task in parallel:

```
gulp.task('default', ['sync', 'thirdTask'], function () {
    // do stuff
});
```

Fortunately, we can install the `run-sequence` Gulp plugin via npm, which will allow us to have better control over the task execution order:

```
var runSequence = require('run-sequence');
gulp.task('default', function(cb) {
    runSequence(
        'lint',                                // lint
        ['tsc', 'tsc-tests'],                  // compile
        ['bundle-js', 'bundle-test'],          // optimize
        'karma',                                // test
        'browser-sync',                         // serve
        cb                                      // callback
    );
});
```

The preceding code snippet will run in the following order:

1. `lint`.
2. `tsc` and `tsc-tests` in parallel.
3. `bundle-js` and `bundle-test` in parallel.
4. `karma`.
5. `browser-sync`.



The Gulp development team announced plans to improve the management of the task execution order without the need for external plugins when this book was about to be published. Refer to the Gulp documentation and release notes on future releases to learn more about it. The documentation can be found at <https://github.com/gulpjs/gulp/blob/master/docs/README.md>.

Test runners

A test runner is a tool that allows us to automate the execution of our application's unit tests.



Unit testing refers to the practice of testing certain functions and areas (units) of our code. This gives us the ability to verify that our functions work as expected. It is assumed that the reader has some understanding of the unit test process, but the topics explored here will be covered in a much higher level of detail in *Chapter 7, Application Testing*.

We can use a test runner to automatically execute our application's test suites in multiple browsers instead of having to manually open each web browser in order to execute the tests.

We will use a test runner known as Karma. Karma is compatible with multiple unit testing frameworks, but we will use the Mocha testing framework together with two libraries: Chai (an assertion library) and Sinon (a mocking framework).



You don't need to worry too much about these libraries right now because we will focus on their usage in *Chapter 7, Application Testing*.

Let's start by using npm to install the testing framework that we are going to use:

```
npm install mocha chai sinon --save-dev
```

We will continue by installing the karma test runner and some dependencies:

```
npm install karma karma-mocha karma-chai karma-sinon karma-coverage  
karma-phantomjs-launcher gulp-karma --save-dev
```

After installing all the necessary packages, we have to add a new Gulp task to the `gulpfile.js` file. The new task will run the application's unit tests using Karma:

```
var karma = require("gulp-karma");

gulp.task('karma', function(cb) {
  gulp.src('./dist/test/**/*.*.test.js')
    .pipe(karma({
      configFile: 'karma.conf.js',
      action: 'run'
    }))
    .on('end', cb)
    .on('error', function(err) {
      // Make sure failed tests cause gulp to exit non-zero
      throw err;
    });
});
```

In the preceding code snippet, we are fetching all the files with the extension `.test.js` under the directory located at `./dist/test/` and all its subdirectories. We will then pass the files to the Karma plugin together with the location of the `karma.conf.js` file, which contains the Karma configuration. We will create a new JavaScript file named `karma.conf.js` in the project's root directory and copy the following code into it:

```
module.exports = function (config) {
  'use strict';
  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
    plugins : [
      'karma-coverage',
      'karma-mocha',
      'karma-chai',
      'karma-sinon',
      'karma-phantomjs-launcher'
    ],
    preprocessors: {
      './dist/test/*.*.test.js' : ['coverage']
    },
    port: 9876,
    colors: true,
    autoWatch: false,
    singleRun: false,
    logLevel: config.LOG_INFO
  });
};
```

The configuration file tells Karma about the application's base path, frameworks (Mocha, Chai, and Sinon.JS), browsers (PhantomJS), plugins, and reporters that we want to use during the tests' execution. PhantomJS is a headless web browser, it is useful because it can execute the unit test without actually having to open a web browser.



We should run the tests in real web browsers along with PhantomJS before doing a production deployment. There are Karma plugins, such as `karma-firefox-launcher` and `karma-chrome-launcher`, which will allow us to run the unit tests in the browsers of our choice.

Karma uses the progress reporter by default to let us know the status of the test execution process. We added the coverage reporter as well because we want to have an idea of what percentage of our application's code has been tested with unit tests. After adding the coverage reporter and running our unit tests we will be able to find the coverage report under a folder named `coverage`, which should be located in the same directory where the `karma.conf.js` file was located.

If we look at the Karma configuration documentation at <http://karma-runner.github.io/0.8/config/configuration-file.html>, we will notice that we are missing the `files` field in our `karma.conf.js` file. We didn't indicate the location of our unit tests because the Gulp task will pass the stream, which contains the unit tests' files to Karma, and then the Karma task is executed.

Synchronized cross-device testing

We will add one last task to the `gulpfile.js` file, which will allow us to run our application in a web browser. We need to install the `browser-sync` package by using `npm`:

```
npm install -g browser-sync
```

We will then create two new tasks. These tasks are just used to group a few tasks into one main task. We are doing this because sometimes we want to refresh a webpage to see the effect of changing some TypeScript code and we need to run a number of tasks (compilation, bundling, and so on) before we can actually see the changes in a web browser. By grouping all these tasks into higher-level tasks, we can save some time and make our configuration files more readable:

```
gulp.task('bundle', function(cb) {
  runSequence('build', [
```

```
'bundle-js', 'bundle-test'  
], cb);  
});  
  
gulp.task('test', function(cb) {  
  runSequence('bundle', ['karma'], cb);  
});
```

The preceding two tasks are used to group all the build-related tasks into a higher-level task (named `bundle`) and to group all the test-related tasks into a higher-level task (named `test`).

After installing the package and implementing the preceding two tasks, we can add a new Gulp task to the `gulpfile.js` file:

```
var browserSync = require('browser-sync');  
gulp.task('browser-sync', ['test'], function() {  
  browserSync({  
    server: {  
      baseDir: "./dist"  
    }  
  });  
  
  return gulp.watch([  
    "./dist/source/js/**/*.*js",  
    "./dist/source/css/**.css",  
    "./dist/test/**/**.test.js",  
    "./dist/data/**/**",  
    "./index.html"  
  ], [browserSync.reload]);  
});
```

In this task, we are configuring `BrowserSync` to host in the local web server all the static files under the `dist` directory. We then use the `gulp watch` function to indicate that, if the content of any of the files under the `dist` directory changes, `BrowserSync` should automatically refresh our web browser.

When some changes are detected, the `test` task is invoked. Because the `test` task invokes the `bundle` tasks, any changes will trigger the entire process (`build` and `test`) before refreshing the webpage and displaying the new files in a web browser.

`BrowserSync` is a really powerful tool, it allows us to test in one device and automatically repeat our actions (clicks, scrolls, and so on) on as many devices as we want. It will also allow us to debug our applications remotely, which can be really useful when we are testing an application on mobile devices.

Synchronizing devices is really simple. If we run the browser-sync task, the application will be launched in the default web browser. If we look at the console output, we will see that the application is running in one URL (`http://localhost:3000`) and the BrowserSync tools are available in a second URL (`http://localhost:3001`):

[BS] Access URLs:

Local: `http://localhost:3000`

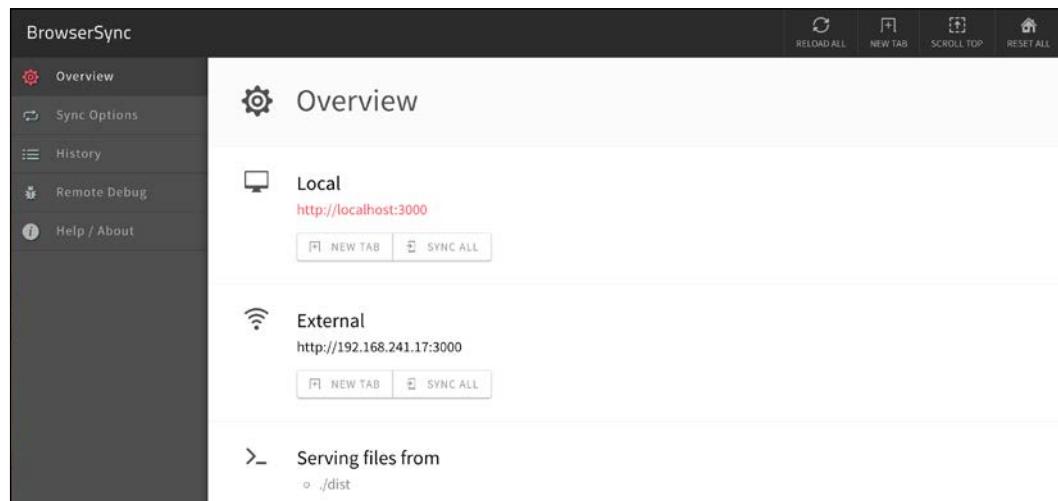
External: `http://192.168.241.17:3000`

UI: `http://localhost:3001`

UI External: `http://192.168.241.17:3001`

[BS] Serving files from: `./dist`

If we open another tab in our browser pointing to the BrowserSync tools URL (`http://localhost:3001`, in the example), we will access the BrowserSync tools user interface:



We can use the BrowserSync tools user interface to access the remote debugging options and device synchronization options. To synchronize a new device, we just need to use a phone or tablet connected to the same local area network and open the indicated external URL in the device's web browser.

If you wish to learn more about BrowserSync, visit the official project documentation at <http://www.browsersync.io/docs/>.

Continuous Integration tools

Continuous Integration (CI) is a development practice that helps to prevent potential integration issues. Software integration issues refers to the difficulties that may arise during the practice of combining individually tested software components into an integrated whole. Software is integrated when components are combined into subsystems or when subsystems are combined into products.

Components may be integrated after all of them are implemented and tested, as in a waterfall model or a big bang approach. On the other hand, CI requires developers to commit their code daily into a remote code repository. Each commit is then verified by an automated build, allowing teams to detect integration issues earlier.

In this chapter, we have created a remote code repository and an automated build, but we haven't configured a tool to observe our commits and run the automate build accordingly. We need a CI server. There are many options when it comes to choosing a CI server, but exploring these options is out of the scope of this book. We will work with Travis CI because it is highly integrated with GitHub and is free for open source projects and learning purposes.

To configure Travis CI, we need to visit the website <https://travis-ci.org> and log in using our GitHub credentials. Once we have logged in, we will be able to see a list of our public GitHub repositories and will also be able to enable the CI.

username/repository-name



To finish the configuration, we need to add a file named `travis.yml` to our application's root directory, which contains the Travis CI configuration:

```
language: node_js  
node_js:  
  - "0.10""
```



There are many other available TravisCI configuration options. Refer to <http://docs.travis-ci.com/> to learn more about the available options.

After completing these two small configuration steps, Travis CI will be ready to observe the commits to our remote code repository.



If the build works in the local development environment, but fails in the CI server, we will have to check the build error log and try to figure out what went wrong. Chances are that the software versions in our environment will be ahead of the ones in the CI server and we will need to indicate to Travis CI that a dependency needs to be installed or updated. We can find the Travis CI documentation at <http://docs.travis-ci.com/user/build-configuration/> to learn how to resolve this kind of issue.

Scaffolding tools

A scaffolding tool is used to autogenerate the project structure, build scripts, and much more. The most popular scaffolding tool these days is Yeoman. Yeoman uses an internal command known as `yo`, a package manager, and a task runner of our choice to generate projects based on templates.

The project templates are known as generators and the open source community has already published many of them, so we should be able to find one that more or less suits our needs. Alternatively, we can write and publish our own Yeoman generator.

We will now create a new project to showcase how Yeoman can help us to save some time. Yeoman will generate the `package.json` and `bower.json` files and automatically install some dependencies for us.

The `yo` command can be installed using `npm`:

```
npm install -g yo
```

After installing the `yo` command, we will need to install at least one generator. We need to find a generator for the kind of project that we wish to create.

We are going to create a new project using Gulp as the task runner and TypeScript to showcase the usage of Yeoman. We can use a generator called `generator-typescript`. The list of available generators can be found online at <http://yeoman.io/generators/>.

We can install a generator by using `npm`:

```
npm install -g generator-typescript
```

After installing the generator, we can use it with the help of the `yo` command:

```
yo typescript
```

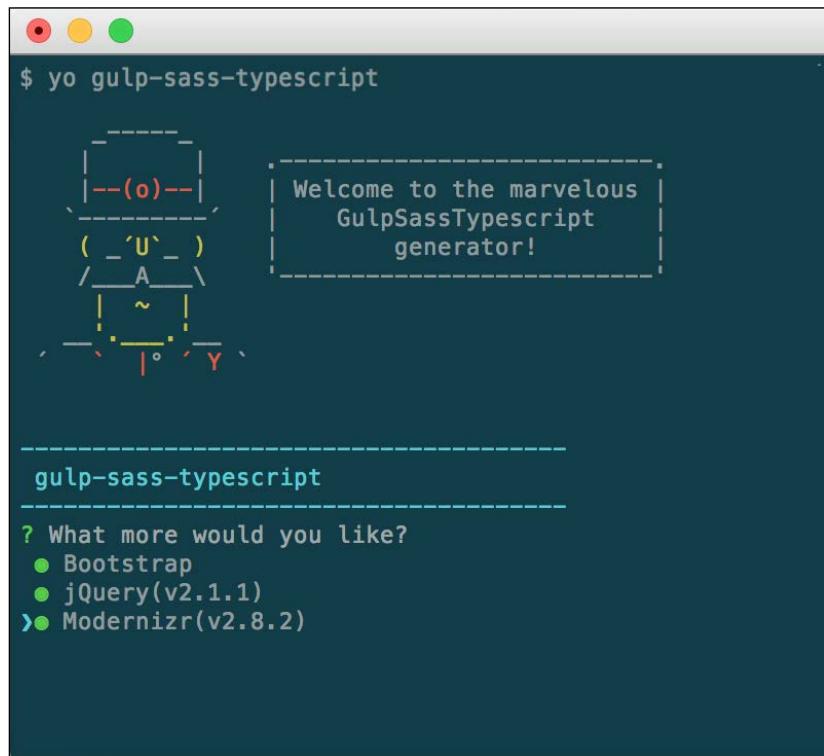
If, for example, we also wanted to use Sass, we could use the generator-gulp-sass-typescript generator instead:

```
npm install -g generator-gulp-sass-typescript
```

Some of the generators are interactive and will allow us to select whether we want to add some optional third-party libraries to the project or not. Let's run the generator to see what it looks like:

```
yo generator-gulp-sass-typescript
```

The screen that is displayed contains a series of steps to guide us through the process of creating a new project, which includes Gulp as the task runner, Sass as the CSS preprocessor, and TypeScript as the programming language:



After executing the generator, the project template will generate a directory tree similar to the following one:

```
└── app
    ├── index.html
    ├── sass
    │   └── styles.scss
    ├── scripts
    │   └── main.js
    ├── styles
    │   └── styles.css
    └── ts
        └── main.ts
└── bower.json
└── bower_components
    └── ...
└── gulpfile.js
└── node_modules
    └── ...
└── package.json
```

The `bower.json`, `package.json`, and `gulpfile.js` files (the Gulp task runner configuration) are autogenerated and will save us a considerable amount of time.



It is never a good idea to let a tool generate some code for us if we don't really understand what that code does. While in the future you should definitely consider using Yeoman to generate a new project, it is recommended to gain a good understanding of task and test runners before using a scaffolding tool.

Summary

In this chapter, you learned how to work with a source control repository and how to use Gulp to manage the tasks in an automated build. The automated build helps us to validate the quality of the TypeScript code, compile it, test it, and optimize it. You also learned how to install third-party packages and TypeScript type definitions for those third-party components.

Towards the end of the chapter, you learned how to use the automated build and a continuous integration server to reduce the impact of potential integration issues.

In the next chapter, you will learn about functions.

3

Working with Functions

In *Chapter 1, Introducing TypeScript*, we took a first look at the usage of functions. Functions are the fundamental building block of any application in TypeScript, and they are powerful enough to deserve the dedication of an entire chapter to explore their potential.

In this chapter, we will learn to work with functions in depth. The chapter is divided into two main sections. In the first section, we will start with a quick recap of some basic concepts and then move onto some less commonly known function features and use cases. The first section includes the following concepts:

- Function declaration and function expressions
- Function types
- Functions with optional parameters
- Functions with default parameters
- Functions with rest parameters
- Function overloading
- Specialized overloading signature
- Function scope
- Immediately invoked functions
- Generics
- Tag functions and tagged templates

The second section focuses on TypeScript asynchronous programming capabilities and includes the following concepts:

- Callbacks and higher order functions
- Arrow functions
- Callback hell
- Promises
- Generators
- Asynchronous functions (async and await)

Working with functions in TypeScript

In this section, we will focus on the declaration and usage of functions, parameters, and arguments. We will also introduce one of the most powerful features of TypeScript: Generics.

Function declarations and function expressions

In the first chapter, we introduced the possibility of declaring functions with (named function) or without (unnamed or anonymous function) explicitly indicating its name, but we didn't mention that we were also using two different types of function.

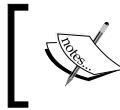
In the following example, the named function `greetNamed` is a function declaration while `greetUnnamed` is a function expression. Ignore the first two lines, which contain two console log statements, for now:

```
console.log(greetNamed("John"));
console.log(greetUnnamed("John"));

function greetNamed(name : string) : string {
    if(name) {
        return "Hi! " + name;
    }
}

var greetUnnamed = function(name : string) : string {
    if(name) {
        return "Hi! " + name;
    }
}
```

We might think that these preceding functions are really similar, but they will behave differently. The interpreter can evaluate a function declaration as it is being parsed. On the other hand, the function expression is part of an assignment and will not be evaluated until the assignment has been completed.



The main cause of the different behavior of these functions is a process known as variable hoisting. We will learn more about the variable hoisting process later in this chapter.

If we compile the preceding TypeScript code snippet into JavaScript and try to execute it in a web browser, we will observe that the first alert statement will work because JavaScript knows about the declaration function and can parse it before the program is executed.

However, the second alert statement will throw an exception, which indicates that `greetUnnamed` is not a function. The exception is thrown because the `greetUnnamed` assignment must be completed before the function can be evaluated.

Function types

We already know that it is possible to explicitly declare the type of an element in our application by using the optional type declaration annotation:

```
function greetNamed(name : string) : string {  
    if(name) {  
        return "Hi! " + name;  
    }  
}
```

In the preceding function, we have specified the type of the parameter `name` (`string`) and its return type (`string`). Sometimes, we will need to not just specify the types of the function elements, but also the function itself. Let's take a look at an example:

```
var greetUnnamed : (name : string) => string;  
  
greetUnnamed = function (name : string) : string {  
    if(name){  
        return "Hi! " + name;  
    }  
}
```

In the preceding example, we have declared the variable `greetUnnamed` and its type. The type of `greetUnnamed` is a function type that takes a string variable called `name` as its only parameter and returns a string after being invoked. After declaring the variable, a function, whose type must be equal to the variable type, is assigned to it.

We can also declare the `greetUnnamed` type and assign a function to it in the same line rather than declaring it in two separate lines like we did in the previous example:

```
var greetUnnamed : (name : string) => string = function(name : string)
: string {
  if(name) {
    return "Hi! " + name;
  }
}
```

Just like in the previous example, the preceding code snippet also declares a variable `greetUnnamed` and its type. We will assign a function to this variable in the same line in which it is declared. The assigned function must be equal to the variable type.



In the preceding example, we have declared the type of the `greetUnnamed` variable and then assigned a function as its value. The type of the function can be inferred from the assigned function, and for this reason, it is unnecessary to add a redundant type annotation. We have done this to facilitate the understanding of this section, but it is important to mention that adding redundant type annotations can make our code harder to read, and it is considered bad practice.

Functions with optional parameters

Unlike JavaScript, the TypeScript compiler will throw an error if we attempt to invoke a function without providing the exact number and type of parameters that its signature declares. Let's take a look at a code sample to demonstrate it:

```
function add(foo : number, bar : number, foobar : number) : number {
  return foo + bar + foobar;
}
```

The preceding function is called add and will take three numbers as parameters: named foo, bar, and foobar. If we attempt to invoke this function without providing exactly three numbers, we will get a compilation error indicating that the supplied parameters do not match the function's signature:

```
add();           // Supplied parameters do not match any signature
add(2, 2);      // Supplied parameters do not match any signature
add(2, 2, 2);   // returns 6
```

There are scenarios in which we might want to be able to call the function without providing all its arguments. TypeScript features optional parameters in functions to help us to increase the flexibility of our functions. We can indicate to TypeScript that we want a function's parameter to be optional by appending the character ? to its name. Let's update the previous function to transform the required parameter foobar into an optional parameter:

```
function add(foo : number, bar : number, foobar? : number) : number {
    var result = foo + bar;
    if(foobar !== undefined){
        result += foobar;
    }
    return result;
}
```

Note how we have changed the foobar parameter name into foobar?, and how we are checking the type of foobar inside the function to identify if the parameter was supplied as an argument to the function or not. After doing these changes, the TypeScript compiler will allow us to invoke the function without errors when we supply two or three arguments to it:

```
add();           // Supplied parameters do not match any signature
add(2, 2);      // returns 4
add(2, 2, 2);  // returns 6
```

It is important to note that the optional parameters must always be located after the required parameters in the function's parameters list.

Functions with default parameters

When a function has some optional parameters, we must check if an argument has been passed to the function (just like we did in the previous example).

There are some scenarios in which it would be more useful to provide a default value for a parameter when it is not supplied than to make it an optional parameter. Let's rewrite the add function (from the previous section) using the inline `if` structure:

```
function add(foo : number, bar : number, foobar? : number) :  
number {  
    return foo + bar + (foobar !== undefined ? foobar : 0);  
}
```

There is nothing wrong with the preceding function, but we can improve its readability by providing a default value for the `foobar` parameter instead of flagging it as an optional parameter:

```
function add(foo : number, bar : number, foobar : number = 0) :  
number {  
    return foo + bar + foobar;  
}
```

To indicate that a function parameter is optional, we just need to provide a default value using the `=` operator when declaring the function's signature. The TypeScript compiler will generate an `if` structure in the JavaScript output to set a default value for the `foobar` parameter if it is not passed as an argument to the function:

```
function add(foo, bar, foobar) {  
    if (foobar === void 0) { foobar = 0; }  
    return foo + bar + foobar;  
}
```

`Void 0` is used by the TypeScript compiler to check if a variable is equal to `undefined`. While most developers use the `undefined` variable, most compilers use `void 0`.

Just like optional parameters, default parameters must be always located after any required parameters in the function's parameter list.

Functions with rest parameters

We have seen how to use optional and default parameters to increase the number of ways that we can invoke a function. Let's return one more time to the previous example:

```
function add(foo : number, bar : number, foobar : number = 0) : number {
    return foo + bar + foobar;
}
```

We have seen how to make possible the usage of the `add` function with two or three parameters, but what if we wanted to allow other developers to pass four or five parameters to our function? We would have to add two extra default or optional parameters. And what if we wanted to allow them to pass as many parameters as they may need? The solution to this possible scenario is the use of rest parameters. The rest parameter syntax allows us to represent an indefinite number of arguments as an array:

```
function add(...foo : number[]) : number {
    var result = 0;
    for(var i = 0; i < foo.length; i++) {
        result += foo[i];
    }
    return result;
}
```

As we can see in the following code snippet, we have replaced the function parameters `foo`, `bar`, and `foobar` with just one parameter: `foo`. Note that the name of the parameter `foo` is preceded by an ellipsis (a set of three periods – not the actual ellipsis character). A rest parameter must be of an array type or we will get a compilation error. We can now invoke the `add` function with as many parameters as we may need:

```
add();           // returns 0
add(2);         // returns 2
add(2, 2);      // returns 4
add(2, 2, 2);   // returns 6
add(2, 2, 2, 2); // returns 8
add(2, 2, 2, 2, 2); // returns 10
add(2, 2, 2, 2, 2, 2); // returns 12
```

Although there is no specific limit to the theoretical maximum number of arguments that a function can take, there are, of course, practical limits. These limits are entirely implementation-dependent and, most likely, will also depend exactly on how we are calling the function.

JavaScript functions have a built-in object called the `arguments` object. This object is available as a local variable named `arguments`. The `arguments` variable contains an object similar to an array, which contains the arguments used when the function was invoked.



The `arguments` object exposes some of the methods and properties provided by a standard array, but not all of them. Refer to the complete reference at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments> to learn more about its peculiarities.

If we examine the JavaScript output, we will notice that TypeScript iterates the `arguments` object in order to add the values to the `foo` variable:

```
function add() {  
    var foo = [];  
    for (var _i = 0; _i < arguments.length; _i++) {  
        foo[_i - 0] = arguments[_i];  
    }  
    var result = 0;  
    for (var i = 0; i < foo.length; i++) {  
        result += foo[i];  
    }  
    return result;  
}
```

We can argue that this is an extra, unnecessary iteration over the function's parameters. Even though is hard to imagine this extra iteration becoming a performance issue, if you think that this could be a problem for the performance of your application, you may want to consider avoiding using rest parameters and use an array as the only parameter of the function instead:

```
function add(foo : number[]) : number {  
    var result = 0;  
    for(var i = 0; i < foo.length; i++) {  
        result += foo[i];  
    }  
    return result;  
}
```

The preceding function takes an array of numbers as its only parameter. The invocation API will be a little different from the rest parameters, but we will effectively avoid the extra iteration over the function's argument list:

```
add();      // Supplied parameters do not match any signature
add(2);     // Supplied parameters do not match any signature
add(2,2);   // Supplied parameters do not match any signature
add(2,2,2); // Supplied parameters do not match any signature

add([]);    // returns 0
add([2]);   // returns 2
add([2,2]); // returns 4
add([2,2,2]); // returns 6
```

Function overloading

Function overloading or method overloading is the ability to create multiple methods with the same name and a different number of parameters or types. In TypeScript, we can overload a function by specifying all function signatures of a function, followed by a signature known as the implementation signature. Let's take a look at an example:

```
function test(name: string) : string;      // overloaded signature
function test(age: number) : string;        // overloaded signature
function test(single: boolean) : string; // overloaded signature
function test(value: (string | number | boolean)) : string; { // implementation signature
  switch(typeof value) {
    case "string":
      return `My name is ${value}.`;
    case "number":
      return `I'm ${value} years old.`;
    case "boolean":
      return value ? "I'm single." : "I'm not single.";
    default:
      console.log("Invalid Operation!");
  }
}
```



You might not be familiar with the syntax used in some of the strings in the preceding code snippet. This syntax is known as **Template Strings**. Template strings are enclosed by the back-tick (` `) character instead of double or single quotes. Template strings can contain placeholders. These are indicated by the dollar sign and curly braces (`$(expression)`). The expressions in the placeholders and the text between them get passed to a function. The default function just concatenates the parts into a single string.

As we can see in the preceding example, we have overloaded the function test three times by adding a signature that takes a string as its only parameter, another function that takes a number, and a final signature that takes a Boolean as its unique parameter. It is important to note that all function signatures must be compatible; so if, for example, one of the signatures tries to return a number while another tries to return a string, we will get a compilation error.

The implementation signature must be compatible with all the overloaded signatures, always be the last in the list, and take any or a union type as the type of its parameters.

Invoking the implementation signature directly will cause a compilation error:

```
test("Remo");           // returns "My name is Remo."  
test(26);              // returns "I'm 26 years old.";  
test(false);            // returns "I'm not single.";  
test({ custom : "custom" }); // error
```

Specialized overloading signatures

We can use a specialized signature to create multiple methods with the same name and number of parameters but a different return type. To create a specialized signature, we must indicate the type of function parameter using a string. The string literal is used to identify which of the function overloads is invoked:

```
interface Document {  
  createElement(tagName: "div"): HTMLDivElement; // specialized  
  createElement(tagName: "span"): HTMLSpanElement; // specialized  
  createElement(tagName: "canvas"): HTMLCanvasElement; //  
  specialized  
  createElement(tagName: string): HTMLElement; // non-specialized  
}
```

In the preceding example, we have declared three specialized overloaded signatures and one non-specialized signature for the function named `createElement`.

When we declare a specialized signature in an object, it must be assignable to at least one non-specialized signature in the same object. This can be observed in the preceding example, as the `createElement` property belongs to a type that contains three specialized signatures, all of which are assignable to the non-specialized signature in the type.

When writing overloaded declarations, we must list the non-specialized signature last.



Remember that, as seen in *Chapter 1, Introducing TypeScript*, we can also use union types to create a method with the same name and number of parameters but a different type.

Function scope

Low-level languages such as C have low-level memory management features. In programming languages with a higher level of abstraction such as TypeScript, values are allocated when variables are created and automatically cleared from memory when they are not used anymore. The process that cleans the memory is known as **garbage collection** and is performed by the JavaScript runtime garbage collector.

The garbage collector generally does a great job, but it is a mistake to assume that it will always prevent us from facing a memory leak. The garbage collector will clear a variable from the memory whenever the variable is out of the scope. Is important to understand how the TypeScript scope works so we understand the lifecycle of the variables.

Some programming languages use the structure of the program source code to determine what variables we are referring to (lexical scoping), while others use the runtime state of the program stack to determine what variable we are referring to (dynamic scoping). The majority of modern programming languages use lexical scoping (including TypeScript). Lexical scoping tends to be dramatically easier to understand for both humans and analysis tools than dynamic scoping.

While in most lexical scoped programming languages, variables are scoped to a block (a section of code delimited by curly braces `{ }`), in TypeScript (and JavaScript), variables are scoped to a function:

```
function foo() : void {  
    if(true){  
        var bar : number = 0;  
    }  
}
```

```
    alert(bar);  
}  
  
foo(); // shows 0
```

The preceding function named `foo` contains an `if` structure. We have declared a numeric variable named `bar` inside the `if` structure, and later we have attempted to show the value of the variable `bar` using the `alert` function.

We might think that the preceding code sample would throw an error in the fifth line because the `bar` variable should be out of the scope when the `alert` function is invoked. However, if we invoke the `foo` function, the `alert` function will be able to display the variable `bar` without errors because all the variables inside a function will be in the scope of the entire function body, even if they are inside another block of code (except a function block).

This might seem really confusing, but it is easy to understand once we know that, at runtime, all the variable declarations are moved to the top of a function before the function is executed. This behavior is called hoisting.



TypeScript is compiled to JavaScript and then executed – this means that a TypeScript application is a JavaScript application at runtime, and for this reason, when we refer to the TypeScript runtime, we are talking about the JavaScript runtime. We will learn in depth about the runtime in *Chapter 5, Runtime*.

So, before the preceding code snippet is executed, the runtime will move the declaration of the variable `bar` to the top of our function:

```
function foo() : void {  
    var bar :number;  
    if(true){  
        bar= 0;  
    }  
    alert(bar);  
}
```

This means that we can use a variable before it is declared. Let's take a look at an example:

```
function foo2() : void {  
    bar = 0;  
    var bar : number;  
    alert(bar);  
}  
  
foo2();
```

In the preceding code snippet, we have declared a function `foo2`, and in its body, we have assigned the value `0` to a variable named `bar`. At this point, the variable has not been declared. In the second line, we are actually declaring the variable `bar` and its type. In the last line, we are displaying the value of `bar` using the `alert` function.

Because declaring a variable anywhere inside a function (except another function) is equivalent to declaring it at the top of the function, the `foo2` function is transformed into the following at runtime:

```
function foo2() : void {  
    var bar : number;  
    bar = 0;  
    alert(bar);  
}  
  
foo2();
```

Because developers with a Java or C# background are not used to the function scope, it is one of the most criticized characteristics of JavaScript. The people in charge of the development of the ECMAScript 6 specification are aware of this and, as a result, they have introduced the keywords `let` and `const`.

The `let` keyword allows us to set the scope of a variable to a block (`if`, `while`, `for...`) rather than a function block. We can update the first example in this section to showcase how `let` works:

```
function foo() : void {  
    if(true){  
        let bar : number = 0;  
        bar = 1;  
    }  
    alert(bar); // error  
}
```

The `bar` variable is now declared using the `let` keyword and, as a result, it is only accessible inside the `if` block. The variable is not hoisted to the top of the `foo` function and cannot be accessed by the `alert` function outside the `if` statement.

While variables defined with `const` follow the same scope rules as variables declared with `let`, they can't be reassigned:

```
function foo() : void {  
    if(true){  
        const bar : number = 0;  
        bar = 1; // error
```

```
    }
    alert(bar); // error
}
```

If we attempt to compile the preceding code snippet, we will get an error because the `bar` variable is not accessible outside the `if` statement (just like when we used the `let` keyword), and a new error occurs when we try to assign a new value to the `bar` variable. The second error is caused because it is not possible to assign a value to a constant variable once the variable has already been initialized.

Immediately invoked functions

An **immediately invoked function expression** (IIFE) is a design pattern that produces a lexical scope using function scoping. IIFE can be used to avoid variable hoisting from within blocks or to prevent us from polluting the global scope. For example:

```
var bar = 0; // global

(function() {
  var foo : number = 0; // in scope of this function
  bar = 1; // in global scope
  console.log(bar); // 1
  console.log(foo); // 0
})();

console.log(bar); // 1
console.log(foo); // error
```

In the preceding example, we have wrapped the declaration of two variables (`foo` and `bar`) with an IIFE. The `foo` variable is scoped to the IIFE function and is not available in the global scope, which explains the error when trying to access it in the last line.

We can also pass a variable to the IIFE to have better control over the creation of variables outside its own scope:

```
var bar = 0; // global

(function(global) {
  var foo : number = 0; // in scope of this function
  bar = 1; // in global scope
  console.log(global.bar); // 1
  console.log(foo); // 0
}) (this);
```

```
console.log(bar); // 1
console.log(foo); // error
```

This time, the IIFE takes the `this` operator as its only argument, which points to the global scope, because we are not invoking the `this` operator from within a function. Inside the IIFE, the `this` operator is passed as a parameter named `global`. We can then achieve much better control over the objects we want to declare in the global scope (`bar`) and those we don't (`foo`).

Furthermore, IIFE can help us to simultaneously allow public access to methods while retaining privacy for variables defined within the function. Let's take a look at an example:

```
class Counter {
    private _i : number;
    constructor() {
        this._i = 0;
    }
    get() : number {
        return this._i;
    }
    set(val : number) : void {
        this._i = val;
    }
    increment() : void {
        this._i++;
    }
}
var counter = new Counter();
console.log(counter.get()); // 0
counter.set(2);
console.log(counter.get()); // 2
counter.increment();
console.log(counter.get()); // 3
console.log(counter._i); // Error: Property '_i' is private
```



By convention, TypeScript and JavaScript developers usually name private variables with names preceded by an underscore (`_`).

We have defined a class named `Counter` that has a private numeric attribute named `_i`. The class also has methods to get and set the value of the private property `_i`. We have also created an instance of the `Counter` class and invoked the methods `set`, `get`, and `increment` to observe that everything is working as expected. If we attempt to access the `_i` property in an instance of `Counter`, we will get an error because the variable is private.

If we compile the preceding TypeScript code (only the class definition) and examine the generated JavaScript code, we will see the following:

```
var Counter = (function () {
    function Counter() {
        this._i = 0;
    }
    Counter.prototype.get = function () {
        return this._i;
    };
    Counter.prototype.set = function (val) {
        this._i = val;
    };
    Counter.prototype.increment = function () {
        this._i++;
    };
    return Counter;
})();
```

This generated JavaScript code will work perfectly in most scenarios, but if we execute it in a browser and try to create an instance of Counter and access its property `_i`, we will not get any errors because TypeScript will not generate runtime private properties for us. Sometimes we will need to write our functions in such a way that some properties are private at runtime, for example, if we release a library that will be used by JavaScript developers. We can use IIFE to simultaneously allow public access to methods while retaining privacy for variables defined within the function:

```
var Counter = (function () {
    var _i : number = 0;
    function Counter() {
    }
    Counter.prototype.get = function () {
        return _i;
    };
    Counter.prototype.set = function (val : number) {
        _i = val;
    };
    Counter.prototype.increment = function () {
        _i++;
    };
    return Counter;
})();
```

In the preceding example, everything is almost identical to TypeScript's generated JavaScript, except that the variable `_i` before was an attribute of the `Counter` class, and now it is an object in the `Counter` closure.



Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure *remembers* the environment (variables in the scope) in which it was created. We will discover more about closures in *Chapter 5, Runtime*.

If we run the generated output in a browser and try to invoke the `_i` property directly, we will notice that the property is now private at runtime:

```
var counter = new Counter();
console.log(counter.get()); // 0
counter.set(2);
console.log(counter.get()); // 2
counter.increment();
console.log(counter.get()); // 3
console.log(counter._i); // undefined
```



In some cases, we will need to have really precise control over scope and closures, and our code will end up looking much more like JavaScript. Just remember that, as long as we write our application components (classes, modules, and so on) to be consumed by other TypeScript components, we will rarely have to worry about implementing runtime private properties. We will look in depth at the TypeScript runtime in *Chapter 5, Runtime*.

Generics

Andy Hunt and Dave Thomas formulated the **don't repeat yourself (DRY)** principle in the book *The Pragmatic Programmer*. The DRY principle aims to reduce the repetition of information of all kinds. We will now take a look at an example that will help us to understand what generics functions are and how they can help us follow the DRY principle.

We will start by declaring a really simple `User` class:

```
class User {
  name : string;
  age : number;
}
```

Now that we have our `User` class in place, let's write a function named `getUsers` that will request a list of users via AJAX:

```
function getUsers(cb : (users : User[]) => void) : void {
  $.ajax({
    url: "/api/users",
    method: "GET",
    success: function(data) {
      cb(data.items);
    },
    error : function(error) {
      cb(null);
    }
  });
}
```

We will use jQuery in this example. Remember to create a package.json file and install the jQuery package using npm. You will also need to install the jQuery type definitions file using tsd. Refer to *Chapter 1, Introducing Typescript* and *Chapter 2, Automating Your Development Workflow* if you need additional help.



The `getUsers` function takes a function as a parameter that will be invoked if the AJAX request has been successful. It can be invoked as follows:

```
getUsers(function(users : User[]){
  for(var i, users.length; i++) {
    console.log(users[i].name);
  }
});
```

Now let's imagine that we need an almost identical operation. But this time, we will use an `Order` entity instead:

```
class Order {
  id : number;
  total : number;
  items : any[]
}
```

The `getOrders` function is almost identical to the `getUsers` function. It uses a different URL and it will pass an array of `Orders` instead of a `User` array:

```
function getOrders(cb : (orders : Order[]) => void) : void {
  $.ajax({
```

```

url: "/api/orders",
method: "GET",
success: function(data) {
    cb(data.items);
},
error : function(error) {
    cb(null);
}
});
}

getOrders(function(orders : Orders[]) {
    for(var i; orders.length; i++){
        console.log(orders[i].total);
    }
});

```

We can use generics to avoid this kind of repetition. Generic programming is a style of computer programming in which algorithms are written in terms of types to be specified later. These types are then instantiated when needed for specific types provided as parameters. We are going to write a generic function named `getEntities` that takes two parameters:

```

function getEntities<T>(url : string, cb : (list : T[]) => void) :
void {
    $.ajax({
        url: url,
        method: "GET",
        success: function(data) {
            cb(data.items);
        },
        error : function(error) {
            cb(null);
        }
    });
}

```

We have added angle brackets (`<>`) after the name of our functions to indicate that it is a generic function. Enclosed in the angle brackets is the character `T`, which is used to refer to a type. The first parameter is named `url` and is a string; the second parameter is a function named `cb`, which takes a parameter list of type `T` as its only parameter.

We can now use this generic function to indicate what type T will represent:

```
getEntities<User>("/api/users", function(users : Users[]) {
  for(var i; users.length; i++) {
    console.log(users[i].name);
  }
});

getEntities<Order>("/api/orders", function(orders : Orders[]) {
  for(var i; orders.length; i++) {
    console.log(orders[i].total);
  }
});
```

Tag functions and tagged templates

We have already seen how to work with template strings such as the following:

```
var name = 'remo';
var surname = jansen;
var html = `<h1>${name} ${surname}</h1>`;
```

However, there is one use of template strings that we deliberately skipped because it is closely related to the use of a special kind of function known as **tag function**.

We can use a tag function to extend or modify the standard behavior of template strings. When we apply a tag function to a template string, the template string becomes a **tagged template**.

We are going to implement a tag function named `htmlEscape`. To use a tag function, we must use the name of the function followed by a template string:

```
var html = htmlEscape `<h1>${name} ${surname}</h1>`;
```

A tag template must return a string and take the following arguments:

- An array which contains all the static literals in the template string (`<h1>` and `</h1>` in the preceding example) is passed as the first argument.
- A rest parameter is passed as the second parameter. The rest parameter contains all the values in the template string (`name` and `surname` in the preceding example).

We now know the signature of a tag function.

```
tag(literals : string[], ...values : any[]) : string
```

Let's implement the `htmlEscape` tag function:

```
function htmlEscape(literals, ...placeholders) {  
    let result = "";  
    for (let i = 0; i < placeholders.length; i++) {  
        result += literals[i];  
        result += placeholders[i]  
            .replace(/&/g, '&amp;')  
            .replace(/\"/g, '"')  
            .replace(/\'/g, '&#39;')  
            .replace(/</g, '&lt;')  
            .replace(/>/g, '&gt;');  
    }  
    result += literals[literals.length - 1];  
    return result;  
}
```

The preceding function iterates through the literals and values and ensures that the HTML code is escaped from the values to avoid possible code injection attacks.

The main benefit of using a tagged function is that it allows us to create custom template string processors.



This feature will be available in the TypeScript 1.6 release.



Asynchronous programming in TypeScript

Now that we have seen how to work with functions, we will explore how we can use them, together with some native objects, to write asynchronous applications.

Callbacks and higher-order functions

In TypeScript, functions can be passed as arguments to another function. The function passed to another as an argument is known as a **callback**. Functions can also be returned by another function. The functions that accept functions as parameters (callbacks) or return functions as an argument are known as **higher-order functions**. Callbacks are usually anonymous functions.

```
var foo = function() { // callback  
    console.log('foo');
```

```
}

function bar(cb : () => void) { // higher order function
    console.log('bar');
    cb();
}

bar(foo); // prints 'bar' then prints 'foo'
```

Arrow functions

In TypeScript, we can declare a function using a function expression or an arrow function. An arrow function expression has a shorter syntax compared to function expressions and lexically binds the value of the `this` operator.

The `this` operator behaves a little differently in TypeScript compared to other languages. When we define a class in TypeScript, we can use the `this` operator to refer to the class's own properties. Let's take a look at an example:

```
class Person {
    name : string;
    constructor(name : string) {
        this.name = name;
    }
    greet() {
        alert(`Hi! My name is ${this.name}`);
    }
}
var remo = new Person("Remo");
remo.greet(); // "Hi! My name is Remo"
```

We have defined a `Person` class that contains a property of type `string` called `name`. The class has a constructor and a method `greet`. We have created an instance named `remo` and invoked the method named `greet`, which internally uses the `this` operator to access the `remo` property's `name`. Inside the `greet` method, the `this` operator points to the object that encloses the `greet` method.

We must be careful when using the `this` operator because in some scenarios it can point to the wrong value. Let's add an extra method to the previous example:

```
class Person {
    name : string;
    constructor(name : string) {
        this.name = name;
    }
}
```

```

greet() {
    alert(`Hi! My name is ${this.name}`);
}

greetDelay(time : number) {
    setTimeout(function() {
        alert(`Hi! My name is ${this.name}`);
    }, time);
}

var remo = new Person("remo");
remo.greet(); // "Hi! My name is remo"
remo.greetDelay(1000); // "Hi! My name is "

```

In the `greetDelay` method, we perform an almost identical operation to the one performed by the `greet` method. This time the function takes a parameter named `time`, which is used to delay the greet message.

In order to delay the message, we use the `setTimeout` function and a callback. As soon as we define an anonymous function (the callback), the `this` keyword changes its value and starts pointing to the anonymous function. This explains why the name `remo` is not displayed by the `greetDelay` message.

As mentioned, an arrow function expression lexically binds the value of the `this` operator. This means that it allows us to add a function without altering the value of this operator. Let's replace the function expression from the previous example with an arrow function:

```

class Person {
    name : string;
    constructor(name : string) {
        this.name = name;
    }
    greet() {
        alert(`Hi! My name is ${this.name}`);
    }
    greetDelay(time : number) {
        setTimeout(() => {
            alert(`Hi! My name is ${this.name}`);
        }, time);
    }
}

var remo = new Person("remo");
remo.greet(); // "Hi! My name is remo"
remo.greetDelay(1000); // "Hi! My name is remo"

```

By using an arrow function, we can ensure that the `this` operator still points to the Person instance and not to the `setTimeout` callback. If we execute the `greetDelay` function, the name property will be displayed as expected.

The following piece of code was generated by the TypeScript compiler. When compiling an arrow function, the TypeScript compiler will generate an alias for the `this` operator named `_this`. The alias is used to ensure that the `this` operator points to the right object.

```
Person.prototype.greetDelay = function (time) {  
    var _this = this;  
    setTimeout(function () {  
        alert("Hi! My name is " + _this.name);  
    }, time);  
};
```

Callback hell

We have seen that callbacks and higher order functions are two powerful and flexible TypeScript features. However, the use of callbacks can lead to a maintainability issue known as **callback hell**. We will now write a real-life example to showcase what a callback hell is and how easily we can end up dealing with it.



Remember that you can find the complete source code for this demo in the companion source code.

We are going to need handlebars and jQuery libraries, so let's install these two libraries and their respective type definition files using npm and tsd. We can then import their type definitions:

```
///<reference path="../typings/handlebars/handlebars.d.ts" />  
///<reference path="../typings/jquery/jquery.d.ts" />
```

To make our code easier to read, we will create an alias for the callback type:

```
type cb = (json : any) => void;
```

Now we need to declare our View class. The `View` class has some properties that allow us to set the following properties:

- **Container**: The DOM selector where we want our view to be inserted
- **Template URL**: The URL that will return a handlebars template

- **Service URL:** The URL of a web service that will return some JSON data
- **Arguments:** The data to be send to the service

We can see the view class implementation as follows:

```
class View {  
    private _container : string;  
    private _templateUrl : string;  
    private _serviceUrl : string;  
    private _args : any;  
    constructor(config) {  
        this._container = config.container;  
        this._templateUrl = config.templateUrl;  
        this._serviceUrl = config.serviceUrl;  
        this._args = config.args;  
    }  
    //...  
}
```

After defining the class constructor and its properties, we will add a private method named `_loadJson` to our class. This method takes the service URL, the arguments, a success callback, and an error callback as its arguments. Inside the method, we will send a jQuery AJAX request using the service URL and argument settings:

```
private _loadJson(url : string, args : any, cb : cb, errorCb :  
cb) {  
    $.ajax({  
        url: url,  
        type: "GET",  
        dataType: "json",  
        data: args,  
        success: (json) => {  
            cb(json);  
        },  
        error: (e) => {  
            errorCb(e);  
        }  
    });  
}  
//...
```



Handlebars is a library that allows us to compile and render HTML templates in a browser. These templates help with JSON-to-HTML transformations. We will mention this library later a couple of times, but don't worry if you have never used it before; this section is not about handlebars.

This section is about a set of tasks and how we can control the execution flow of those tasks using callbacks. If you want to learn more about handlebars, visit <http://handlebarsjs.com/>.

This function is almost identical to the previous one, but instead of loading some JSON, we will load a handlebars template:

```
private _loadHbs(url : string, cb : cb, errorCb : cb) {  
    $.ajax({  
        url: url,  
        type: "GET",  
        dataType: "text",  
        success: (hbs) => {  
            cb(hbs);  
        },  
        error: (e) => {  
            errorCb(e);  
        }  
    });  
}  
//...
```

This function takes a handlebar template code as input and tries to compile it using the handlebars compile function. Just like in the previous example, we use callbacks, which will be invoked after the success or failure of the operation:

```
private _compileHbs(hbs : string, cb : cb, errorCb : cb) {  
    try  
    {  
        var template = Handlebars.compile(hbs);  
        cb(template);  
    }  
    catch(e) {  
        errorCb(e);  
    }  
}  
//...
```

In this function, we take the already compiled template and the already loaded JSON data and put them together to transform JSON into HTML following the template formatting rules. Just like in the previous example, we use callbacks that will be invoked after the success or failure of the operation:

```
private _jsonToHtml(template : any, json : any, cb : cb, errorCallback : cb) {
    try {
        var html = template(json);
        cb(html);
    }
    catch(e) {
        errorCallback(e);
    }
}
//...
```

The following function takes the HTML generated by the `_jsonToHtml` function and appends it to a DOM element:

```
private _appendHtml = function (html : string, cb : cb, errorCallback : cb) {
    try {
        if($(this._container).length === 0) {
            throw new Error("Container not found!");
        }
        $(this._container).html(html);
        cb($(this._container));
    }
    catch(e) {
        errorCallback(e);
    }
}
//...
```

Now that we have a few functions that use callbacks, we will use all of them together in one single function named `render`. The `render` method controls the execution flow of the tasks, and executes them in the following order:

1. Loads the JSON data.
2. Loads the template.
3. Compiles the template.

4. Transforms JSON into HTML.
5. Appends HTML to the DOM.

Each task takes a success callback, which invokes the following tasks in the list if it is successful, and an error callback, which is invoked when something goes wrong:

```
public render (cb : cb, errorCallback : cb) {  
    try  
    {  
        this._loadJson(this._serviceUrl, this._args, (json) => {  
            this._loadHbs(this._templateUrl, (hbs) => {  
                this._compileHbs(hbs, (template) => {  
                    this._jsonToHtml(template, json, (html) => {  
                        this._appendHtml(html, cb);  
                    }, errorCallback);  
                }, errorCallback);  
            }, errorCallback);  
        }, errorCallback;  
    }  
    catch(e){  
        errorCallback(e);  
    }  
}  
}
```

In general, you should try to avoid nesting callbacks like in the preceding example because it will:

- Make the code harder to understand
- Make the code harder to maintain (refactor, reuse, and so on)
- Make exception handling more difficult

Promises

After seeing how the use of callbacks can lead to some maintainability problems, we will now look at promises and how they can be used to write better asynchronous code. The core idea behind promises is that a promise represents the result of an asynchronous operation. Promise must be in one of the three following states:

- **Pending:** The initial state of a promise
- **Fulfilled:** The state of a promise representing a successful operation
- **Rejected:** The state of a promise representing a failed operation

Once a promise is fulfilled or rejected, its state can never change again. Let's take a look at the basic syntax of a promise:

```
function foo() {  
    return new Promise((fulfill, reject) => {  
        try  
        {  
            // do something  
            fulfill(value);  
        }  
        catch(e){  
            reject(reason);  
        }  
    })  
}  
  
foo().then(function(value){ console.log(value); })  
.catch(function(e){ console.log(e); });
```

A `try...catch` statement is used here to showcase how we can explicitly fulfill or reject a promise. The `try...catch` statement is not really needed in a `Promise` function because when an error is thrown in a promise, the promise will automatically be rejected.

The preceding code snippet declares a function named `foo` that returns a promise. The promise contains a method named `then`, which accepts a function to be invoked when the promise is fulfilled. Promises also provide a method named `catch`, which is invoked when a promise is rejected.

We will now return to the callback hell example and make some changes in the code to use promises instead of callbacks.

Just like before, we are going to need handlebars and jQuery; so let's import their type definitions. In addition, this time, we will also need the declarations of a library known as `Q`:

```
///<reference path="../typings/handlebars/handlebars.d.ts" />  
///<reference path="../typings/jquery/jquery.d.ts" />  
///<reference path="../typings/q/q.d.ts" />
```

We will use the `Promise` object from a library instead of the native object because the libraries implement fallbacks so our code can work in old browsers. We will use a promises library known as `Q` (version 1.0.1) in this example. If you want to learn more about it, visit <https://github.com/kriskowal/q>.

The class name has changed from view to ViewAsync but everything else is still identical to the previous example:

```
class ViewAsync {  
    private _container : string;  
    private _templateUrl : string;  
    private _serviceUrl : string;  
    private _args : any;  
    constructor(config) {  
        this._container = config.container;  
        this._templateUrl = config.templateUrl;  
        this._serviceUrl = config.serviceUrl;  
        this._args = config.args;  
    }  
    //...
```

 Many developers append the word `Async` to the name of a function as a code convention, which is used to indicate that a function is an asynchronous function.

We will use our first promise in the function `_loadJsonAsync`. This function was named `_loadJson` in the callback example. We have removed the callbacks for success and error previously declared in the function signature. Finally, we have wrapped the function with a promise object and invoked the `resolve` and `reject` methods when the promise succeeds or fails respectively.

```
private _loadJsonAsync(url : string, args : any) {  
    return Q.Promise(function(resolve, reject) {  
        $.ajax({  
            url: url,  
            type: "GET",  
            dataType: "json",  
            data: args,  
            success: (json) => {  
                resolve(json);  
            },  
            error: (e) => {  
                reject(e);  
            }  
        });  
    }  
    //...
```

We will then refactor (rename, remove callbacks, wrap logic with a promise, and so on) each of the class functions (`_loadHbsAsync`, `compileHbsAsync`, and `_appendHtmlAsync`):

```
private _loadHbsAsync(url : string) {
  return Q.Promise(function(resolve, reject) {
    $.ajax({
      url: url,
      type: "GET",
      dataType: "text",
      success: (hbs) => {
        resolve(hbs);
      },
      error: (e) => {
        reject(e);
      }
    });
  }
}

private _compileHbsAsync(hbs : string) {
  return Q.Promise(function(resolve, reject) {
    try {
      var template : any = Handlebars.compile(hbs);
      resolve(template);
    }
    catch(e) {
      reject(e);
    }
  });
}

private _jsonToHtmlAsync(template : any, json : any) {
  return Q.Promise(function(resolve, reject) {
    try {
      var html = template(json);
      resolve(html);
    }
    catch(e) {
      reject(e);
    }
  });
}
```

```
private _appendHtmlAsync(html : string, container : string) {
    return Q.Promise((resolve, reject) => {
        try {
            var $container : any = $(container);
            if($container.length === 0) {
                throw new Error("Container not found!");
            }
            $container.html(html);
            resolve($container);
        }
        catch(e) {
            reject(e);
        }
    });
}
//...
```

The `RenderAsync` method (previously named `render`) will present some significant differences.

In the following function, we start by wrapping the function's logic with a promise, invoke the function `_loadJsonAsync`, and assign its return value to the variable `getJSON`. If we return to the `_loadJsonAsync` function, we will notice that the return type is a promise. Therefore, the `getJSON` variable is a promise that once fulfilled will return the JSON data required to render our view.

This time, we will invoke the `then` method, which belongs to the promise returned by the `_loadHbsAsync` method. This will allow us to pass the output of the function `_loadHbsAsync` to `_compileHbsAsync` when the promise's state changes to fulfilled.

```
public renderAsync() {
    return Q.Promise((resolve, reject) => {
        try {
            // assign promise to getJson
            var getJson = this._loadJsonAsync(this._serviceUrl,
                this._args);

            // assign promise to getTemplate
            var getTemplate = this._loadHbsAsync(this._templateUrl)
                .then(this._compileHbsAsync);

            // execute promises in parallel
            Q.all([getJson, getTemplate]).then((results) => {

```

```

        var json = results[0];
        var template = results[1];

        this._jsonToHtmlAsync(template, json)
        .then((html : string) => {
            return this._appendHtmlAsync(html, this._container);
        })
        .then(($container : any) => { resolve($container); });
    });

    catch(error) {
        reject(error);
    }
));
}
}
}

```

Once we have declared the `getJSON` and `getTemplate` variables (each containing a promise as a value) we will use the `all` method from the `Q` library to execute the `getJSON` and `getTemplate` promises in parallel.

`Q`'s `all` method takes a list of promises and a callback as input. Once all the promises in the list have been fulfilled, the callback is invoked and an array named `results` is passed to the fulfillment callback. The array contains the results of each of the promises in the same order that they were passed to the `all` method.

Inside `Q`'s `all` method callback, we will use the loaded JSON and the compiled template and arguments when invoking the `_jsonToHtmlAsync` promise. We will finally use the `then` method to call the `_appendHtmlAsync` method and resolve the promise.

As observed in the example, using promises gives us better control over the execution flow of each of the operations in our `render` method. Remember that you can use four different types of asynchronous flow control:

- **Concurrent:** The tasks are executed in parallel. We saw this in the example when we used the `all` method in the `getJSON` and `getTemplate` promises.
- **Series:** A group of tasks is executed in sequence but the preceding tasks do not pass arguments to the next task.
- **Waterfall:** A group of tasks is executed in sequence and each task passes arguments to the next task. This approach is useful when the tasks have dependencies on each other. In the preceding example, we find this asynchronous flow control approach when the `_loadHbsAsync` promise passes its output to the `_compileHbsAsync` promise.

- **Composite:** This is any combination of the previous concurrent, series, and waterfall approaches. The `render` method in the example uses a combination of all the asynchronous flow control approaches in this list.

Generators

If we invoke a function in TypeScript, we can assume that once the function starts running, it will always run to completion before any other code can run. This has been the case until now. However, a new kind of function which may be paused in the middle of execution – one or many times – and resumed later, allowing other code to run during these paused periods, is about to arrive in TypeScript and ES6. These new kinds of functions are known as **generators**.

A generator represents a sequence of values. The interface of a generator object is just an iterator. The `next()` function can be invoked until it runs out of values.

We can define the constructor of a generator by using the `function` keyword followed by an asterisk (*). The `yield` keyword is used to stop the execution of the function and return a value. Let's take a look at an example:

```
function *foo() {  
    yield 1;  
    yield 2;  
    yield 3;  
    yield 4;  
    return 5;  
}  
  
var bar = new foo();  
bar.next(); // Object {value: 1, done: false}  
bar.next(); // Object {value: 2, done: false}  
bar.next(); // Object {value: 3, done: false}  
bar.next(); // Object {value: 4, done: false}  
bar.next(); // Object {value: 5, done: true}  
bar.next(); // Object { done: true }
```

As you can see, this iterator has five steps. The first time we call `next`, the function will be executed until it reaches the first `yield` statement, and then it will return the value 1 and stop the execution of the function until we invoke the generator's `next` method again. As we can see, we are now able to stop the function's execution at a given point. This allows us to write infinite loops without causing a stack overflow as in the following example:

```
function* foo() {  
    var i = 1;
```

```
while (true) {
    yield i++;
}

var bar = new foo();
bar.next(); // Object {value: 1, done: false}
bar.next(); // Object {value: 2, done: false}
bar.next(); // Object {value: 3, done: false}
bar.next(); // Object {value: 4, done: false}
bar.next(); // Object {value: 5, done: false}
bar.next(); // Object {value: 6, done: false}
bar.next(); // Object {value: 7, done: false}
// ...
```

Generators will open possibilities for synchronicity as we can call a generator's next method after some asynchronous event has occurred.

Asynchronous functions – `async` and `await`

Asynchronous functions are a TypeScript feature that is scheduled to arrive with the upcoming TypeScript releases. An asynchronous function is a function that is expected to be invoked in a synchronous operation. Developers can use the `await` keyword to wait for the function results without blocking the normal execution of the program.

Asynchronous functions will be implemented using promises when targeting ES6, and promise fallbacks when targeting ES3 and ES5.

Using asynchronous functions generally helps to increase the readability of a piece of code when compared with the use of promises; but technically we can achieve the same features using both promises and synchronous code.

Let's take a sneak-peek at this upcoming feature:

```
var p: Promise<number> = /* ... */;

async function fn(): Promise<number> {
    var i = await p;
    return 1 + i;
}
```

The preceding code snippet declares a promise named `p`. This promise is the piece of code that will wait to be executed. While waiting, the program execution will not be blocked because we will wait from an asynchronous function named `fN`. As we can see, the `fN` function is preceded by the `async` keyword, which is used to indicate to the compiler that it is an asynchronous function.

Inside the function, the `await` keyword is used to suspend execution until `p` is settled. As we can see, the syntax is much more minimalistic and cleaner than it would be if we used the promises API (then and catch methods and callbacks).



Refer to the TypeScript roadmap to learn more about the stages of development of this feature.



Summary

In this chapter, we saw how to work with functions in depth. We started with a quick recap of some basic concepts and then moved to some lesser known function features and use cases.

Once we saw how to work with functions, we focused on the usage of callbacks, promises, and generators to take advantage of the asynchronous programming capabilities of Typescript.

In the next chapter, we will look at how to work with classes, interfaces, and other object-oriented programming features of the TypeScript programming language.

Object-Oriented Programming with TypeScript

In the previous chapter, we explored the use of functions and some asynchronous techniques. In this chapter, we will see how to group our functions in reusable components, such as classes or modules. This chapter is divided into two main sections. The first part will cover the following topics:

- SOLID principles
- Classes
- Association, aggregation, and composition
- Inheritance
- Mixins
- Generic classes
- Generic constraints
- Interfaces

In the second part, we will focus on the declaration and use of namespaces and external modules. The second part will cover the following topics:

- Namespaces (internal modules)
- External modules
- **Asynchronous module definition (AMD)**
- CommonJS modules
- ES6 modules
- Browserify and **universal module definition (UMD)**
- Circular dependencies

SOLID principles

In the early days of software development, developers used to write code with procedural programming languages. In procedural programming languages, the programs follow a top-to-bottom approach and the logic is wrapped with functions.

New styles of computer programming, such as modular programming or structured programming, emerged when developers realized that procedural computer programs could not provide them with the desired level of abstraction, maintainability, and reusability.

The development community created a series of recommended practices and design patterns to improve the level of abstraction and reusability of procedural programming languages, but some of these guidelines required a certain level of expertise. In order to facilitate adherence to these guidelines, a new style of computer programming known as **object-oriented programming (OOP)** was created.

Developers quickly noticed some common OOP mistakes and came up with five rules that every OOP developer should follow to create a system that is easy to maintain and extend over time. These five rules are known as the SOLID principles. SOLID is an acronym introduced by Michael Feathers, which stands for the following principles:

- **Single responsibility principle (SRP):** This principle states that a software component (function, class, or module) should focus on one unique task (have only one responsibility).
- **Open/closed principle (OCP):** This principle states that software entities should be designed with application growth (new code) in mind (should be open to extension), but the application growth should require the fewer possible number of changes to the existing code (be closed for modification).
- **Liskov substitution principle (LSP):** This principle states that we should be able to replace a class in a program with another class as long as both classes implement the same interface. After replacing the class, no other changes should be required, and the program should continue to work as it did originally.
- **Interface segregation principle (ISP):** This principle states that we should split interfaces that are very large (general-purpose interfaces) into smaller and more specific ones (many client-specific interfaces) so that clients will only need to know about the methods that are of interest to them.
- **Dependency inversion principle (DIP):** This principle states that entities should depend on abstractions (interfaces) as opposed to depending on concretion (classes).

In this chapter, we will see how to write TypeScript code that adheres to these principles so that our applications are easy to maintain and extend over time.

Classes

We should already be familiar with the basics about TypeScript classes, as we have declared some of them in previous chapters. So we will now look at some details and OOP concepts through examples. Let's start by declaring a simple class:

```
class Person {  
    public name : string;  
    public surname : string;  
    public email : string;  
    constructor(name : string, surname : string, email : string){  
        this.email = email;  
        this.name = name;  
        this.surname = surname;  
    }  
    greet() {  
        alert("Hi!");  
    }  
}  
  
var me : Person = new Person("Remo", "Jansen",  
"remo.jansen@wolksoftware.com");
```

We use classes to represent the type of an object or entity. A **class** is composed of a name, attributes, and methods. The class in the preceding example is named `Person` and contains three attributes or properties (`name`, `surname`, and `email`) and two methods (`constructor` and `greet`). Class attributes are used to describe the object's characteristics, while class methods are used to describe its behavior.

A **constructor** is a special method used by the `new` keyword to create instances (also known as objects) of our class. We have declared a variable named `me`, which holds an instance of the `Person` class. The `new` keyword uses the `Person` class's constructor to return an object whose type is `Person`.

A class should adhere to the single responsibility principle (SRP). The `Person` class in the preceding example represents a person, including all their characteristics (attributes) and behaviors (methods). Now let's add some `email` as validation logic:

```
class Person {  
    public name : string;  
    public surname : string;
```

```

public email : string;
constructor(name : string, surname : string, email : string) {
    this.surname = surname;
    this.name = name;
    if(this.validateEmail(email)) {
        this.email = email;
    }
    else {
        throw new Error("Invalid email!");
    }
}
validateEmail() {
    var re = /\S+@\S+\.\S+/;
    return re.test(this.email);
}
greet() {
    alert("Hi! I'm " + this.name + ". You can reach me at " +
    this.email);
}
}

```

When an object doesn't follow the SRP and it knows too much (has too many properties) or does too much (has too many methods), we say that the object is a God object. The `Person` class here is a God object because we have added a method named `validateEmail` that is not really related to the `Person` class's behavior.

Deciding which attributes and methods should or should not be part of a class is a relatively subjective decision. If we spend some time analyzing our options, we should be able to find a way to improve the design of our classes.

We can refactor the `Person` class by declaring an `Email` class, responsible for e-mail validation, and use it as an attribute in the `Person` class:

```

class Email {
    public email : string;
constructor(email : string){
    if(this.validateEmail(email)) {
        this.email = email;
    }
    else {
        throw new Error("Invalid email!");
    }
}
validateEmail(email : string) {
    var re = /\S+@\S+\.\S+/;

```

```
        return re.test(email);
    }
}
```

Now that we have an `Email` class, we can remove the responsibility of validating the emails from the `Person` class and update its `email` attribute to use the type `Email` instead of `string`:

```
class Person {
    public name : string;
    public surname : string;
    public email : Email;
    constructor(name : string, surname : string, email : Email){
        this.email = email;
        this.name = name;
        this.surname = surname;
    }
    greet() {
        alert("Hi!");
    }
}
```

Making sure that a class has a single responsibility makes it easier to see what it does and how we can extend/improve it. We can further improve our `Person` and `Email` classes by increasing the level of abstraction of our classes. For example, when we use the `Email` class, we don't really need to be aware of the existence of the `validateEmail` method; so this method could be invisible from outside the `Email` class. As a result, the `Email` class would be much simpler to understand.

When we increase the level of abstraction of an object, we can say that we are encapsulating the object's data and behavior. Encapsulation is also known as information hiding. For example, the `Email` class allows us to use emails without having to worry about e-mail validation because the class will deal with it for us. We can make this clearer by using access modifiers (`public` or `private`) to flag as private all the class attributes and methods that we want to abstract from the use of the `Email` class:

```
class Email {
    private email : string;
    constructor(email : string) {
        if(this.validateEmail(email)) {
            this.email = email;
        }
        else {
            throw new Error("Invalid email!");
        }
    }
}
```

```
}

private validateEmail(email : string) {
    var re = /\S+@\S+\.\S+/;
    return re.test(email);
}

get():string {
    return this.email;
}

}
```

We can then simply use the `Email` class without needing to explicitly perform any kind of validation:

```
var email = new Email("remo.jansen@wolksoftware.com");
```

Interfaces

The feature that we will miss the most when developing large-scale web applications with JavaScript is probably interfaces. We have seen that following the SOLID principles can help us to improve the quality of our code, and writing good code is a must when working on a large project. The problem is that if we attempt to follow the SOLID principles with JavaScript, we will soon realize that without interfaces, we will never be able to write SOLID OOP code. Fortunately for us, TypeScript features interfaces.

Traditionally, in OOP, we say that a class can extend another class and implement one or more interfaces. An interface can implement one or more interfaces and cannot extend another class or interface. Wikipedia's definition of interfaces in OOP is as follows:

In object-oriented languages, the term interface is often used to define an abstract type that contains no data or code, but defines behaviors as method signatures.

Implementing an interface can be understood as signing a contract. The interface is a contract, and when we sign it (implement it), we must follow its rules. The interface rules are the signatures of the methods and properties, and we must implement them.

We will see many examples of interfaces later in this chapter.

In TypeScript, interfaces don't strictly follow this definition. The two main differences are that in TypeScript:

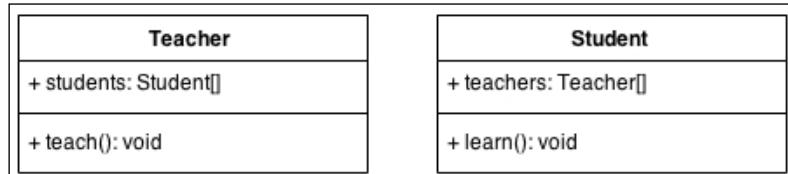
- An interface can extend another interface or class
- An interface can define data and behaviors as opposed to only behaviors

Association, aggregation, and composition

In OOP, classes can have some kind of relationship with each other. Now, we will take a look at the three different types of relationships between classes.

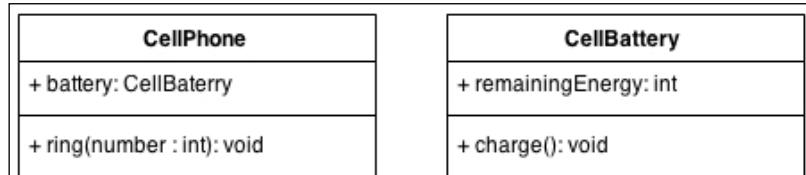
Association

We call **association** those relationships whose objects have an independent lifecycle and where there is no ownership between the objects. Let's take an example of a teacher and student. Multiple students can associate with a single teacher, and a single student can associate with multiple teachers, but both have their own lifecycles (both can be created and deleted independently); so when a teacher leaves the school, we don't need to delete any students, and when a student leaves the school, we don't need to delete any teachers.



Aggregation

We call **aggregation** those relationships whose objects have an independent lifecycle, but there is ownership, and child objects cannot belong to another parent object. Let's take an example of a cell phone and a cell phone battery. A single battery can belong to a phone, but if the phone stops working, and we delete it from our database, the phone battery will not be deleted because it may still be functional. So in aggregation, while there is ownership, objects have their own lifecycle.

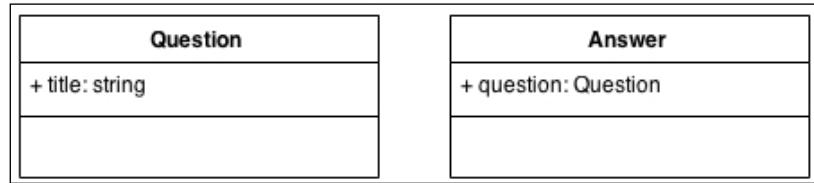


Composition

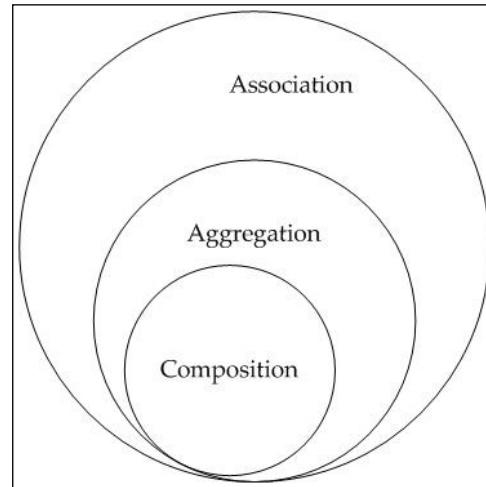
We use the term **composition** to refer to relationships whose objects don't have an independent lifecycle, and if the parent object is deleted, all child objects will also be deleted.

Let's take an example of the relationship between questions and answers. Single questions can have multiple answers, and answers cannot belong to multiple questions. If we delete questions, answers will automatically be deleted.

Objects with a dependent life cycle (answers, in the example) are known as **weak entities**.



Sometimes, it can be a complicated process to decide if we should use association, aggregation, or composition. This difficulty is caused in part because aggregation and composition are subsets of association, meaning they are specific cases of association.



Inheritance

One of the most fundamental object-oriented programming features is its capability to extend existing classes. This feature is known as inheritance and allows us to create a new class (child class) that inherits all the properties and methods from an existing class (parent class). Child classes can include additional properties and methods not available in the parent class. Let's return to our previously declared `Person` class. We will use the `Person` class as the parent class of a child class named `Teacher`:

```
class Person {  
    public name : string;  
    public surname : string;  
    public email : Email;  
    constructor(name : string, surname : string, email : Email){  
        this.name = name;  
        this.surname = surname;  
        this.email = email;  
    }  
    greet() {  
        alert("Hi!");  
    }  
}
```



This example is included in the companion source code.

Once we have a parent class in place, we can extend it by using the reserved keyword `extends`. In the following example, we declare a class called `Teacher`, which extends the previously defined `Person` class. This means that `Teacher` will inherit all the attributes and methods from its parent class:

```
class Teacher extends Person {  
    teach() {  
        alert("Welcome to class!");  
    }  
}
```

Note that we have also added a new method named `teach` to the class `Teacher`. If we create instances of the `Person` and `Teacher` classes, we will be able to see that both instances share the same attributes and methods with the exception of the `teach` method, which is only available for the instance of the `Teacher` class:

```
var teacher = new Teacher("remo", "jansen", new  
Email("remo.jansen@wolksoftware.com"));
```

```
var me = new Person("remo", "jansen", new
Email("remo.jansen@wolksoftware.com"));

me.greet();
teacher.greet();
me.teach(); // Error : Property 'teach' does not exist on type
'Person'
teacher.teach();
```

Sometimes, we will need a child class to provide a specific implementation of a method that is already provided by its parent class. We can use the reserved keyword `super` for this purpose. Imagine that we want to add a new attribute to list the teacher's subjects, and we want to be able to initialize this attribute through the teacher constructor. We will use the `super` keyword to explicitly reference the parent class constructor inside the child class constructor. We can also use the `super` keyword when we want to extend an existing method, such as `greet`. This OOP language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by its parent classes is known as **method overriding**.

```
class Teacher extends Person {
    public subjects : string[];
    constructor(name : string, surname : string, email : Email, subjects
: string[]){
        super(name, surname, email);
        this.subjects = subjects;
    }
    greet() {
        super.greet();
        alert("I teach " + this.subjects);
    }
    teach() {
        alert("Welcome to Maths class!");
    }
}

var teacher = new Teacher("remo", "jansen", new
Email("remo.jansen@wolksoftware.com"), ["math", "physics"]);
```

We can declare a new class that inherits from a class that is already inheriting from another. In the following code snippet, we declare a class called `SchoolPrincipal` that extends the `Teacher` class, which extends the `Person` class:

```
class SchoolPrincipal extends Teacher {  
    manageTeachers() {  
        alert("We need to help students to get better results!");  
    }  
}
```

If we create an instance of the `SchoolPrincipal` class, we will be able to access all the properties and methods from its parent classes (`SchoolPrincipal`, `Teacher`, and `Person`):

```
var principal = new SchoolPrincipal("remo", "jansen", new  
Email("remo.jansen@wolksoftware.com"), ["math", "physics"]);  
principal.greet();  
principal.teach();  
principal.manageTeachers();
```

It is not recommended to have too many levels in the inheritance tree. A class situated too deeply in the inheritance tree will be relatively complex to develop, test, and maintain. Unfortunately, we don't have a specific rule that we can follow when we are unsure whether we should increase the **depth of inheritance tree (DIT)**.

We should use inheritance in such a way that it helps us to reduce the complexity of our application and not the opposite. We should try to keep the DIT between 0 and 4 because a value greater than 4 would compromise encapsulation and increase complexity.

Mixins

Sometimes, we will find scenarios in which it would be a good idea to declare a class that inherits from two or more classes simultaneously (known as **multiple inheritance**).

Let's take a look at an example. We will not add any code to the methods in this example because we want to avoid the possibility of getting distracted by it; we should focus on the inheritance tree:

```
class Animal {  
    eat() {  
        // ...  
    }  
}
```

We started by declaring a class named `Animal`, which only has one method named `eat`. Now, let's declare two new classes:

```
class Mammal extends Animal {  
    breathe() {  
        // ...  
    }  
}  
  
class WingedAnimal extends Animal {  
    fly() {  
        // ...  
    }  
}
```

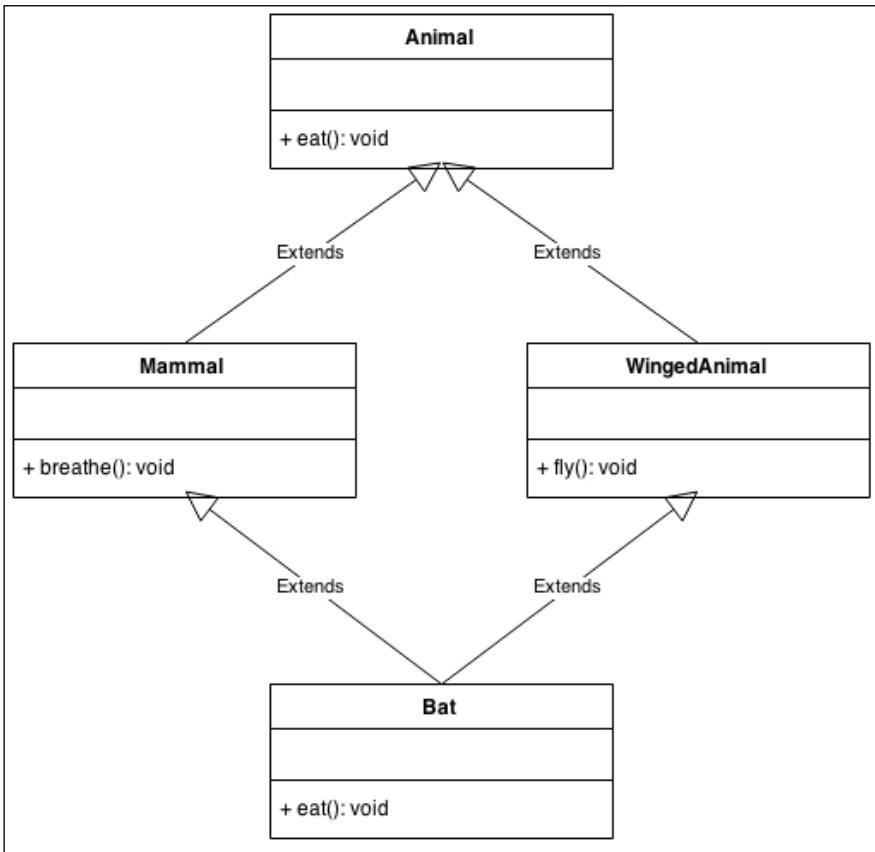
We have declared two new classes named `WingedAnimal` and `Mammal`. Both classes inherit from the `Animal` class.

Now that we have our classes ready, we are going to try to implement a class named `Bat`. Bats are mammals and have wings – creating a new class named `Bat`, which will extend both the `Mammal` and `WingedAnimal` classes, seems logical. However, if we attempt to do so, we will encounter a compilation error:

```
// Error: Classes can only extend a single class.  
class Bat extends WingedAnimal, Mammal {  
    // ...  
}
```

This error is thrown because TypeScript doesn't support multiple inheritance. This means that a class can only extend one class. The designers of programming languages such as C# or TypeScript decided to not support multiple inheritance because it can potentially increase the complexity of applications.

Sometimes, a class inheritance diagram can take a diamond-like shape (as seen in the following figure). This kind of class inheritance diagram can potentially lead us to design issue known as the **diamond problem**.



We will not face any problems if we call a method that is exclusive to only one of the classes in the inheritance tree:

```

var bat = new Bat();
bat.fly();
bat.eat();
bat.breathe();

```

The diamond problem takes place when we try to invoke one of the `Bat` class's parent's methods, and it is unclear or ambiguous which of the parent's implementations of that method should be invoked. If we add a method named `move` to both the `Mammal` and the `WingedAnimal` class and try to invoke it from an instance of `Bat`, we will get an ambiguous call error.

Now that we know why multiple inheritance can be potentially dangerous, we will introduce a feature known as **mixin**. Mixins are alternatives to multiple inheritance, but this feature has some limitations.

Let's return to the Bat class example to showcase the usage of mixins:

```
class Mammal {  
    breathe() : string {  
        return "I'm alive!";  
    }  
}  
  
class WingedAnimal {  
    fly() : string{  
        return "I can fly!";  
    }  
}
```



This example is included in the companion source code.



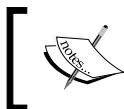
The two classes presented in the preceding example are not much different from the previous example; we have added some logic to the `breathe` and `fly` methods, so we can have some output to help us to understand this demonstration. Also, note that the classes no longer extend the `Animal` class:

```
class Bat implements Mammal, WingedAnimal {  
    breathe : () => string;  
    fly : () => string;  
}
```

The `Bat` class has some important additions. We have used the reserved keyword `implements` (as opposed to `extends`) to indicate that `Bat` will implement the functionality declared in both the `Mammal` and `WingedAnimal` classes. We have also added the signature of each of the methods that the `Bat` class will implement.

We need to copy the following function somewhere in our code to be able to apply mixins:

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {  
    baseCtors.forEach(baseCtor => {  
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name =>  
        {  
            if (name !== 'constructor') {  
                derivedCtor.prototype[name] = baseCtor.prototype[name];  
            }  
        })  
    })  
}
```



The preceding function is a well-known pattern and can be found in many books and online references, including the official TypeScript handbook.

This function will iterate each property of the parent classes (contained in an array named `baseCtors`) and copy the implementation to a child class (`derivedCtor`).

We only need to declare this function once in our application. Once we have done it, we can use it as follows:

```
applyMixins(Bat, [Mammal, WingedAnimal]);
```

The child class (`Bat`) will then contain the implementation of each property and method of the two parent classes (`WingedAnimal` and `Mammal`):

```
var bat = new Bat();
bat.breathe(); // I'm alive!
bat.fly(); // I can fly!
```

As we said at the beginning of this section, mixins have some limitations. The first limitation is that we can only inherit the properties and methods from one level in the inheritance tree. Now we can understand why we removed the `Animal` class prior to applying the mixin. The second limitation is that, if two or more of the parent classes contain a method with the same name, the method that is going to be inherited will be taken from the last class passed in the `baseCtors` array to the `applyMixins` function. We will now see an example that presents both these limitations.

In order to show the first limitation, we will declare the `Animal` class:

```
class Animal {
  eat() : string {
    return "Delicious!";
  }
}
```

We will then declare the `Mammal` and `WingedAnimal` classes, but this time, they will extend the `Animal` class:

```
class Mammal extends Animal {
  breathe() : string {
    return "I'm alive!";
  }
  move() : string {
    return "I can move like a mammal!";
  }
}
```

```
class WingedAnimal extends Animal {
    fly() : string{
        return "I can fly!";
    }
    move() : string {
        return "I can move like a bird!";
    }
}
```

We will then declare a `Bat` class but we will name it `Bat1`. This class will implement both the `Mammal` and `WingedAnimal` classes:

```
class Bat1 implements Mammal, WingedAnimal {
    eat : () => string;
    breathe : () => string;
    fly : () => string;
    move : () => string;
}
```

We are ready to invoke the `applyMixins` function. Notice how we pass `Mammal` before `WingedAnimal` in the array:

```
applyMixins(Bat1, [Mammal, WingedAnimal]);
```

We can now create an instance of `Bat1`, and we will be able to observe that the `eat` method has not been inherited from the `Animal` class due to the first limitation:

```
var bat1 = new Bat();
bat1.eat(); // Error: not a function
```

Each of the parent class's methods has been inherited without issues:

```
bat1.breathe(); // I'm alive!
bat1.fly(); // I can fly!"
```

Except the `move` method because according to the second limitation, only the implementation of the last parent class passed to the `applyMixins` method will be implemented. In this case, the implementation is inherited from the `WingedAnimal` class:

```
bat1.move(); // I can move like a bird
```

To finalize, we will see the effect of switching the order of the parent classes when invoking the `applyMixins` method:

```
class Bat2 implements WingedAnimal, Mammal {
    eat : () => string;
    breathe : () => string;
```

```
fly : () => string;
move : () => string;
}
```

Notice how we have passed `WingedAnimal` before `Mammal` in the array:

```
applyMixins(Bat2, [WingedAnimal, Mammal]);
var bat2 = new Bat2();
bat2.eat(); // Error: not a function
bat2.breathe(); // I'm alive!
bat2.fly(); // I can fly!
bat2.move() // I can move like a mammal
```

Generic classes

In the previous chapter, we saw how to work with generic functions. We will now take a look at how to work with generic classes.

Just like with generic functions, generic classes can help us to avoid the duplication of code. Let's take a look at an example:

```
class User {
  public name : string;
  public password : string;
}
```



This example is included in the companion source code.



We have declared a `User` class, which contains two properties: `name` and `password`. We will now declare a class named `NotGenericUserRepository` without using generics. This class takes a URL via its constructor and has a method named `getAsync`. The `getAsync` method will request a list of users stored in a JSON file using AJAX:

```
class NotGenericUserRepository {
  private _url : string;
  constructor(url : string) {
    this._url = url;
  }
  public getAsync() {
    return Q.Promise((resolve : (users : User[]) => void, reject) => {
      $.ajax({
        url: this._url,
```

```

        type: "GET",
        dataType: "json",
        success: (data) => {
            var users = <User[]>data.items;
            resolve(users);
        },
        error: (e) => {
            reject(e);
        }
    });
}
}

```

Once we have finished declaring the `NotGenericUserRepository` user repository, we can create an instance and invoke the `getAsync` method:

```

var notGenericUserRepository = new NotGenericUserRepository("./demos/
shared/users.json");
notGenericUserRepository.getAsync()
.then(function(users : User[]){
    console.log('notGenericUserRepository => ', users);
});

```

If we also need to request another list of entities different from `User`, we could end up duplicating a lot of code. Imagine that we also need to request a list of conference talks. We could create an entity named `Talk` and an almost identical repository class:

```

class Talk {
    public title : string;
    public description : string;
    public language : string;
    public url : string;
    public year : string;
}

class NotGenericTalkRepository {
    private _url : string;
    constructor(url : string) {
        this._url = url;
    }
    public getAsync() {
        return Q.Promise((resolve : (talks : Talk[]) => void, reject) => {
            $ajax({

```

```
        url: this._url,
        type: "GET",
        dataType: "json",
        success: (data) => {
            var users talks = <Talk[]>data.items;
            resolve(userstalks);
        },
        error: (e) => {
            reject(e);
        }
    });
});
```

If the number of entities grows, we will continue to repeatedly duplicate code. We may think that we could use the `any` type to avoid this problem, but then we would be losing the security provided by the checking type performed by TypeScript at compilation time. A much better solution is to create a Generic repository:

```
class GenericRepository<T> {
    private _url : string;
    constructor(url : string) {
        this._url = url;
    }
    public getAsync() {
        return Q.Promise((resolve : (entities : T[]) => void, reject) => {
            $.ajax({
                url: this._url,
                type: "GET",
                dataType: "json",
                success: (data) => {
                    var list = <T[]>data.items;
                    resolve(list);
                },
                error: (e) => {
                    reject(e);
                }
            });
        });
    }
}
```

The repository code is identical to `NotGenericUserRepository`, except for the entity type. We have removed the hardcoded reference to the `User` and `Talk` entities and replaced them with the generic type `T`. We can now declare as many repositories as we wish without duplicating a single line of code:

```
var userRepository = new GenericRepository<User>("./demos/shared/users.json");
userRepository.getAsync()
  .then((users : User[]) => {
    console.log('userRepository => ', users);
  });

var talkRepository = new GenericRepository<Talk>("./demos/shared/talks.json");
talkRepository.getAsync()
  .then((talks : Talk[]) => {
    console.log('talkRepository => ', talks);
  });
}
```

Generic constraints

Sometimes, we might need to restrict the use of a generic class. Take the generic repository from the previous section as an example. We have a new requirement: we need to add some changes to validate the entities loaded via AJAX, and we will return only the valid entities.

One possible solution is to use the `typeof` operator to identify the type of the generic type parameter `T` within a generic class or function:

```
// ...
success: (data) => {
  var list : T[];
  var items = <T[]>data.items;
  for(var i = 0; i < items.length; i++){
    if(items[i] instanceof User) {
      // validate user
    }
    if(items[i] instanceof Talk) {
      // validate talk
    }
  }
  resolve(list);
}
// ...
```

The problem is that we will have to modify our `GenericRepository` class to add extra logic with each new entity. We will not add the validation rules into the `GenericRepository` repository class because a generic class should not be aware of the type used as the generic type.

A better solution is to add a method named `isValid` to the entities, which will return true if the entity is valid:

```
// ...
success: (data) => {
  var list : T[];
  var items = <T[]>data.items;
  for(var i = 0; i < items.length; i++) {
    if(items[i].isValid()) { // error
      // ...
    }
  }
  resolve(list);
}
// ...
```

The second approach follows the second SOLID principle, the open/close principle, as we can create new entities and the generic repository will continue to work (open for extension), but no additional changes to it will be required (closed for modification). The only problem with this approach is that, if we attempt to invoke an entity's `isValid` method inside the generic repository, we will get a compilation error.

The error is thrown because we are allowed to use the generic repository with any type, but not all types have a method named `isValid`. Fortunately, this issue can easily be resolved by using a generic constraint. Constraints will restrict the types that we are allowed to use as the generic type parameter `T`. We are going to declare a constraint, so only types that implement an interface named `ValidatableInterface` can be used with the generic method.

Let's start by declaring an interface:

```
interface ValidatableInterface {
  isValid() : boolean;
}
```



This example is included in the companion source code.

Now we can proceed to implement the interface. In this case, we must implement the `isValid` method:

```
class User implements ValidatableInterface {
    public name : string;
    public password : string;
    public isValid() : boolean {
        // user validation...
        return true;
    }
}

class Talk implements ValidatableInterface {
    public title : string;
    public description : string;
    public language : string;
    public url : string;
    public year : string;
    public isValid() : boolean {
        // talk validation...
        return true;
    }
}
```

Now, let's declare a generic repository and add a type constraint so only types derived from `ValidatableInterface` will be accepted as the generic type parameter `T`:

```
class GenericRepositoryWithConstraint<T extends
ValidatableInterface> {
    private _url : string;
    constructor(url : string) {
        this._url = url;
    }
    public getAsync() {
        return Q.Promise((resolve : (talks : T[]) => void, reject) =>
        {
            $.ajax({
                url: this._url,
                type: "GET",
                dataType: "json",
                success: (data) => {
                    var items = <T[]>data.items;
                    for(var i = 0; i < items.length; i++) {
```

```
        if(items[i].isValid()) {
            list.push(items[i]);
        }
    }
    resolve(list);
},
error: (e) => {
    reject(e);
}
));
});
}
}
```

 Even though we have used an interface, we used the `extends` keyword and not the `implements` keyword to declare the constraint in the preceding example. There is no special reason for that. This is just the way the TypeScript constraint syntax works.

We can then create as many repositories as we want:

```
var userRepository = new
GenericRepositoryWithConstraint<User>("./users.json");

userRepository.getAsync()
.then(function(users : User[]){
    console.log(users);
});

var talkRepository = new
GenericRepositoryWithConstraint<Talk>("./talks.json");

talkRepository.getAsync()
.then(function(talks : Talk[]){
    console.log(talks);
});
```

If we attempt to use a class that doesn't implement the `ValidatableInterface` of the generic type parameter `T`, we will get a compilation error.

Multiple types in generic type constraints

We can only refer to one type when declaring a generic type constraint. Let's imagine that we need a generic class to be constrained, so it only allows types that implement the following two interfaces:

```
interface IMyInterface {  
    doSomething();  
};  
interface IMySecondInterface {  
    doSomethingElse();  
};
```

We may think that we can define the required generic constraint as follows:

```
class Example<T extends IMyInterface, IMySecondInterface> {  
    private genericProperty : T;  
    useT() {  
        this.genericProperty.doSomething();  
        this.genericProperty.doSomethingElse(); // error  
    }  
}
```

However, this code snippet will throw a compilation error. We cannot specify multiple types when declaring a generic type constraint. However, we can work around this issue by transforming `IMyInterface`, `IMySecondInterface` in super-interfaces:

```
interface IChildInterface extends IMyInterface, IMySecondInterface {  
}
```

`IMyInterface` and `IMySecondInterface` are now super-interfaces because they are the parent interfaces of the `IChildInterface` interface. We can then declare the constraint using the `IChildInterface` interface:

```
class Example<T extends IChildInterface> {  
    private genericProperty : T;  
    useT() {  
        this.genericProperty.doSomething();  
        this.genericProperty.doSomethingElse();  
    }  
}
```

The new operator in generic types

To create a new object within generic code, we need to indicate that the generic type T has a constructor function. This means that instead of using `type:T`, we should use `type: { new(): T; }` as follows:

```
function factoryNotWorking<T>(): T {
    return new T(); // compile error could not find symbol T
}

function factory<T>(): T {
    var type: { new(): T; };
    return new type();
}

var myClass: MyClass = factory<MyClass>();
```

Applying the SOLID principles

As we have previously mentioned, interfaces are fundamental features when it comes to following the SOLID principles, and we have already put the first two SOLID principles into practice.

We have already discussed the single responsibility principle. Now, we will see real examples of the three remaining principles.

The Liskov substitution principle

The Liskov substitution principle (LSP) states, *Subtypes must be substitutable for their base types*. Let's take a look at an example to understand what this means.

We will declare a class named `PersistanceService`, the responsibility of which is to persist some object into some sort of storage. We will start by declaring the following interface:

```
interface PersistanceServiceInterface {
    save(entity : any) : number;
}
```

After declaring the `PersistanceServiceInterface` interface, we can implement it. We will use cookies as the storage for the application's data:

```
class CookiePersistanceService implements PersistanceServiceInterface{  
    save(entity : any) : number {  
        var id = Math.floor((Math.random() * 100) + 1);  
        // Cookie persistance logic...  
        return id;  
    }  
}
```

We will continue by declaring a class named `FavouritesController`, which has a dependency on `PersistanceServiceInterface`:

```
class FavouritesController {  
    private _persistanceService : PersistanceServiceInterface;  
    constructor(persistanceService : PersistanceServiceInterface) {  
        this._persistanceService = persistanceService;  
    }  
    public saveAsFavourite(articleId : number) {  
        return this._persistanceService.save(articleId);  
    }  
}
```

We can finally create an instance of `FavouritesController` and pass an instance of `CookiePersistanceService` via its constructor:

```
var favController = new FavouritesController(new  
CookiePersistanceService());
```

The LSP allows us to replace a dependency with another implementation as long as both implementations are based in the same base type; so, if we decide to stop using cookies as storage and use the HTML5 local storage API instead, we can declare a new implementation:

```
class LocalStoragePersistanceService implements  
PersistanceServiceInterface{  
    save(entity : any) : number {  
        var id = Math.floor((Math.random() * 100) + 1);  
        // Local storage persistance logic...  
        return id;  
    }  
}
```

We can then replace it without having to add any changes to the `FavouritesController` controller class.

```
var favController = new FavouritesController(new  
LocalStoragePersistanceService());
```

The interface segregation principle

Interfaces are used to declare how two or more software components cooperate and exchange information with each other. This declaration is known as **application programming interface (API)**. In the previous example, our interface was `PersistanceServiceInterface`, and it was implemented by the classes `LocalStoragePersistanceService` and `CookiePersitanceService`. The interface was consumed by the `FavouritesController` class; so we say that this class is a client of the `PersistanceServiceInterface`'s API.

The **interface segregation principle (ISP)** states that no client should be forced to depend on methods it does not use. To adhere to the ISP, we need to keep in mind that when we declare the API (how two or more software components cooperate and exchange information with each other) of our application's components, the declaration of many client-specific interfaces is better than the declaration of one general-purpose interface. Let's take a look at an example.

If we design an API to control all the elements in a vehicle (engine, radio, heating, navigation, lights...), we could have one general-purpose interface, which allows us to control every single element of the vehicle:

```
interface VehicleInterface {  
    getSpeed() : number;  
    getVehicleType: string;  
    isTaxPayed() : boolean;  
    isLightsOn() : boolean;  
    isLightsOff() : boolean;  
    startEngine() : void;  
    acelerate() : number;  
    stopEngine() : void;  
    startRadio() : void;  
    playCd : void;  
    stopRadio() : void;  
}
```



This example is included in the companion source code.

If a class has a dependency (client) in the `VehicleInterface` interface but it only wants to use the radio methods, we will be facing a violation of the ISP because, as we have already seen, no client should be forced to depend on methods it does not use.

The solution is to split the `VehicleInterface` interface into many client-specific interfaces so that our class can adhere to the ISP by depending only on the `RadioInterface` interface:

```
interface VehicleInterface {
    getSpeed() : number;
    getVehicleType: string;
    isTaxPayed() : boolean;
    isLightsOn() : boolean;
}

interface LightsInterface {
    isLightsOn() : boolean;
    isLightsOff() : boolean;
}

interface RadioInterface {
    startRadio() : void;
    playCd : void;
    stopRadio() : void;
}

interface EngineInterface {
    startEngine() : void;
    acelerate() : number;
    stopEngine() : void;
}
```

The dependency inversion principle

The **dependency inversion (DI)** principle states, *Depend upon abstractions. Do not depend upon concretions.* In the previous section, we implemented `FavouritesController` and we were able to replace an implementation of `PersistanceServiceInterface` with another without having to perform any additional change to `FavouritesController`. This was possible because we followed the DI principle, as `FavouritesController` has a dependency upon `PersistanceServiceInterface` (abstractions) rather than `LocalStoragePersistanceService` or `CookiePersistanceService` (concretions).



Depending on your background, you may wonder if there are any **Inversion of Control (IoC)** containers available for TypeScript. We can indeed find some IoC containers available online. However, because Typescript's runtime doesn't support reflection or interfaces, they can arguably be considered pseudo IoC containers rather than real IoC containers.

If you want to learn more about inversion of control, I highly recommend the article, Inversion of Control Containers and the Dependency Injection pattern, by Martin Fowler, available at <http://martinfowler.com/articles/injection.html>.

Namespaces

TypeScript features namespaces (previously known as internal modules). Namespaces are mainly used to organize our code.

If we are working on a large application, as the code base grows we will need to introduce some kind of organization scheme to avoid naming collisions and make our code easier to follow and understand.

We can use namespaces to encapsulate interfaces, classes, and objects that are somehow related. For example, we could wrap all our application models inside an internal module named model:

```
namespace app {  
    export class UserModel {  
        // ...  
    }  
}
```

When we declare a namespace, all its entities are private by default. We can use the `export` keyword to declare what parts of our namespace we wish to make public.

We are allowed to nest a namespace inside another. Let's create a file named `models.ts` and add the following code snippet to it:

```
namespace app {  
    export namespace models {  
        export class UserModel {  
            // ...  
        }  
    }  
}
```

```
export class TalkModel {  
    // ...  
}  
}  
}
```

In the preceding example, we have declared a namespace named `app`, and inside it, we have declared a public namespace named `models`, which contains two public classes: `UserModel` and `TalkModel`. We can then call the namespace from another TypeScript file by indicating the full namespace name:

```
var user = new app.models.UserModel();  
var talk = new app.models.TalkModel();
```

If an internal module becomes too big, it can be divided into multiple files to increase its maintainability. If we take the preceding example, we could add more contents to the internal module named `app` by referencing it in another file.

Let's create a new file named `validation.ts` and add the following code to it:

```
namespace app {  
    export namespace validation {  
        export class UserValidator{  
            // ...  
        }  
  
        export class TalkValidator {  
            // ...  
        }  
    }  
}
```

Let's create a file named `main.ts` and add the following code to it:

```
var user = new app.models.UserModel();  
var talk = new app.models.TalkModel();  
var userValidator = new app.validation.UserValidator();  
var talkValidator = new app.validation.TalkValidator();
```

Even though the namespaces' `models` and `validation` are in two different files, we are able to access them from a third file.

Namespace can contain periods. For example, instead of nesting the namespaces (`validation` and `models`) inside the `app` module, we could have used periods in the `validation` and `model` internal module names:

```
namespace app.validation {
```

```
// ...
}
namespace app.models {
    // ...
}
```

The `import` keyword can be used within an internal module to provide an alias for another module:

```
import TalkValidatorAlias = app.validation.TalkValidator;
var talkValidator = new TalkValidatorAlias();
```

Once we have finished declaring our namespaces, we can decide if we want to compile each one into JavaScript or if we prefer to concatenate all the files into one single file.

We can use the `--out` flag to compile all the input files into a single JavaScript output file:

```
tsc --out output.js input.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. We can then import our files or file using an HTML `<script>` tag.

Modules

TypeScript also has the concept of external modules or just modules. The main difference between using modules (instead of namespaces) is that after declaring all our modules, we will not import them using an HTML `<script>` tag and we will be able to use a module loader instead.

A **module loader** is a tool that allows us to have better control over the module loading process. This allows us to perform tasks such as loading files asynchronously or combining multiple modules into a single highly optimized file with ease.

Using the `<script>` tag is not recommended because when a web browser finds a `<script>` tag, it downloads the file using asynchronous requests. We should attempt to load as many files as possible using asynchronous requests because doing so will significantly improve the network performance of a web application.

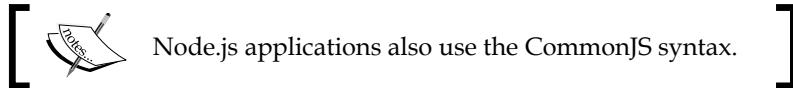


We will discover more about network performance in
Chapter 6, Application Performance.



The JavaScript versions prior to ECMAScript 6 (ES6) don't include native module support. Developers were forced to develop their own module loaders. The open source community tried to come up with improved solutions over the years. As a result, today there are several module loaders available, and each one uses a different module definition syntax. The most popular ones are as follows:

- **RequireJS**: RequireJS uses a syntax known as asynchronous module definition (AMD)
- **Browserify**: Browserify uses a syntax known as CommonJS.
- **SystemJS**: SystemJS is a universal module loader, which means that it supports all the available module syntaxes (ES6, CommonJS, AMD, and UMD).



Node.js applications also use the CommonJS syntax.

Fortunately, TypeScript allows us to choose which kind of module definition syntax (ES6, CommonJS, AMD, SystemJS, or UMD) we want to use at runtime.

We can indicate our preference by using the `--module` flag when compiling:

```
tsc --module commonjs main.ts // use CommonJS  
tsc --module amd main.ts      // use AMD  
tsc --module umd main.ts      // use UMD  
tsc --module system main.ts   // use SystemJS
```

While we can select four different module definition syntaxes at runtime. However, only two are available at design time:

- External module syntax (The default module syntax in the TypeScript versions prior to 1.5)
- ES6 module syntax (The recommended external module syntax in TypeScript 1.5 or higher)

It is important to understand that we can use one kind of module definition syntax at design time (ES6, CommonJS, AMD, SystemJS, or UMD) and another at runtime (external modules or ES6).

Since the release of TypeScript 1.5, it is recommended you use the ECMAScript 6 module definition syntax because it is based on standards, and in the future, we will be able to use this syntax at both design time and runtime.

We will now take a look at each of the available module definition syntaxes.

ES6 modules – runtime and design time

TypeScript 1.5 introduces support for the ES6 module syntax. Let's define an external module using it:

```
class UserModel {  
    // ...  
}  
export { UserModel };
```

We have defined an external module. We don't need to use the `namespace` keyword, but we must continue to use the `export` keyword. We used the `export` keyword at the bottom of the module, but it is also possible to use it just before the `class` keyword like we did in the internal module example:

```
export class UserModel {  
    // ...  
}
```

We can also export an entity using an alias:

```
class UserModel {  
    // ...  
}  
export { UserModel as User }; // UserModel exported as User
```

An `export` declaration exports all meanings of a name:

```
interface UserModel {  
    // ...  
}  
  
class UserModel {  
    // ...  
}  
export { UserModel }; // Exports both interface and function
```

To import a module, we must use the `import` keyword as follows:

```
import { UserModel } from "./models";
```

The `import` keyword creates a variable for each imported component. In the preceding code snippet, a new variable named `UserModel` is declared and its value contains a reference to the `UserModel` class, which was declared and exported in the `models.ts` file.

We can use the `export` keyword to import multiple entities from one module:

```
class UserValidator {  
    // ...  
}  
  
class TalkValidator {  
    // ...  
}  
  
export { UserValidator, TalkValidator };
```

Furthermore, we can use the `import` keyword to import multiple entities from a single module as follows:

```
import { UserValidator, TalkValidator } from "./validation.ts"
```



Throughout the rest of this book, we will use the ES6 syntax at design-time and the CommonJS syntax at runtime.

External modules – design time only

Before TypeScript 1.5, modules were declared using a kind of module syntax known as external module syntax. This kind of syntax was used at design time (TypeScript code). However, once compiled into JavaScript, it was transformed and executed (runtime) into AMD, CommonJS, UMD, or SystemJS modules.

We should try to avoid using this syntax and use the new ES6 syntax instead. However, we will take a quick look at the external module syntax because we may have to work on old applications or outdated documentation.

We can import a module using the `import` keyword:

```
import User = require("./user_class");
```

The preceding code snippet declares a new variable named `User`. The `User` variable takes the exported content of the `user_class` module as its value.

To export a module, we need to use the `export` keyword. We can apply the `export` keyword directly to a class or interface:

```
export class User {  
    // ...  
}
```

We can also use the `export` keyword on its own by assigning to it the value that we desire to export:

```
class User {  
    // ...  
}  
export = User;
```

External modules can be compiled into any of the available module definition syntaxes (AMD, CommonJS, SystemJS, or UMD).

AMD modules – runtime only

If we compile the initial external module into an AMD module (using the flag `--compile amd`), we will generate the following AMD module:

```
define(["require", "exports"], function (require, exports) {  
    var UserModel = (function () {  
        function UserModel() {  
        }  
        return UserModel;  
    })();  
    return UserModel;  
});
```

The `define` function takes an array as its first argument. This array contains a list of the names of the module dependencies. The second argument is a callback that will be invoked once all the module dependencies have been loaded. The callback takes each of the module dependencies as its parameters and contains all the logic from our TypeScript component. Notice how the return type of the callback matches the components that we declared as public by using the `export` keyword. AMD modules can then be loaded using the RequireJS module loader.



We will not discuss AMD and RequireJS further in this book, but if you want to learn more about them, you can do so by visiting <http://requirejs.org/docs/start.html>.

CommonJS modules – runtime only

We begin by compiling our external module into a CommonJS module (using the flag `--compile commonjs`). We will compile the following code snippet:

```
class User {  
    // ...  
}  
export = User;
```

As a result, the following CommonJS module is generated:

```
var UserModel = (function () {  
    function UserModel() {  
        //...  
    }  
    return UserModel;  
})();  
module.exports = UserModel;
```

As we can see in the preceding code snippet, the CommonJS module definition syntax is almost identical to the deprecated TypeScript (1.4 or prior) external module syntax.

The preceding CommonJS module can be loaded by a Node.js application without any additional changes using the `import` keyword and the `require` function:

```
import UserModel = require('./UserModel');  
var user = new UserModel();
```

However, if we attempt to use the `require` function in a web browser, an exception will be thrown because the `require` function is undefined. We can easily solve this problem by using Browserify.

All that we need to follow is three simple steps:

1. Install Browserify using npm:

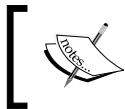
```
npm install -g browserify
```

2. Use Browserify to bundle all your CommonJS modules into a JavaScript file that you can import using an HTML `<script>` tag. We can do this by executing the following command:

```
browserify main.js -o bundle.js
```

In the preceding command, `main.js` is the file that contains the root module within our application's dependency tree. The `bundle.js` file is the output file that we will be able to import using a HTML script tag.

3. Import the bundle.js file using a HTML <script> tag.



If you need more information about Browserify, visit the official documentation at <https://github.com/substack/node-browserify#usage>.

UMD modules – runtime only

If we want to release a JavaScript library or framework, we will need to compile our TypeScript application into both CommonJS and AMD modules. Our library should also allow developers to load it directly in a web browser using a HTML script tag.

The web development community has developed the following code snippet to help us to achieve **universal module definition (UMD)** support:

```
(function (root, factory) {
  if (typeof exports === 'object') {
    // CommonJS
    module.exports = factory(require('b'));
  } else if (typeof define === 'function' && define.amd) {
    // AMD
    define(['b'], function (b) {
      return (root.returnExportsGlobal = factory(b));
    });
  } else {
    // Global Variables
    root.returnExportsGlobal = factory(root.b);
  }
}(this, function (b) {
  // Your actual module
  return {};
}));
```

This code snippet is great, but we want to avoid manually adding it to every single module in our application. Fortunately, there are a few options available to achieve UMD support with ease.

The first option is to use the flag `--compile_umd` to generate one UMD module for each module in our application. The second option is to create one single UMD module that will contain all the modules in the application using a module loader known as Browserify.



Refer to the official Browserify project website at <http://browserify.org/> to learn more about Browserify. Refer to the Browserify-standalone option to learn more about the generation of one unique optimized file.

SystemJS modules – runtime only

While UMD gives you a way to output a single module that works in both AMD and CommonJS, SystemJS will allow you to use ES6 modules closer to their native semantics without requiring an ES6-compatible browser engine.

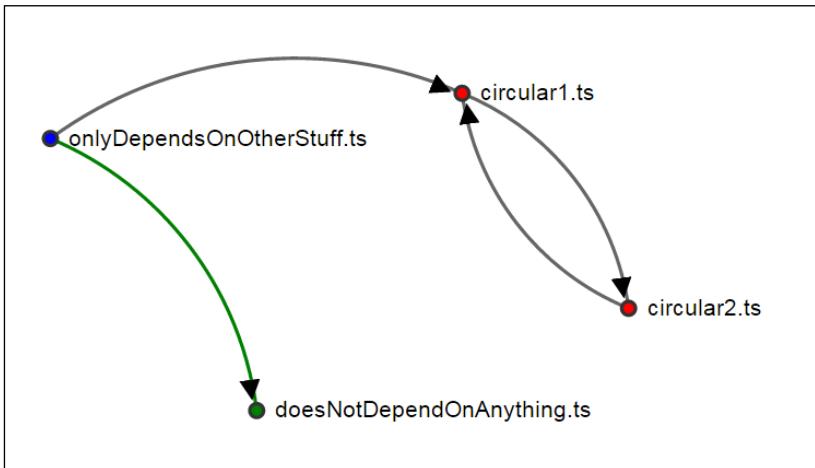
SystemJS is used by Angular 2.0, which is the upcoming version of a popular web application development framework.

Refer to the official SystemJS project website at <https://github.com/systemjs/systemjs> to learn more about SystemJS.

There is a free list of common module mistakes available online at <http://www.typescriptlang.org/Handbook#modules-pitfalls-of-modules>.

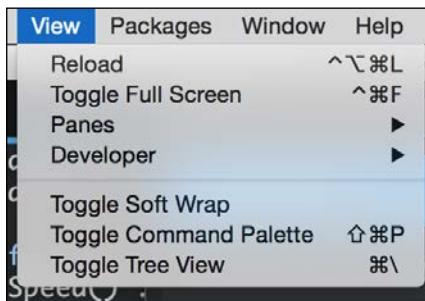
Circular dependencies

A circular dependency is an issue that we can encounter when working with multiple components and dependencies. Sometimes, it is possible to reach a point in which one component (A) has a dependency on a second component (B), which depends on the first component (A). In the following graph, each node is a component, and we can observe that the nodes `circular1.ts` and `circular2.ts` have a circular dependency. The node named `doesNotDependOnAnything.ts` doesn't have dependencies and the node named `onlyDependsOnOtherStuff.ts` has a dependency on `circular1.ts` but doesn't have circular dependencies..

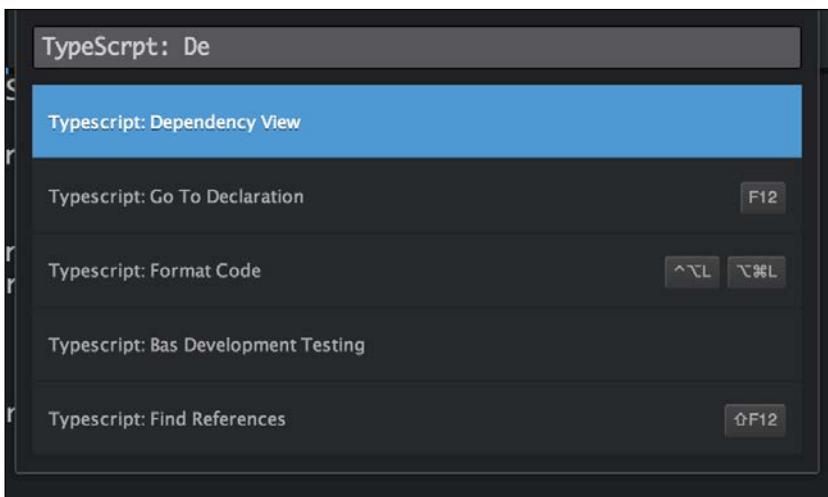


The circular dependencies don't need to necessarily involve just two components. We can encounter scenarios in which a component depends on another component, which depends on other components, and some of the components in the dependency tree end up pointing to one of their parent components in the tree.

Identifying a circular dependency is very time consuming. Fortunately, Atom includes a command-line tool that will generate a dependency tree graph for us like the preceding one. In order to access the Atom command line, we need to navigate to **View** (in the top menu) and then to **Toggle Command Palette**.



After opening the **Toggle Command Palette**, we need to type **TypeScript: Dependency View** to display the graph:



If you want to learn more about dependency graphs, you can visit its official documentation at <https://github.com/TypeStrong/atom-typescript/blob/master/docs/dependency-view.md>.

Summary

In this chapter, we saw how to work with classes, interfaces, and modules in depth. We were able to reduce the complexity of our application by using techniques such as encapsulation and inheritance.

We were also able to create external modules and manage our application dependencies using tools such as RequireJS or Browserify.

In the next chapter, we will discuss the TypeScript runtime.

5

Runtime

After completing this book, you will probably be eager to start a new project to put into practice all your new knowledge. As the new project grows and you develop more complex features, you might encounter some runtime issues.

We should be able to resolve design-time issues with ease because in the previous chapter, we looked at the main TypeScript features.

However, we have not learned much about the TypeScript runtime. The good news is that, depending on your background, you may already know a lot about it, as the TypeScript runtime is the JavaScript runtime. TypeScript is only used at design time; the TypeScript code is then compiled into JavaScript and finally executed. The JavaScript runtime is in charge of the execution. Is important to understand that we never execute TypeScript code and we always execute JavaScript code. For this reason, when we refer to the TypeScript runtime, we will, in fact, be talking about the JavaScript runtime.

When we compile our TypeScript code, we will generate JavaScript code, which will be executed on the server side (with Node.js) or on the client side (in a web browser). It is then that we may encounter some challenging runtime issues.

In this chapter, we will cover the following topics:

- The environment
- The event loop
- The `this` operator
- Prototypes
- Closures

Let's start by learning about the environment.

The environment

The runtime environment is one of the first things that we must consider before we can start developing a TypeScript application. Once we have compiled our TypeScript code, it can be executed in many different JavaScript engines. While the majority of those engines will be web browsers, such as Chrome, Internet Explorer, or Firefox, we might also want to be able to run our code on the server side or in a desktop application in environments such as Node.js or RingoJS.

It is important to keep in mind that there are some variables and objects available at runtime that are environment-specific. For example, we could create a library and access the document.layers variable. While document is part of the W3C **Document Object Model (DOM)** standard, the layers property is only available in Internet Explorer and is not part of the W3C DOM standard.

The W3C defines the DOM as follows:

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.

In a similar manner, we can also access a set of objects known as the **Browser Object Model (BOM)** from a web browser runtime environment. The BOM consists of the objects navigator, history, screen, location, and document, which are properties of the window object.

You need to realize that the DOM is part of the web browsers but not part of JavaScript. If we want to run our application in a web browser, we will be able to access the DOM and BOM. However, in environments like Node.js or RingoJS, they will not be available, since they are standalone JavaScript environments completely independent of a web browser. We can also find other objects on the server-side environments (such as process.stdin in Node.js) that will not be available if we attempt to execute our code in a web browser.

As if this wasn't enough work, we also need to keep in mind the existence of multiple versions of these JavaScript environments. We will have to support multiple browsers and multiple versions of Node.js. The recommended practice when dealing with this problem is to add logic that looks for the availability of features rather than the availability of a particular environment or version.

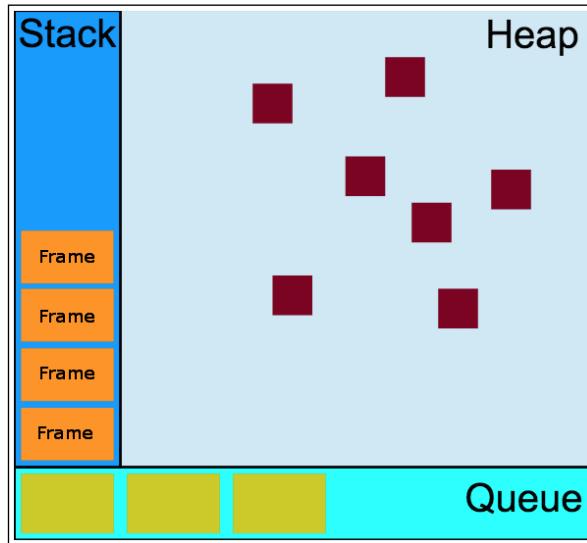


A really good library is available that can help us to implement feature detection when developing for web browsers. The library is called **Modernizr** and can be downloaded at <http://modernizr.com/>.

The runtime

The TypeScript runtime (JavaScript) has a concurrency model based on an **event loop**. This model is quite different to the models in other languages such as C or Java. Before we focus on the event loop itself, you must understand some runtime concepts.

What follows is a visual representation of some important runtime concepts: heap, stack, queue, and frame:



We will now look at the role of each of these runtime concepts.

Frames

A frame is a sequential unit of work. In the preceding diagram, the frames are represented by the blocks inside the stack.

When a function is called in JavaScript, the runtime creates a frame in the stack. The frame holds that particular function's arguments and local variables. When the function returns, the frame is popped out of the stack. Let's take a look at an example:

```
function foo(b) {  
    var a = 12;  
    return a+b+35;  
}  
  
function bar(x) {
```

```
var m = 4;  
return foo(m*x);  
}
```

After declaring the `foo` and `bar` functions, we invoke the `bar` function:

```
bar(21);
```

When `bar` is executed, the runtime will create a new frame containing the arguments of `bar` and all the local variables. The frame (represented as a square in the preceding diagram) is then added to the top of the stack.

Internally, `bar` invokes `foo`. When `foo` is invoked, a new frame is created and allocated in the top of the stack. When the execution of `foo` is finished (`foo` has returned), the top frame is removed from the stack. When the execution of `bar` is also complete, it is removed from the stack as well.

Now, let's imagine what would happen if the `foo` function invoked the `bar` function. We would create a never-ending function call loop. With each function call, a new frame would be added to the stack, and eventually, there would be no more space in the stack, and an error would be thrown. Most developers are familiar with this error, known as a stack overflow error.

Stack

The stack contains the sequential steps (frames) that a message needs to execute. A stack is a data structure that represents a simple **Last In First Out (LIFO)** collection of objects. Therefore, when a frame is added to the stack, it is always added to the top of the stack.

Since the stack is a LIFO collection, the event loop processes the frames stored in it from top to bottom. The dependencies of a frame are added to the top of it in the stack to ensure that all the dependencies of each of the frames are met.

Queue

The queue contains a list of messages waiting to be processed. Each message is associated with a function. When the stack is empty, a message is taken out of the queue and processed. The processing consists of calling the associated function and adding the frames to the stack. The message processing ends when the stack becomes empty again.

In the previous runtime diagram, the blocks inside the queue represent the messages.

Heap

The heap is a memory container that is not aware of the order of the items stored in it. The heap contains all the variables and objects currently in use. It may also contain frames that are currently out of scope but have not yet been removed from memory by the garbage collector.

The event loop

Concurrency is the ability for two or more operations to be executed simultaneously. The runtime execution takes place on one single thread, which means that we cannot achieve real concurrency.

The event loop follows a run-to-completion approach, which means that it will process a message from beginning to end before any other message is processed.



As we discussed in *Chapter 3, Working with Functions*, we can use the `yield` keyword and generators to pause the execution of a function.

Every time a function is invoked, a new message is added to the queue. If the stack is empty, the function is processed (the frames are added to the stack).

When all the frames have been added to the stack, the stack is cleared from top to bottom. At the end of the process, the stack is empty and the next message is processed.



Web workers can perform background tasks in a different thread. They have their own queue, heap, and stack.

One of the advantages of the event loop is that the execution order is quite predictable and easy to follow. Another important advantage of the event loop approach is that it features non-blocking I/O. This means that when the application is waiting for an input and output (I/O) operation to finish, it can still process other things, such as user input.

A disadvantage of this approach is that if a message takes too long to complete, the application becomes unresponsive. Good practice is to make message processing short, and if possible, split one message function into several messages functions.

The `this` operator

In JavaScript, the `this` operator behaves a little differently than other languages. The value of the `this` operator is often determined by the way a function is invoked. Its value cannot be set by assignment during execution, and it may be different each time a function is invoked.



The `this` operator also has some differences when using the strict and nonstrict modes. To learn more about the strict mode, refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.

The `this` operator in the global context

In the global context, the `this` operator will always point to the global object. In a web browser, the `window` object is the global object:

```
console.log(this === window); // true
this.a = 37;
console.log(window.a); // 37
console.log(this.document === document === window.document); // true
```

The `this` operator in a function context

The value of `this` inside a function depends on how the function is invoked. If we simply invoke a function in the nonstrict mode, the value of `this` within the function will point to the global object:

```
function f1(){
  return this;
}
f1() === window; // true
```

However, if we invoke a function in the strict mode, the value of `this` within the function's body will point to `undefined`:

```
console.log(this); // global (window)

function f2(){
  "use strict";
  return this; // undefined
}
console.log(f2()); // undefined
console.log(this); // window
```

However, the value of the `this` operator inside a function invoked as an instance method points to the instance. In other words, the value of the `this` operator within a function that is part of a class points to that class:

```
var p = {
  age: 37,
  getAge: function() {
    return this.age; // this points to the class instance (p)
  }
};
console.log(p.getAge()); // 37
```

In the preceding example, we have used object literal notation to define an object named `p`, but the same applies when declaring objects using prototypes:

```
function Person() {}
Person.prototype.age = 37;
Person.prototype.getAge = function () {
  return this.age;
}
var p = new Person();
p.age; // 37
p.getAge(); // 37
```

When a function is used as a constructor (with the `new` keyword), the `this` operator points to the object being constructed:

```
function Person() { // function used as a constructor
  this.age = 37;
}
var p = new Person();
console.log(p.age); // logs 37
```

The `call`, `apply`, and `bind` methods

All the functions inherit the `call`, `apply`, and `bind` methods from `Function.prototype`. We can use these methods to set the value of the `this` operator when it is used inside the body of a function.

The `call` and `apply` methods are almost identical; both methods allow us to invoke a function and set the value of the `this` operator within the function. The main difference between `call` and `apply` is that while `apply` lets us invoke the function with arguments as an array and `call` requires the function parameters to be listed explicitly.



A useful mnemonic is *A* (*apply*) for *array* and *C* (*call*) for *comma*.

Let's take a look at an example. We will start by declaring a class named `Person`. This class has two properties (`name` and `surname`) and one method (`greet`). The `greet` method uses the `this` operator to access the `name` and `surname` instance properties:

```
class Person {  
    public name : string;  
    public surname : string;  
  
    constructor(name : string, surname : string) {  
        this.name = name;  
        this.surname = surname;  
    }  
  
    public greet(city : string, country : string) {  
        // we use the this operator to access name and surname  
        var msg = `Hi, my name is ${this.name} ${this.surname}. `;  
        msg += `I'm from ${city} (${country}).`;  
        console.log(msg);  
    }  
}
```

After declaring the `Person` class, we will create an instance:

```
var person = new Person("remo", "jansen");
```

If we invoke the `greet` method, it will work as expected:

```
person.greet("Seville", "Spain");  
// Hi, my name is remo jansen. I'm from Seville (Spain).
```

Alternatively, we can invoke the method using the `call` and `apply` functions. We have supplied the `person` object as the first parameter of both functions because we want the `this` operator (inside the `greet` method) to take `person` as its value:

```
person.greet.call(person, "seville", "spain");  
person.greet.apply(person, ["seville", "spain"]);
```

If we provide a different value to be used as the value of `this`, we will not be able to access the `name` and `surname` properties within the `greet` function:

```
person.greet.call(null, "seville", "spain");  
person.greet.apply(null, ["seville", "spain"]);  
// Hi, my name is undefined. I'm from seville spain.
```

The two preceding examples may seem useless because the first one invoked the function directly and the second one caused an unexpected behavior. The `apply` and `call` methods make sense only when we want the `this` operator to take a different value when a function is invoked:

```
var valueOfThis = { name : "anakin", surname : "skywalker" };  
person.greet.call(valueOfThis, "mos espa", "tatooine");  
person.greet.apply(valueOfThis, ["mos espa", "tatooine"]);  
// Hi, my name is anakin skywalker. I'm from mos espa tatooine.
```

The `bind` method can be used to set the value of the `this` operator (within a function) regardless of how it is invoked.

When we invoke a function's `bind` method, it returns a new function with the same body and scope as the original function, but the `this` operator (within the body function) is permanently bound to the first argument of `bind`, regardless of how the function is invoked.

Let's take a look at an example. We will start by creating an instance of the `Person` class that we declared in the previous example:

```
var person = new Person("remo", "jansen");
```

Then, we can use `bind` to set the `greet` function to be a new function with the same scope and body:

```
var greet = person.greet.bind(person);
```

If we try to invoke the `greet` function using `bind` and `apply`, just like we did in the previous example, we will be able to observe that this time the `this` operator will always point to the object instance independent of how the function is invoked:

```
greet.call(person, "seville", "spain");  
greet.apply(person, ["seville", "spain"]);  
// Hi, my name is remo jansen. I'm from seville spain.
```

```
greet.call(null, "seville", "spain");  
greet.apply(null, ["seville", "spain"]);  
// Hi, my name is remo jansen. I'm from seville spain.
```

```
var valueOfThis = { name: "anakin", surname: "skywalker" };  
greet.call(valueOfThis, "mos espa", "tatooine");  
greet.apply(valueOfThis, ["mos espa", "tatooine"]);  
// Hi, my name is remo jansen. I'm from mos espa tatooine.
```



Using the `apply`, `call`, and `bind` functions is not recommended unless you really know what you are doing, because they can lead to complex runtime issues for other developers.

Once we bind an object to a function with `bind`, we cannot override it:

```
var valueOfThis = { name: "anakin", surname: "skywalker" };
var greet = person.greet.bind(valueOfThis);
greet.call(valueOfThis, "mos espa", "tatooine");
greet.apply(valueOfThis, ["mos espa", "tatooine"]);
// Hi, my name is remo jansen. I'm from mos espa tatooine.
```



The use of the `bind`, `apply`, and `call` methods is often discouraged because it can lead to confusion. Modifying the default behavior of the `this` operator can lead to really unexpected results. Remember to use these methods only when strictly necessary and to document your code properly to reduce the risk caused by potential maintainability issues.

Prototypes

When we compile a TypeScript program, all classes and objects become JavaScript objects. Sometimes, we will encounter our application behaving unexpectedly at runtime, and we will not be able to identify and understand the root cause of this behavior without a good understanding of how inheritance works in JavaScript. This understanding will allow us to have much better control over our application at runtime.

The runtime inheritance system uses a prototypal inheritance model. In a prototypal inheritance model, objects inherit from objects, and there are no classes available. However, we can use prototypes to simulate classes. Let's see how it works.

At runtime, almost every JavaScript object has an internal property called `prototype`. The value of the `prototype` attribute is an object, which contains some attributes (data) and methods (behavior).

In TypeScript, we can use a class-based inheritance system:

```
class Person {
  public name : string;
  public surname : string;
  public age : number = 0;
  constructor(name : string, surname : string) {
    this.name = name;
```

```

        this.surname = surname;
    }
    greet() {
        var msg = `Hi! my name is ${this.name} ${this.surname}`;
        msg += `I'm ${this.age}`;
    }
}

```

We have defined a class named `Person`. At runtime, this class is declared using prototypes instead of classes:

```

var Person = (function () {
    function Person(name, surname) {
        this.age = 0;
        this.name = name;
        this.surname = surname;
    }
    Person.prototype.greet = function () {
        var msg = "Hi! my name is " + this.name +
            " " + this.surname;
        msg += "I'm " + this.age;
    };
    return Person;
})();

```

The TypeScript compiler wraps the object definition (we will not refer it as the class definition because technically, it is not a class) with an **immediately invoked function expression (IIFE)**. Inside the IIFE, we can find a function named `Person`. If we examine the function and compare it to the TypeScript class, we will notice that it takes the same parameters, like the constructor in the TypeScript class. This function is used to create new instances of the `Person` class.

After the constructor, we can see the definition of the `greet` method. As you can see, the `prototype` attribute is used to attach the `greet` method to the `Person` class.

Instance properties versus class properties

As JavaScript is a dynamic programming language, we can add properties and methods to an instance of an object at runtime; and they don't need to be part of the object (class) itself. Let's take a look at an example:

```

function Person(name, surname) {
    // instance properties
    this.name = name;
}

```

```
    this.surname = surname;  
}  
var me = new Person("remo", "jansen");  
me.email = "remo.jansen@wolksoftware.com";
```

Here, we defined a constructor function for an object named `Person`, which takes two variables (`name` and `surname`) as arguments. Then, we have created an instance of the `Person` object and added a new property named `email` to it. We can use a `for...in` statement to check the properties of `me` at runtime:

```
for(var property in me) {  
    console.log("property: " + property + ", value: '" +  
    me[property] + "'");  
}  
// property: name, value: 'remo'  
// property: surname, value: 'jansen'  
// property: email, value: 'remo.jansen@wolksoftware.com'  
// property: greet, value: 'function (city, country) {  
//     var msg = "Hi, my name is " + this.name + " " +  
// this.surname;  
//     msg += "\nI'm from " + city + " " + country;  
//     console.log(msg);  
// }'
```

All these properties are **instance properties** because they hold a value for each new instance. If, for example, we create a new instance of `Person`, both instances will hold their own values:

```
var hero = new Person("John", "117");  
hero.name; // "John"  
me.name; // "remo"
```

We have defined these instance properties using the `this` operator, because in the class constructor, the `this` operator points to the object's prototype. This explains why we can alternatively define instance properties through the object's prototype:

```
Person.prototype.name = name; // instance property  
Person.prototype.surname = surname; // instance property
```

We can also declare class properties and methods. The main difference is that the value of class properties and methods is shared between all the instances of an object. Class properties and methods are sometimes called static properties and methods.

Class properties are often used to store static values:

```
function MathHelper() {  
    /* ... */  
}  
  
// class property  
MathHelper.PI = 3.14159265359;
```

Class methods are also often used as utility functions that perform calculations upon supplied parameters and return a result:

```
function MathHelper() { /* ... */ }  
  
// class method  
MathHelper.areaOfCircle = function(radius) {  
    return radius * radius * this.PI;  
}  
  
// class property  
MathHelper.PI = 3.14159265359;
```

In the preceding example, we have accessed a class attribute (`PI`) from a class method (`areaOfCircle`). We can access class properties from instance methods, but we cannot access instance properties or methods from class properties or methods. We can demonstrate this by declaring `PI` as an instance property instead of a class property:

```
function MathHelper() {  
    // instance property  
    this.PI = 3.14159265359;  
}
```

If we then attempt to access `PI` from a class method, it will be undefined:

```
// class method  
MathHelper.areaOfCircle = function(radius) {  
    return radius * radius * this.PI; // this.PI is undefined  
}  
  
MathHelper.areaOfCircle(5); // NaN
```

We are not supposed to access class methods or properties from instance methods, but there is a way to do it. We can achieve it using the prototype's constructor property. We can also demonstrate this as follows:

```
function MathHelper () { /* ... */ }

// class property
MathHelper.PI = 3.14159265359;

// instance method
MathHelper.prototype.areaOfCircle = function(radius) {
    return radius * radius * this.constructor.PI;
}

var math = new MathHelper ();
console.log(MathHelper.areaOfCircle(5)); // 78.53981633975
```

We can access PI (the class property) from areaOfCircle (the instance method) using the prototype's constructor property because this property returns a reference to the object's constructor.

Inside areaOfCircle, the this operator returns a reference to the object's prototype:

```
this === MathHelper.prototype //true
```

We may deduce that this.constructor is equal to MathHelper.prototype.constructor and, therefore, MathHelper.prototype.constructor is equal to MathHelper.

Prototypal inheritance

You might be wondering how the extends keyword works. Let's create a new TypeScript class, which inherits from the Person class, to help you understand it:

```
class SuperHero extends Person {
    public superpower : string;
    constructor(name : string, surname : string, superpower : string) {
        super(name, surname);
        this.superpower = superpower;
    }
    userSuperPower() {
        return `I'm using my ${this.superpower}`;
    }
}
```

The preceding class is named `SuperHero` and extends the `Person` class. It has one extra attribute (`superpower`) and method (`useSuperPower`). If we compile the code, we will notice the following piece of code:

```
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 __();
};
```

This piece of code is generated by TypeScript. Even though it is a really small piece of code, it showcases almost every concept contained in this chapter, and understanding it can be quite challenging. We might need to examine it multiple times to understand it, but the effort is worth it. Let's take a look at the function.

Before the function expression is evaluated for the first time, the `this` operator points to the global object, which does not contain a method named `__extends`. This means that the `__extends` variable is undefined at this point:

```
console.log(this.__extends); // undefined
```

When the function expression is evaluated for the first time, the value of the function expression (an anonymous function) is assigned to the `__extends` property in the global scope:

```
console.log(this.__extends); // extends(n, e, t);
```

TypeScript generates the function expression once for each TypeScript file containing the `extends` keyword. However, the function expression is only evaluated once (when the `__extends` variable is undefined). This behavior is implemented in the first line of code:

```
var __extends = this.__extends || function (d, b) { // ...
```

The first time this line of code is executed, the function expression is evaluated. The value of the function expression is an anonymous function, which is assigned to the `__extends` variable in the global scope. As we are in the global scope, `var __extends` and `this.__extends` refer to the same variable at this point.

When a new file is executed, the `__extends` variable is already available in the global scope and the function expression is not evaluated. This means that the value of the function expression is only assigned to the `__extends` variable once.

As you already know, the value of the function expression is an anonymous function. Let's now focus on it:

```
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 __();  
}
```

This function takes two arguments named `d` and `b`. When we invoke it, we should pass a derived object constructor (`d`) and a base object constructor (`b`).

The first line inside the anonymous function iterates each class property and method from the base class and creates their copy in the derived class:

```
for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
```

 When we use a `for...in` statement to iterate an instance of an object, it will iterate the object's instance properties. However, if we use a `for...in` statement to iterate the properties of an object's constructor, the statement will iterate its class properties. In the preceding example, the `for...in` statement is used to inherit the object's class properties and methods. To inherit the instance properties, we will copy the object's prototype.

The second line declares a new constructor function named `__`, and inside it, the `this` operator is used to access its prototype:

```
function __() { this.constructor = d; }
```

The prototype contains a special property named `constructor`, which returns a reference to the object's constructor. The function named `__` and `this.constructor` are pointing to the same variable at this point. The value of the derived object constructor (`d`) is then assigned to the `__` constructor.

In the third line, the value of the prototype object from the base object constructor is assigned to the prototype of the `__` object constructor:

```
__.prototype = b.prototype;
```

In the last line, a new `__()` is invoked, and the result is assigned to the derived class (`d`) prototype. By performing all these steps, we have achieved all that we need to invoke the following:

```
var instance = new d();
```

Upon doing so, we will get an object that contains all the properties from both the derived class (d) and the base class (b). Furthermore, the instance of operator will work as we would expect:

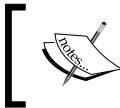
```
var superHero = new SuperHero();
console.log(superHero instanceof Person);      // true
console.log(superHero instanceof SuperHero); // true
```

We can see the function in action by examining the runtime code that defines the `SuperHero` class:

```
var SuperHero = (function (_super) {
  __extends(SuperHero, _super);
  function SuperHero(name, surname, superpower) {
    _super.call(this, name, surname);
    this.superpower = superpower;
  }
  SuperHero.prototype.userSuperPower = function () {
    return "I'm using my " + superpower;
  };
  return SuperHero;
}) (Person);
```

We can see an IIFE here again. This time, the IIFE takes the `Person` object constructor as the argument. Inside the function, we will refer to this argument using the name `_super`. Inside the IIFE, the `__extends` function is invoked and the `SuperHero` (derived class) and `_super` (base class) arguments are passed to it.

In the next line, we can find the declaration of the `SuperHero` object constructor and the `useSuperPower` function. We can use `SuperHero` as an argument of `__extend` before it is declared, because functions declarations are hoisted to the top of the scope.



Function expressions are not hoisted. When we assign a function to a variable in a function expression, the variable is hoisted, but its value (the function itself) is not hoisted.

Inside the `SuperHero` constructor, the base class (`Person`) constructor is invoked using the `call` method:

```
_super.call(this, name, surname);
```

As we discussed previously in this chapter, we can use `call` to set the value of the `this` operator in a function context. In this case, we are passing the `this` operator, which points to the instance of `SuperHero` being created:

```
function Person(name, surname) {  
    // this points to the instance of SuperHero being created  
    this.name = name;  
    this.surname = surname;  
}
```

The prototype chain

When we try to access a property or method of an object, the runtime will search for that property or method in the object's own properties and methods. If it is not found, the runtime will continue searching through the object's inherited properties by navigating the entire inheritance tree. As a derived object is linked to its base object through the `prototype` property, we refer to this inheritance tree as the `prototype` chain.

Let's take a look at an example. We will declare two simple TypeScript classes named `Base` and `Derived`:

```
class Base {  
    public method1(){ return 1; };  
    public method2(){ return 2; };  
}  
  
class Derived extends Base {  
    public method2(){ return 3; };  
    public method3(){ return 4; };  
}
```

Now, we will examine the JavaScript code generated by TypeScript:

```
var Base = (function () {  
    function Base() {  
    }  
    Base.prototype.method1 = function () { return 1; };  
    ;  
    Base.prototype.method2 = function () { return 2; };  
    ;  
    return Base;  
})();
```

```
var Derived = (function (_super) {
    __extends(Derived, _super);
    function Derived() {
        _super.apply(this, arguments);
    }
    Derived.prototype.method2 = function () { return 3; };
    ;
    Derived.prototype.method3 = function () { return 4; };
    ;
    return Derived;
}) (Base);
```

We can then create an instance of the `Derived` class:

```
var derived = new Derived();
```

If we try to access the method named `method1`, the runtime will find it in the instance's own properties:

```
console.log(derived.method1()); // 1
```

The instance also has its own property named `method2` (with value 2), but there is also an inherited property named `method2` (with value 3). The object's own property (`method2` with value 3) prevents access to the prototype property (`method2` with value 2). This is known as **property shadowing**:

```
console.log(derived.method2()); // 3
```

The instance does not have its own property named `method3`, but it has a property named `method3` in its prototype:

```
console.log(derived.method3()); // 4
```

Both the instance and the objects in the prototype chain (the `Base` class) don't have a property named `method4`:

```
console.log(derived.method4()); // error
```

Accessing the prototype of an object

Prototypes can be accessed in three different ways:

- `Person.prototype`: We can access the prototype of a function directly using the `prototype` attribute
- `Person.getPrototypeOf(person)`: We want this function to access the prototype of an instance of an object we can use the `getPrototypeOf` function

- `person.__proto__`: This is a property that exposes the internal prototype of the object through which it is accessed



The use of `__proto__` is controversial and has been discouraged by many. It was never originally included in the ECMAScript language spec, but modern browsers decided to implement it anyway. Today, the `__proto__` property has been standardized in the ECMAScript 6 language specification and will be supported in the future, but it is still a slow operation that should be avoided if performance is a concern.

The new operator

We can use the `new` operator to generate an instance of `Person`:

```
var person = new Person("remo", "jansen");
```

The runtime does not follow a class-based inheritance model. When we use the `new` operator, the runtime creates a new object that inherits from the `Person` class prototype.

We may conclude that the behavior of the `new` operator at runtime (JavaScript) is not really different from the `extends` keyword at design time (TypeScript).

Closures

Closures are one of the most powerful features available at runtime, but they are also one of the most misunderstood. The Mozilla developer network defines closures as follows:

"Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure 'remembers' the environment in which it was created."

We understand **independent (free) variables** as variables that persist beyond the lexical scope from which they were created. Let's take a look at an example:

```
function makeArmy() {  
    var shooters = []  
    for(var i = 0; i < 10; i++) {  
        var shooter = function() { // a shooter is a function  
            alert(i) // which should alert it's number  
        }  
    }  
}
```

```
        shooters.push(shooter)
    }
    return shooters;
}
```

We have declared a function named `makeArmy`. Inside the function, we have created an array of functions named `shooters`. Each function in the `shooters` array will alert a number, the value of which was set from the variable `i` inside a `for` statement. We will now invoke the `makeArmy` function:

```
var army = makeArmy();
```

The `army` variable should now contain the array of functions `shooters`. However, we will notice a problem if we execute the following piece of code:

```
army[0](); // 10 (expected 0)
army[5](); // 10 (expected 5)
```

The preceding code snippet does not work as expected because we made one of the most common mistakes related to closures. When we declared the `shooter` function inside the `makeArmy` function, we created a closure without knowing it.

The reason for this is that the functions assigned to `shooter` are closures; they consist of the function definition and the captured environment from the `makeArmy` function's scope. Ten closures have been created, but each one shares the same single environment. By the time the `shooter` functions are executed, the loop has run its course and the `i` variable (shared by all the closures) has been left pointing to the last entry (10).

One solution in this case is to use more closures:

```
function makeArmy() {
    var shooters = []
    for(var i = 0; i < 10; i++) {
        (function(i){
            var shooter = function() {
                alert(i);
            }
            shooters.push(shooter)
        })(i);
    }
    return shooters;
}

var army = makeArmy();
army[0](); // 0
army[5](); // 5
```

This works as expected. Rather than the shooter functions sharing a single environment, the immediately invoked function creates a new environment for each one, in which `i` refers to the corresponding value.

Static variables with closures

In the previous section, we saw that when a variable is declared in a closure context it can be shared between multiple instances of a class, or in other words, the variable behaves as a static variable.

We will now see how we can create variables and methods that behave like static variables. Let's start by declaring a TypeScript class named `Counter`:

```
class Counter {
    private static _COUNTER = 0;
    constructor() {}
    private _changeBy(val) {
        Counter._COUNTER += val;
    }
    public increment() {
        this._changeBy(1);
    }
    public decrement() {
        this._changeBy(-1);
    }
    public value() {
        return Counter._COUNTER;
    }
}
```

The preceding class contains a static member named `_COUNTER`. The TypeScript compiler transforms it into the following resulting code:

```
var Counter = (function () {
    function Counter() {}
    Counter.prototype._changeBy = function (val) {
        Counter._COUNTER += val;
    };
    Counter.prototype.increment = function () {
        this._changeBy(1);
    };
    Counter.prototype.decrement = function () {
        this._changeBy(-1);
    };
})
```

```
Counter.prototype.value = function () {
    return Counter._COUNTER;
};

Counter._COUNTER = 0;
return Counter;
})();
```

As you can observe, the static variable is declared by the TypeScript compiler as a class property (as opposed to an instance property). The compiler uses a class property because class properties are shared across all instances of a class.

Alternatively, we could write some JavaScript (remember that all valid JavaScript is valid TypeScript) code to emulate static properties using closures:

```
var Counter = (function() {
    // closure context
    var _COUNTER = 0;

    function changeBy(val) {
        _COUNTER += val;
    }

    function Counter() {};

    Counter.prototype.increment = function() {
        changeBy(1);
    };
    Counter.prototype.decrement = function() {
        changeBy(-1);
    };
    Counter.prototype.value = function() {
        return _COUNTER;
    };
    return Counter;
})();
```

The preceding code snippet declares a class named `Counter`. The class has some methods used to increment, decrement, and read the variable named `_COUNTER`. The `_COUNTER` variable itself is not part of the object prototype.

The `Counter` constructor function is part of a closure. As a result, all the instances of the `Counter` class will share the same closure context, which means that the context (the variable `counter` and the function `changeBy`) will behave as a singleton.



The singleton pattern requires an object to be declared as a static variable to avoid the need to create its instance whenever it is required. The object instance is, therefore, shared by all the components in the application. The singleton pattern is frequently used in scenarios where it is not beneficial, which introduces unnecessary restrictions in situations where a unique instance of a class is not actually required, and introduces global states into an application.

So, you now know that it is possible to use closures to emulate static variables:

```
var counter1 = new Counter();
var counter2 = new Counter();
console.log(counter1.value()); // 0
console.log(counter2.value()); // 0
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2
console.log(counter2.value()); // 2 (expected 0)
counter1.decrement();
console.log(counter1.value()); // 1
console.log(counter2.value()); // 1 (expected 0)
```

Private members with closures

We have seen that the closure function can access variables that persist beyond the lexical scope from which they were created. These variables are not part of the function prototype or body, but they are part of the closure function context.

As there is no way to directly access the context of a closure function, the context variables and methods can be used to emulate private members. The main advantage of using closures to emulate private members (instead of the TypeScript private access modifier) is that closures will prevent access to private members at runtime.

TypeScript avoids emulating private properties at runtime. The TypeScript compiler will throw an error at compilation time if we attempt to access a private member.

However, TypeScript avoids the use of closures to emulate private members to improve the application performance. If we add or remove an access modifier to or from one of our classes, the resulting JavaScript code will not change at all. This means that private members of a class become public members at runtime.

However, it is possible to use closures to emulate private properties at runtime. Just like when we emulated a static variable using closures, we can only achieve this kind of advanced control over the behavior of closures by writing pure JavaScript. Let's take a look at an example:

```
function makeCounter() {  
  
    // closure context  
    var _COUNTER = 0;  
    function changeBy(val) {  
        _COUNTER += val;  
    }  
  
    function Counter() {};  
  
    Counter.prototype.increment = function() {  
        changeBy(1);  
    };  
    Counter.prototype.decrement = function() {  
        changeBy(-1);  
    };  
    Counter.prototype.value = function() {  
        return _COUNTER;  
    };  
    return new Counter();  
};
```

The preceding class is almost identical to the class that we previously declared to demonstrate how to emulate static variables at runtime using closures.

This time, a new closure context is created every time we invoke the `makeCounter` function, so each new instance of `Counter` will remember an independent context (`counter` and `changeBy`):

```
var counter1 = makeCounter();  
var counter2 = makeCounter();  
console.log(counter1.value()); // 0  
console.log(counter2.value()); // 0  
counter1.increment();  
counter1.increment();  
console.log(counter1.value()); // 2  
console.log(counter2.value()); // 0 (expected 0)  
counter1.decrement();  
console.log(counter1.value()); // 1  
console.log(counter2.value()); // 0 (expected 0)
```

Since the context cannot be accessed directly, we can say that the variable `counter` and the `changeBy` function are private members:

```
console.log(counter1.counter); // undefined
counter1.changeBy(2); // changeBy is not a function
console.log(counter1.value()); // 1
```

Summary

In this chapter, we discovered how to understand the runtime, which allows us not only to resolve runtime issues with ease but also to be able to write better TypeScript code. A deep understanding of closures and prototypes will allow you to develop some complex features that it would have not been possible to develop without this knowledge.

In the next chapter, we will focus on performance, memory management, and exception handling.

6

Application Performance

In this chapter, we will take a look at how can we manage available resources in an efficient manner to achieve great performance. You will understand the different types of resource, performance factors, performance profiling and automation.

The chapter begins by introducing some core performance concepts, such as latency or bandwidth, and continues by showcasing how to measure and monitor performance as part of the automated build process.

As we discussed in previous chapters, we can use TypeScript to generate JavaScript code that can be executed in many different environments (web browsers, Node.js, mobile devices, and so on). In this chapter, we will explore performance optimization, which is mainly applicable to the development of web applications. The following topics will be covered in this chapter:

- Performance and resources
- Aspects of performance
- Memory profiling
- Network Profiling
- CPU and GPU profiling
- Performance testing
- Performance recommendations
- Performance automation

Prerequisites

Before we get started, we need to install Google Chrome because we will use its developer tools to perform web performance analysis.

Performance and resources

Before we get our hands dirty doing some performance analysis, monitoring, and automation, we must first spend some time understanding some core concepts and aspects about performance.

A good application is one that has a set of desirable characteristics, which includes functionality, reliability, usability, reusability, efficiency, maintainability, and portability. Over the course of this book so far, we have understood a lot about maintainability and reusability. In this chapter, we will focus on performance, which is closely related to reliability and maintainability.

The term performance refers to the amount of useful work accomplished compared to the time and resources used. A resource is a physical (CPU, RAM, GPU, HDD, and so on) or virtual (CPU times, RAM regions, files, and so on) component with limited availability. As the availability of a resource is limited, each resource is shared between processes. When a process finishes using a resource, it must release the resource before any other process can use it. Managing available resources in an efficient manner will help to reduce the time other processes spend waiting for the resources to become available.

When we work on a web application, we need to keep in mind that the following resources will have limited availability:

- **Central Processing Unit (CPU):** This carries out the instructions of a computer program by performing the basic arithmetic, logical, control, and input/output (I/O) operations specified by the instructions.
- **Graphics Processor Unit (GPU):** This is a specialized processor used in the manipulation and alteration of memory to accelerate the creation of images in a frame buffer intended for output to a display. The GPU is used when we create applications that use the WebGL API or when we use some CSS3 animations.
- **Random Access Memory (RAM):** This allows data items to be read and written in approximately the same amount of time regardless of the order in which data items are accessed. When we declare a variable, it will be stored in RAM memory; when the variable is out of the scope, it will be removed from RAM by the garbage collector.

- **Hard Disk Drive (HDD) and Solid State Drive (SSD):** Both of these are data storage devices used to store and retrieve information. When developing client-side web applications, we will not have to worry about these resources really often because these applications don't usually extensively use persistent data storage. However, we should keep in mind that, whenever we store an object in a persistent manner (cookies, local storage, IndexedDB, and so on), the performance of our application will be affected by the availability of the HDD or SSD.
- **Network throughput:** This determines how much actual data can be sent per unit of time across a network. The network throughput is determined by factors such as the network latency or bandwidth (we will discuss more about these factors later in this chapter).

All the resources presented in the preceding list are also limited when working on a Node.js application or a hybrid application. However, it is not really common to extensively use the GPU while working on a Node.js application, but it is a possible scenario.

Performance metrics

As performance is influenced by the availability of multiple types of physical and virtual device, we can find a few different performance metrics (factors to measure performance). Some popular performance metrics include availability, response time, processing speed, latency, bandwidth, and scalability. These measurement mechanisms are usually directly related to one of the general resources (CPU, network throughput, and so on) that were mentioned in the previous section. We will now look at each of these performance metrics in detail.

Availability

The availability of a system is related to its performance, because if the system is not available at some stage, we will perceive it as bad performance. The availability can be improved by improving the reliability, maintainability, and testability of the system. If the system is easy to test and maintain, it will be easy to increase its reliability.

The response time

The response time is the amount of time that it takes to respond to a request for a service. A service here does not refer to a web service; a service can be any unit of work. The response time can be divided into three parts:

- **Wait time:** This is the amount of time that the requests will spend waiting for other requests that took place earlier to be completed.
- **Service time:** This is the amount of time that it takes for the service (unit of work) to be completed.
- **Transmission time:** Once the unit of work has been completed, the response will be sent back to the requestor. The time that it takes for the response to be transmitted is known as the transmission time.

Processing speed

Processing speed (also known as clock rate) refers to the frequency at which a processing unit (CPU or GPU) runs. An application contains many units of work. Each unit of work is composed of instructions for the processor; usually, the processors can perform an instruction in each clock tick. Since a few clock ticks are required for an operation to be completed, the higher the clock rate (processing speed), the more instructions will be completed.

Latency

Latency is a term we can apply to many elements in a system; but when working on web applications, we will use this term to refer to network latency. Network latency indicates any kind of delay that occurs in data communication over the network.

High latency creates bottlenecks in the communication bandwidth. The impact of latency on network bandwidth can be temporary or persistent, based on the root cause of the delays. High latency can be caused by problems in the medium (cables or wireless signals), problems with routers and gateways, and anti-virus, among other things.

Bandwidth

Just like in the case of latency, whenever we mention bandwidth in this chapter, we will be referring to the network bandwidth. The bandwidth, or data transfer rate, is the amount of data that can be carried from one point to another in a given time. The network bandwidth is usually expressed in bits per second.



Network performance can be affected by many factors. Some of these factors can degrade the network throughput. For example, a high packet loss, latency, and jitter will reduce the network throughput, while a high bandwidth will increase it.

Scalability

Scalability is the ability of a system to handle a growing amount of work. A system with good scalability will be able to pass some performance tests, such as spike or stress testing.

We will discover more about performance tests (such as spike and stress) later in this chapter.

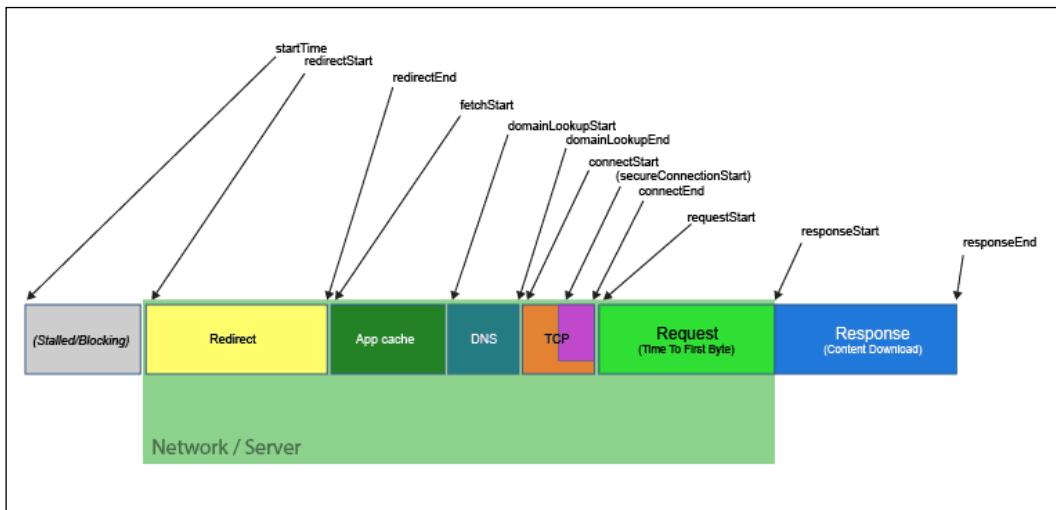
Performance analysis

Performance analysis (also known as performance profiling) is the observation and study of resource usage by an application. We will perform profiling in order to identify performance issues in our applications. A different performance profiling process will be carried out for each type of resource using specific tools. We will now take a look at how we can use Google Chrome's developer tools to perform network profiling.

Network performance analysis

We are going to start by analyzing network performance. Not so long ago, in order to be able to analyze the network performance of an application, we would have had to write a small network logging application ourselves. Today, things are much easier thanks to the arrival of the performance timing API (<http://www.w3.org/TR/resource-timing/>). The performance timing API allows us to access detailed network timing data for each loaded resource.

The following diagram illustrates the network timing data points that the API provides:



We can access the performance timing API via the global object:

```
window.performance
```

The `performance` attribute in the global object has some properties (`memory`, `navigation`, and `timing`) and methods (`clearMarks`, `clearMeasures`, and `getEntries`). We can use the `getEntries` function to get an array that contains the timing data points of each request:

```
window.performance.getEntries()
```

Each entity in the array is an instance of `PerformanceResourceTiming`, which contains the following information:

```
{
  connectEnd: 1354.525000002468
  connectStart: 1354.525000002468
  domainLookupEnd: 1354.525000002468
  domainLookupStart: 1354.525000002468
  duration: 179.89400000078604
  entryType: "resource"
  fetchStart: 1354.525000002468
  initiatorType: "link"
  name: "https://developer.chrome.com/static/css/out/site.css"
  redirectEnd: 0
  redirectStart: 0
  requestStart: 1380.8379999827594
```

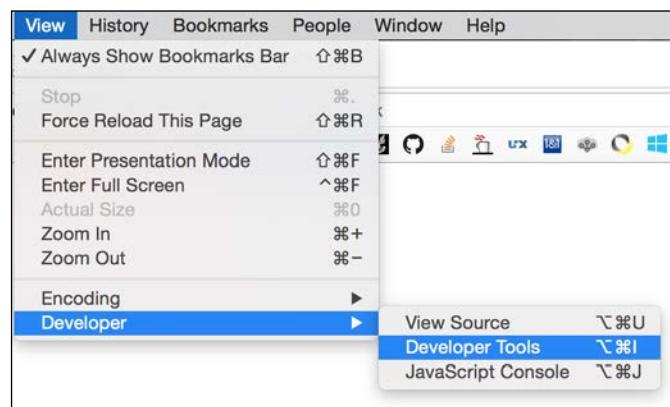
```
responseEnd: 1534.419000003254
responseStart: 1533.6550000065472
secureConnectionStart: 0
startTime: 1354.525000002468
}
```

Unfortunately, the timing data points in the preceding format may not be really useful, but there are tools that can help us to analyze them with ease. The first of these tools is a browser extension called **performance-bookmarklet**. This extension is open source and is available for Chrome and Firefox. The extension download links can be found at <https://github.com/micmro/performance-bookmarklet>.

In the following screenshot, you can see one of the graphs generated by the extension. The graphs display the performance typing API information in a much better way, allowing us to spot performance issues with ease:

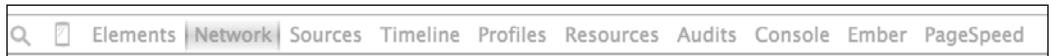


Alternatively, you can use the network panel in the Chrome developer tools to perform network performance profiling. To access the network panel, navigate to **View**, **Developer**, and then **Developer Tools**:

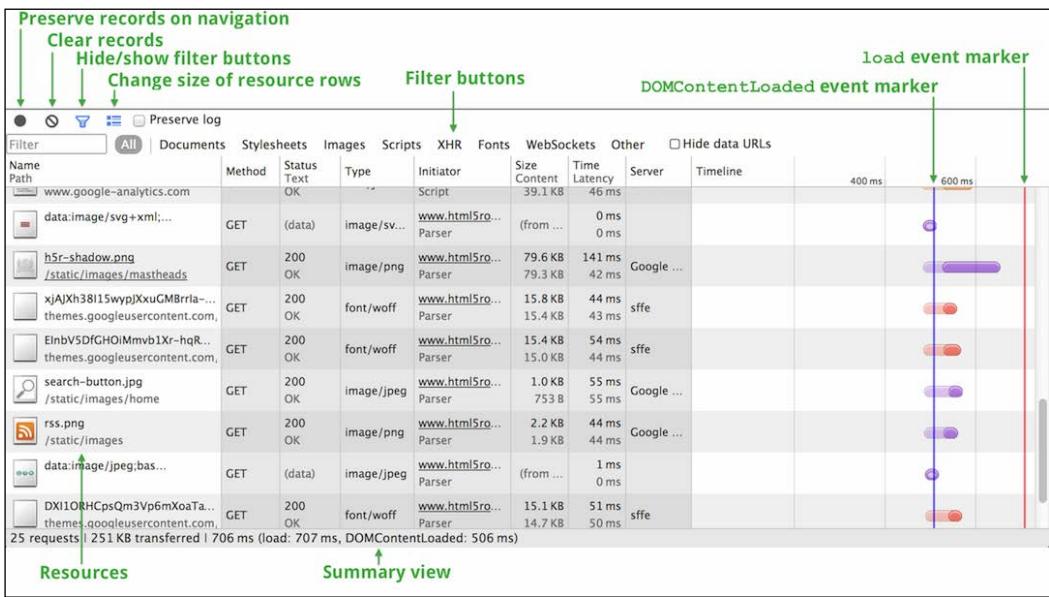


 Windows users can access the developer tools by pressing the *F12* key.
OS X users can access it using the *Alt + Cmd + I* shortcut.

Once the developer tools are visible, you can access the **Network** tab by clicking on it:

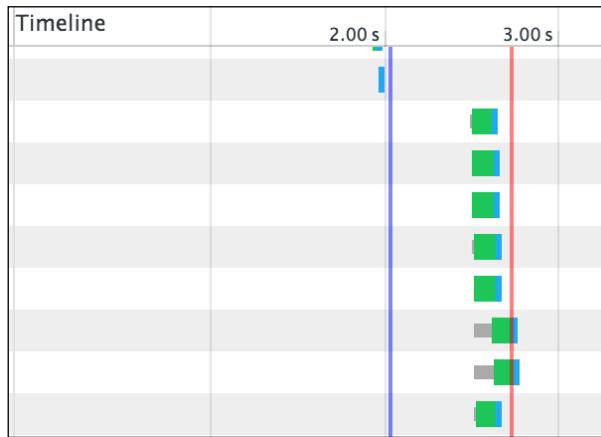


Clicking on the **Network** tab will lead you to a screen similar to the one seen here:



As you can observe, the information is presented in a table in which each file loaded is displayed as a row. On the right-hand side, you can see that one of the columns is the timeline. The timeline displays the performance timing API in a similar way to the way that the performance-bookmarklet extension did.

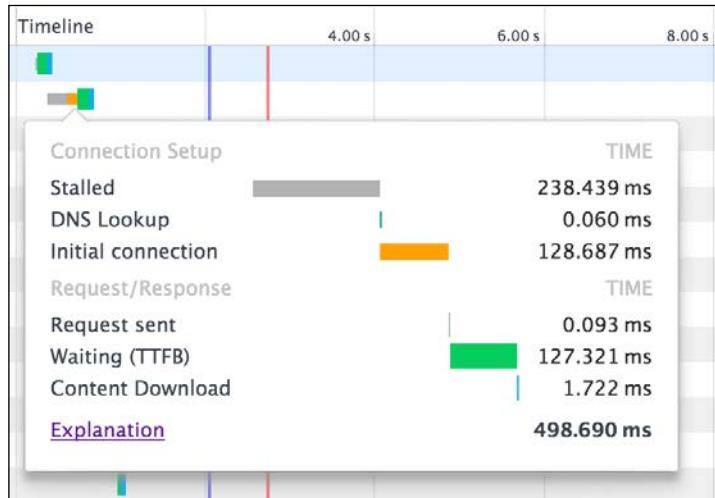
Two important elements in the timeline are the red and blue lines. These lines let us know when the `DOMContentLoaded` event is triggered (the blue line), following which the `load` event is triggered (the red line):



These two events are important because we can examine which requests were completed when the event was fired to get an idea of which contents were available for the user when they took place:

- The `DOMContentLoaded` event is fired when the engine has completed parsing of the main document
- The `load` event is fired when all the page's resources have been loaded

If you hover over one of the cells of the timing column, you will be able to see each of the performance timing API data points:



It is interesting to know that this developer tool actually reads this information using the performance timing API. Let's understand the meaning of each of the data points:

Performance timing API data point	Description
Stalled/Blocking	This is the time the request spent waiting before it could be sent; there is a maximum number of open TCP connections for an origin. When the limit is reached, some requests will display blocking time rather than stalled time.
Proxy Negotiation	This is the time spent negotiating a connection with a proxy server.
DNS Lookup	This is the time spent resolving a DNS address; resolving a DNS requires a full round-trip to the DNS server for each domain in the page.
Initial Connection / Connecting	This is the time it took to establish a connection.
SSL	This is the time spent establishing an SSL connection.
Request Sent / Sending	This is the time spent issuing the network request, typically a fraction of a millisecond.
Waiting (TTFB)	This is the time spent waiting for the initial byte to be received – the time to first byte (TTFB). The TTFB can be used to find out the latency of a round-trip to the server in addition to the time spent waiting for the server to deliver the response.
Content Download / Downloading	This is the time taken for the response data to be received.

Network performance and user experience

Now that you know how we can analyze network performance, it is time to identify the performance goals we should aim for. Numerous studies have proved that it is really important to keep loading times as low as possible. The Akamai study, published in September 2009, interviewed 1,048 online shoppers and found the following:

- 47 percent of people expect a web page to load in two seconds or less
- 40 percent will abandon a web page if it takes more than three seconds to load
- 52 percent of online shoppers claim that quick page loads are important for their loyalty to a site
- 14 percent will start shopping at a different site if page loads are slow; 23 percent will stop shopping or even walk away from their computer
- 64 percent of shoppers who are dissatisfied with their site visit will go somewhere else to shop next time

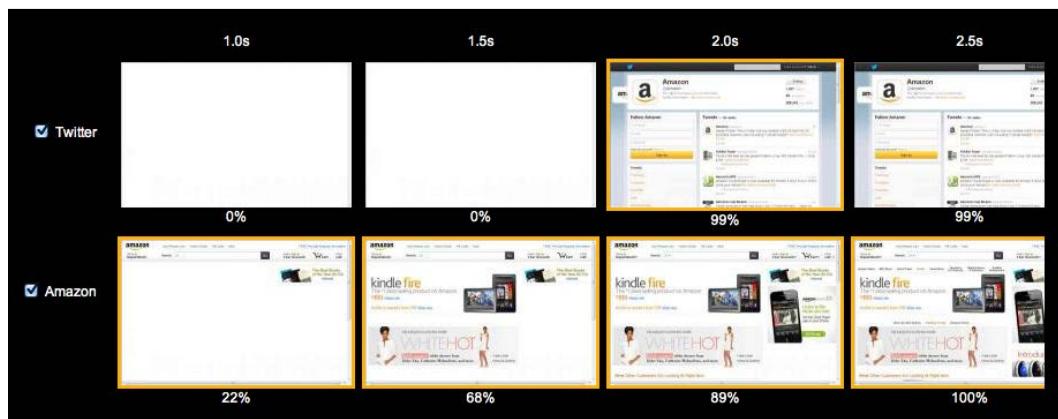


You can read the full Akamai study at http://www.akamai.com/html/about/press/releases/2009/press_091409.html.

From the preceding study conclusions, we should assume that network performance matters. Our first priority should be to try to improve the loading speed.

If we try to improve the performance of a site to make sure that it loads in less than two seconds, we might make a common mistake: trying to get the `onLoad` event to be triggered in under two seconds.

While triggering the `onLoad` event as early as possible will probably improve the network performance of an application, it doesn't mean that the user experience will be equally improved. The `onLoad` event is insufficient to determine performance. We can demonstrate this by comparing the loading performance of the Twitter and Amazon websites. As you can see in the following screenshot, users have the opportunity to engage with Amazon much sooner than with Twitter. Even though the `onLoad` event is the same on both sites, the user experience is drastically different:



This example demonstrates that to improve the user experience, we must try to reduce the loading times, but we must also try to load the web contents in such a way that the user engagement can begin as early as possible. To achieve this, we should load all the secondary content in an asynchronous manner.



Refer to *Chapter 3, Working with Functions* to learn more about asynchronous programming with TypeScript.

Network performance best practices and rules

Another easy way to analyze the performance of a web application is by using a best-practices tool for network performance, such as the Google PageSpeed Insights application or the Yahoo YSlow application.

Google PageSpeed Insights can be used online or as a Google Chrome extension. To try this tool, you can visit the online version at <https://developers.google.com/speed/pagespeed/insights/> and insert the URL of the web application that you want to analyze. In just a few seconds, you will get a report like the one in the following screenshot:

The screenshot shows the Google PageSpeed Insights interface. At the top, there's a navigation bar with the Google Developers logo, followed by 'Products > PageSpeed Insights'. Below that is a search bar containing 'PageSpeed Insights' and a 'g+1' button. A URL input field contains 'http://www.remojansen.com/' and a blue 'ANALYZE' button to its right. Underneath, there are two tabs: 'Mobile' (selected) and 'Desktop'. The main content area displays a red box with the score '31 / 100 Speed'. It then lists recommendations under 'Should Fix:' and 'Consider Fixing:', each with a 'Show how to fix' link. On the right side, there's a preview of a smartphone displaying the mobile version of the website 'WWW.REMOJANSSEN.COM' with a circular profile picture of three people and some descriptive text below it.

The report contains some effective recommendations that will help us to improve the network performance and overall user experience of our web applications. Google PageSpeed Insights uses the following rules to rate the speed of a web application:

- Avoid landing page redirects
- Enable compression

- Improve server response time
- Leverage browser caching
- Minify resources
- Optimize images
- Optimize CSS Delivery
- Prioritize visible content
- Remove render-blocking JavaScript
- Use asynchronous scripts

When you use this tool, if you click on the score of each rules, you can see recommendations and details that will help you to understand what is wrong and what you need to do to increase the score achieved for one particular rule.

On the other hand, Yahoo YSlow is available as a browser extension, a Node.js module, and a PhantomJS plugin, among others. We can find the right version for our needs at <http://yslow.org/>. When we run YSlow, it will generate a report that will provide us with a general score and a detailed score of the website, like the one in the following screenshot:

Grade B Overall performance score 82 Ruleset applied: YSlow(V2) URL: <http://www.remojansen.com/>

[ALL \(23\)](#) FILTER BY: [CONTENT \(6\)](#) | [COOKIE \(2\)](#) | [CSS \(6\)](#) | [IMAGES \(2\)](#) | [JAVASCRIPT \(4\)](#) | [SERVER \(6\)](#)

C Make fewer HTTP requests	
F	Use a Content Delivery Network (CDN)
A	Avoid empty src or href
F	Add Expires headers
A	Compress components with gzip
A	Put CSS at top
A	Put JavaScript at bottom
A	Avoid CSS expressions
n/a	Make JavaScript and CSS external
A	Reduce DNS lookups
A	Minify JavaScript and CSS
A	Avoid URL redirects
A	Remove duplicate JavaScript and CSS

Grade C on Make fewer HTTP requests

This page has 4 external Javascript scripts. Try combining them into one. This page has 8 external stylesheets. Try combining them into one.

Decreasing the number of components on a page reduces the number of HTTP requests. Some ways to reduce the number of components include: combine files, compress files, use a style sheet, and use CSS Sprites and image maps.

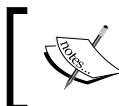
[»Read More](#)

Copyright © 2015 Yahoo! Inc. All rights reserved.

YSlow uses the following set of rules to rate the speed of a web application:

- Minimize HTTP requests
- Use a content delivery network
- Avoid empty `src` or `href`
- Add an expires or a cache-control header
- Gzip components
- Put stylesheets at the top
- Put scripts at the bottom
- Avoid CSS expressions
- Make JavaScript and CSS external
- Reduce DNS lookups
- Minify JavaScript and CSS
- Avoid redirects
- Remove duplicate scripts
- Configure ETags
- Make AJAX cacheable
- Use GET for AJAX requests
- Reduce the number of DOM elements
- Prevent 404 errors
- Reduce cookie size
- Use cookie-free domains for components
- Avoid filters
- Do not scale images in HTML
- Make `favicon.ico` small and cacheable

Just like before, when you use this tool, if you click on each of the rules scored you can see recommendations and details that will help you to understand what is wrong and what you need to do to increase the score achieved for one particular rule.



If you want to learn more about network performance optimization, please take a look at the book *High Performance Browser Networking* by Ilya Grigorik.

GPU performance analysis

The rendering of some elements in web applications is accelerated by the use of the GPU. The GPU is specialized in the processing of graphics-related instructions and can, therefore, deliver much better performance than the CPU when it comes to graphics. For example, CSS3 animations in modern web browsers are accelerated by the GPU, while the CPU performs JavaScript animations. In the past, the only way to achieve some animations was via JavaScript. But today, we should avoid using them when possible and use CSS3 instead because it will help us to achieve great web performance.

In recent years, access to the GPU has been added to browsers via the WebGL API. This API allows web developers to create 3D games and other highly visual applications by using the power of the GPU.

Frames per second (FPS)

We will not go into much detail about the performance of 3D applications because it is a really extensive field and we could write an entire book talking about it.

However, we will mention an important concept that can be applied to any kind of web application: **frames per second (FPS)** or frame rate. When a web application is displayed on screen, it is done at a number of images (frames) per second. A low frame rate can be detrimental to the overall user experience when perceived by the users. A lot of research has been carried out on this topic, and 60 frames per second seems to be the optimum frame rate for a great user experience.

Whenever we develop a web application, we should take a look at the frame rate and try to prevent it from dropping below 40 FPS. This is especially important during animations and user actions.

An open source library called `stats.js` can help us to see the frame rate while developing a web application. This library can be downloaded from GitHub at <https://github.com/mrdoob/stats.js/>. We need to download the library and load it in a web page. We can then load the following code snippet by adding a new file or just execute it in the developer console:

```
var stats = new Stats();
stats.setMode(1); // 0: fps, 1: ms

// position of the frame rate counter (align top-left)
stats.domElement.style.position = 'absolute';
stats.domElement.style.left = '0px';
stats.domElement.style.top = '0px';
```

```
document.body.appendChild( stats.domElement );  
  
var update = function () {  
    stats.begin();  
    // monitored code goes here  
    stats.end();  
    requestAnimationFrame( update );  
};  
requestAnimationFrame( update );
```

If everything goes well, we will be able to see the frame rate counter in the top-left corner of the screen. Clicking on it will switch from the FPS view to the millisecond (MS) view:

- The FPS view displays the frames rendered in the last second. The higher this number is, the better.
- The MS view displays the milliseconds needed to render a frame. The lower this number is, the better.



[ Some advanced WebGL applications may require an in-depth performance analysis. For such cases, Chrome provides the Trace Event Profiling Tool. If you wish to learn more about this tool, visit the official page at <https://www.chromium.org/developers/how-tos/trace-event-profiling-tool>.]

CPU performance analysis

To analyze the usage of the processing time, we will take a look at the execution path of our application. We will examine each of the functions invoked and how long it takes to complete their execution. We can access all this information by opening the Chrome developer tools' **Profiles** tab:

Profiles

Select profiling type

 Collect JavaScript CPU Profile

CPU profiles show where the execution time is spent in your page's JavaScript functions.

 Take Heap Snapshot

Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.

 Record Heap Allocations

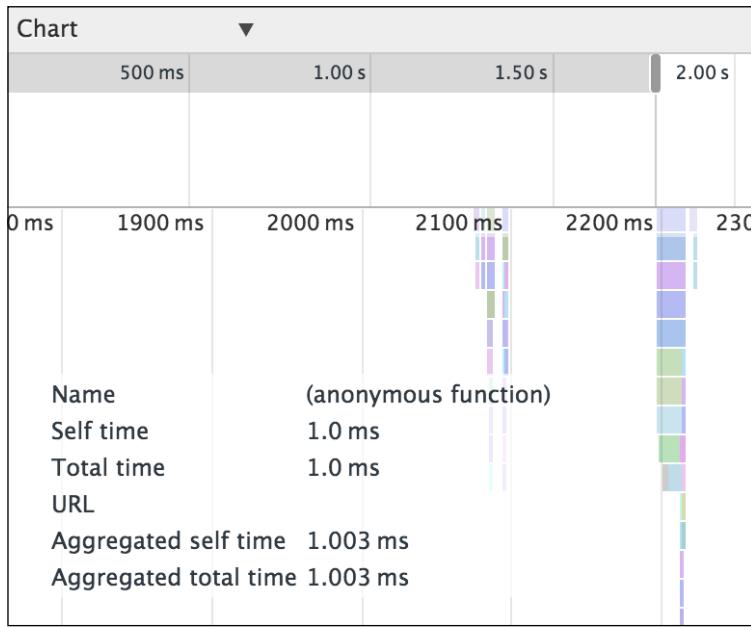
Record JavaScript object allocations over time. Use this profile type to isolate memory leaks.

Start**Load**

In this tab, we can select **Collect JavaScript CPU Profile** and then click on the **Start** button to start recording the CPU usage. Being able to select when we want to start and stop recording the CPU usage helps us select the specific functions that we want to analyze. If, for example, we want to analyze a function named `foo`, all we need to do is start recording the CPU usage, invoke the `foo` function and stop recording. A timeline like the one in the following screenshot will then be displayed:



The timeline displays (horizontally) the functions invoked in the chronological order. If the function invokes other functions, the function's call-stack is displayed vertically. When we hover over one of these functions, we will be able to see its details in the bottom-left corner of the timeline:



The details include the following information:

- **Name:** The name of the function.
- **Self time:** The time spent on the completion of the current invocation of the function. We will take into account the time spent in the execution of the statements within the function, not including any functions that it called.
- **Total time:** The total time spent on the completion of the current invocation of the function. We will take into account the time spent in the execution of the statements within the function, including functions that it called.
- **Aggregated self time:** The time for all invocations of the function across the recording, not including functions called by this function.
- **Aggregated total time:** The time for all invocations of the function across the recording, including functions called by this function.

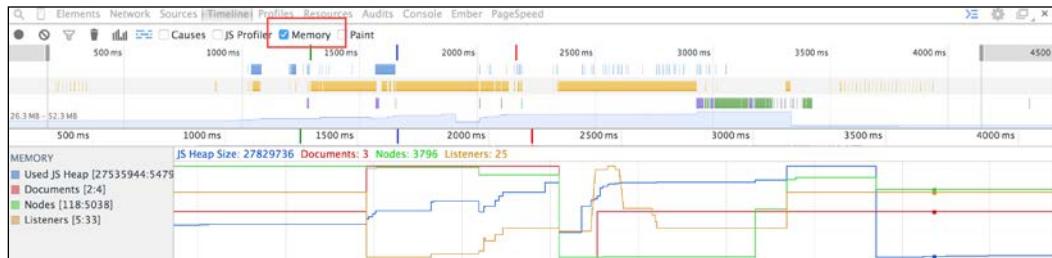
As we saw in the previous chapter, all the JavaScript code is executed in one single thread at runtime. For this reason, when a function is executed, no other function will be executed. Sometimes, the execution of a function takes too long to be completed, and the application becomes unresponsive. We can use the CPU profile report to identify which functions are consuming too much processing time. Once we have identified these functions, we can refactor and then to try to improve the application responsiveness. Some common improvements include using an asynchronous execution flow when possible and reducing the size of the functions.

Memory performance analysis

When we declare a variable, it is allocated in the RAM. Some time after the variable is out of the scope, it is cleared from memory by the garbage collector. Sometimes, we can generate a scenario in which a variable never goes out of scope. If the variable never goes out of scope, it will never be cleared from memory. This can eventually lead to some serious memory leaking issues. A **memory leak** is the continuous loss of available memory.

When dealing with memory leaks, we can take advantage of the Google Chrome developer tools to identify the root cause of the problem with ease.

The first thing that we might wonder is whether our application has memory leaks or not. We can find out by visiting the timeline tab and clicking on the top-left icon to start recording the resource usage. Once we stop recording, a timeline graph like the one in the following screenshot will be displayed:



In the timeline, we can select **Memory** to see the memory usage (**Used JS Heap**) over time (the blue line in the image). In the preceding example, we can see a notable drop towards the end of the line. This is a good sign because it indicates that the majority of the used memory has been cleared when the page has finished loading.

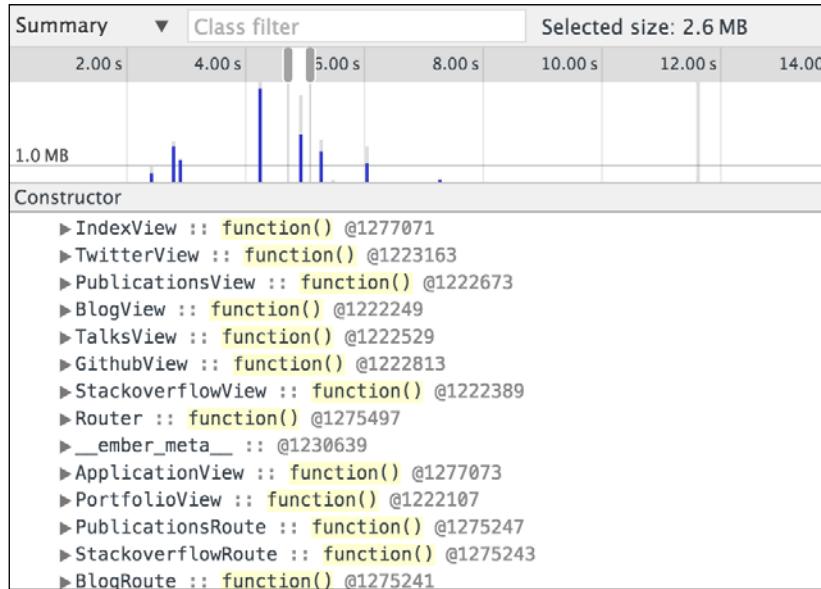
The memory leaks can also take place after loading; in that case, we can use the application for a while and observe how the memory usage varies in the graph to identify the cause of the leak.

An alternative way to detect memory leaks is by observing the memory allocations. We can access this information by recording the heap allocations in the **Profiles** tab:

Record Heap Allocations
Record JavaScript object allocations over time. Use this profile type to isolate memory leaks.

Start **Load**

The report will be displayed after we have recorded some usage of the resources. We can do this by clicking on the **Start** and **Stop** buttons. The memory allocation report will display a timeline like the one in the following screenshot. Each of the blue lines is a memory allocation that took place during the recorded period. The height of the line represents the amount of memory used. As you can see, the memory is almost cleared completely around the eighth second:



If we click on one of the blue lines, we will be able to navigate through all the variables that were stored in memory when the allocation took place and examine their values. It is also possible to take a memory snapshot at any given point from the **Profiles** tab:

Take Heap Snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.

This feature is particularly useful when we are debugging and we want to see the memory usage at a particular breakpoint. The memory snapshot works like the details view in the previously explained allocations view:

The screenshot shows a memory snapshot interface with the following structure:

- Summary** ▼ Class filter All objects
- Constructor**
- Error
- Window / about:blank
- ▼ u
 - u @1444185
 - u @1438303
 - u @1695301
 - u @1646261
- Retainers**
- Object**
- ▼ renderer in function() @1506093
 - PublicationsView in n @1461035
 - app in Window / www.remojansen.com/#/portfolio @1420981
 - source in @1741023
 - 2 in [] @1752981
 - application:main in @1741035
 - application:main in @1741033
 - application:main in @1741031
 - application:main in @1741029
 - [1] in Array @1448393
 - 0 in (object properties)[] @1742809
 - 1 in (object elements)[] @1480317
 - namespace in n @1740333
 - 39 in (object properties)[] @1742819
 - 39 in (object properties)[] @1742817
 - 39 in (object properties)[] @1742815

As you can see in the preceding screenshot, the memory snapshot allows us to navigate through all the variables that were stored in memory when the snapshot was taken and examine their values.

The garbage collector

Programming languages with a low level of abstraction have low-level memory management mechanisms. On the other hand, in languages with a higher level of abstraction, such as C# or JavaScript, the memory is automatically allocated and freed by a process known as the garbage collector.

The JavaScript garbage collector does a great job when it comes to memory management, but it doesn't mean that we don't need to care about memory management.

Independent of which programming language we are working with, the memory life cycle pretty much follows the same pattern:

- Allocate the memory you need
- Use the memory (read/write)
- Release the allocated memory when it is not needed any more

The garbage collector will try to release the allocated memory when is not needed any more using a variation of an algorithm known as the **mark-and-sweep algorithm**. The garbage collector performs periodical scans to identify objects that are out of the scope and can be freed from the memory. The scan is divided in two phases: the first one is known as **mark** because the garbage collector will flag or mark the items that can be freed from the memory. During the second phase, known as **sweep**, the garbage collector will free the memory consumed by the items marked in the previous phase.

The garbage collector is usually able to identify when an item can be cleared from the memory; but we, as developers, must try to ensure that objects get out of scope when we don't need them any more. If a variable never gets out of the scope, it will be allocated in memory forever, potentially leading to a severe memory leak issue.

The number of references pointing to an item in memory will prevent it from being freed from memory. For this reason, most cases of memory leaks can be fixed by ensuring that there are no permanent references to variables. Here are a few rules that can help us to prevent potential memory leak issues:

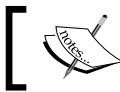
- Remember to clear intervals when you don't need them any more.
- Remember to clear event listeners when you don't need them any more.
- Remember that when you create a closure, the inner function will remember the context in which it was declared. This means that there will be some extra items allocated in memory.
- Remember that when using object composition, if circular references are created, you can end up having some variables that will never be cleared from memory.

Performance automation

In this section we will understand how we can automate many of the performance optimization tasks, from concatenation and compression of contents to the automation of the performance monitoring and performance testing processes.

Performance optimization automation

After analyzing the performance of our application, we will start working on some performance optimizations. Many of these optimizations involve the concatenation and compression of some of the application's components. The problem with compressed components is that they are more complicated to debug and maintain. We will also have to create a new version of the concatenated and compressed contents every time one of the original components (not concatenated and not compressed) changes. As these include many highly repetitive tasks, we can use the task runner Gulp to perform many of these tasks for us. We can find online plugins that will allow us to concatenate and compress components, optimize images, generate a cache manifest, and perform many other performance optimization tasks.



If you would like to learn more about Gulp, refer to *Chapter 2, Automating Your Development Workflow*.



Performance monitoring automation

We have seen that we can automate many of the performance optimization tasks using the Gulp task runner. In a similar way, we can also automate the performance monitoring process.

In order to monitor the performance of an existing application, we will need to collect some data that will allow us to compare the application performance over time. Depending on how we collect the data, we can identify three different types of performance monitoring:

- **Real user monitoring (RUM):** This is a type of solution used to capture performance data from real user visits. The collection of data is performed by a small JavaScript code snippet loaded in the browser. This type of solution can help us to collect data and discover performance trends and patterns.
- **Simulated browsers:** This type of solution is used to capture performance data from simulated browsers. This is the most economic option, but it is limited because simulated browsers cannot offer as accurate a representation of the real user experience.
- **Real-browser monitoring:** This is used to capture the performance data of real browsers. This information provides a more accurate representation of the real user experience, as the data is collected using exactly what a user would see if they visited the site with the given environment (browser, geographic location, and network throughput).

In *Chapter 2, Automating Your Development Workflow*, we saw how to configure a Gulp task that used the Karma test runner to execute a test suite in a headless browser known as **PhantomJS**.

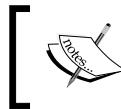
PhantomJS is a simulated browser that can be configured to generate **HTTP Archive (HAR)** files. A HAR file uses a common format for recording HTTP tracing information. This file contains a variety of information, but for our purposes, it has a record of each object being loaded by a browser.

There are multiple scripts available online that showcase how to collect the data and reformat it using the PhantomJS API. One of the examples, `netsniff.js`, exports the network traffic in HAR format. The `netsniff.js` file (and other examples) can be found at <https://github.com/ariya/phantomjs/blob/master/examples/netsniff.js>.

Once we have generated the HAR files, we can use another application to see the collected performance information on a visual timeline. This application is called HAR viewer, and it can be found at <https://github.com/janodvarko/harviewer>.

Alternatively, we could write a custom script or Gulp task to read the HAR files and break the automated build if the application performance doesn't meet our needs.

It is also possible to configure PhantomJS to run the YSlow performance analysis report and integrate it with the automated build. To learn more about PhantomJS and performance monitoring, refer to the official documentation at <http://phantomjs.org/network-monitoring.html>.



If you are considering using RUM, take a look at the New Relic solutions at <http://newrelic.com/>, or Google Analytics at <http://www.google.com/analytics/>.



Performance testing automation

Another way to improve the performance of an application is to write automated performance tests. These tests can be used to guarantee that the system meets a set of performance goals. There are multiple types of performance testing, but some of the most common ones include the following:

- **Load testing:** This is the most basic form of performance testing. We can use a load test to understand the behavior of the system under a specific expected load (number of concurrent users, number of transactions, and duration). There are multiple types of load testing:

- **Stress testing:** This is normally used to understand the maximum capacity limits of an application. This kind of test determines if an application is able to handle an extreme load by using an extreme load for an extended period of time.
Stress testing is not really useful when working on a client-side application. However, it can be really helpful when working on a Node.js application, since Node.js applications can have many simultaneous users.
- **Soak testing:** This is also known as endurance testing. This kind of test is similar to the stress test, but instead of using an extreme load, it uses the expected load for an extended period of time. It is a common practice to collect memory usage data during this kind of test to detect potential memory leaks. This kind of test helps us to detect if the performance suffers some kind of degradation after an extended period of time.
- **Spike testing:** This is also similar to the stress test, but instead of using an extreme time load during an extended time period, it uses sudden intervals of extreme and expected load. This kind of test helps us to determine if an application is able to handle dramatic changes in load.
- **Configuration testing:** This is used to determine the effects of configuration changes on the performance and behavior of an application. A common example would be experimenting with different methods of load balancing.



This kind of test can also be automated by using tools such as JMeter (<http://jmeter.apache.org>) or Locust (<http://locust.io>).

Exception handling

Understanding how to use the available resources in an efficient manner will help us to create better applications. In a similar manner, understanding how to handle runtime errors will help us to improve the overall quality of our applications. Exception handling in TypeScript involves three main language elements.

The Error class

When a runtime error takes place, an instance of the `Error` class is thrown:

```
throw new Error();
```

We can create custom errors in a couple of different ways. The easiest way to achieve it is by passing a string as argument to the `Error` class constructor:

```
Throw new Error("My basic custom error");
```

If we need more customizable and advanced control over custom exceptions, we can use inheritance to achieve it:

```
module CustomException {  
    export declare class Error {  
        public name: string;  
        public message: string;  
        public stack: string;  
        constructor(message?: string);  
    }  
  
    export class Exception extends Error {  
  
        constructor(public message: string) {  
            super(message);  
            this.name = 'Exception';  
            this.message = message;  
            this.stack = (<any>new Error()).stack;  
        }  
        toString() {  
            return this.name + ': ' + this.message;  
        }  
    }  
}
```

In the preceding code snippet, we have declared a class named `Error`. This class is available at runtime but is not declared by TypeScript, so we will have to do it ourselves. Then, we have created an `Exception` class, which inherits from the `Error` class.

Finally, we can create `customError` by inheriting from our `Exception` class:

```
class CustomError extends CustomException.Exception {  
    // ...  
}
```

The try...catch statements and throw statements

A catch clause contains statements that specify what to do if an exception is thrown in the `try` block. We should perform some operations in the `try` block, and if they fail, the program execution flow will move from the `try` block to the `catch` block.

Additionally, there is an optional block known as `finally`, which is executed after both the `try` and `catch` (if there was an exception in `catch`) blocks:

```
try {
    // code that we want to work
    throw new Error("Oops!");
}
catch (e) {
    // code executed if expected to work fails
    console.log(e);
}
finally {
    // code executed always after try or try and catch (when
    errors)
    console.log("finally!");
}
```

It is also important to mention that in the majority of programming languages, including TypeScript, throwing and catching exceptions is an expensive operation in terms of resource consumption. We should use these statements if we need them, but sometimes it is necessary to avoid them because they can potentially negatively affect the performance of our applications. Therefore, we should keep in mind that it is a good idea to avoid the use of `try...catch` and `throw` statements in performance-critical functions and loops.

Summary

In this chapter, we saw what performance is and how the availability of resources can influence it. We also looked at how to use some tools to analyze the way a TypeScript application uses available resources. These tools allow us to spot some possible issues, such as a low frame rate, memory leaks, and high loading times. We have also discovered that we can automate many kinds of performance optimization task, as well as the performance monitoring and testing processes.

In the following chapter, we will see how we can automate the testing process of our TypeScript applications to achieve great application maintainability and reliability.

Application Testing

In this chapter, we are going to take a look at how to write unit tests for TypeScript applications. We will see how to use tools and frameworks to facilitate the testing process of our applications.

The contents of this chapter cover the following topics:

- Setting up a test infrastructure
- Testing planning and methodologies
- How to work with Mocha, Chai, and Sinon.JS
- How to work with test assertions, specs, and suites
- Test spies
- Test stubs
- Testing on multiple environments
- How to work with Karma and PhantomJS
- End-to-end testing
- Generating test coverage reports

We will get started by installing some necessary third-party software dependencies.

Software testing glossary

Across this chapter, we will use some concepts that may not be familiar to those readers without previous software testing experience. Let's take a quick look at some of the most popular testing concepts before we get started.

Assertions

An **assertion** is a condition that must be tested to confirm that a certain piece of code behaves as expected or, in other words, to confirm conformance to a requirement.

Let's imagine that we are working as part of one of the Google Chrome development team and we have to implement the JavaScript `Math` object. If we are working on the `pow` method, the requirement could be something like the following:

"The `Math.pow(base, exponent)` function should return the base (the base number) to the exponent (the exponent used to raise the base power – that is, $\text{base}^{\text{exponent}}$)."

With this information, we could create the following implementation:

```
class Math1 {  
    public static pow(base: number, exponent: number) {  
        var result = base;  
        for(var i = 1; i < exponent; i++) {  
            result = result * base;  
        }  
        return result;  
    }  
}
```

To ensure that the method is correctly implemented, we must test it conforms with the requirement. If we analyze the requirements closely, we should identify at least two necessary assertions.

The function should return the base to the exponent:

```
var actual = Math1.pow(3, 5);  
var expected = 243;  
var assertion1 = (Math1.pow(base1, exponent1) === expected1);
```

The exponent is not used as the base (or the base is not used as the exponent):

```
var actual = Math1.pow(5, 3);  
var expected = 125;  
var assertion2 = (Math1.pow(base2, exponent2) === expected2);
```

If both assertions are valid, then our code adheres to the requirements, and we know that it will work as expected:

```
var isValidCode = (assertion1 && assertion2);  
console.log(isValidCode);
```

Specs

Spec is a term used by software development engineers to refer to test specifications. A test specification (not to be confused with a test plan) is a detailed list of all the scenarios that should be tested, how they should be tested, and so on. We will see later in this chapter how we can use a testing framework to define a test spec.

Test cases

A **test case** is a set of conditions used to determine whether one of the features of an application is working as it was originally established to work. We might wonder what the difference between a test assertion and a test case is. While a test assertion is a single condition, a test case is a set of conditions. We will see later in this chapter how we can use a testing framework to define test cases.

Suites

A **suite** is a collection of test cases. While a test case should focus on only one test scenario, a test suite can contain test cases for many test scenarios.

Spies

Spies are a feature provided by some testing frameworks. They allow us to wrap a method and record its usage (input, output, number of times invoked). When we wrap a function with a spy, the underlying method's functionality does not change.

Dummies

A **dummy** object is an object that is passed around during the execution of a test but is never actually used.

Stubs

A **stub** is a feature provided by some testing frameworks. Stubs also allow us to wrap a method to observe its usage. Unlike spies, when we wrap a function with a stub, the underlying method's functionality is replaced with a new behavior.

Mocks

Mocks are often confused with stubs. Martin Fowler once wrote the following in an article titled Mocks Aren't Stubs:

In particular I see them often (mocks) confused with stubs - a common helper to testing environments. I understand this confusion - I saw them as similar for a while too, but conversations with the mock developers have steadily allowed a little mock understanding to penetrate my tortoiseshell cranium. This difference is actually two separate differences. On the one hand there is a difference in how test results are verified: a distinction between state verification and behavior verification. On the other hand is a whole different philosophy to the way testing and design play together, which I term here as the classical and mockist styles of Test Driven Development.

Both mocks and stubs provide some sort of input to the test case; but, despite their similarities, the flow of information from each is very different:

- Stubs provide input for the application under test so that the test can be performed on something else
- Mocks provide input to the test to decide whether the test should pass or fail

The difference between mocks and stubs will become clearer as we move towards the end of this chapter.

Test coverage

The term test coverage refers to a unit of measurement, which is used to illustrate the number of portions of code in an application that have been tested via automated tests. Test coverage can be obtained by automatically generating test coverage reports. Towards the end of the chapter, we will see how to create such reports using a tool called Istanbul (<http://gotwarlost.github.io/istanbul/>).

Prerequisites

Throughout this chapter, we will use some third-party tools, including some frameworks and automation tools. We will start by looking at each tool in detail. Before we get started, we need to use npm to create a package.json file in the folder that we are going to use to implement the examples in this chapter.

Let's create a new folder named app and run the npm init command inside it to generate a new package.json file:

```
npm init
```



Refer to *Chapter 2, Automating Your Development Workflow* for additional help on npm.

Gulp

We will use the Gulp task runner to run some tasks necessary to execute our tests. We can install Gulp using npm:

```
npm install gulp -g
```



If you are not familiar with task runners and continuous integration build servers, take a look at *Chapter 2, Automating Your Development Workflow*.

Karma

Karma is a test runner. We will use Karma to automatically execute our tests. This is useful because sometimes the execution of the test will not be started by one of the members of our software development team. Instead, it will be triggered by a continuous integration build server (usually via a task runner).

Karma can be used with multiple testing frameworks, thanks to the installation of plugins. Let's install Karma using the following command:

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

We will also install another Karma plugin that facilitates the creation of test coverage reports:

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

Istanbul

Istanbul is a tool that identifies which lines of our application are processed during the execution of the automated test. It can generate reports known as test coverage reports. These reports can help us to get an idea of the level of testing of a project because they show which lines of code were not executed and a percentage value that represents the fraction of the application that has been tested. It is recommended that a test coverage value of at least 75 percent of the overall application should be achieved, while many open source projects target a test coverage of 100 percent.

Mocha

Mocha is a popular JavaScript testing library that facilitates the creation of test suites, test cases, and test specs. Mocha can be used to test TypeScript in the frontend and backend, identify performance issues, and generate different types of test reports, among many other features.

Let's install Mocha and the Karma-Mocha plugin using the following command:

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

Chai

Chai is a test assertion library that supports **test-driven development (TDD)** and **behavior-driven development (BDD)** test styles.



We will see more about TDD and BDD later in this chapter.

The main goal of Chai is to reduce the amount of work necessary to create a test assertion and make the test more readable.

We can install Chai and the Karma-Chai plugin using the following command:

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

Sinon.JS

Sinon.JS is an isolation framework that provides us with a set of APIs (test spies, stubs, and mocks) that can help us to test a component in isolation. Testing isolated software components is difficult because there is a high level of coupling between the components. A mocking library such as Sinon.JS can help us isolate the components in order to test individual features.

We can install Sinon.JS and the Karma-Sinon plugin using the following command:

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

Type definitions

To be able to work with third-party libraries in JavaScript with a good support, we need to import the type definitions of each library. We will use the `tsd` package manager to install the necessary type definitions:

```
tsd install mocha --save
```

```
tsd install chai --save  
tsd install sinon --save  
tsd install jquery - -save
```



Refer to *Chapter 2, Automating Your Development Workflow* for additional help on tsd.

PhantomJS

PhantomJS is a headless browser. We can use PhantomJS to run our tests in a browser without having to actually open a browser. Being able to do this is useful for a few reasons; the main one is that PhantomJS can be executed via a command interface, and it is really easy to integrate with task runners and continuous integration servers. The second reason is that not having to open a browser potentially reduces the time required to complete the execution of the tests suites.

We need to install the Karma plugin that will run the test in PhantomJS:

```
npm install --save-dev phantomjs  
npm install --save-dev karma-phantomjs-launcher
```

Selenium and Nightwatch.js

Selenium is a test runner but it was especially designed to run a particular type of test known as an **end-to-end** (E2E) test.



We will learn more about E2E testing later on this chapter, so we don't need to worry too much about this topic for now.

Though we will see how to use selenium towards the end of the chapter, we can install it now. We will not work with Selenium directly because we are going to use another tool (known as Nightwatch.js) for E2E testing, which will automatically run Selenium for us.

Nightwatch.js is an automated testing framework, written in Node.js for web applications and websites, which uses the Selenium WebDriver API. It is a complete browser automation (end-to-end) solution.

We can install Nightwatch.js and Selenium by executing the following commands:

```
npm install --save-dev gulp-nightwatch  
npm install selenium-standalone -g  
selenium-standalone install
```

The Selenium standalone requires the Java binaries to be installed in the development environment and accessible through the \$PATH variable. Refer to the official Java documentation at https://www.java.com/en/download/help/index_installing.xml to learn more about the Java installation.



Testing planning and methodologies

When it comes to software development, we usually have many choices. Every time we have to develop a new application, we can choose the type of database, the architecture, and frameworks that we will use. Not all our choices are about technologies. For example, we can also choose a software development methodology such as extreme programming or scrum. When it comes to testing, there are two major styles or methodologies: test-driven development (TDD) and behavior-driven development (BDD).

Test-driven development

Test-driven development is a testing methodology that focuses on encouraging developers to write tests before they write application code. Usually, the process of writing code in TDD consists of the following basic steps:

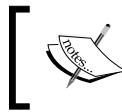
1. Write a test that fails.
2. Run the test and ensure that it fails (there is no code at this point so it should fail).
3. Write the code to make the test pass.
4. Run the test and ensure that it passes.
5. Run all the other tests to ensure that no other parts of the application break.
6. Repeat the process.

The difference between using TDD or not is really a mindset. Many developers don't like writing tests, so chances are that if we leave their implementation as the last task in the development process, the tests will not be implemented or the application will just be partially tested.

TDD is recommended because it effectively helps you and your team to increase the test coverage of your applications and, therefore, significantly reduce the number of potential issues.

Behavior-driven development (BDD)

Behavior-driven development appeared after TDD with the mission of being a refined version of TDD. BDD focuses on the way tests are described (specs) and states that the tests should focus on the application requirements and not the test requirements. Ideally, this will encourage developers to think less about the tests themselves and more about the application as a whole.



The original article in which the BDD principles were introduced for the first time by *Dan North* is available online at <http://dannorth.net/introducing-bdd/>.

As we have already seen, Mocha and Chai provide APIs for the TDD and BDD approaches. Later in this chapter, we will further explore these two approaches.

Recommending one of these methodologies is not trivial because TDD and BDD are both really good testing methodologies. However, BDD was developed after TDD with the objective to improve it, so we can argue that BDD has some additional advantages over TDD. In BDD, the description of a test focuses on what the application should do and not what the test code is testing. This can help the developers to identify tests that reflect the behavior desired by the customer. BDD tests are then used to document the requirements of a system in a way that can be understood and validated by both the developer and the customer. On the other hand, TDD tests cannot be understood with ease by the customer.

Tests plans and test types

The term test plan is sometimes incorrectly used to refer to a test specification. While tests specifications define the scenarios that will be tested and how they will be tested, the test plan is a collection of all the test specs for a given area.

It is recommended to create an actual planning document because a test plan can involve many processes, documents, and practices. One of the main goals of a test plan is to identify and define what kind of test is adequate for a particular component or set of components in an application.

Following are the most commonly used test types:

- **Unit tests:** These are used to test an isolated component. If the component is not isolated – or in other words, the component has some dependencies – we will have to use some tools and practices such as mocks or dependency injection to try to isolate it as much as we can during the test.
If it is not possible to manipulate the component dependencies, we will use spies to facilitate the creation of the unit tests.
Our main goal should be to achieve the total isolation of a component when it is tested. A unit test should also be fast, and we should try to avoid input/output, network usage, and any other operation that could potentially affect the speed of the test.
- **Partial integration tests and full integration tests:** These are used to test a set of components (partial integration test) or the entire application as a whole (full integration test). In integration, we will normally use known test data to feed the backend with information that will be displayed in the frontend. We will then assert that the displayed information is correct.
- **Regression tests:** These tests are used to verify that an issue has been fixed. If we are using TDD or BDD, whenever we encounter an issue we should create a unit test that reproduces the issue, and then change the code. By doing this, we will be able to run attempts to reproduce past issues and ensure that everything is still working.
- **Performance / Load tests:** These tests verify if the application meets our performance expectations. We can use performance tests to verify that our application will be able to handle many concurrent users or activity spikes. To learn more about this type of test, take a look at the previous chapter: *Chapter 6, Application Performance*.
- **End-to-end (E2E) tests:** These tests are not really different from full integration tests. The main difference is that in an E2E testing session, we will try to emulate an environment almost identical to the real user environment. We will use Nightwatch.js and Selenium for this purpose.
- **User acceptance tests (UAT):** These are used so that the system meets all the requirements of the end user.

Setting up a test infrastructure

As we saw previously in this chapter when we talked about unit tests, usually, testing requires being able to isolate the individual software component of our applications.

In order to be able to isolate the components of our application, we will need to adhere to some principles (such as the dependency inversion principle) that will help us to increase the level of decoupling between the components.

We will now configure a testing environment using Gulp and Karma and write some automated test using Mocha and Chai. By the end of this chapter, we will know how writing unit tests can help us to increase the level of decoupling and isolation between the components of an application, and how they can lead us to the development of great applications, especially when it comes to maintainability and reliability.

Let's get started by creating the folder structure of a new application. We will create two folders inside the app folder that we created at the beginning of this chapter.

Let's name the first folder `source` and the second folder `test`. Here, we can see how our directory tree should look by the end of the chapter:

```
|-- app
  |-- gulpfile.js
  |-- index.html
  |-- karma.conf.js
  |-- nightwatch.json
  |-- package.json
  |-- source
    |-- calculator_widget.ts
    |-- demos.ts
    |-- interfaces.d.ts
    |-- math_demo.ts
  |-- style
    |-- demo.css
  |-- test
    |-- bdd.test.ts
    |-- e2e.test.ts
    |-- tdd.test.ts
  |-- tsd.json
  |-- typings
```

We are going to develop a really small application to be able to write a unit test. We are going to write a unit test and an end-to-end test.



The source code of the entire demo can be found in the companion code samples.



Once we have completed our application, we will be able to open it in a browser, where we should see a form like the one in the following screenshot. This form allows us to find the result of a number (base) to the power of another (exponent).

Test Application

Base	Exponent	Result	
2	8	256	<button>Submit</button>

Building the application with Gulp

We will get started by creating a new `gulpfile.js` file as we did in *Chapter 2, Automating Your Development Workflow*. The first thing that we are going to do is import all the necessary node modules:

```
var gulp      = require("gulp"),
    browserify = require("browserify"),
    source     = require("vinyl-source-stream"),
    buffer     = require("vinyl-buffer"),
    run        = require("gulp-run"),
    nightwatch = require('gulp-nightwatch'),
    tslint     = require("gulp-tslint"),
    tsc        = require("gulp-typescript"),
    browserSync = require('browser-sync'),
    karma      = require("karma").server,
    uglify     = require("gulp-uglify"),
    docco      = require("gulp-docco"),
    runSequence= require("run-sequence"),
    header     = require("gulp-header"),
    pkg        = require(__dirname + "/package.json");
```



Remember that we need to install all necessary packages by using the npm package manager. We can take a look at the `package.json` file to see all the dependencies and their respective versions.



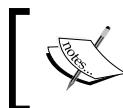
The second thing that we are going to do is to create some tasks to compile our TypeScript code. Here, we should notice that we are going to compile the application code into the /build/source folder and the application tests into the /build/test folder:

```
var tsProject = tsc.createProject({
    removeComments : false,
    noImplicitAny : false,
    target : "ES5",
    module : "commonjs",
    declarationFiles : false
});

gulp.task("build-source", function() {
    return gulp.src(__dirname + "/source/*.ts")
        .pipe(tsc(tsProject))
        .pipe(gulp.dest(__dirname + "/build/source/"));
});
```

The previous Gulp task compiles the TypeScript files under the source folder into JavaScript files that will be stored in inside the build/source folder. We should be able to run the task by executing the following command:

```
gulp build-source
```



The preceding command will fail if no source files are available. You can copy project source files from the companion source code or continue reading this chapter and create the files as we progress.

We will also declare a second task to compile our unit tests, but the output will be stored under the build/test folder:

```
var tsTestProject = tsc.createProject({
    removeComments : false,
    noImplicitAny : false,
    target : "ES5",
    module : "commonjs",
    declarationFiles : false
});

gulp.task("build-test", function() {
    return gulp.src(__dirname + "/test/*.test.ts")
        .pipe(tsc(tsTestProject))
        .pipe(gulp.dest(__dirname + "/build/test/"));
});
```

We should be able to run this new task using Gulp by using the following command:

```
gulp build-test
```

Once the JavaScript is under the build folder, we need to bundle the external modules (as we used { module : "commonjs" } in the preceding compiler settings) into bundled libraries that can be executed in a web browser.

Browserify needs a unique entry point for each library. For this reason, we are going to create three tasks—one for each bundled library.

We will create a task to bundle the application itself:

```
gulp.task("bundle-source", function () {
  var b = browserify({
    standalone : 'demos',
    entries: __dirname + "/build/source/demos.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("demos.js"))
    .pipe(buffer())
    .pipe(gulp.dest(__dirname + "/bundled/source/"));
}) ;
```

Just like we did with the previous Gulp tasks, we can invoke the new task by using the following command:

```
gulp bundle-source
```

We will also create another task to bundle all the unit tests in our application into a single bundled suite of tests:

```
gulp.task("bundle-test", function () {
  var b = browserify({
    standalone : 'test',
    entries: __dirname + "/build/test/bdd.test.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("bdd.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest(__dirname + "/bundled/test/"));
}) ;
```



The companion code has tests using both the TDD and BDD styles in two independent files named `tdd.test.ts` and `bdd.test.ts`. However, in the examples in this chapter, we will only focus on the BDD style.

We can invoke the new task by using the following command:

```
gulp bundle-test
```

Finally, we will create another task to bundle all the E2E tests in the application into a single bundled E2E test suite:

```
gulp.task("bundle-e2e-test", function () {  
  
  var b = browserify({  
    standalone : 'test',  
    entries: __dirname + "/build/test/e2e.test.js",  
    debug: true  
  }) ;  
  
  return b.bundle()  
    .pipe(source("e2e.test.js"))  
    .pipe(buffer())  
    .pipe(gulp.dest(__dirname + "/bundled/e2e-test/"));  
}) ;
```

We can invoke the new task by using the following command:

```
gulp bundle-e2e-test
```

Running the unit test with Karma

We have already covered the basics of Karma in *Chapter 2, Automating Your Development Workflow*. We are going to create a task to execute Karma:

```
gulp.task("run-unit-test", function(cb) {  
  karma.start({  
    configFile : __dirname + "/karma.conf.js",  
    singleRun: true  
  }, cb);  
}) ;
```

The Karma task configuration is really simple because the majority of the configuration is located in the karma.conf.js file, which is included in the companion code. Let's take a look at the configuration file:

```
module.exports = function (config) {
  'use strict';

  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
    coverageReporter: {
      type : 'lcov',
      dir : __dirname + '/coverage/'
    },
    plugins : [
      'karma-coverage',
      'karma-mocha',
      'karma-chai',
      'karma-sinon',
      'karma-phantomjs-launcher'
    ],
    preprocessors: {
      '**/bundled/test/bdd.test.js' : 'coverage'
    },
    files : [
      {
        pattern: "/bundled/test/bdd.test.js",
        included: true
      },
      {
        pattern: "/node_modules/jquery/dist/jquery.min.js",
        included: true
      },
      {
        pattern:
          "/node_modules/bootstrap/dist/js/bootstrap.min.js",
        included: true
      }
    ],
  },
}
```

```
client : {
  mocha : {
    ui : "bdd"
  }
},
port: 9876,
colors: true,
autoWatch: false,
logLevel: config.DEBUG
}) ;
} ;
```

If we take a look at the configuration file, we will see that we have configured the path where the tests are located and the browser that we want to use to run the test (PhantomJS). Declaring what browser we want to use is not enough; we also need to install a plugin so Karma can launch that browser.

Since we are going to write test using Mocha, Chai, and Sinon.JS, we have loaded the plugins to integrate Karma with each of these frameworks. There are many other popular testing frameworks, and the majority of them are compatible with Karma via the use of plugins.

Another interesting setting in the preceding configuration file is the client entry. We use it to configure the options of Mocha and indicate that we are going to use a BDD testing style.

When Karma executes the Mocha unit tests, it generates an HTML page internally and adds all the required files indicated in the files field as well as some files indicated by the plugins field. For the preceding example, Karma will generate an HTML page that will contain reference (using the `<script>` tags) to Mocha, Chai, and Sinon.JS (indicated by the plugins) as well as jQuery, Bootstrap, and the `bdd-test.js` file (indicated by the files field).

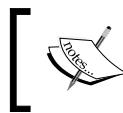


The companion source code includes the `package.json` file. We can use this file to run the `npm install` command and download all the third-party dependencies (including jQuery and Bootstrap).

It is important to understand that only files loaded via the files field will be available during the test execution, and that all the files will be loaded using a script tag. Sometimes, we may encounter issues related to missing files or parsing errors (when a non-JavaScript file is loaded using a script tag). We can have a better control over the file inclusion process using the settings pattern, included, served, and watched:

Settings	Description
pattern	The pattern to use to match files.
included	If autoWatch is true, all files that have set watched to true will be watched for changes.
served	Should the files be served by Karma's webserver?
watched	Should the files be included in the browser using the <script> tag? We will use false if we want to load them manually (for example, using RequireJS).

The karma.conf.js file also contains some settings to generate test coverage reports, but we will skip those for now and focus on them towards the end of the chapter.



Remember that you can find all the details about each field in the karma.conf.js file at <http://karma-runner.github.io/0.8/config/configuration-file.html>.



Running E2E tests with Selenium and Nightwatch.js

Karma (in combination with Mocha, Chai, and Sinon.JS) is a great tool when it comes to writing and executing unit tests and partial integration tests. However, Karma is not the best tool when it comes to writing E2E tests. For this reason, we will write a collection of E2E tests that will be written and executed using a separate set of tools: Selenium and Nightwatch.js.

To configure Nightwatch.js, we will start by creating a new Gulp task that will be in charge of the execution of the E2E tests. We only need to specify the location of an external configuration file named Nightwatch.js:

```
gulp.task('run-e2e-test', function() {
  return gulp.src('')
    .pipe(nightwatch({
      configFile: __dirname + '/nightwatch.json'
    }));
});
```



We are going to focus on Nightwatch.js because it is designed to work with the majority of frameworks; but if you are working with AngularJS, I would recommend you to take a look at Protractor. Protractor is a great E2E testing framework that has a high level of integration with AngularJS.

The `nightwatch.js` file contains the entire required configuration necessary to execute our E2E tests. We need to specify the location of the E2E test suites and the basic Selenium configuration.

We need to think that Selenium is more or less like Karma; it is a tool that can execute a unit test in a browser. The main difference is that Selenium allows us to write tests in a way that simulates much better how a real user would behave. It is important to understand that Nightwatch.js is not the tool directly in charge of execution of the test. Nightwatch.js is a framework that helps to write E2E tests and can communicate with Selenium to execute the tests.

In this case, we will tell Nightwatch.js not to run Selenium for us using the `start_process` entry in the `nightwatch.json` configuration file. The `nightwatch.json` file should look as follows:

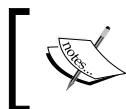
```
{
  "src_folders" : ["bundled/e2e-test/"],
  "output_folder" : "reports",
  "selenium" : {
    "start_process" : false
  },
  "test_settings" : {
    "default" : {
      "silent": true,
      "screenshots" : {
        "enabled" : true,
        "path" : "screenshots"
      },
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled" : true,
        "acceptSslCerts" : true
      }
    },
    "phantomjs" : {
      "desiredCapabilities": {
        "browserName": "phantomjs",
        "javascriptEnabled" : true,
      }
    }
  }
}
```

```
        "acceptSslCerts" : true,
        "phantomjs.binary.path" :
        "./node_modules/phantomjs/bin/phantomjs"
    },
},
"chrome" : {
    "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
    }
}
}
```

We will run Selenium manually using the selenium-standalone npm package (we can check the prerequisites section for installation details):

selenium-standalone start

Besides configuring Selenium, we need to configure which web browsers we are going to use during the execution of our E2E tests and to run the web application on a web server.



If you wish to learn more about all the available Nightwatch.js configuration parameters, please visit the official documentation at <http://nightwatchjs.org/guide#settings-file>.

Finally, to be able to run the E2E test, we will also need to run the application itself on a web server. As we saw in *Chapter 2, Automating Your Development Workflow*, we can use browserSync for that purpose; so we will add a task to deploy browserSync:

```
gulp.task('serve', function(cb) {
  browserSync({
    port: 8080,
    server: {
      baseDir: "./"
    }
  });
}

gulp.watch([
  "./**/*.js",
  "./**/*.css",
  "./index.html"
], browserSync.reload, cb);
});
```

If one test is failing and we don't know what is causing it to fail, we will be able to test it manually by running the application in a web browser.

It is important to run the tasks in the correct order. We need to open a console or terminal and start Selenium:

```
selenium-standalone start
```

Open another console or terminal and run the following commands:

```
gulp build-source  
gulp build-test  
gulp bundle-source  
gulp bundle-e2e-test  
gulp serve
```

Finally, open a third console and run the following command:

```
gulp run-e2e-test
```

Creating test assertions, specs, and suites with Mocha and Chai

Now that the test infrastructure is ready, we will start writing a unit test. We need to remember that we are going to follow the BDD development testing style, which means that we will write the test before we actually write the code.

We will write a web calculator; because we want to keep it simple, we will only implement one of its features. After doing some analysis, we have come up with a design interface that will help us to understand the requirements. We will declare the following interface in the `interfaces.d.ts` file:

```
interface MathInterface {  
    PI : number;  
    pow( base: number, exponent: number);  
}
```

As we can see, the calculator will allow us to calculate the exponent of a number and to get the number `PI`. Now that we know the requirements, we can start writing some unit tests. Let's create a file named `bdd.test.ts` and add the following code:

```
///<reference path="../typings/tsd.d.ts" />  
///<reference path="../source/interfaces.d.ts" />
```

```
import { MathDemo } from "../source/math_demo";

var expect = chai.expect;

describe('BDD test example for MathDemo class \n', () => {

  before(function(){ /* invoked once before ALL tests */ });
  after(function(){ /* invoked once after ALL tests */ });
  beforeEach(function(){ /* invoked once before EACH test */ });
  afterEach(function(){ /* invoked once before EACH test */ });

  it('should return the correct numeric value for PI \n', () => {
    var math : MathInterface = new MathDemo();
    expect(math.PI).to.equals(3.14159265359);
    expect(math.PI).to.be.a('number');
  });

  // ...
});

});
```

In the preceding code snippet, we have imported the necessary type definition files and an external module named `MathDemo`. This external module will declare the `MathDemo` class, which will implement the `MathInterface` that we are about to test.

We can also see a shortcut for `expect`, so we don't need to write `chai.expect` every time we need to invoke `expect`:

```
var expect = chai.expect;
```

Just below the shortcut we can find the first test suite:

```
describe('BDD test example for MathDemo class \n', () => {
```

Test suites are declared using the `describe()` function and are used to wrap a set of unit tests; and the unit tests themselves are declared using the `it()` function:

```
  it('should return the correct numeric value for PI \n', () => {
```

Inside the unit test, we can perform one or more assertions. The Chai assertions provide easily readable code thanks to the usage of a chainable style:

```
    expect(math.PI).to.equals(3.14159265359);
    expect(math.PI).to.be.a('number');
```

There are cases in which we will notice that we are repeating a certain test initialization logic across multiple unit tests within a test suite. There are some helper functions that we can use to avoid code duplication.

The `before()` function will be invoked before any test in the suite case is executed. The `after()` function will be executed after all the tests in the test suite have been executed:

```
before(function(){ /* invoked once before ALL tests */ });
after(function(){ /* invoked once after ALL tests */ });
```

The `beforeEach()` function is executed once (before the test is executed) for each test in the test suite, while the `afterEach()` function is executed once (after the test is executed) for each test in the test suite:

```
beforeEach(function(){ /* invoked once before EACH test */ });
afterEach(function(){ /* invoked once before EACH test */ });
```

If we run the test at this point, it will fail because the feature being tested (`PI`) is not implemented. Let's create a file named `math_demo.ts` and add the following code:

```
///<reference path="./interfaces.d.ts" />

class MathDemo implements MathInterface{
    public PI : number;

    constructor() {
        this.PI = 3.14159265359;
    }

    //...
}

export { MathDemo };
```

If we execute the test with Karma, it should pass without errors. It is important to run the tasks in the correct order. To do this, we need to open a console or terminal and run the following commands:

```
gulp build-source
gulp build-test
gulp bundle-source
gulp bundle-test
```

Finally, we can run the unit tests using the following command:

```
gulp run-unit-test
```

There was another requirement in the `MathInterface` interface, so we are going to repeat the entire BDD process once more; but this time, we will test a function named `pow` instead of a property. We will start by adding a new test to the test suite that we have previously created:

```
it('should return the correct numeric value for pow \n', () => {
  var math : MathInterface = new MathDemo();
  var result = math.pow(3,5);
  var expected = 243;
  expect(result).to.be.a('number');
  expect(result).to.equal(expected);
});
```

As we can see in the previously declared `MathInterface` interface, the function that we are going to test is named `pow` and takes two numeric arguments. So we have created a test that will create a new instance of `MathDemo` and invoke its `pow` method, passing the numeric values 3 and 5 as arguments. The expected value of calculating $3^3 \cdot 3^3 \cdot 3^3 \cdot 3^3$ is 243; for this reason, we have asserted that the `pow()` function returns a numeric value and its value is 243.

At this point, the preceding test will fail because the `pow` method has not been implemented. Let's return to the `math_demo.ts` file and implement the `pow` method:

```
///<reference path="./interfaces.d.ts" />

class MathDemo implements MathInterface{
  public PI : number;

  constructor() {
    this.PI = 3.14159265359;
  }

  public pow(base: number, exponent: number) {
    var result = base;
    for(var i = 1; i < exponent; i++){
      result = result * base;
    }
    return result;
  }

  // ...
}

export { MathDemo };
```

If we run the tests again, we will be able to see the number of tests that have been executed, how many of them have failed, and how long it took to finish the execution of all the tests:

```
Executed 2 of 2 SUCCESS (0.007 secs / 0.008 secs)
```

Testing the asynchronous code

In *Chapter 3, Working with Functions*, we learned how to work with a synchronous code; and in *Chapter 6, Application Performance*, we saw that using asynchronous code is one of the golden rules of web application performance. We should aim to write asynchronous code as much as we can, and for this reason, it is important to learn how to test asynchronous code.

Let's write an asynchronous version of the `pow` function to demonstrate how we can test an asynchronous function. We will start with the requirements:

```
interface MathInterface {  
    // ..  
    powAsync(base: number, exponent: number, cb : (result : number)  
    => void);  
}
```

We need to implement a function named `powAsync`, which takes two numeric values as parameters (just like before) and a callback function. The test for the asynchronous version is almost identical to the test that we wrote for the synchronous function:

```
it('should return the correct numeric value for pow (async) \n',  
(done) => {  
    var math : MathInterface = new MathDemo();  
    math.powAsync(3, 5, function(result) {  
        var expected = 243;  
        expect(result).to.be.a('number');  
        expect(result).to.equal(expected);  
        done(); // invoke done() inside your call back or fulfilled  
        promises  
    });  
});
```

The main thing that we need to notice is that, this time, the callback passed to the `it` method receives an argument named `done`. The argument is a function that we need to execute to indicate that the test execution is finished.

By default, the `it` method waits for the callback to return, but when testing asynchronous code, the function may return before the test execution is finished:

```
public powAsyncSlow(base: number, exponent: number, cb : (result : number) => void) {
    var delay = 45; //ms
    setTimeout(() => {
        var result = this.pow(base, exponent);
        cb(result);
    }, delay);
}
```

When testing asynchronous code, Mocha will consider the test as failed (timeout) if it takes more than 2,000 milliseconds to invoke the `done` function. The time limit before a timeout can be configured, as can be warnings for slow functions.

Mocha recommends that, when a function takes more than 40 milliseconds, we should consider investigating how to improve its performance. If the function execution takes over 100 milliseconds, we must investigate. Execution times of over 2,000 milliseconds are not tolerated by default.

Asserting exceptions

Asserting the types or values of variables is straightforward, as we have been able to explore in the previous examples; but there is one scenario that perhaps is not as intuitive as the previous one. This scenario is testing for an exception.

Let's add a new method to the `MathInterface` interface with the only purpose of illustrating how to test for an exception:

```
interface MathInterface {
    // ...
    bad(foo? : any) : void;
}
```

The `bad` method throws an exception when it is invoked with a non-numeric argument:

```
public bad(foo? : any) {
    if(isNaN(foo)){
        throw new Error("Error!");
    }
    else {
        //...
    }
}
```

In the following test, we can see how we can use Chai's expect API to assert that an exception is thrown:

```
it('should throw an exception when no parameters passed \n', () => {
  var math : MathInterface = new MathDemo();
  var throwsF = function() { math.bad(/* missing args */) };
  expect(throwsF).to.throw(Error);
});
```



If you wish to learn more about assertions, visit the Chai official documentation available at <http://chaijs.com/api/bdd/>.



TDD versus BDD with Mocha and Chai

TDD and BDD follow many of the same principles but have some differences in their style. While these two styles provide the same functionality, BDD is considered to be easier to read by many of the members of a software development team (not just developers).

The following table compares the naming and style of suites, tests, and assertions between the TDD and BDD styles:

TDD	BDD
suite	describe
setup	before
teardown	after
suiteSetup	beforeEach
suiteTeardown	afterEach
test	it
assert.equal(math.PI, 3.14159265359);	expect(math.PI).to.equals(3.14159265359);



In the companion code samples, you will find all the examples in this chapter following both the TDD and BDD styles.



Test spies and stubs with Sinon.JS

We have been working on the `MathDemo` class. We have implemented and tested its features using unit tests and assertions. Now we are going to create a little web widget that will internally use the `MathDemo` class to perform a mathematical operation. We can think of this new class as a graphical user interface for the `MathDemo` class. We need the following HTML:

```
<div id="widget">
  <input type="text" id="base" />
  <input type="text" id="exponent" />
  <input type="text" id="result" />
  <button id="submit" type="submit">Submit</button>
</div>
```



In the companion code, the HTML code contains more attributes, such as CSS classes; but they been have removed here for clarity.



Let's create a file named `calculator_widget.ts` under the source directory. We are going to store the HTML code in a string variable located in the scope of the web widget. The new class will be called `CalculatorWidget`, and it will implement the `CalculatorWidgetInterface` interface:

```
interface CalculatorWidgetInterface {
  render(id : string);
  onSubmit() : void;
}
```

We should write the unit test before we implement the `CalculatorWidget` class, but this time we will break the BDD rules in an attempt to facilitate the understanding of stubs and spies:

```
///<reference path="../interfaces.d.ts" />
///<reference path=".../typings/tsd.d.ts" />

var template = 'HTML...';

class CalculatorWidget implements CalculatorWidgetInterface{

  private _math : MathInterface;
  private $base: JQuery;
  private $exponent: JQuery;
  private $result: JQuery;
  private $btn: JQuery;
```

```

constructor(math : MathInterface) {
    if(math == null) throw new Error("Argument null exception!");
    this._math = math;
}

public render(id : string) {
    $(id).html(template);
    this.$base = $("#base");
    this.$exponent = $("#exponent");
    this.$result = $("#result");
    this.$btn = $("#submit");
    this.$btn.on("click", (e) => {
        this.onSubmit();
    });
}

public onSubmit() {
    var base = parseInt(this.$base.val());
    var exponent = parseInt(this.$exponent.val());

    if(isNaN(base) || isNaN(exponent)) {
        alert("Base and exponent must be a number!");
    }
    else {
        this.$result.val(this._math.pow(base, exponent));
    }
}
export { CalculatorWidget };

```

As we can see, we have defined a variable that contains the HTML that we previously examined but it is not displayed for brevity. A new class named `CalculatorWidget` is also defined together with the class constructor. We can observe that the class has two properties: a variable named `_dom` and an implementation of `MathInterface` named `_math`. We are depending on an interface because as we saw in *Chapter 4, Object-Oriented Programming with TypeScript*, it is a good practice (dependency inversion principle) to do so.

Notice that the class constructor takes an implementation of `MathInterface` as its only argument. Passing the dependencies of a component via its constructor is also a good practice and is used to reduce the coupling between components.

The first method in the class is named `render` and takes the ID (string) of an HTML element as its only argument. The ID is used to select the node that matches the ID using a jQuery selector. Once it has been selected, the HTML that we previously examined is inserted into the selected node. We can say that the component is in charge of rendering its own HTML and can be reused easily just by changing its container. This is how web widgets usually work: they are independent components that can be considered as reusable standalone applications within a parent application that is no more than just a collection of web widgets.

After rendering the HTML, the `render` method creates shortcuts for each component of the widget's form and initializes a click event listener:

```
public render(id : string) {  
    $(id).html(template);  
    this._dom.$base = $("#base");  
    this._dom.$exponent = $("#exponent");  
    this._dom.$result = $("#result");  
    this._dom.$btn = $("#submit");  
  
    this._dom.$btn.on("click", (e) => {  
        this.onSubmit();  
    });  
}
```

When a user clicks on the button with id equals to `submit`, an event is triggered, and the event listener invokes the `onSubmit` function that we can find in the following code snippet. This function will read the values for base and exponent using the shortcuts previously declared:

```
public onSubmit() {  
    var base = parseInt(this._dom.$base.val());  
    var exponent = parseInt(this._dom.$exponent.val());  
  
    if(isNaN(base) || isNaN(exponent)) {  
        alert("Base and exponent must be a number!");  
    }  
    else {  
        this._dom.$result.val(this._math.pow(base, exponent));  
    }  
}
```

If the values of the inputs (base and exponent) are not numeric values, an alert message is displayed to provide the users with error feedback. If the values are numeric, the `pow` method of the `MathDemo` class is invoked, and the result is assigned to the `result` field value via one of the previously created shortcuts.

Writing unit tests can become a complex task when the components being tested are highly coupled with other components. In the previous section, we tried to follow some good practices such as the dependency inversion principle or injecting dependencies via the constructor of the dependent; but sometimes, even when using good practices, we will have to deal with highly coupled code.

Spies, mocks, and stubs can help us to take away some of the pain caused by highly coupled components. These features can also help us to identify the root cause of an issue. If we replace all the dependencies of a component with stubs and a test fail, we will know that the issue is located in the component being tested and not in one of its dependencies.

For example, the `CalculatorWidget` class has a dependency on the `MathDemo` class. If there is an issue in the calculator website, we will not be able to know if the root cause of the issue is located in the `CalculatorWidget` class or the `MathDemo` class. However, if we write some unit tests for the `CalculatorWidget` class in isolation (replacing its `MathDemo` dependency with a stub) and some of the tests fail, we will know for sure that the root cause of the issue is located in the `CalculatorWidget` and not in the `MathDemo` class.

Let's take a look at some test examples.

Spies

We are going to start by taking a look at the use of spies by creating a new test suite. This time we will use the `before()` and `beforeEach()` functions. When the `before()` function is invoked (before any unit test is executed), a new HTML node is created to hold the widget's HTML.

The `beforeEach()` function is used to reset the container before each test. This way, we can ensure that a new widget is created for each test in the test suite. This is a good idea because it will prevent one test from potentially affecting the results of another.

```
describe('BDD test example for CalculatorWidget class \n', () => {  
  before(function() {  
    $("body").append('<div id="widget"/>');  
  });  
});
```

```
beforeEach(function() {
  $("#widget").empty();
});
```



Usually, testing frameworks (regardless of the language we are working with) won't allow us to control the order in which the unit tests and test suites are executed. The tests can even be executed in parallel by using multiple threads. For this reason, it is important to ensure that the unit tests in our test suites are independent of each other.

Now that the test suite is ready, we can create unit tests for the `render()` and `onSubmit()` methods. The test starts by the creation of an instance of `MathDemo`, which is then passed to `CalculatorWidget` constructor to create a new instance named `calculator`.

The `render` method is then invoked to render the widget inside the HTML node with the ID `widget`. The HTML node should be available at this stage because it was created by the `before()` method. After the widget has been rendered, a value is set for the inputs with IDs `base` and `exponent`.

The test specification (`onSubmit` should be invoked when `#submit` is clicked) should help us understand that we are testing the click event. We are going to use a spy to observe the `onSubmit()` function; so, when the button with ID `submit` is clicked, the spy will detect that the `onSubmit()` function was invoked.

To finish the test, we are going to trigger a click event on the button with ID `submit` and assert that the `onSubmit()` function was actually only invoked once:

```
it('onSubmit should be invoked when #submit is clicked', () => {
  var math : MathInterface = new MathDemo();
  var calculator = new CalculatorWidget(math);
  calculator.render("#widget");
  $('#base').val("2");
  $('#exponent').val("3");

  // spy on onSubmit
  var onSubmitSpy = sinon.spy(calculator, "onSubmit");
  $("#submit").trigger("click");

  // assert calculator.onSubmit was invoked when click on #submit
  expect(onSubmitSpy.called).to.equal(true);
  expect(onSubmitSpy.callCount).to.equal(1);
  expect($("#result").val()).to.equal("8");
}) ;
```

Spies will allow us to perform many operations: from checking how many times a function has been invoked to checking if it was invoked using the new operator, or if it was invoked with a set of specific parameters.

The last assertion helps us guarantee that `onSubmit()` is setting the correct result in the result input.



All the possible operations are detailed in the Sinon.JS online documentation found at <http://sinonjs.org/docs/#sinonspy>.

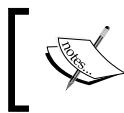
Stubs

It may look like we have already tested the entire application by now, but that is usually never the case. Let's analyze what exactly we have tested so far:

- We have tested the entire `MathDemo` class, and we know that it returns the correct value when `pow` is invoked
- We know that the `CalculatorWidget` class is rendering the HTML correctly
- We know that the `CalculatorWidget` class is setting up some events and reading some values from the HTML inputs as expected

So far, we have created some tests for the `MathDemo` class and the `CalculatorWidget` class, but we have forgotten to test the integration between them.

We have been testing using 2 as base and 3 as exponent, but if we wrongly used the same value as base and exponent, we could have missed one potential issue: maybe the `CalculatorWidget` class is passing the arguments in incorrect order to the `MathDemo` class when the function `pow()` is invoked in the body of the `onSubmit()` function.



Later on in this chapter, we will see how to generate a kind of report (a test coverage report) that can help us to identify areas of our application that have not been tested.

We can test this scenario by isolating the `CalculatorWidget` class from its dependency on the `MathDemo` class. We can achieve this by using a stub. Let's take a look at the upcoming unit tests to see a stub in action.

At the beginning of the method, a new instance of MathDemo is created, and a stub is used against its pow method. The stub will replace the pow method with a new method. The new method will assert that the parameters received are in the correct order:

```
it('pass the right args to Math.pow', (done) => {
  var math : MathInterface = new MathDemo();

  // replace pow method with a method for testing
  sinon.stub(math, "pow", function(a, b) {
    // assert that CalculatorWidget.onSubmit invokes
    // math.pow with the right arguments
    expect(a).to.equal(2);
    expect(b).to.equal(3);
    done();
  });

  var calculator = new CalculatorWidget(math);
  calculator.render("#widget");
  $('#base').val("2");
  $('#exponent').val("3");

  $("#submit").trigger("click");
}) ;
```

Once the stub is ready, a new instance of the CalculatorWidget class is created, but instead of passing a normal instance of MathDemo as its only argument, we are injecting the stub. By doing this, we are no longer testing the MathDemo class, and we are testing the CalculatorWidget class in an isolated environment. This would have been much more complicated without a design that facilitates replacing the class dependencies via a constructor injection.

To finish the test, we render the calculator widget, set the value of the inputs with IDs base and exponent, and trigger a click on the button with ID submit. The event will invoke the onSubmit function, which will then invoke the pow method. When the parameters are in the incorrect order, we will be able to be 100 percent sure about the location of the root cause of this issue: the onSubmit function.

Creating end-to-end tests with Nightwatch.js

Writing an E2E test with Nightwatch.js is an intuitive process. We should be able to read an E2E test and be able to understand it even if it is the first time that we encounter one.

If we take a look at the following code snippet, we will see that, once we have reached the page, the test will wait 1 second for the body of the page to be visible. The test will then wait 0.1 seconds for some elements to be visible. The elements can be selected using CSS selectors or XPath syntax. If the elements are visible, the `setValue` method will insert 2 in the text input with base as ID and 3 in the text input with exponent as ID:

```
var test = {
  'Calculator pow e2e test example' : function (client) {
    client
      .url('http://localhost:8080/')
      .waitForElementVisible('body', 1000)
      .assert.waitForElementVisible('TypeScriptTesting', 100)
      .assert.waitForElementVisible('input#base', 100)
      .assert.waitForElementVisible('input#exponent', 100)
      .setValue('input#base', '2')
      .setValue('input#exponent', '3')
      .click('button#submit')
      .pause(100)
      .assert.value('input#result', '8')
      .end();
  }
};

export = test;
```

The test will then find the submit button and trigger an on-click event. After 0.1 seconds, the test asserts that the correct value has been inserted into the text input with result as ID. We can see each of these steps in the console during the test execution.

We can run the tests using the following command:

```
gulp run-e2e-test
```



Remember that we must run the tasks to compile and bundle the E2E tests as well as run the application in a web server with BrowserSync and execute Selenium before being able to run E2E tests.

Generating test coverage reports

Earlier in this chapter, when we configured Karma, we added some settings to generate test coverage reports. Let's take a look at the `karma.conf.js` file to identify test coverage-related configuration:

```
module.exports = function (config) {
  'use strict';

  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
    coverageReporter: {
      type : 'lcov',
      dir : __dirname + '/coverage/'
    },
    plugins : [
      'karma-coverage',
      'karma-mocha',
      'karma-chai',
      'karma-sinon',
      'karma-phantomjs-launcher'
    ],
    preprocessors: {
      '**/bundled/test/bdd.test.js' : 'coverage'
    },
    files : [
      {
        pattern: "/bundled/test/bdd.test.js",
        included: true
      },
      {
        pattern: "/node_modules/jquery/dist/jquery.min.js",
        included: true
      },
      {
        pattern:
          "/node_modules/bootstrap/dist/js/bootstrap.min.js",
        included: true
      }
    ],
    client : {
      mocha : {
        ui : "bdd"
      }
    },
  });
}
```

```
port: 9876,
colors: true,
autoWatch: false,
logLevel: config.DEBUG
}) ;
};
```

As we can see, we need to set the folder in which the test coverage report will be stored. We also need to add coverage to the reporter's setting and a new entry named coverageReport to configure the format of the report.

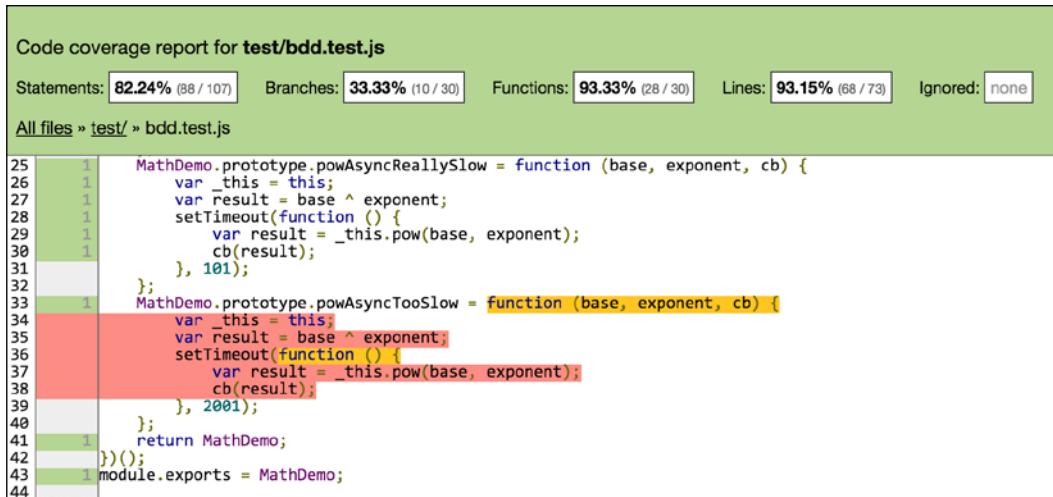
We cannot forget to install the karma-coverage plugin using npm and adding a reference in the karma.conf.js under the plugins field. Finally, we need to add coverage to the preprocessor field:

```
npm karma-coverage
```

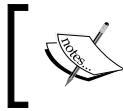
To generate the report, we just need to execute the Gulp tasks used to run all the unit tests in the application. We can do so by using the following command:

```
gulp run-unit-test
```

Once the execution of the test has been completed, we can open the folder in which we decided to store the coverage reports and open the available index.html file in a web browser. The HTML report allows us to navigate to the coverage statistics of a specific file by clicking on the name of one of the source files.



The report can help us to identify with ease the parts of our code that have not been tested (lines are highlighted in red). The test coverage report also calculates the number of lines tested against the number of lines in the application. As we can see in the preceding screenshot, only 82.24 percent of the statements are tested in the example.



If you would like to learn more about all the tools that we have discussed in this chapter, I highly recommend checking out the book *Backbone.js Testing* written by *Ryan Roemer*.

Summary

In this chapter, we discussed some core testing concepts (including stubs, spies, suites, and more). We also looked at the test-driven development and behavior-driven development approaches and how to work with some of the leading JavaScript testing frameworks, such as Mocha, Chai, Sinon.JS, Karma, Selenium, and Nightwatch.js.

Towards the end of the chapter we explored how to test across multiple devices and how to generate test coverage reports.

In the next chapter, we will look at decorators and the metadata reflection API – two exciting new features introduced by TypeScript 1.5.

8

Decorators

In this chapter, you are going to learn about annotations and decorators – the two new features based on the future ECMAScript 6 specification, but we can use them today with TypeScript 1.5.

You will learn about the following topics:

- Annotations and decorators:
 - Class decorators
 - Method decorators
 - Property decorators
 - Parameter decorators
 - Decorator factory
 - Decorators with parameters
- The reflection metadata API

Prerequisites

The TypeScript features in this chapter require TypeScript 1.5 or higher. We can use Gulp as we have done in previous chapters, but we need to ensure that the latest version of TypeScript is used by the `gulp-typescript` package. Let's start by creating a package.json file and installing the required packages:

```
npm init
npm install --save-dev gulp gulp-typescript typescript
npm install --save reflect-metadata
```

Once we have installed the packages, we can create a `gulpfile.js` file and add a new task to compile our code.

The following code snippet shows the required compiler configuration. The compilation target must be ES5 and the `emitDecoratorMetadata` setting must be set as `true`. We also need to specify the package that provides the TypeScript compiler to ensure that the latest version is used:

```
var gulp      = require("gulp"),
    tsc       = require("gulp-typescript"),
    typescript = require("typescript");

var tsProject = tsc.createProject({
    removeComments : false,
    noImplicitAny : false,
    target : "es5",
    module : "commonjs",
    declarationFiles : false,
    emitDecoratorMetadata : true,
    typescript: typescript
});
```

Once the compiler settings are ready, we can write a gulp task using the `gulp-typescript` plugin:

```
gulp.task("build-source", function() {
    return gulp.src(__dirname + "/file.ts")
        .pipe(tsc(tsProject))
        .js.pipe(gulp.dest(__dirname + "/"));
});
```

Annotations and decorators

Annotations are a way to add metadata to class declarations. The metadata can then be used by tools such as dependency injection containers.

The annotations API was proposed by the Google AtScript team but annotations are not a standard. However, decorators are a proposed standard for ECMAScript 7 by Yehuda Katz, to annotate and modify classes and properties at design time.

Annotations and decorators are pretty much the same:

Annotations and decorators are nearly the same thing. From a consumer perspective we have exactly the same syntax. The only thing that differs is that we don't have control over how annotations are added as metadata to our code. A decorator is rather an interface to build something that ends up as annotation.

Over a long term, however, we can just focus on decorators, since those are a real proposed standard. AtScript is TypeScript and TypeScript implements decorators.

- "The difference between Annotations and Decorators" by Pascal Precht

We are going to use the following class to showcase how to work with decorators:

```
class Person {  
  
    public name: string;  
    public surname: string;  
  
    constructor(name : string, surname : string) {  
        this.name = name;  
        this.surname = surname;  
    }  
  
    public saySomething(something : string) : string {  
        return this.name + " " + this.surname + " says: " + something;  
    }  
}
```

There are four types of decorators that can be used to annotate: classes, properties, methods, and parameters.

The class decorators

The official TypeScript decorator proposal defines a class decorator as follows:

A class decorator function is a function that accepts a constructor function as its argument, and returns either undefined, the provided constructor function, or a new constructor function. Returning undefined is equivalent to returning the provided constructor function.

- "Decorators Proposal – TypeScript" by Ron Buckton

A class decorator is used to modify the constructor of class in some way. If the class decorator returns undefined, the original constructor remains the same. If the decorator returns, the return value will be used to override the original class constructor.

We are going to create a class decorator named `logClass`. We can start by defining the decorator as follows:

```
function logClass(target: any) {  
    // ...  
}
```

The class decorator above does not have any logic yet, but we can already apply it to a class. To apply a decorator, we need to use the at (@) symbol:

```
@logClass  
class Person {  
    public name: string;  
    public surname: string;  
    //...
```

If we have declared and applied a decorator, a function named `__decorate` will be generated by the TypeScript compiler, which will then compile our code in JavaScript. We are not going to examine the internal implementation of the `__decorate` function, but we need to understand that it is used to apply a decorator at runtime. We can see it in action by examining the JavaScript code that is generated when we compile the decorated `Person` class mentioned previously:

```
var Person = (function () {  
    function Person(name, surname) {  
        this.name = name;  
        this.surname = surname;  
    }  
    Person.prototype.saySomething = function (something) {  
        return this.name + " " + this.surname + " says: " +  
            something;  
    };  
    Person = __decorate([  
        logClass  
    ], Person);  
    return Person;  
})();
```

Now that we know how the class decorator will be invoked, let's implement it:

```
function logClass(target: any) {  
  
    // save a reference to the original constructor  
    var original = target;  
  
    // a utility function to generate instances of a class  
    function construct(constructor, args) {  
        var c : any = function () {  
            return constructor.apply(this, args);  
        }  
        c.prototype = constructor.prototype;  
        return new c();  
    }  
  
    // the new constructor behaviour  
    var f : any = function (...args) {  
        console.log("New: " + original.name);  
        return construct(original, args);  
    }  
  
    // copy prototype so instanceof operator still works  
    f.prototype = original.prototype;  
  
    // return new constructor (will override original)  
    return f;  
}
```

The class decorator takes the constructor of the class being decorated as its only argument. This means that the argument (named `target`) is the constructor of the `Person` class.

The decorator starts by creating a copy of the class constructor, then it defines a utility function (named `construct`) that can be used to generate instances of a class.

Decorators are used to add some extra logic or metadata to the decorated element. When we try to extend the functionality of a function (methods or constructors), we need to wrap the original function with a new function that contains the additional logic and invokes the original function.

In the preceding decorator, we added extra logic to log in the console, the name of the class when a new instance is created. To achieve this, a new class constructor (named `f`) was declared. The new constructor contains the additional logic and uses the `construct` function to invoke the original class constructor.

At the end of the decorator, the prototype of the original constructor function is copied to the new constructor function to ensure that the `instanceof` operator continues to work when it is applied to an instance of the decorated class. Finally, the new constructor is returned and some code generated by the TypeScript compiler uses it to override the original class constructor.

After decorating the class constructor, a new instance is created:

```
var me = new Person("Remo", "Jansen");
```

On doing so, the following text appears in the console:

```
"New: Person"
```

The method decorators

The official TypeScript decorator proposal defines a method decorator as follows.

A method decorator function is a function that accepts three arguments:

The object that owns the property, the key for the property (a string or a symbol), and optionally the property descriptor of the property. The function must return either undefined, the provided property descriptor, or a new property descriptor. Returning undefined is equivalent to returning the provided property descriptor.

- "Decorators Proposal - TypeScript" by Ron Buckton

The method decorator is really similar to the class decorator but it is used to override a method, as opposed to using it to override the constructor of a class.

If the method decorator returns a value different from `undefined`, the returned value will be used to override the property descriptor of the method.



Note that a property descriptor is an object that can be obtained by invoking the `Object.getOwnPropertyDescriptor()` method.

Let's declare a method decorator named `logMethod` without any behavior for now:

```
function logMethod(target: any, key: string, descriptor: any) {  
    // ...  
}
```

We can apply the decorator to one of the methods in the Person class:

```
//...
@logMethod
public saySomething(something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
}
// ...
```

The method decorator is invoked using the following arguments:

- The prototype of the class that contains the method being decorated is `Person.prototype`
- The name of the method being decorated is `saySomething`
- The property descriptor of the method being decorated is `Object.getOwnPropertyDescriptor(Person.prototype, saySomething)`

Now that we know the value of the decorator parameters, we can proceed to implement it:

```
function logMethod(target: any, key: string, descriptor: any) {

    // save a reference to the original method
    var originalMethod = descriptor.value;

    // editing the descriptor/value parameter
    descriptor.value = function (...args: any[]) {

        // convert method arguments to string
        var a = args.map(a => JSON.stringify(a)).join();

        // invoke method and get its return value
        var result = originalMethod.apply(this, args);

        // convert result to string
        var r = JSON.stringify(result);

        // display in console the function call details
        console.log(`Call: ${key}(${a}) => ${r}`);

        // return the result of invoking the method
        return result;
    }

    // return edited descriptor
    return descriptor;
}
```

Just like we did when we implemented the class decorator, we start by creating a copy of the element being decorated. Instead of accessing the method via the class prototype (`target["key"]`), we will access it via the property descriptor (`descriptor.value`).

We then create a new function that will replace the method being decorated. The new function invokes the original method but also contains some additional logic used to log in the console, the method name, and the value of its arguments every time it is invoked.

After applying the decorator to the method, the method name and arguments will be logged in the console when it is invoked:

```
var me = new Person("Remo", "Jansen");
me.saySomething("hello!");
// Call: saySomething("hello!") => "Remo Jansen says: hello!"
```

The property decorators

The official TypeScript decorator proposal defines a property decorator as follows:

A property decorator function is a function that accepts two arguments: The object that owns the property and the key for the property (a string or a symbol). A property decorator does not return.

- "Decorators Proposal – TypeScript" by Ron Buckton

A property decorator is really similar to a method decorator. The main differences are that a property decorator doesn't return a value and that the third parameter (the property descriptor) is not passed to the property decorator.

Let's create a property decorator named `logProperty` to see how it works:

```
function logProperty(target: any, key: string) {
  // ...
}
```

We can use it in one of the `Person` class's properties as follows:

```
class Person {
  @logProperty
  public name: string;
  // ...
```

As we have been doing so far, we are going to implement a decorator that will override the decorated property with a new property that will behave exactly as the original one, but will perform an additional task—logging the property value in the console whenever it changes:

```
function logProperty(target: any, key: string) {  
  
    // property value  
    var _val = this[key];  
  
    // property getter  
    var getter = function () {  
        console.log(`Get: ${key} => ${_val}`);  
        return _val;  
    };  
  
    // property setter  
    var setter = function (newVal) {  
        console.log(`Set: ${key} => ${newVal}`);  
        _val = newVal;  
    };  
  
    // Delete property. The delete operator throws  
    // in strict mode if the property is an own  
    // non-configurable property and returns  
    // false in non-strict mode.  
    if (delete this[key]) {  
        Object.defineProperty(target, key, {  
            get: getter,  
            set: setter,  
            enumerable: true,  
            configurable: true  
        });  
    }  
}
```

In the preceding decorator, we created a copy of the original property value and declared two functions: `getter` (invoked when we change the value of the property) and `setter` (invoked when we read the value of the property) respectively.

In the previous decorator, the return value was used to override the element being decorated. Because the property decorator doesn't return a value, we can't override the property being decorated but we can replace it. We have manually deleted the original property and created a new property using the `Object.defineProperty` function and the previously declared `getter` and `setter` functions.

After applying the decorator to the name property, we will be able to observe any changes to its value in the console:

```
var me = new Person("Remo", "Jansen");
// Set: name => Remo
me.name = "Remo H.";
// Set: name => Remo H.
var n = me.name;
// Get: name Remo H.
```

The parameter decorators

The official decorator proposal defines a parameter decorator as follows:

A parameter decorator function is a function that accepts three arguments: The object that owns the method that contains the decorated parameter, the property key of the property (or undefined for a parameter of the constructor), and the ordinal index of the parameter. The return value of this decorator is ignored.

Decorators Proposal – TypeScript by Ron Buckton

Let's create a parameter decorator named addMetadata to see how it works:

```
function addMetadata(target: any, key : string, index : number) {
    // ...
}
```

We can apply the property decorator to a parameter as follows:

```
public saySomething(@addMetadata something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
}
```

The parameter decorator doesn't return, which means that we will not be able to override the method that contains the parameter being decorated.

We can use parameter decorators to add some metadata to the prototype (`target`) class. In the following implementation, we will add an array named `log_${key}_parameters` as a class property where `key` is the name of the method that contains the parameter being decorated:

```
function addMetadata(target: any, key : string, index : number) {
    var metadataKey = `log_${key}_parameters`;
    if (Array.isArray(target[metadataKey])) {
        target[metadataKey].push(index);
    }
}
```

```
    else {
      target[metadataKey] = [index];
    }
}
```

To allow more than one parameter to be decorated, we check whether the new field is an array. If the new field is not an array, we create and initialize the new field to be a new array containing the index of the parameter being decorated. If the new field is an array, the index of the parameter being decorated is added to the array.

A parameter decorator is not really useful on its own; it needs to be combined with a method decorator, so the parameter decorator adds the metadata and the method decorator reads it:

```
@readMetadata
public saySomething(@addMetadata something : string) : string {
  return this.name + " " + this.surname + " says: " + something;
}
```

The following method decorator works like the method decorator that we implemented previously in this chapter, but it will read the metadata added by the parameter decorator and instead of displaying all the arguments passed to the method in the console when it is invoked, it will only log the ones that have been decorated:

```
function readMetadata (target: any, key: string, descriptor: any) {

  var originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {

    var metadataKey = `_log_${key}_parameters`;
    var indices = target[metadataKey];

    if (Array.isArray(indices)) {

      for (var i = 0; i < args.length; i++) {

        if (indices.indexOf(i) !== -1) {

          var arg = args[i];
          var argStr = JSON.stringify(arg) || arg.toString();
          console.log(`${key} ${arg[$i]}: ${argStr}`);
        }
      }
    }

    var result = originalMethod.apply(this, args);
    return result;
  }
}
```

```
        }
    }
    return descriptor;
}
```

If we apply the `saySomething` method:

```
var person = new Person("Remo", "Jansen");
```

```
person.saySomething("hello!");
```

The `readMetadata` decorator will display the value of the parameters that were added to the metadata (the class property named `_log_saySomething_parameters`) in the console by the `addMetadata` decorator:

```
saySomething arg[0]: "hello!"
```

Note that, in the previous example, we used a class property to store some metadata. Later in this chapter, you will learn how to use the reflection metadata API; this API has been designed specifically to generate and read metadata and it is, therefore, recommended to use it when we need to work with decorators and metadata.



The decorator factory

The official decorator proposal defines a decorator factory as follows:

A decorator factory is a function that can accept any number of arguments, and must return one of the above types of decorator function.

Decorators Proposal – TypeScript by Ron Buckton

You learned to implement class, property, method, and parameter decorators. In the majority of cases, we will consume decorators, not implement them. For example, in Angular 2.0, we will use an `@view` decorator to declare that a class will behave as a View, but we will not implement the `@view` decorator ourselves.

We can use the decorator factory to make decorators easier to consume. Let's consider the following code snippet:

```
@logClass
class Person {

    @logProperty
    public name: string;
```

```

public surname: string;

constructor(name : string, surname : string) {
    this.name = name;
    this.surname = surname;
}

@logMethod
public saySomething(@logParameter something : string) : string {
    return this.name + " " + this.surname + " says: " + something;
}
}

```

The problem with the preceding code is that we, as developers, need to know that the `logMethod` decorator can only be applied to a method. This might seem trivial because the decorator naming used above makes it easier for us.

A better solution is to enable developers to use an `@log` decorator without having to worry about using the right kind of decorator:

```

@log
class Person {

    @log
    public name: string;
    public surname: string;

    constructor(name : string, surname : string) {
        this.name = name;
        this.surname = surname;
    }

    @log
    public saySomething(@log something : string) : string {
        return this.name + " " + this.surname + " says: " + something;
    }
}

```

We can achieve this by creating a decorator factory. A decorator factory is a function that is able to identify what kind of decorator is required and return it:

```

function log(...args : any[]) {
    switch(args.length) {
        case 1:
            return logClass.apply(this, args);
    }
}

```

```

case 2:
    // break instead of return as property
    // decorators don't have a return
    logProperty.apply(this, args);
    break;
case 3:
    if(typeof args[2] === "number") {
        logParameter.apply(this, args);
    }
    return logMethod.apply(this, args);
default:
    throw new Error("Decorators are not valid here!");
}
}

```

As we can observe in the preceding code snippet, the decorator factory uses the number and type of arguments passed to the decorator to identify the required kind of decorator.

Decorators with arguments

We can use a special kind of decorator factory to allow developers to configure the behavior of a decorator. For example, we could pass a string to a class decorator as follows:

```

@logClass("option")
class Person {
    // ...

```

In order to be able to pass some parameters to a decorator, we need to wrap the decorator with a function. The wrapper function takes the parameters of our choice and returns a decorator:

```

function logClass(option : string) {
    return function (target: any) {

        // class decorator logic goes here
        // we have access to the decorator parameters
        console.log(target, option);
    }
}

```

This can be applied to all the kinds of decorator that you learned about in this chapter.

The reflection metadata API

You learned that decorators can be used to modify and extend the behavior of a class's methods or properties. You also learned that we can use decorators to add metadata to the class being decorated.

For less experienced developers, the possibility of adding metadata to a class might not seem really useful or exciting but it is one of the greatest things that has happened to JavaScript in the past few years.

As we already know, TypeScript only uses types at design time. However, some features such as dependency injection, runtime type assertions, reflection, and testing are not possible without the type information being available at runtime. This is not a problem anymore because we can use decorators to generate metadata and that metadata can contain type information. The metadata can then be processed at runtime.

When the TypeScript team started to think about the best possible way to allow developers to generate type information metadata, they reserved a few special decorator names for this purposes.

The idea was that, when an element was decorated using these reserved decorators, the compiler would automatically add the type information to the element being decorated. The reserved decorators were the following:

TypeScript compiler will honor special decorator names and will flow additional information into the decorator factory parameters annotated by these decorators.

@type – The serialized form of the type of the decorator target

@returnType – The serialized form of the return type of the decorator target if it is a function type, undefined otherwise

@parameterTypes – A list of serialized types of the decorator target's arguments if it is a function type, undefined otherwise

@name – The name of the decorator target

- "Decorators brainstorming" by Jonathan Turner

Shortly after, the TypeScript team decided to use the reflection metadata API (one of the proposed ES7 features) instead of the reserved decorators.

The idea is almost identical but instead of using the reserved decorator names, we will use some reserved metadata keys to retrieve the metadata using the reflection metadata API. The TypeScript documentation defines three reserved metadata keys:

Type metadata uses the metadata key "design:type".

Parameter type metadata uses the metadata key "design:paramtypes".

Return type metadata uses the metadata key "design:returntype".

- Issue #2577 - *TypeScript Official Repository at GitHub.com*

Let's see how we can use the reflection metadata API. We need to start by referencing and importing the required `reflect-metadata` npm package:

```
/// <reference path="./node_modules/reflect-metadata/reflect-
metadata.d.ts"/>
import 'reflect-metadata';
```

We can then create a class for testing purposes. We are going to get the type of one of the class properties at runtime. We are going to decorate the class using a property decorator named `logType`:

```
class Demo {
    @logType
    public attr1: string;
}
```

Instead of using a reserved decorator, `@type`, we need to invoke the `Reflect.getMetadata()` method and pass the `design:type` key. The types are returned as functions, for example, for the type `string`, the function `String() {}` function is returned. We can use the `function.name` property to get the type as a string:

```
function logType(target: any, key: string) {
    var t = Reflect.getMetadata('design:type', target, key);
    console.log(`#${key} type: ${t.name}`);
}
```

If we compile the preceding code and run the resulting JavaScript code in a web browser, we will be able to see the type of the `attr1` property in the console:

```
'attr1 type: String'
```



Remember that, in order to run this example, the `reflect-metadata` library must be imported.

We can apply the other reserved metadata keys in a similar manner. Let's create a method with many parameters to use the `design:paramtypes` reserved metadata key to retrieve the types of the parameters

```
class Demo {  
    @logParamTypes  
    public doSomething(  
        param1 : string,  
        param2 : number,  
        param3 : Foo,  
        param4 : { test : string },  
        param5 : IFoo,  
        param6 : Function,  
        param7 : (a : number) => void  
    ) : number {  
  
        return 1;  
    }  
}
```

This time, we will use the `design:paramtypes` reserved metadata key, and because we are querying the types of multiple parameters, the types will be returned as an array by the `Reflect.getMetadata()` function:

```
function logParamTypes(target : any, key: string) {  
    var types = Reflect.getMetadata('design:paramtypes', target, key);  
    var s = types.map(a => a.name).join();  
    console.log(` ${key} param types: ${s}`);  
}
```

If we compile and run the preceding code in a web browser, we will be able to see the types of the parameters in the console:

```
'doSomething param types: String, Number, Foo, Object, Object,  
Function, Function'
```

The types are serialized and follow some rules. We can see that functions are serialized as `Function`, objects literals (`{test : string}`) and interfaces are serialized as `Object`, and so on:

Type	Serialized
void	undefined
string	String
number	Number

Type	Serialized
boolean	Boolean
symbol	Symbol
any	Object
enum	Number
Class C{}	C
Object literal { }	Object
interface	Object

 Note that some developers have required the possibility of accessing the type of interfaces and the inheritance tree of a class via metadata. This feature is known as **complex type serialization** and is not available at the time of writing this book, but the TypeScript team has already started to work on it.

To conclude, we are going to create a method with a return type and use the `design:returntype` reserved metadata key to retrieve the types of the return type:

```
class Demo {
  @logReturnType
  public doSomething2() : string {
    return "test";
  }
}
```

Just like in the two previous decorators, we need to invoke the `Reflect.getMetadata()` function, passing the `design:returntype` reserved metadata key:

```
function logReturnType(target, key) {
  var returnType = Reflect.getMetadata('design:returntype', target,
  key);
  console.log(`#${key} return type: ${returnType.name}`);
}
```

If we compile and run the preceding code in a web browser, we will be able to see the types of the return type in the console:

```
'doSomething2 return type: String'
```

Summary

In this chapter, you learned how to consume and implement the four available types of decorators (class, method, property, and parameter) and how to create a decorator factory to abstract developers from the decorator types when they are consumed.

You also learned how to use the reflection metadata API to access type information at runtime.

In the next chapter, you will learn about the architecture of a TypeScript application. You will also learn about how to work with some design patterns and how to create a single-page web application.

9

Application Architecture

In previous chapters, we have covered several aspects of TypeScript, and we should now feel confident enough to create a small application.

As we know, TypeScript was created by Microsoft to facilitate the creation of large-scale JavaScript applications. Some TypeScript features such as modules or classes can facilitate the process of creating large applications, but it is not enough. We need good application architecture if we want to succeed in the long term.

This chapter is divided into two main parts. In the first part, we are going to look at the **single-page application (SPA)** architecture and some design patterns that will help us create scalable and maintainable applications. This section covers the following topics:

- The single-page web application architecture
- The MV^{*} architecture
- Models and collections
- Item views and collection views
- Controllers
- Events
- Router and hash navigation
- Mediator
- Client-side rendering and virtual DOM
- Data binding and data flow
- The web component and shadow DOM
- Choosing an MV^{*} framework

In the second part of this chapter, we are going to put in to practice many of the theoretical concepts explored in the first part of this chapter. We are going to develop a single-page web application framework, from scratch, which will be used to create an application in *Chapter 10, Putting Everything Together*.

The single-page application architecture

We are going to start by exploring what **single-page applications (SPAs)** are and how they work. Numerous SPA frameworks are available that can help us develop applications with a good architecture.

We could jump directly into the use of one of these frameworks, but it is always a good thing to understand how a third-party software component works before we use it. For this reason, we are going to use the first part of this chapter to study the internal architecture of an SPA. Let's start by understanding what an SPA is.

An SPA is a web application in which all the resources (HTML, CSS, JavaScript, and so on) are either loaded in one single request, or loaded dynamically without fully reloading the page. We use the term **single-page** to refer to this kind of application because the web page is never fully reloaded after the initial page load.

In the past, the Web was just a collection of static HTML files and hyperlinks; every time we clicked on a hyperlink, a new page was loaded. This affected web application performance negatively because many of the contents of the page (for example, page headers, page footers, side menus, scripts) were loaded again with each new page.

When AJAX support arrived for web browsers, developers started to load some of the page content via AJAX requests to avoid unnecessary page reloads and provide better user experience. AJAX applications and SPAs work in a very similar way. The significant difference is that AJAX applications load sections of the web application as HTML. These sections are ready to be appended to the DOM as soon as they finish loading. On the other hand, SPAs avoid loading the HTML; instead, they load data and client-side templates. The templates and data are processed and transformed into HTML in the web browser in a process known as **client-side rendering**. The data is usually in XML or JSON format, and there are many available client-side template languages.

Let's compare both approaches in detail. For example, to show a list of clients and orders in an HTML table using the AJAX application approach, we could load the initial page containing the list of clients in HTML format, ready to be displayed. In the table, we would use a row for each client:

```
<tr>
    <td>Client Name 1</td>
    <td>
        <a href="javascript: void(0);" class="orders_link" data-client-
id="1">
            View Orders
        </a>
    </td>
    <!-- more columns... -->
</tr>
```



You don't need to create new folders or files for now. This is a theoretical example and is not meant to be implemented or executed.

We would also need some JavaScript code to load the client orders via AJAX when a user clicks on the `View Orders` link:

```
$(document).ready() {

    // load and display client orders
    function displayOrders(userId) {
        $.ajax({
            method: "GET",
            url: `/client/orders.aspx?id=${userId}`,
            dataType: "html",
            success : function(html) {
                $("#page_container").html(html);
            },
            error : function(e) {
                var msg = "<h1>Sorry, there has been an error!</h1>";
                $("#page_container").html(msg);
            }
        });
    }

    // set click event
```

```
$('.orders_link').on('click', function(e) {  
    var userId = $(e.currentTarget).data("client-id");  
    displayOrders(userId);  
});  
}
```



Refer to the Handlebars.js (<http://handlebarsjs.com/>) and JQuery AJAX (<http://api.jquery.com/jquery.ajax/>) documentation if you need additional help to understand the preceding example.

The preceding code snippet waits for the page to finish loading by using a document-ready event handler. Then it adds an event handler for click events on elements with a class attribute equal to `orders_link`.

The event handler takes the user ID from the `data-client-id` attribute and passes it to the `displayOrders` function. The `displayOrders` function uses an AJAX request to load the list of orders. The list of orders is in HTML format and can be inserted into the DOM without changing its format.

In an SPA, the process is very similar. The initial HTML page (containing the list of clients) is loaded just like in the AJAX application. In SPAs, the navigation to a new page is also managed by JavaScript events, but it is usually managed by a component known as **Router**.

Let's ignore navigation in SPAs for now and focus on the loading and rendering. In an SPA, we will not load a list of orders in HTML format; we will load it using the XML or JSON formats. If we use JSON, the response may look like the following one:

```
{  
  "orders" : [  
    {  
      "order_id" : 32423234,  
      "currency" : "EUR",  
      "date" : "13-02-2015,  
      "items" : [  
        { "product_id" : 13223523, "price" : 150.00, "quantity": 2 }  
        { "product_id" : 62352355, "price" : 50.00, "quantity": 1 }  
      ]  
    },  
    {  
      "order_id" : 32423786,  
      "currency" : "EUR",  
      "date": "13-02-2015,
```

```

        "items" : [
            { "product_id" : 13228898, "price" : 60.00, "quantity" : 1 }
        ]
    }
]
}

```

We can use an AJAX request almost identical to the one that we used to load HTML in the AJAX application:

```

function getOrdersData(userId : number, cb) {
    $.ajax({
        method: "GET",
        url: `/api/orders/${userId}`,
        dataType: "json",
        success : function(json) {
            cb(json);
        },
        error : function(e) {
            var msg = "<h1>Sorry, there has been an error!</h1>";
            $("#page_container").html(msg);
        }
    });
}

```

Before we can show the list of orders in the web browser, we need to transform it into HTML. To transform the JSON into HTML, we can use a template system. There are many template systems, but we are going to use a Handlebars template for this example. Let's take a look at the syntax of one of these templates:

```

{{#each orders}}
<tr>
<td>{{order_id}}</td>
<td>{{date}}</td>
<td>
<ul>
{{#each items}}<li> {{product_id}} x {{quantity}}
</li>{{/each}}
</ul>
</td>
</tr>
{{/each}}

```

The elements of the Handlebars template language are wrapped with double brackets ({{ and }}). The preceding template starts with an each flow control statement. The each statement is used to repeat some instructions for each of the elements in an array. If we take a look at the JSON response, we will be able to see that the orders element is an array. The template will repeat the operations between {{#each orders}} and {{/each}} once for each object in the orders array.

Each repetition creates a new HTML table row. To display the value of one of the JSON fields in the HTML output, we just need to refer to the field wrapped around double brackets. For example, when we render the cell containing the order ID, we use {{order_id}}.

When referring to a JSON field in a template, the field must be in the current scope. The scope can be explicitly accessed using the this keyword, for example, {{this.order_id}} is equal to {{order_id}}. The scope in a template changes when we use some of the available flow control sentences. For example, the {{#each orders}} statement assigns the current item in the array to the this keyword.

In order to use a Handlebars template, we need to load and compile it. We can load the template using a regular AJAX request:

```
function getOrdersTemplate(cb) {
  $.ajax({
    method: "GET",
    url: "/client/orders.hbs",
    dataType: "text",
    success : function(templateSource) {
      var template = Handlebars.compile(source);
      cb(template);
    },
    error : function(e) {
      var msg = "<h1>Sorry, there has been an error!</h1>";
      $("#page_container").html(msg);
    }
  });
}
```

In the preceding example, we have loaded a template using an AJAX request and compiled it using the Handlebars compile method.



In a real production website, templates are usually precompiled by the continuous integration build. The templates are then ready to be used when they finish loading. Precompiling the templates can help to improve the application's performance.

We have created two functions: one to load the template and compile it and the other to load the JSON data. The last step is to create a function that puts together the template and the JSON data to generate the HTML table, which contains the list of client orders:

```
function displayOrders(userId) {
  getOrdersData(userId, function(data) {
    getOrdersTemplate(data, function(template) {
      var html = template(json);
      $("#page_container").html(html);
    });
  });
}
```

It may seem like SPAs require much more work and that they could cause poor performance compared with AJAX applications because there are both more operations and requests to be performed in the web browser. However, that is far from the reality. To understand the benefits of SPAs, we need to understand why they were created in the first place.

The creation of SPAs was highly influenced by two events: the first one is the exponential increase of the popularity of mobile devices and tablets with Internet access and powerful hardware. The second event is the improvement of JavaScript performance that took place during the same period of time.

As mobile devices gained popularity, companies were forced to develop a mobile version of the same client application. Companies started developing web services to generate JSON and XML (instead of HTML pages) that could be consumed by each of these client applications. These web services could be used by all applications, thus allowing companies to reduce costs.

The problem was that the existing AJAX applications could not take advantage of the web services without a client-side rendering system. Template systems such as Mustache (the predecessor of Handlebars) were released for the first time to solve this problem.

One of the main advantages of SPAs is that we need an HTTP API. An HTTP API has many advantages over an application that renders HTML pages in the server side. For example, we can write unit tests for a web service with ease because asserting data is much easier than asserting some user interaction functionality. HTTP APIs can be used by many client applications, which can reduce costs and open new lines of business, such as selling the HTTP API as a product.

Another important advantage of SPAs is that because a lot of the work is performed in the web browser, the server performs fewer tasks and is able to handle a higher number of requests. Client-side performance is not negatively affected because personal computers and mobile devices have become really powerful and JavaScript performance has improved significantly over the last few years.

Network performance in SPAs can be both better and worse when compared to network performance in AJAX applications. The response formatted in the HTML format can sometimes be heavier than the data in JSON or XML formats.

The price to pay when using JSON or XML is that but we will perform an extra web request to fetch the template. We can solve these problems by pre-compiling the templates, implementing caching mechanisms and joining small template files into larger template files to reduce the number of requests.

The MV* architecture

As we have seen, many tasks that were traditionally performed on the server side are performed on the client side in SPAs. This has caused an increase in the size of JavaScript applications and the need for a better code organization.

As a result, developers have started using in the frontend some of the design patterns that have been used with success in the backend over the last decade. Among those, we can highlight the **Model-View-Controller (MVC)** design pattern and some of its derivative versions, such as **Model-View-ViewModel (MVVM)** and **Model-View-Presenter (MVP)**.

Developers around the world started to share some SPA frameworks that somehow try to implement the MVC design pattern but do not necessarily follow the MVC pattern strictly. The majority of these frameworks implement Models and Views, but since not all of them implement Controllers, we refers to this family of frameworks as MV*.



We will cover concepts such as MVC, Models, and Views later in this chapter.

We will now look at other architecture principles, design patterns, and components commonly present in MV* frameworks.

Common components and features in the MV* frameworks

We have seen that single-page web applications are usually developed using a family of frameworks known as MV*, and we have covered the basics of some common SPA architecture principles.

Let's delve further into some components and features that are commonly found in MV* frameworks.

In this section, we will use some small code snippets from some of the most popular MV* frameworks. We are not attempting to learn how to use each of these frameworks, and no previous experience with an MV* framework is required.

Our goal should be to understand the common components and features of an MV* framework and not focus on a particular framework.

Models

A **model** is a component used to store data. The data is retrieved from an HTTP API and displayed in the view. Some frameworks include a model entity that we, as developers, must extend. For example, in Backbone.js (a popular MV* framework), a model must extend the `Backbone.Model` class:

```
class TaskModel extends Backbone.Model {  
    public created : number;  
    public completed : boolean;  
    public title : string;  
    constructor() {  
        super();  
    }  
}
```

A model inherits some methods that can help us interact with the web services. For example, in the case of a Backbone.js model, we can use a method named `fetch` to set the values of a model using the data returned by a web service. In some frameworks, models include logic to retrieve data from an HTTP API, while others include an independent component responsible for the communication with an HTTP API.

In other frameworks, models are plain entities, and it is not necessary to extend or instantiate one of the framework's classes:

```
class TaskModel {  
    public created : number;  
    public completed : boolean;  
    public title : string;  
}
```

Collections

Collections are used to represent a list of models. In the previous section, we saw an example of a model named `TaskModel`. While this model could be used to represent a single task in a list of things to do, a collection could be used to represent the list of tasks.

In the majority of MV* frameworks that support collections, we need to specify the model of the items of a collection when the collection is declared. For example, in the case of `Backbone.js`, the `Task` collection could look like the following:

```
class TaskCollection extends Backbone.Collection<TaskModel> {  
    public model : TaskModel;  
    constructor() {  
        this.model = TodoModel;  
        super();  
    }  
}
```

Just like in the case of models, some frameworks' collections are plain arrays, and we will not need to extend or instantiate one of the framework's classes. Collections can also inherit some methods to facilitate interaction with web services.

Item views

The majority of frameworks feature an item view (or just view) component. Views are responsible for rendering the data stored in the models as HTML. Views usually require a model, a template, and a container to be passed as a constructor argument, property, or setting.

- The model and the template are used to generate the HTML, as we discovered earlier on in this chapter
- The container is usually the selector of one of the DOM elements in the page; the selected DOM element is then used as a *container* for the HTML, which is inserted or appended to it

For example, in Marionette.js (a popular MV* framework based on Backbone.js), a view is declared as follows:

```
class NavBarItemView extends Marionette.ItemView {  
    constructor(options: any = {}) {  
        options.template = "#navBarItemViewTemplate";  
        super(options);  
    }  
}
```

Collection views

A collection view is a special type of view. The relationship between collection views and views is somehow comparable with the relationship between collections and models. Collection views usually require a collection, an item view, and a container to be passed as a constructor argument, property, or setting.

A collection loops through the models in the specified collection, renders each of them using a specified item view, and then appends the results of the container.

In the majority of frameworks, when a collection view is rendered, an item view is rendered for each item in the collection; this can sometimes create a performance bottleneck.



An alternative solution is to use an item view and a model in which one of its attributes is an array. We can then use the `{ {#each} }` statement in the view template to render a collection in one single operation, as opposed to one operation for each item in the collection.

The following code snippet is an example of a collection view in Marionette.js:

```
class SampleCollectionView extends Marionette.  
CollectionView<SampleModel> {  
    constructor(options: any = {}) {  
        super(options);  
    }  
}  
  
var view = new SampleCollectionView({  
    collection : collection,  
    el:$("#divOutput"),  
    childView : SampleView  
});
```

Controllers

Some frameworks feature Controllers. Controllers are usually in charge of handling the lifecycle of specific models and their associated views. They are responsible for instantiating connection models and collections with their respective views and collection views as well as disposing them before handing the control over to another controller.

Interaction in MVC applications is organized around controllers and actions. Controllers can include as many action methods as needed, and an action typically has one-to-one mapping with user interactions.

We are going to take a look at a small code snippet that uses an MV* framework known as **Chaplin**. Just like Marionette.js, Chaplin is a framework based on Backbone.js. The following code snippet defines a class that inherits from the base Controller class, which is defined by Chaplin:

```
class LikesController extends Chaplin.Controller {  
  
    public beforeAction() {  
        this.redirectUnlessLoggedIn();  
    }  
  
    public index(params) {  
        this.collection = new Likes();  
        this.view = new LikesView({collection: this.collection});  
    }  
  
    public show(params) {  
        this.model = new Like({id: params.id});  
        this.view = new FullLikeView({model: this.model});  
    }  
}
```

In the preceding code snippet, we can see that the controller is named `LikesController`, and it has two actions named `index` and `show` respectively. We can also observe a method named `beforeAction` that is executed by Chaplin before an action is invoked.

Events

An event is an action or occurrence detected by the program that may be handled by the program. MV* frameworks usually distinguish two kinds of events:

- **User events:** Applications allow users to interact with it by triggering and handling user events, such as clicking on a button, scrolling, or submitting a form. User events are usually handled in a view.
- **Application events:** The application can also trigger and handle events. For example, some frameworks trigger an `onRender` event when a view has been rendered or an `onBeforeRouting` event when a controller action is about to be invoked.

Application events are a good way to adhere to the Open/Closed principle of the SOLID principle. We can use events to allow developers to extend a framework (by adding event handlers) without having to modify the framework itself.

Application events can also be used to avoid direct communication between two components. We will cover more about them later in this chapter when we focus on a component known as Mediator.

Router and hash (#) navigation

The router is responsible for observing URL changes and passing the execution flow to a controller's action that matches the URL.

The majority of frameworks use a combination of a technique known as hash navigation and the usage of the HTML5 History API to handle changes in the URL without reloading the page.

In an SPA, the links usually contain the hash (#) character. This character was originally designed to set the focus on one of the DOM elements on a page, but it is used by MV* frameworks to navigate without needing to fully reload the web page.

In order to understand this concept, we are going to implement a really basic Router from scratch. We are going to start by taking a look at how a route—a plain object used to represent a URL—looks in the majority of MV* frameworks:

```
class Route {  
    public controllerName : string;  
    public actionPerformed : string;
```

```

public args : Object[] ;

constructor(controllerName : string, actionName : string, args :
Object[])
{
    this.controllerName = controllerName;
    this.actionName = actionName;
    this.args = args;
}
}

```

The router observes the changes in the web browser's URL. When the URL changes, the router parses it and generates a new route instance.

A really basic router could look as follows:

```

class Router {
    private _defaultController : string;
    private _defaultAction : string;

    constructor(defaultController : string, defaultAction : string) {
        this._defaultController = defaultController || "home";
        this._defaultAction = defaultAction || "index";
    }

    public initialize() {

        // observe URL changes by users
        $(window).on('hashchange', () => {
            var r = this.getRoute();
            this.onRouteChange(r);
        });
    }

    // Encapsulates reading the URL
    private getRoute() {
        var h = window.location.hash;
        return this.parseRoute(h);
    }

    // Encapsulates parsing an URL
    private parseRoute(hash : string) {
        var comp, controller, action, args, i;
        if (hash[hash.length - 1] === "/") {
            hash = hash.substring(0, hash.length - 1);
        }
    }
}

```

```

comp = hash.replace("#", '').split('/');
controller = comp[0] || this._defaultController;
action = comp[1] || this._defaultAction;

args = [];
for (i = 2; i < comp.length; i++) {
    args.push(comp[i]);
}
return new Route(controller, action, args);
}

private onRouteChange(route : Route) {
    // invoke controller here!
}
}

```

[ In the second part of this chapter, we are going to develop an entire SPA framework from scratch, and we will use an extended version of the preceding class.]

The preceding class takes the name of the default constructor and the name of the default action as its constructor arguments. The controller named `home` and the action named `index` are used as the default values when no arguments are passed to the constructor.

The method named `initialize` is used to create an event listener for the `hashchange` event. Web browsers trigger this event when the `window.location.hash` value changes.

For example, let's consider the current URL to be `http://localhost:8080`. A user then clicks on the following link:

```
<a href="#tasks/index">View Tasks</a>
```

When the link is clicked, the `window.location.hash` value will change to `"task/index"`. The URL in the browser navigation panel will change, but the hash character will prevent the page from fully reloading. The router will then invoke its `getRoute` method to transform the URL into a new instance of the `Route` class by using the `parseRoute` method.

The URL follows the following name convention:

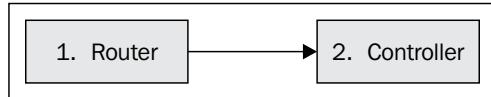
```
#controllerName/actionName/arg1/arg2/arg3/argN
```

This means that the task/index URL is transformed into:

```
new Route("task", "index", []);
```

[ The majority of MV* frameworks use the HTML History API to hide the hash (#) character from the URL, but we will not implement this feature in our framework.]

The instance of the `Route` class is passed to the `onRouteChange` method, which is responsible for invoking the controller that matches the route.



[ We have omitted the implementation of the `onRouteChange` method on purpose but will refer to this function in the Mediator and Dispatcher sections later in this chapter.]

This is basically how hash navigation and routers work. As we can expect, in a real framework, a router has many additional features, but the preceding example should help us gain a good understanding of how routing works in the majority of MV* frameworks.

Mediator

Some MV* frameworks introduce a component known as **Mediator**. The mediator is a simple object all other modules use to communicate with each other.

The mediator usually implements the publish/subscribe design pattern (also known as **pub/sub**). This pattern enables modules to not depend on each other. Instead of making direct use of other parts of the application, modules communicate through events.

Modules can listen for and react to events but also publish events of their own to give other modules the chance to react. This ensures loose coupling of application modules, while still allowing for ease of information exchange.

The mediator can also help us to allow developers to extend our framework (by subscribing to events) without actually having to modify the framework itself. As we saw in *Chapter 4, Object-Oriented Programming with TypeScript*, this is a good thing because it adheres to the Open/Closed principle in the SOLID principles.

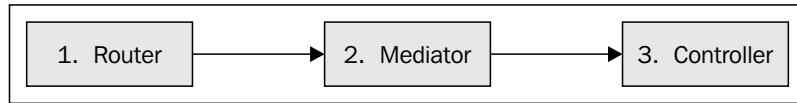
We are going to avoid the internal details of how a mediator works for now, but we can take a look at an example of the public interface of a mediator:

```
interface IMediator {  
    publish(e : IAppEvent) : void;  
    subscribe(e : IAppEvent) : void;  
    unsubscribe(e : IAppEvent) : void;  
}
```

In the previous section, we omitted the details about how the router invokes a controller because the framework that we are going to develop will use a mediator:

```
class Router {  
    // ...  
    private onRouteChange(route : Route) {  
        this.mediator.publish(new AppEvent("app.dispatch", route, null));  
    }  
}
```

The preceding code snippet showcases how the router avoids invoking the controller's action directly, and instead, it publishes an event using a mediator.



Dispatcher

There was something in the previous code snippet that may have caught your attention: the event name is `app.dispatch`.

The `app.dispatch` event refers to an entity known as **Dispatcher**. This means that the router is sending an event to the dispatcher and not to a controller:

```
class Dispatcher {  
    // ...  
    public initialize() {  
        this.mediator.subscribe(  
            new AppEvent("app.dispatch", null, (e: any, data? : any) => {  
                this.dispatch(data);  
            })  
    }  
}
```

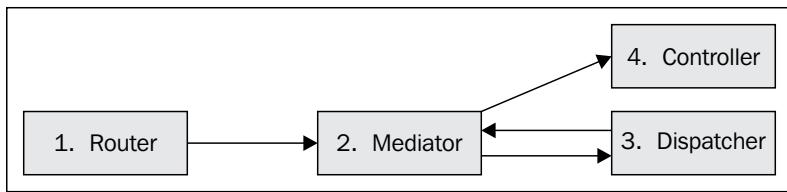
```

        })
    );
}

// Create and dispose controller instances
private dispatch(route : IRoute) {
    // 1. Dispose previous controller
    // 2. Create instance of new controller
    // 3. Invoke controller action using Mediator
}
// ...
}

```

As we can see in this code snippet, the dispatcher is responsible for the creation of new controllers and the disposal of old controllers. When a router finishes parsing a URL, it will pass an instance of the `Route` class to the dispatcher using a mediator. The dispatcher then disposes the previous controller creates an instance of the new controller, and invokes the controller action using a mediator.



Client-side rendering and Virtual DOM

We are already familiar with the basics of client-side rendering. We know client-side rendering requires a template and some data to generate HTML as output, but we haven't mentioned some performance details that we need to consider when selecting an MV* framework.

Manipulating the DOM is one of the main potential performance bottlenecks in SPAs. For this reason, it is interesting to compare how frameworks render the views internally before we decide to work with one or another.

Some frameworks render a view whenever the model changes, and there are two possible ways to know when a model has changed:

- The first one is to check for changes using an interval (this operation is sometimes referred as a dirty check)
- The second option is to use an observable model

The observable approach is much more efficient than using a time interval because the observable model will only consume processing time when it has actually changed. On the other hand, the interval will consume processing time even when the model has not changed.

When to render is important, but we also need to consider how to render. Some frameworks manipulate the DOM directly and others use an in-memory representation of the DOM known as **Virtual DOM**. Virtual DOM is much more efficient because JavaScript is able to manipulate the in-memory representation of the DOM much faster than the DOM itself.

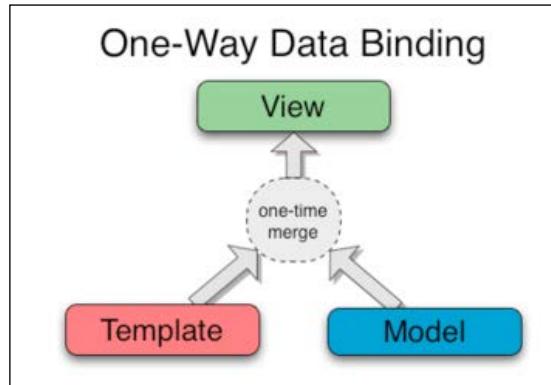
User interface data binding

User interface (UI) data binding is a design pattern that aims to simplify development of graphic UI applications. UI data binding binds UI elements to an application domain model.

A binding creates a link between two properties such that when one changes, the other one is updated to the new value automatically. Bindings can connect properties on the same object, or across two different objects. Most MV* frameworks include some sort of binding implementation between views and models.

One-way data binding

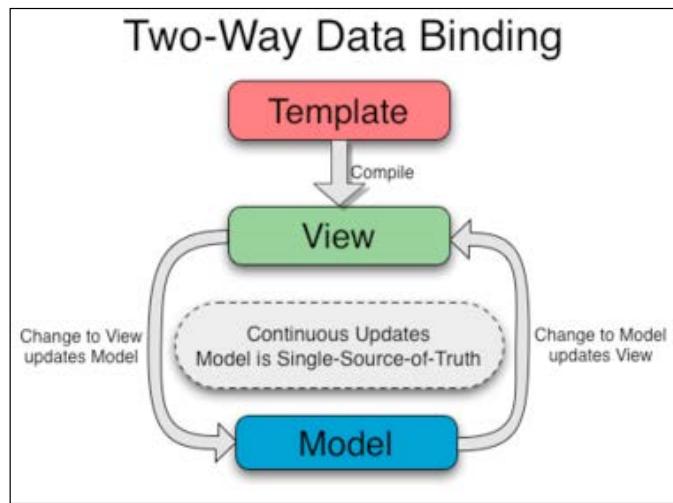
One-way data binding is a type of UI data binding. This type of data binding only propagates changes in one direction.



In the majority of MV* frameworks, this means that any changes in the model are propagated to the view. On the other hand, any changes in the view are not propagated to the model.

Two-way data binding

Two-way binding is used to ensure that any changes to the view are propagated to the model and any changes in the model are propagated to the view.

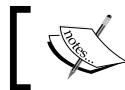
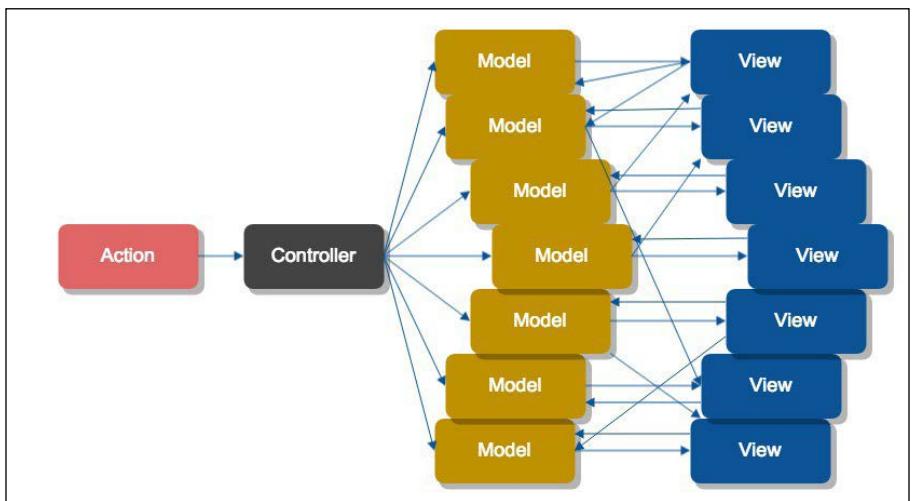


Data flow

Some of the latest MV* frameworks have introduced new approaches and techniques. One of these new concepts is the unidirectional data flow architecture (introduced by Flux).

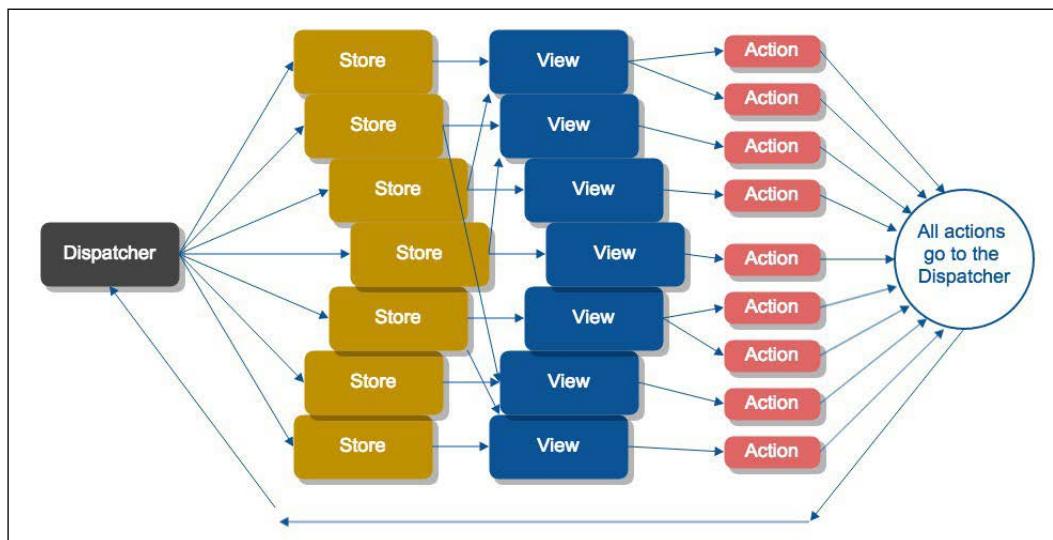
This unidirectional data flow architecture is based on the idea that changing the value of a variable should automatically force recalculation of the values of variables that depend on its value.

In an MVC application, a controller handles multiple Models and Views. Sometimes, a View uses more than one model, and when two-way data binding is used, we can end up with a complicated flow of data to follow. The following diagram illustrates such a scenario:



In this diagram, action does not refer to the actions in a controller.
Action here refers to user or application events.

Dataflow architecture attempts to solve this problem by restricting the flow of data to one unique channel and direction. By doing so, the flow of data within the application components becomes much easier to follow. The following diagram illustrates the flow of data in an application that uses unidirectional data flow architecture:



The preceding diagram illustrates how the data always moves in the same direction.

In Flux's unidirectional data flow architecture, all the actions are directed to the dispatcher. The dispatcher in Flux is like the dispatcher in our framework, but instead of passing the execution flow to a controller, it passes the execution flow to a store.

Stores are in charge of retrieving and manipulating data and can be compared with Models in MVC. Once the data has been modified in some way, it is passed to the views.

Views, just like in MVC, are responsible for rendering the data as HTML and handling user events (actions). If the event requires some data to be modified in some way, the Views will send an action to the dispatcher instead of manipulating its model, as would happen in an application with two-way data binding support.

The data always moves in the same direction and in circles, which makes the execution flow of a large dataflow application much easier to debug and predict than that of a two-way data binding MVC application.

Web components and shadow DOM

Some frameworks use the term web component to refer to reusable user interface widgets. Web components allow developers to define custom HTML elements. For example, we could define a new HTML `<map>` tag to display a map. Web components can import their own dependencies and use client-side templates to render their own HTML using a technology known as **shadow DOM**.

Shadow DOM allows the browser to use HTML, CSS, and JavaScript within a web component. Shadow DOM is useful when developing large applications because it helps to prevent CSS, HTML, and JavaScript conflicts between components.



Some of the existing MV* frameworks (for example, Polymer) can be used to implement real web components. While other frameworks (for example, React) use the term web components to refer to reusable user interface widgets, those components cannot be considered real web components because they don't use the web components technology stack (custom elements, HTML templates, shadow DOM and HTML imports).

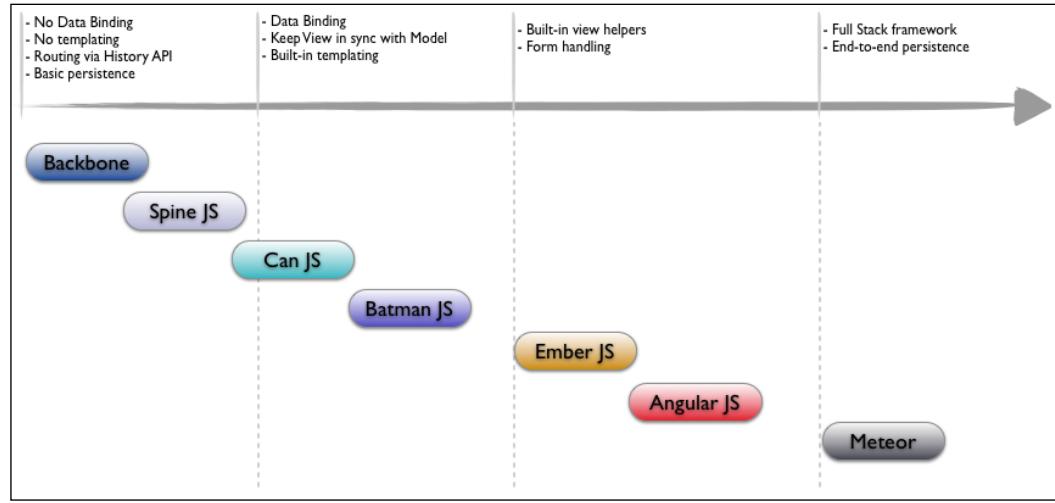
Choosing an application framework

We can create a SPA from scratch, but usually we pick up an existing framework before creating our own. One of the main problems of choosing a JavaScript SPA framework is that there are too many choices.

The latest and greatest JavaScript framework comes around every sixteen minutes.

- Allen Pike

I would personally recommend considering a framework or another depending on the features that you think that you will need to achieve your goals.



For example, if we are going to work on an application with not really complex views and forms, Backbone.js or one of its derivations (Marionette.js, Chaplin, and so on) should work for us. However, if our application is expected to have many forms and complex views, Ember.js or AngularJS might be a better option.



If you need some extra help when choosing one framework over another, you should visit <http://todomvc.com>. TodoMVC is a project that offers the same application (a task manager) implemented using MV* concepts in most of the popular JavaScript MV* frameworks today.

Writing an MVC framework from scratch

Now that we have a good idea about the common components of an MV* application framework, we are going to try to implement our own framework from scratch.

The framework that we are about to develop has not been designed to be used in a real professional environment. Real MV* frameworks have thousands of features and have been under intense development for months and even years before becoming stable.

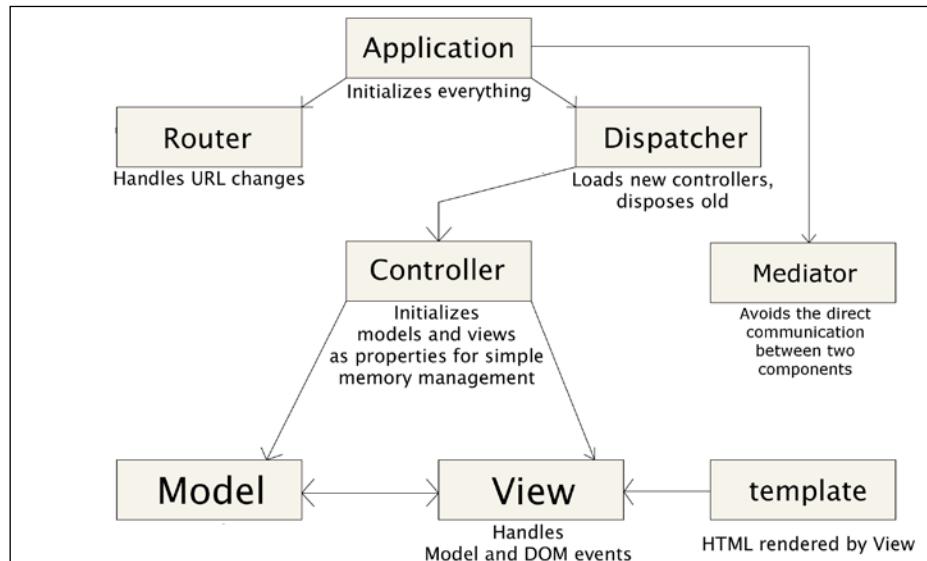
This framework has been developed not to be the most efficient or the most maintainable MV* framework available, but to be a good learning resource.

Our application will feature controllers, templates, views, and models as well as a router, a mediator, and a dispatcher. Let's take a look at the role of each of these components in our framework:

- **Application:** This is the root component of an application. The application component is in charge of the initialization of all the internal components of the framework (mediator, router, and dispatcher).
- **Mediator:** The mediator is in charge of the communication between all the other components in the application.
- **Application Events:** Application events are used to send information from one component to another. An application event is identified by an identifier known as a topic. The components can publish application events as well as subscribe and unsubscribe to application events.
- **Router:** The router observes the changes in the browser URL and creates instances of the Route class that are then sent to the Dispatcher using an application event.
- **Routes:** These are used to represent a URL. The URLs use naming conventions that can be used to identify which controller and action should be invoked.
- **Dispatcher:** The dispatcher receives instances of the Route class, which are used to identify the required controller. The dispatcher can then dispose the previous controller and create a new controller instance if necessary. Once the controller has been initialized, the dispatcher passes the execution flow to the controller using an application event.

- **Controllers:** Controllers are used to initialize views and models. Once the views and models are initialized, the controller passes the execution flow to one or more models using an application event.
- **Models:** Models are in charge of the interaction with the HTTP API as well as data manipulation in memory. This involves data formatting as well as operations such as the addition or deletion of data. Once the Model has finished manipulating the data, it is passed to one or more views using an application event.
- **Views:** Views are in charge of the load and compilation of templates. Once the template has been loaded, the views wait for data to be sent by the models. When the data is received, it is combined with the templates to generate HTML code, which is appended to the DOM. Views are also in charge of the binding and unbinding of UI events (click, focus, and so on).

The following diagram can help us to understand the interaction between the available components:



Now that we have a basic idea about the overall architecture of our framework, let's start a new project.

Prerequisites

Just like we have been doing in the previous chapters of this book, it is recommended to create a new project and configure an automated development workflow using Gulp.

You can try to create the framework and final application following the steps described in the following sections, or you can download the companion source code to get a copy of the finished application.

We are going to start by installing the following runtime dependencies with npm:

```
npm init  
npm install animate.css bootstrap datatables handlebars jquery q --  
save
```

We also need to install the following development dependencies:

```
npm browser-sync browserify chai gulp gulp-coveralls gulp-tslint  
gulp-typescript gulp-uglify karma karma-chai karma-mocha karma-sinon  
mocha run-sequence sinon vinyl-buffer vinyl-source-stream --save-dev
```

Now, let's install the required type definition files using tsd:

```
tsd init  
tsd install jquery bootstrap handlebars q chai sinon mocha  
jquery.dataTables highcharts --save
```

The application uses the following directory tree:

```
├── LICENSE  
├── README.md  
├── css  
│   └── site.css  
├── data  
│   ├── nasdaq.json  
│   └── nyse.json  
├── gulpfile.js  
├── index.html  
├── karma.conf.js  
├── node_modules  
├── package.json  
└── source  
    ├── app  
    │   └── // Chapter 10  
    └── framework  
        ├── app.ts  
        └── app_event.ts
```

```
└── controller.ts
└── dispatcher.ts
└── event_emitter.ts
└── framework.ts
└── interfaces.ts
└── mediator.ts
└── model.ts
└── route.ts
└── router.ts
└── tsconfig.json
└── view.ts

└── test
└── tsd.json
└── typings
```

We will be working on the files located under the source folder during this chapter. In the next chapter, we will create an application using our framework. Most of the files of this application will be located under the app folder.

Now that we have a basic idea about the overall architecture of our framework, let's proceed to implement each of its components.



The final version of the entire framework and application is included in the companion source code.

Application events

We are going to use application events that allow the communication between two components. For example, when a model finishes receiving the response of an HTTP API, the response of the request will be sent from the model to a view using an application event.

As we saw in *Chapter 4, Object-Oriented Programming with TypeScript*, one of the SOLID principles is the dependency inversion principle, which states that we should not depend upon concretions (classes) and should depend upon abstractions instead (interfaces). We are going to try to follow the SOLID principles, so let's get started by creating a new file named `interfaces.ts` inside the `framework` folder and declaring the `IAppEvent` interface:

```
interface IAppEvent {
  topic : string;
  data : any;
  handler: (e: any, data : any) => void;
}
```

An application event contains an identifier or topic and some data or an event handler. We will understand these properties better once we get to publish and subscribe to some events.

Let's continue by creating a new file named `app_event.ts` inside the framework folder and copy the following code into it:

```
/// <reference path="./interfaces"/>

class AppEvent implements IAppEvent {
    public guid : string;
    public topic : string;
    public data : any;
    public handler: (e: Object, data? : any) => void;

    constructor(topic : string, data : any, handler: (e: any, data? : any) => void) {
        this.topic = topic;
        this.data = data;
        this.handler = handler;
    }
}
export { AppEvent };
```

The preceding code snippet declares a class named `AppEvent` which implements the `IappEvent` interface.

Mediator

As we already know, the mediator is a component that implements the pub/sub design pattern and is used to avoid the direct communication between two components.

Let's add a new interface to the `interfaces.ts` file:

```
interface IMediator {
    publish(e : IAppEvent) : void;
    subscribe(e : IAppEvent) : void;
    unsubscribe(e : IAppEvent) : void;
}
```

As we can see in this code snippet, the `IMediator` interface exposes the three methods necessary to implement the publish/subscribe design pattern, as follows:

- `publish`: This is used to trigger events. When we publish an event, all the event subscribers receive it.
- `subscribe`: This is used to subscribe to an event, or in other words, set an event handler for an event.
- `unsubscribe`: This is used to unsubscribe to an event, or in other words, remove an event handler for an event type.

Now, let's proceed to create a new file named `mediator.ts` under the framework folder and add the following code to it:

```
/// <reference path="./interfaces"/>

class Mediator implements IMediator {
    private _$ : JQuery;
    private _isDebug;

    constructor(isDebug : boolean = false) {
        this._$ = $([]);
        this._isDebug = isDebug;
    }

    public publish(e : IAppEvent) : void {
        if(this._isDebug === true) { console.log(new Date().getTime(),
            "PUBLISH", e.topic, e.data); }
        this._$.trigger(e.topic, e.data);
    }

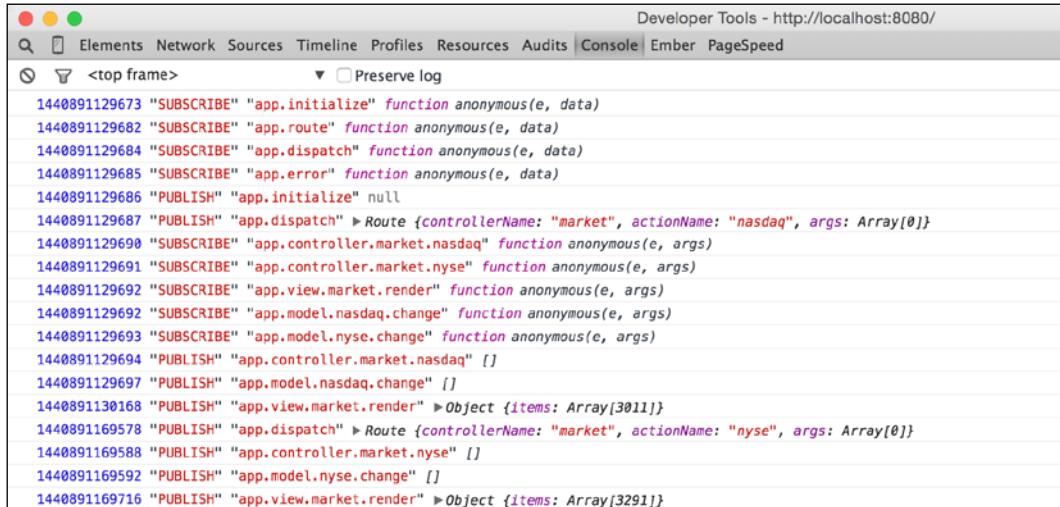
    public subscribe(e : IAppEvent) : void {
        if(this._isDebug === true) { console.log(new Date().getTime(),
            "SUBSCRIBE", e.topic, e.handler); }
        this._$.on(e.topic, e.handler);
    }

    public unsubscribe(e : IAppEvent) : void {
        if(this._isDebug === true) { console.log(new Date().getTime(),
            "UNSUBSCRIBE", e.topic, e.data); }
        this._$.off(e.topic);
    }
}

export { Mediator };
```

The preceding code snippet declares a class named `Mediator`, which implements the `IMediator` interface. The `Mediator` constructor has a default (`false`) parameter that is used to indicate if we are using the debug mode.

The debug mode is useful because when it is enabled, we will be able to observe all the calls to the `publish`, `subscribe`, and `unsubscribe` methods of the mediator without the need to use a debugger. In the following screenshot, we can observe the kind of information that we can expect to see in the browser console when the debug mode is enabled:



A screenshot of a browser's developer tools console window. The title bar says "Developer Tools - http://localhost:8080/". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, Console (which is selected), Ember, and PageSpeed. Below the tabs, there are two dropdown menus: one for the frame (set to <top frame>) and one for log level (set to Preserve log). The main area contains a list of log entries, each starting with a timestamp (e.g., 1440891129673) followed by a message type ("SUBSCRIBE" or "PUBLISH"), a method name ("app.initialize", "app.route", etc.), and a function call. The log entries show the flow of events between the application's controller, model, and view layers.

```
1440891129673 "SUBSCRIBE" "app.initialize" function anonymous(e, data)
1440891129682 "SUBSCRIBE" "app.route" function anonymous(e, data)
1440891129684 "SUBSCRIBE" "app.dispatch" function anonymous(e, data)
1440891129685 "SUBSCRIBE" "app.error" function anonymous(e, data)
1440891129686 "PUBLISH" "app.initialize" null
1440891129687 "PUBLISH" "app.dispatch" ▶ Route {controllerName: "market", actionName: "nasdaq", args: Array[0]}
1440891129690 "SUBSCRIBE" "app.controller.market.nasdaq" function anonymous(e, args)
1440891129691 "SUBSCRIBE" "app.controller.market.nyse" function anonymous(e, args)
1440891129692 "SUBSCRIBE" "app.view.market.render" function anonymous(e, args)
1440891129692 "SUBSCRIBE" "app.model.nasdaq.change" function anonymous(e, args)
1440891129693 "SUBSCRIBE" "app.model.nyse.change" function anonymous(e, args)
1440891129694 "PUBLISH" "app.controller.market.nasdaq" []
1440891129697 "PUBLISH" "app.model.nasdaq.change" []
1440891130168 "PUBLISH" "app.view.market.render" ▶ Object {items: Array[3011]}
1440891169578 "PUBLISH" "app.dispatch" ▶ Route {controllerName: "market", actionName: "nyse", args: Array[0]}
1440891169588 "PUBLISH" "app.controller.market.nyse" []
1440891169592 "PUBLISH" "app.model.nyse.change" []
1440891169716 "PUBLISH" "app.view.market.render" ▶ Object {items: Array[3291]}
```

The `publish`, `subscribe`, and `unsubscribe` methods use the `jQuery trigger`, `on`, and `off` methods respectively to execute event listeners as well as create and remove event listeners when requested.

The default constructor also initializes a private property named `_$_`. The value of this property is just an empty `jQuery` object in memory. This object is used by `jQuery` to add and remove event handlers when the `trigger`, `on`, and `off` method are invoked.

It is important to mention that if the mediator is cleared from memory, its `_$_` property will also be cleared from memory and all the application event handlers will be lost. In the following section, we will see how the `App` class ensures that the mediator is never cleared from memory.

Application

The application class is the root component of an application. The application class is in charge of the initialization of the main components of an application (router, mediator, and dispatcher).

We are going to start by declaring a couple of interfaces required by the application class, so let's add the following interfaces to the `interfaces.td` file:

```
interface IAppSettings {
    isDebug : boolean,
    defaultController : string,
    defaultAction : string,
    controllers : Array<IControllerDetails>;
    onErrorHandler : (o : Object) => void;
}

interface IControllerDetails {
    controllerName : string;
    controller : { new(...args : any[]): IController ;};
}
```

The `IAppSettings` interface is used to indicate the available application settings. We can use the application settings to enable the debug mode, set the name of the default controller and action, set the available controllers, and set a global error handler. Let's take a look at the actual implementation of the application class:

```
/// <reference path=".//interfaces"/>

import { Dispatcher } from "./dispatcher";
import { Mediator } from "./mediator";
import { AppEvent } from "./app_event";
import { Router } from "./router";

class App {
    private _dispatcher : IDispatcher;
    private _mediator : IMediator;
    private _router : IRouter;
    private _controllers : IControllerDetails[];
    private _onErrorHandler : (o : Object) => void;

    constructor(appSettings : IAppSettings) {
        this._controllers = appSettings.controllers;
        this._mediator = new Mediator(appSettings.isDebugEnabled || false);
    }
}
```

```

        this._router = new Router(this._mediator,
        appSettings.defaultController, appSettings.defaultAction);
        this._dispatcher = new Dispatcher(this._mediator,
        this._controllers);
        this._onErrorHandler = appSettings.onErrorHandler;
    }

    public initialize() {
        this._router.initialize();
        this._dispatcher.initialize();
        this._mediator.subscribe(new AppEvent("app.error", null, (e:
        any, data? : any) => {
            this._onErrorHandler(data);
        }));
        this._mediator.publish(new AppEvent("app.initialize", null,
        null));
    }
}

export { App };

```

The preceding code snippet declares a class named `App` that takes the implementation of `IAppSettings` as its only constructor argument. The class constructor initializes the class properties (dispatcher, mediator, router, controller and global error handler).

When we create a new application, it automatically creates a new mediator, and it is passed to both the router and the dispatcher. This means that one unique instance of the mediator is shared by all the components in the application, or in other words, the mediator is a singleton: it stays in memory for the entire application lifecycle.

After creating an instance of the `App` class, we must invoke the `initialize` method to start the execution of the application. We will later see that when the router is initialized, it uses the mediator to subscribe to the `app.initialize` event.

The `initialize` method calls the `initialize` method of some of the application components (router and dispatcher). It then sets an event handler for global errors and publishes the `app.initialize` event.

The mediator then invokes the event handler for the `app.initialize` event by the router. This explains how the execution flow is passed from the application class to the router class.

Route

In order to be able to understand the implementation of the router class, we need to learn about some of its dependencies first. The first of these dependencies is the `Route` class.

The `Route` class implements the `Route` interface. This interface was previously explained in this chapter, so we will not go into its details again.

```
interface IRoute {  
    controllerName : string;  
    actionName : string;  
    args : Object[];  
    serialize() : string;  
}
```

We have also included the implementation of the `Route` class previously in this chapter, but the method named `serialize` was omitted on purpose. The `serialize` method transforms an instance of the `Route` class into a URL.

```
/// <reference path="../interfaces"/>  
  
class Route implements IRoute {  
    public controllerName : string;  
    public actionName : string;  
    public args : Object[];  
  
    constructor(controllerName : string, actionName : string, args :  
Object[]) {  
        this.controllerName = controllerName;  
        this.actionName = actionName;  
        this.args = args;  
    }  
  
    public serialize() : string {  
        var s, sargs;  
        sargs = this.args.map(a => a.toString()).join("/");  
        s = `${this.controllerName}/${this.actionName}/${sargs}`;  
        return s;  
    }  
}  
export { Route };
```

Event emitter

The router also has a dependency in the `EventEmitter` class. This class is particularly important because every single component (except the application component) in the entire framework extends it.

As we already know, all the components use a mediator to communicate with each other. The mediator is a singleton, which means that every single component in our application needs to be provided with access to the mediator instance.

The `EventEmitter` class is used to reduce the amount of boilerplate code that is necessary to achieve this and to provide developers with some helpers that facilitate the publication and subscription of multiple application events:

```
interface IEventEmitter {  
    triggerEvent(event : IAppEvent);  
    subscribeToEvents(events : Array<IAppEvent>);  
    unsubscribeToEvents(events : Array<IAppEvent>);  
}
```

Now, let's create a file named `event_emitter.ts` under the framework directory and copy the following code into it:

```
/// <reference path="./interfaces"/>  
  
import { AppEvent } from "./app_event";  
  
class EventEmitter implements IEventEmitter{  
    protected _metiator : IMediator;  
    protected _events : Array<IAppEvent>;  
  
    constructor(metiator : IMediator) {  
        this._metiator = metiator;  
    }  
  
    public triggerEvent(event : IAppEvent){  
        this._metiator.publish(event);  
    }  
  
    public subscribeToEvents(events : Array<IAppEvent>) {  
        this._events = events;  
        for(var i = 0; i < this._events.length; i++) {  
            this._metiator.subscribe(this._events[i]);  
        }  
    }  
}
```

```
    }

    public unsubscribeToEvents() {
        for(var i = 0; i < this._events.length; i++) {
            this._mediator.unsubscribe(this._events[i]);
        }
    }
}

export { EventEmitter };
```

When the `subscribeToEvents` method is invoked, the `_events` property is used to store the events to which a component is subscribed.

When a component decides to remove its event handlers by using the `unsubscribeToEvents` method, we don't need to pass the full list of events again because the event emitter uses the `events` property to remember them.

Router

The router observes the URL for changes and generates instances of the `Route` class that are then passed to the dispatcher using an application event. The `Router` class implements the `IRouter` interface:

```
interface IRouter extends IEventEmitter {
    initialize(): void;
}
```

Let's take a look at the internal implementation of the `Router` class:

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";
import { Route } from "./route";

class Router extends EventEmitter implements IRouter {
    private _defaultController: string;
    private _defaultAction: string;

    constructor(mediator: IMediator, defaultController: string,
    defaultAction: string) {
        super(mediator);
        this._defaultController = defaultController || "home";
        this._defaultAction = defaultAction || "index";
```

```
}

public initialize() {

    // observe URL changes by users
    $(window).on('hashchange', () => {
        var r = this.getRoute();
        this.onRouteChange(r);
    });

    // be able to trigger URL changes
    this.subscribeToEvents([
        // used to trigger routing on app start
        new AppEvent("app.initialize", null, (e: any, data? : any) => {
            this.onRouteChange(this.getRoute());
        }),
        // used to trigger URL changes from other components
        new AppEvent("app.route", null, (e: any, data? : any) => {
            this.setRoute(data);
        }),
    ]);
}

// Encapsulates reading the URL
private getRoute() {
    var h = window.location.hash;
    return this.parseRoute(h);
}

// Encapsulates writting the URL
private setRoute(route : Route) {
    var s = route.serialize();
    window.location.hash = s;
}

// Encapsulates parsing an URL
private parseRoute(hash : string) {
    var comp, controller, action, args, i;
    if (hash[hash.length - 1] === "/") {
        hash = hash.substring(0, hash.length - 1);
    }
}
```

```

comp = hash.replace("#", '').split('/');
controller = comp[0] || this._defaultController;
action = comp[1] || this._defaultAction;

args = [];
for (i = 2; i < comp.length; i++) {
    args.push(comp[i]);
}
return new Route(controller, action, args);
}

// Pass control to the Dispatcher via the Mediator
private onRouteChange(route : Route) {
    this.triggerEvent(new AppEvent("app.dispatch", route, null));
}
}

export { Router };

```

We have seen this class previously in this chapter, but there are some significant differences here. This time the `Route` class extends the `EventEmitter` class takes a mediator and the names of the default controller and default action as its constructor arguments.

The `initialize` method now includes a call to the `subscribeToEvents` method, which is used to add an application event handler for the `app.initialize` event. This event is used to ensure that the router parses the URL when the application launches for the first time. The router observes the URL for changes, but when the application is launched for the first time, there are no changes in the URL, and the application does not invoke any controller. The router uses the `app.initialize` event handler to solve this problem.

The router is also subscribed to the `app.route` event. The event handler of this event uses a method named `setRoute` to set the browser's URL. The `app.route` application event is used to allow other components to navigate to a route.

Finally, we can find the method named `parseRoute`, which is used to transform a URL into an instance of the `Route` class, and the `onRouteChange` method, which is used to publish an `app.dispatch` application event.

Dispatcher

The dispatcher is a component used to create and dispose controllers when needed. Disposing controllers is important because a controller can use a large number of models and views, which can consume a considerable amount of memory.

If we have many controllers, the amount of memory consumed could become a performance issue. One of the main goals of the dispatcher is to prevent this potential issue.

The dispatcher implements the `IDispatcher` and `IEventEmitter` interfaces:

```
interface IDispatcher extends IEventEmitter {  
    initialize() : void;  
}
```

Let's take a look at the implementation of the `dispatcher` class:

```
/// <reference path="./interfaces"/>  
  
import { EventEmitter } from "./event_emitter";  
import { AppEvent } from "./app_event";  
  
class Dispatcher extends EventEmitter implements IDispatcher {  
    private _controllersHashMap : Object;  
    private _currentController : IController;  
    private _currentControllerName : string;  
  
    constructor(mediator : IMediator, controllers :  
    IControllerDetails[]) {  
        super(mediator);  
        this._controllersHashMap = this.getController(controllers);  
        this._currentController = null;  
        this._currentControllerName = null;  
    }  
}
```

We should be starting to become familiar with how the mediator works at this point. Every component inherits from the `EventEmitter` class and uses its methods to subscribe to some events in the method named `initialize`.

Later in this chapter, we will be able to observe that some classes (Controllers, Views, and Models) also have a method named `dispose`, which is used to unsubscribe to the methods to which the component subscribed in the `initialize` method.

```
// listen to app.dispatch events  
public initialize() {  
    this.subscribeToEvents([
```

```

        new AppEvent("app.dispatch", null, (e: any, data? : any) => {
            this.dispatch(data);
        })
    ]);
}

```

This hash map is used to be able to find a controller as fast as possible when a new route needs to be dispatched. The following method is used to generate a hash map that uses the controller name as the key and the controller constructor as values:

```

private getController(controllers : IControllerDetails[]) : Object {
    var hashMap, hashMapEntry, name, controller, l;

    hashMap = {};
    l = controllers.length;

    if(l <= 0) {
        this.triggerEvent(new AppEvent(
            "app.error",
            "Cannot create an application without at least one
            controller.",
            null));
    }

    for(var i = 0; i < l; i++) {
        controller = controllers[i];
        name = controller.controllerName;
        hashMapEntry = hashMap[name];
        if(hashMapEntry !== null && hashMapEntry !== undefined) {
            this.triggerEvent(new AppEvent(
                "app.error",
                "Two controller cannot use the same name.",
                null));
        }
        hashMap[name] = controller.controller;
    }
    return hashMap;
}

```

The following method is responsible for the creation, initialization, and disposal of controller instances; the code is commented to facilitate its understanding:

```

private dispatch(route : IRoute) {
    var Controller =
        this._controllersHashMap[route.controllerName];

    // try to find controller

```

```

if (Controller === null || Controller === undefined) {
    this.triggerEvent(new AppEvent(
        "app.error",
        `Controller not found: ${route.controllerName}`,
        null));
}
else {
    // create a controller instance
    var controller : IController = new
    Controller(this._mediator);

    // action is not available
    var a = controller[route.actionName];
    if (a === null || a === undefined) {
        this.triggerEvent(new AppEvent(
            "app.error",
            `Action not found in controller: ${route.controllerName}
            - + ${route.actionName}`,
            null));
    }
    // action is available
    else {
        if(this._currentController == null) {
            // initialize controller
            this._currentControllerName = route.controllerName;
            this._currentController = controller;
            this._currentController.initialize();
        }
        else {
            // dispose previous controller if not needed
            if(this._currentControllerName !== route.controllerName) {
                this._currentController.dispose();
                this._currentControllerName = route.controllerName;
                this._currentController = controller;
                this._currentController.initialize();
            }
        }
        // pass flow from dispatcher to the controller
        this.triggerEvent(new AppEvent(
            `app.controller.${this._currentControllerName}
            .${route.actionName}`,
            route.args,
            null
        ));
    }
}

```

```
        }
    }
}

export { Dispatcher };
```

After disposing the previous controller (if necessary) and creating a new controller, this controller is initialized. When a controller is initialized, its `initialize` method is invoked, and as we know, it is then that a component subscribes to some events.

When the dispatcher publishes the following application event, the controller is already subscribed to it and the execution flow is passed to the controller's event handler:

```
`app.controller.${this._currentControllerName} .
${route.actionName}`
```

Controller

Controllers are in charge of the initialization and disposal of views and models. Since controllers must be disposable by the dispatcher, a controller must implement the `dispose` method from the `IController` interface:

```
interface IController extends IEventEmitter {
    initialize() : void;
    dispose() : void;
}
```

The models and views are set as properties of the classes that extend the `Controller` class. The `Controller` class itself does not provide us with any functionality, as it is meant to be implemented by developers when working on an application.

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";

class Controller extends EventEmitter implements IController {

    constructor(mediator : IMediator) {
        super(mediator);
    }

    public initialize() : void {
```

```
    throw new Error('Controller.prototype.initialize() is abstract you  
must implement it!');  
}  
  
public dispose() : void {  
    throw new Error('Controller.prototype.dispose() is abstract you  
must implement it!');  
}  
}  
}  
export { Controller };
```

Even though it is not forced by the framework, it is recommended you use the mediator to pass the control to one of the models (not views) from the controller.

Model and model settings

Models are used to interact with a web service and transform the data returned by it. Models allow us to read, format, update, or delete the data returned by a web service. Models implement the `IModel` and `IEventEmitter` interfaces:

```
interface IMModel extends IEventEmitter {  
    initialize() : void;  
    dispose() : void;  
}
```

A model needs to be provided with the URL of the web service that it consumes. We are going to use a class decorator named `ModelSettings` to set the URL of the service to be consumed.

We could inject the service URL via its constructor, but it is considered a bad practice to inject data (as opposed to a behavior) via a class constructor. The decorator includes some comments to facilitate its understanding:

```
/// <reference path="./interfaces"/>  
  
import { EventEmitter } from "./event_emitter";  
  
function ModelSettings(serviceUrl : string) {  
    return function(target : any) {  
        // save a reference to the original constructor  
        var original = target;  
  
        // a utility function to generate instances of a class  
        function construct(constructor, args) {  
            var c : any = function () {
```

```

        return constructor.apply(this, args);
    }
    c.prototype = constructor.prototype;
    var instance = new c();
    instance._serviceUrl = serviceUrl;
    return instance;
}

// the new constructor behaviour
var f : any = function (...args) {
    return construct(original, args);
}

// copy prototype so instanceof operator still works
f.prototype = original.prototype;

// return new constructor (will override original)
return f;
}
}

```

In the next chapter, we will be able to apply the decorator as follows:

```

@ModelSettings("./data/nasdaq.json")
class NasdaqModel extends Model implements IModel {
//...

```

Let's take a look at the internal implementation of the Model class:

```

class Model extends EventEmitter implements IModel {

    // the values of _serviceUrl must be set using the ModelSettings
    // decorator
    private _serviceUrl : string;

    constructor(mediator : IMediator) {
        super(mediator);
    }

    // must be implemented by derived classes
    public initialize() {
        throw new Error('Model.prototype.initialize() is abstract and
        must implemented.');
    }

    // must be implemented by derived classes

```

```

public dispose() {
    throw new Error('Model.prototype.dispose() is abstract and
    must implemented.');
}

protected requestAsync(method : string, dataType : string, data) {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        $.ajax({
            method: method,
            url: this._serviceUrl,
            data : data || {},
            dataType: dataType,
            success: (response) => {
                resolve(response);
            },
            error : (...args : any[]) => {
                reject(args);
            }
        });
    });
}

protected getAsync(dataType : string, data : any) {
    return this.requestAsync("GET", dataType, data);
}

protected postAsync(dataType : string, data : any) {
    return this.requestAsync("POST", dataType, data);
}

protected putAsync(dataType : string, data : any) {
    return this.requestAsync("PUT", dataType, data);
}

protected deleteAsync(dataType : string, data : any) {
    return this.requestAsync("DELETE", dataType, data);
}

export { Model, ModelSettings };

```

Just like in the case of the controllers, the `initialize` and `dispose` methods are meant to be implemented by the derived models, so they don't contain any logic here.

The `requestAsync` method is used to retrieve data from a web service or static file. As we can see, the method uses the jQuery AJAX API and Q's Promises.

The class also includes the `getAsync`, `postAsync`, `putAsync`, and `deleteAsync` methods, which are helpers to perform GET, POST, PUT, and DELETE requests respectively.

Even though it is not forced by the framework, it is recommended you use the mediator to pass the control to one of the views from the model.

View and view settings

Views are used to render templates and handle UI events. Just like the rest of the components in our application, the `View` class extends the `EventEmitter` class:

```
interface IView extends IEventEmitter {  
    initialize() : void;  
    dispose() : void;  
}
```

A view needs to be provided with the URL of the template that it consumes. We are going to use a class decorator named `ViewSettings` to set the URL of the template to be consumed.

We could inject the template URL via its constructor, but it is considered a bad practice to inject data (as opposed to a behavior) via a class constructor. The decorator includes some comments to facilitate its understanding:

```
/// <reference path="./interfaces"/>  
  
import { EventEmitter } from "./event_emitter";  
import { AppEvent } from "./app_event";  
  
function ViewSettings(templateUrl : string, container : string) {  
    return function(target : any) {  
        // save a reference to the original constructor  
        var original = target;  
  
        // a utility function to generate instances of a class  
        function construct(constructor, args) {  
            var c : any = function () {  
                return constructor.apply(this, args);  
            }  
            c.prototype = constructor.prototype;  
        }  
        Object.defineProperty(target, "constructor", {  
            value: construct,  
            enumerable: false,  
            writable: true,  
            configurable: true  
        });  
        target.$url = templateUrl;  
        target.$container = container;  
    };  
}  
  
// a utility function to generate instances of a class  
function construct(ctor, args) {  
    var c : any = function () {  
        return ctor.apply(this, args);  
    }  
    c.prototype = ctor.prototype;  
}
```

```

var instance = new c();
instance._container = container;
instance._templateUrl = templateUrl;
return instance;
}

// the new constructor behaviour
var f : any = function (...args) {
    return construct(original, args);
}

// copy prototype so instanceof operator still works
f.prototype = original.prototype;

// return new constructor (will override original)
return f;
}
}

```

In the next chapter, we will be able to apply the decorator as follows:

```

@ViewSettings("./source/app/templates/market.hbs", "#outlet")
class MarketView extends View implements IView {
//...

```

Let's take a look at the `View` class. Just like in the case of the controllers and models, the `initialize` and `dispose` methods are meant to be implemented by the derived views, so they don't contain any logic here.

```

class View extends EventEmitter implements IView {

    // the values of _container and _templateUrl must be set using
    // the ViewSettings decorator
    protected _container : string;
    private _templateUrl : string;

    private _templateDelegate : HandlebarsTemplateDelegate;

    constructor(mediator : IMediator) {
        super(mediator);
    }

    // must be implemented by derived classes
    public initialize() {
        throw new Error('View.prototype.initialize() is abstract and
            must implemented.');
    }
}

```

```
}

// must be implemented by derived classes
public dispose() {
    throw new Error('View.prototype.dispose() is abstract and must
    implemented.');
}
```

The view class includes two new methods (named `bindDomEvents` and `unbindDomEvents`) that must be implemented by their derived classes. As we can guess from their names, these methods should be used to set (`bindDomEvents`) and unset (`unbindDomEvents`) UI event handlers:

```
// must be implemented by derived classes
protected bindDomEvents(model : any) {
    throw new Error('View.prototype.bindDomEvents() is abstract
    and must implemented.');
}

// must be implemented by derived classes
protected unbindDomEvents() {
    throw new Error('View.prototype.unbindDomEvents() is abstract
    and must implemented.');
}
```

The following asynchronous methods use promises and are used to load a template (`loadTemplateAsync`), compile it (`compileTemplateAsync`), cache it (`getTemplateAsync`), and render it (`renderAsync`)—all the methods are private except `renderAsync`, which is meant to be used by the derived views:

```
// asynchronously loads a template
private loadTemplateAsync() {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        $.ajax({
            method: "GET",
            url: this._templateUrl,
            dataType: "text",
            success: (response) => {
                resolve(response);
            },
            error : (...args : any[]) => {
                reject(args);
            }
        });
    });
}
```

```

    }

// asynchronously compile a template
private compileTemplateAsync(source : string) {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        try {
            var template = Handlebars.compile(source);
            resolve(template);
        }
        catch(e) {
            reject(e);
        }
    });
}

// asynchronously loads and compile a template if not done
already
private getTemplateAsync() {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        if(this._templateDelegate === undefined ||
           this._templateDelegate === null) {
            this.loadTemplateAsync()
                .then((source) => {
                    return this.compileTemplateAsync(source);
                })
                .then((templateDelegate) => {
                    this._templateDelegate = templateDelegate;
                    resolve(this._templateDelegate);
                })
                .catch((e) => { reject(e); });
        }
        else {
            resolve(this._templateDelegate);
        }
    });
}

// asynchronously renders the view
protected renderAsync(model) {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        this.getTemplateAsync()
            .then((templateDelegate) => {
                // generate html and append to the DOM
                var html = this._templateDelegate(model);

```

```

        $(this._container).html(html);

        // pass model to resolve so it can be used by
        // subviews and DOM event initializer
        resolve(model);
    })
    .catch((e) => { reject(e); });
})
}

export { View, ViewSettings };

```

Framework

The framework file is used to provide access to all the components in the framework from one single file. This means that when we implement an application using our framework, we will not need to import a different file for each component:

```

/// <reference path="./interfaces"/>

import { App } from "./app";
import { Route } from "./route";
import { AppEvent } from "./app_event";
import { Controller } from "./controller";
import { View, ViewSettings } from "./view";
import { Model, ModelSettings } from "./model";

export { App, AppEvent, Controller, View, ViewSettings, Model,
ModelSettings, Route };

```

Summary

In this chapter, we understood what a single-page web application is, what its common components are, and what the main characteristics of this architecture are.

We also created our own MV* framework. This practical experience and knowledge will help us to understand many of the available MV* frameworks.

In the next chapter, we will try to put in practice many of the concepts that we have learned in this book by creating a full SPA using the framework that we created in this chapter.

10

Putting Everything Together

In this chapter, we are going to put into practice the majority of the concepts that we have covered in the previous chapters.

We will develop a small single-page web application using the SPA framework that we developed in *Chapter 9, Application Architecture*.

This application will allow us to find out how the NASDAQ and NYSE stocks are doing on a particular day. It will not be a very large application, but it will be big enough to demonstrate the advantages of working with TypeScript and using a good application architecture.

We will write some classes and several functions. Some of these functions will be asynchronous (*Chapter 1, Introducing TypeScript*; *Chapter 3, Working with Functions*; *Chapter 4, Object-Oriented Programming with TypeScript*; and *Chapter 5, Runtime*). We will also consume some decorators provided by our SPA framework (*Chapter 8, Decorators*).

To complete the chapter, we will create an automated build to facilitate the development process (*Chapter 2, Automating Your Development Workflow*), improve the application performance (*Chapter 6, Application Performance*), and ensure that it works correctly by writing some unit and integration tests (*Chapter 7, Application Testing*).

In this chapter, we will aim to help you gain confidence with TypeScript and the SPA architecture. We need to focus on the SOLID principles and the separation of concerns. Our goal is to create an application that is maintainable and testable, and an application that can grow over time and which components can be reused in future applications.

Prerequisites

In this application, we will use the tools and the directory tree that we created in the previous chapter. You can use the `tsd.json` and `package.json` files included in the companion source code to install the required npm packages and type definition files. Refer to the prerequisites section under the *Writing an MVC framework from scratch* section in *Chapter 9, Application Architecture*, for additional information about the prerequisites of this application.

The application's requirements

We will develop a small application that will allow users to see a list of stock symbols. A stock symbol represents a company that trades its shares on a stock exchange.

The application home page will display stock symbols from two popular stock exchanges: **NASDAQ (National Association of Securities Dealers Automated Quotations)** and **NYSE (New York stock exchange)**.

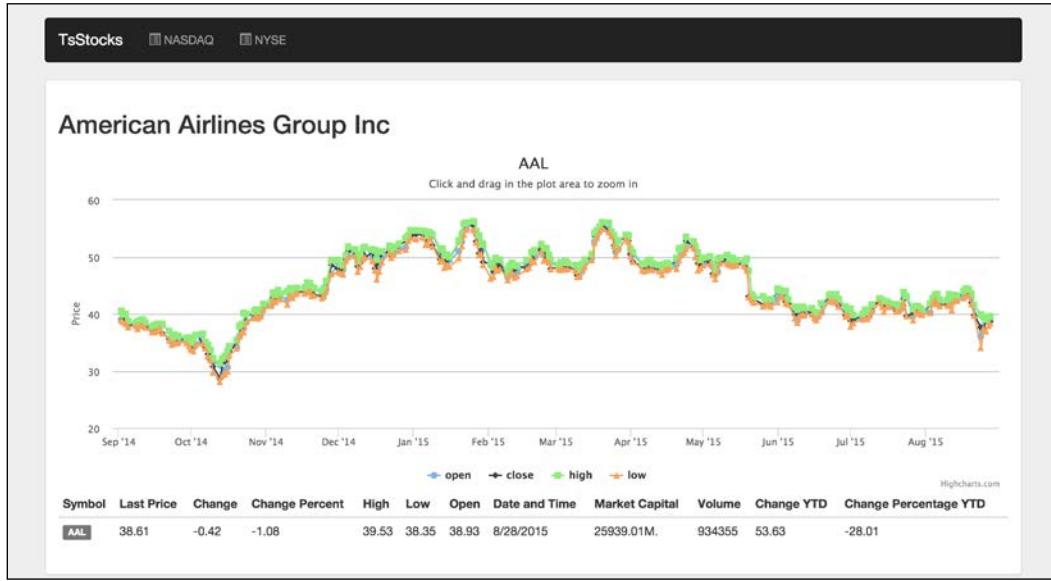
As you can see in the following screenshot, the web application requires a top menu containing links that allow the user to see the stock symbols in one of the aforementioned stock exchanges. The list of stock symbols will be displayed in a table, which will include some basic details about the stocks, such as the price of a share in the last sale or the name or the company:

The screenshot shows a web application interface for 'TsStocks'. At the top, there is a dark header bar with three tabs: 'TsStocks' (selected), 'NASDAQ', and 'NYSE'. Below the header is a search bar labeled 'Search:' with a placeholder 'Type to search' and a magnifying glass icon. The main content area is titled 'NASDAQ'. It features a table with ten rows of stock data. The columns are: 'Symbol' (with a dropdown arrow icon), 'Name', 'Last Sale', 'Market Capital', 'IPO year', 'Sector', 'Industry', and 'Quote' (with a blue button icon). The data rows are as follows:

Symbol	Name	Last Sale	Market Capital	IPO year	Sector	Industry	Quote
AAL	American Airlines Group, Inc.	40.02	\$27.73B	n/a	Transportation	Air Freight/Delivery Services	
AAME	Atlantic American Corporation	3.67	\$75.49M	n/a	Finance	Life Insurance	
AAOI	Applied Optoelectronics, Inc.	18.31	\$273.61M	2013	Technology	Semiconductors	
AAON	AAON, Inc.	24	\$1.3B	n/a	Capital Goods	Industrial Machinery/Components	
AAPO	Atlantic Alliance Partnership Corp.	10.2	\$105.96M	2015	Finance	Business Services	
AAPL	Apple Inc.	128.59	\$740.81B	1980	Technology	Computer Manufacturing	
AAVL	Avalanche Biotechnologies, Inc.	40.79	\$1.04B	2014	Health Care	Biotechnology: Biological Products (No Diagnostic Substances)	
AAWW	Atlas Air Worldwide Holdings	55.29	\$1.38B	n/a	Transportation	Transportation Services	
ABAC	Aoxin Tianli Group, Inc.	1.49	\$49.44M	n/a	Consumer Non-Durables	Farming/Seeds/Milling	

The last column in the table contains some buttons that will allow users to navigate to a second screen that displays a stock quote. A stock quote is just a summary of the pricing performance details of the stock for a given period of time.

The stock quote screen will display a line graph that is used by the brokers to see how the price of the shares (the y axis) has evolved over time (the x axis). We can display multiple lines to visualize the evolution of the opening price (the price of the shares at the beginning of the day), the closing price (the price of a share at the end of the day), the high price (the highest selling price of the share in a given day), and the low price (the lowest selling price of the share in a given day).



The application's data

As we explained in the previous chapter, we need an application backend that allows us to query the data from a web browser using AJAX requests in order to develop an SPA. This means that we are going to need an HTTP API.

We will use a freely available public HTTP API that will allow us to obtain real stock quote data. For the list of available stock symbols, we will use static JSON files. These JSON files have been generated by transforming a CSV file available on the NASDAQ website. The external HTTP API will also provide the line graph data.

In total, we will be using three sets of data:

- **Market data:** This data is stored in static JSON files. These files have been generated from a CSV file provided by the NASDAQ official website and can be found in the companion example.
- **Stock quote data:** This has been provided by an external web service. The external data provider that we will use in this example is a company called **Markit**, specializing in financial information services. We will use their market data API (v2), which is available for free and has been well documented at <http://dev.markitondemand.com/>.
- **Chart data:** This is also provided in a web service by Markit.

The application's architecture

We will develop an SPA using our own framework. As we saw in the previous chapter, our framework can map a URL with an action in a controller.

Our application will have three main screens. Each screen uses a different URL, as follows:

- `#market/nasdaq` displays stocks in the NASDAQ stock market
- `#market/nyse` displays stocks in the NYSE stock market
- `#symbol/quote/{symbol}` displays a stock quote for the selected stock symbol

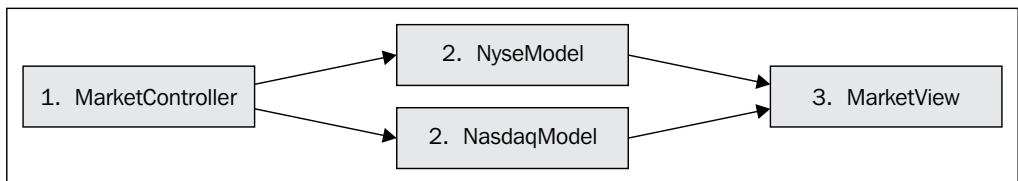
Each of the main URLs mentioned earlier will be implemented as a controller's action in our application. In the previous chapter, you saw that URLs adhere to the following naming convention: `#controllerName/actionName/arg1/arg2/argN`.

If we extrapolate this naming convention to the URLs mentioned in the preceding list, we can deduce that our application will have two controllers: `MarketController` and `SymbolController`.

The `MarketController` controller will be implemented using two models and one view:

- **NasdaqModel:** This loads a list of NASDAQ stocks from a static JSON file
- **NyseModel:** This loads a list of NYSE stocks from a static JSON file
- **MarketView:** This renders the list of either the NASDAQ or NYSE stocks

Each component communicates with the other using application events and the mediator. The execution order of the market screen looks as follows:



The `SymbolController` controller will be implemented using two models and two views:

- **QuoteModel**: This loads a stock quote for the selected symbol
- **ChartModel**: This loads symbol performance data points for the last year
- **ChartView**: This displays stock performance in an interactive chart
- **SymbolView**: This displays the last price change for the selected symbol

Each component communicates with the other using application events and the mediator. The execution order of the stock quote screen looks as follows:



The application's file structure

Presented in this section is the folder structure of the application we are going to build. In the root directory, you can find the application access point (`index.html`), as well as some of the automation tools' configuration files (`gulpfile.js`, `karma.conf.js`, `package.json`, and so on). You can also observe the `typings` folder, which contains some type definition files.

Just as in the previous chapters, the application source code is located under the source directory. The unit and integration tests are located in the test folder. The following is the folder structure of the application:

```
|- LICENSE  
|- README.md  
|- css  
|   |- site.css  
|- data  
|   |- nasdaq.json
```

```
|- └ nyse.json
|- gulpfile.js
|- index.html
|- karma.conf.js
|- node_modules
|- package.json
|- source
|   |- app
|   |   |- controllers
|   |   |   |- market_controller.ts
|   |   |   └ symbol_controller.ts
|   |   |- main.ts
|   |   |- models
|   |   |   |- chart_model.ts
|   |   |   |- nasdaq_model.ts
|   |   |   |- nyse_model.ts
|   |   |   └ quote_model.ts
|   |   |- templates
|   |   |   |- market.hbs
|   |   |   └ symbol.hbs
|   |   └ views
|   |       |- chart_view.ts
|   |       |- market_view.ts
|   |       └ symbol_view.ts
|   |- framework
|       └ framework.ts (Chapter 9)
|- test
|   |- app
|   |- framework
|- tsd.json
|- typings
```

Under the `source` directory, you can observe two folders, named `app` and `framework`. We created all the files under the `framework` directory in the previous chapter. This time, we will focus on the application, which means we will be working under the `app` directory most of the time.

Inside the `app` directory, you can find some directories named `controllers`, `models`, `templates`, and `views`. As you can guess, these directories are used to store controllers, models, templates, and views respectively.

You can also find the `main.ts` file inside the `app` directory. This file is the application's entry point, but because we are going to use ES6 modules, we are not going to be able to load this file in a web browser using a `<script>` tag.

Configuring the automated build

Just as we did in *Chapter 2, Automating Your Development Workflow*, we need to create a configuration file to configure the desired Gulp tasks. So let's create a file named `gulpfile.js` and import the required Gulp plugins:

```
var gulp          = require("gulp"),
    browserify   = require("browserify"),
    source        = require("vinyl-source-stream"),
    buffer        = require("vinyl-buffer"),
    tslint        = require("gulp-tslint"),
    tsc           = require("gulp-typescript"),
    karma         = require("karma").server,
    coveralls     = require('gulp-coveralls'),
    uglify         = require("gulp-uglify"),
    runSequence   = require("run-sequence"),
    header         = require("gulp-header"),
    browserSync   = require("browser-sync"),
    reload         = browserSync.reload,
    pkg            = require(__dirname + "/package.json");
```

We need to remember that before we can import one of these packages, we must first install them using npm.

Once the plugins have been imported, we can proceed to write our first task, which is used to check for some basic name convention rules and to avoid some bad practices (the TypeScript files are under the `source` and `tests` directories):

```
gulp.task("lint", function() {
  return gulp.src([
    "source/**/**.ts",
    "test/**/**.test.ts"
  ])
    .pipe(tslint())
    .pipe(tslint.report("verbose"));
});
```

We also need another task to compile our TypeScript code into JavaScript code. As we are working with decorators, we need to ensure that we are using TypeScript 1.5 or higher and that the `experimentalDecorators` compiler settings and target are configured as in the following code snippet:

```
var tsProject = tsc.createProject({
  target : "es5",
  module : "commonjs",
```

```
experimentalDecorators: true,  
typescript: typescript  
});
```

Once we have set up the compiler options, we can proceed to write some tasks. The first one will compile the application code:

```
gulp.task("build", function() {  
    return gulp.src("src/**/*.{ts,js}")  
        .pipe(tsc(tsProject))  
        .js.pipe(gulp.dest("build/source/"));  
});
```

The second one will compile the unit test and integration test code. We need to use a new project object to avoid potential runtime issues:

```
var tsTestProject = tsc.createProject({  
    target : "es5",  
    module : "commonjs",  
    experimentalDecorators: true,  
    typescript: typescript  
});  
  
gulp.task("build-test", function() {  
    return gulp.src("test/**/*.{test,ts}")  
        .pipe(tsc(tsTestProject))  
        .js.pipe(gulp.dest("/build/test/"));  
});
```

The two previous tasks should be enough to generate JavaScript, but because we are using CommonJS modules, we need to write a task to bundle the CommonJS modules into a package that can be loaded and executed in a web browser. As we saw in *Chapter 2, Automating Your Development Workflow*, we will create a few Gulp tasks that use Browserify for this purpose.

We need a task to bundle the application code:

```
gulp.task("bundle-source", function () {  
    var b = browserify({  
        standalone : 'TsStock',  
        entries: "build/source/app/main.js",  
        debug: true  
});  
  
    return b.bundle()  
        .pipe(source("bundle.js"))
```

```
.pipe(buffer())
.pipe(gulp.dest("bundled/source/")));
});
```

We further need a task to bundle the application's unit tests:

```
gulp.task("bundle-unit-test", function () {
  var b = browserify({
    standalone : 'test',
    entries: "build/test/bdd.test.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("bdd.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest("bundled/test/"));
});
```

We need a final task to bundle the application's integration tests:

```
gulp.task("bundle-e2e-test", function () {
  var b = browserify({
    standalone : 'test',
    entries: "build/test/e2e.test.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("e2e.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest("bundled/e2e-test/"));
});
```

We will return to the `gulpfile.js` configuration file later in this chapter to add some additional tasks that will be in charge of running the application and its automated tests, as well as some optimizations.



Until now, we have been working on the configuration of an automated development workflow. From now on, we will focus on the application components. A component is composed of four core elements: template, style rules, services, and the component's logic. You will be able to find the style rules and templates in the companion code samples, but we will mainly focus on the TypeScript files (services and the component's logic) here.

The application's layout

Let's create a new file, named `index.html`, under the application's root directory. The following code snippet is an altered version of the real `index.html` page, which is included with the companion source code:

```
<ul class="nav navbar-nav">
  <li>
    <a href="#market/nasdaq">NASDAQ</a>
  </li>
  <li>
    <a href="#market/nyse">NYSE</a>
  </li>
</ul>
<div id="outlet">
  <!-- HTML GENERATED BY VIEWS GOES HERE -->
</div>
```

As you can see in the preceding HTML snippet, the code has two important elements. The first significant element is the URL of the two links. These links include the hash character (#), and they will be processed by the application's router.

The second significant element is the element that uses `outlet` as ID. This node is used by our framework as a container where the DOM of each new page is dynamically generated and added to the page.

Implementing the root component

As you saw in the previous chapter, the root component of our custom MVC framework is the `App` component. So, let's create a new file, named `main.ts`, under the `source/app` directory.

We can access all the interfaces in the framework by adding a reference to the `source/interfaces.ts` as follows:

```
/// <reference path="../framework/interfaces"/>
```

We can then access all the components in the framework by importing the `framework/framework.ts` file:

```
import { App, View } from "../framework/framework";
```

Our application will have two controllers. The files don't exist yet but we can add the two import statements anyway:

```
import { MarketController } from  
"./controllers/market_controller";  
import { SymbolController } from  
"./controllers/symbol_controller";
```

At this point, we need to create an object literal that implements the `IAppSettings` interface. This object allows us to set some basic configuration, such as the name of the default controller or action, or a global error handler. However, the most important field in the object literal is the `controller` field, which must be an array of `IControllerDetails`. If you need additional details about the `IControllerDetails`, refer to the previous chapter.

```
var appSettings : IAppSettings = {  
    isDebug : true,  
    defaultController : "market",  
    defaultAction : "nasdaq",  
    controllers : [  
        { controllerName : "market", controller : MarketController },  
        { controllerName : "symbol", controller : SymbolController }  
    ],  
    onErrorHandler : function(e : Object) {  
        alert("Sorry! there has been an error please check out the console  
for more info!");  
        console.log(e.toString());  
    }  
};
```

We can then create the `App` instance and invoke the `initialize` method to start executing it:

```
var myApp = new App(appSettings);  
myApp.initialize();
```

At this point, our code does not compile because we have not defined the `MarketController` and `SymbolController` controllers yet. Let's define our first controller.

Implementing the market controller

Let's create a new file named `market_controller.ts` under the `app/controllers` directory. We need to import the `Controller` and `AppEvent` entities from the framework along with some entities that are not available yet (`NyseModel`, `NasdaqModel` and `MarketView`).

```
/// <reference path="../../framework/interfaces"/>

import { Controller, AppEvent } from "../../framework/framework";
import { MarketView } from "../views/market_view";
import { NasdaqModel } from "../models/nasdaq_model";
import { NyseModel } from "../models/nyse_model";
```

In an application that uses our framework, a controller must extend the base `Controller` class and implement the `IController` class:

```
class MarketController extends Controller implements IController {
```

We are not forced to declare the views and models used by the controller as its properties, but it is recommended:

```
private _marketView : IView;
private _nasdaqModel : IMModel;
private _nyseModel : IMModel;
```

It is also recommended that you set the value of all the controller's dependencies inside the controller constructor:

```
constructor(mediator : IMediator) {
    super(mediator);
    this._marketView = new MarketView(mediator);
    this._nasdaqModel = new NasdaqModel(mediator);
    this._nyseModel = new NyseModel(mediator);
}
```

Instead of setting the value of all the controller's dependencies inside the controller constructor, it would be even better to use an IoC container to automatically inject the controller's dependencies via its constructor. Though, implementing an IoC container is not a simple task, it is beyond the scope of this book.

We must implement the `initialize` method. The `initialize` method is the place where a controller should do the following:

- Subscribe to one application event for each action available in the controller. In this case, the controller has two actions (the `nasdaq` and `nyse` methods).
- Initialize views by invoking the `View.initialize()` method. In this case, there is only one view (`marketView`).
- Initialize models by invoking the `Model.initialize()` method. In this case, there are two models (`nasdaqModel` and `nyseModel`).

```
public initialize() : void {  
  
    // subscribe to controller action events  
    this.subscribeToEvents([
        new AppEvent("app.controller.market.nasdaq", null, (e, args : string[]) => { this.nasdaq(args); }),
        new AppEvent("app.controller.market.nyse", null, (e, args : string[]) => { this.nyse(args); })
    ]);  
  
    // initialize view and models events  
    this._marketView.initialize();  
    this._nasdaqModel.initialize();  
    this._nyseModel.initialize();
}
```

The `dispose` method is the opposite of the `initialize` method. If an event handler was created in the `initialize` method, it should be destroyed in the `dispose` method. The `unsubscribeToEvents` helper will unsubscribe all the events that were subscribed using the `subscribeToEvents` helper:

```
// dispose views/models and stop listening to controller actions  
public dispose() : void {  
  
    // dispose the controller events  
    this.unsubscribeToEvents();  
  
    // dispose views and model events  
    this._marketView.dispose();  
    this._nasdaqModel.dispose();  
    this._nyseModel.dispose();
}
```

As you saw in the previous chapter, the dispatcher uses the controller's `initialize` and `dispose` methods to free some memory when it is not needed any more. If we forget to dispose one of the views used by the controller in its `dispose` method, the view could end up staying in memory forever.

The actions of a controller should not perform any kind of data manipulation (models should be in charge of that) or user interface events management (views should be in charge of that). Ideally, a controller's actions should only publish one or more application events so the execution flow goes from the controller to one or more models.

In the case of the `nasdaq` action, the controller publishes one of the events to which the `nasdaq` model subscribed when the `initialize` method of `NasdaqModel` was invoked:

```
// display NASDAQ stocks
public nasdaq(args : string[]) {
    this._metiator.publish(new AppEvent("app.model.nasdaq.change",
        null, null));
}
```

In the case of the `nyse` action, the controller publishes one of the events to which the `nyse` model was subscribed when the `initialize` method of `NyseModel` was invoked:

```
// display NYSE stocks
public nyse(args : string[]) {
    this._metiator.publish(new AppEvent("app.model.nyse.change",
        null, null));
}
export { MarketController };
```

Implementing the NASDAQ model

Let's create a new file named `nasdaq_model.ts` under the `app/models` directory. We can then import the `Model`, `AppEvent`, and `ModelSettings` from our framework and declare a new class named `NasdaqModel`. The new class must extend the base `Model` class and implement the `IModel` interface.

We will also use the `ModelSettings` decorator to indicate the path of a web service or static data file. In this case, we will use a static data file, which can be found in the companion source code:

```
/// <reference path="../../framework/interfaces"/>
import { Model, AppEvent, ModelSettings } from "../../framework/
```

```
framework";
@ModelSettings("./data/nasdaq.json")
class NasdaqModel extends Model implements IModel {

    constructor(mediator : IMediator) {
        super(mediator);
    }
}
```

The model will subscribe to the `app.model.nasdaq.change` event when the `initialize` method is invoked. This is actually the event that the controller's action published to pass the execution flow from the controller to the model:

```
// listen to model events
public initialize() {
    this.subscribeToEvents([
        new AppEvent("app.model.nasdaq.change", null, (e, args) => {
            this.onChange(args);
        })
    ]);
}
```

Just like in the previous controller, the `unsubscribeToEvents` helper will unsubscribe all the events that were subscribed using the `subscribeToEvents` helper:

```
// dispose model events
public dispose() {
    this.unsubscribeToEvents();
}
```

This is the event handler of the `app.model.nasdaq.change` event. The event handler uses the `getAsync` method to load the data from the service URL that we previously specified using the `ModelSettings` decorator. The `getAsync` method is inherited from the base `Model` class, which we implemented in the previous chapter.

The `getAsync` method returns a promise; if the promise is fulfilled, the data is formatted and then passed to a view:

```
private onChange(args) : void {
    this.getAsync("json", args)
        .then((data) => {

            // format data
            var stocks = { items : data, market : "NASDAQ" };

            // pass control to the market view
            this.triggerEvent(new AppEvent("app.view.market.render",
                stocks, null));
        })
}
```

```

        .catch((e) => {
            // pass control to the global error handler
            this.triggerEvent(new AppEvent("app.error", e, null));
        });
    }
}

export { NasdaqModel };

```

Implementing the NYSE model

Let's create a new file named `nyse_model.ts` under the `app/models` directory. The `NyseModel` class is almost identical to the `NasdaqModel` class, so we will not go into too much detail:

```

@ModelSettings("./data/nyse.json")
class NyseModel extends Model implements IModel {
    ...
}
export { NyseModel };

```

All we need to do is copy the contents of the `nasdaq_model.ts` file into the `nyse_model.ts` file and replace (case sensitive) `nasdaq` with `nyse`.

This kind of code duplication is known as a code smell. A code smell indicates that something is wrong and we need to refactor (improve) it. We could avoid a lot of code duplication by using Generic types. However generic types were not used here because we thought that showcasing the usage of decorators would be more valuable for the readers of this book.

Implementing the market view

Let's create a new file named `market_view.ts` under the `app/views` directory. We can then import the `AppEvent`, `ViewSettings`, and `Route` components from our framework and declare a new class named `MarketView`. The new class must extend the base `View` class and implement the `IView` interface.

We will also use the `ViewSettings` decorator to indicate the path, a Handlebars template, and a selector, which is used to find the DOM element that will be used as the parent node of the view's HTML:

```
/// <reference path="../../framework/interfaces"/>
```

```
import { View, AppEvent, ViewSettings, Route } from "../../framework/framework";
```

```
@ViewSettings("./source/app/templates/market.hbs", "#outlet")
class MarketView extends View implements IView {

    constructor(mediator : IMediator) {
        super(mediator);
    }
}
```

This view is subscribed to the `app.view.market.render` event and its handler invokes the `renderAsync` method, which has been inherited from the base `view` class. This method returns a promise, which is fulfilled if the template passed to the `ViewSettings` decorator has been loaded and compiled successfully.

For the promise to be fulfilled, the view must be successfully rendered and appended to the DOM element that matches the selector passed to the `ViewSettings` decorator:

```
initialize() : void {
    this.subscribeToEvents([
        new AppEvent("app.view.market.render", null, (e, args : any)
        => {
            this.renderAsync(args)
                .then((model) => {
                    // set DOM events
                    this.bindDomEvents(model);
                })
                .catch((e) => {
                    // pass control to the global error handler
                    this.triggerEvent(new AppEvent("app.error", e,
                        null));
                });
        }),
    ]);
}
```

Just like in the previous controller and model, the `unsubscribeToEvents` helper will unsubscribe all the events that were subscribed to using the `subscribeToEvents` helper:

```
public dispose() : void {
    this.unbindDomEvents();
    this.unsubscribeToEvents();
}
```

Views are responsible for the management of user events. The components in our framework use the `initialize` method to subscribe to application events, and the `dispose` method to unsubscribe to application events. In the case of user events, we will use the `bindDomEvents` method to set the user events, and the `unbindDomEvents` method to dispose of them:

```
// initializes DOM events
protected bindDomEvents(model : any) {
    var scope = $(this._container);
    // handle click on "quote" button
    $(".getQuote").on('click', scope, (e) => {
        var symbol = $(e.currentTarget).data('symbol');
        this.getStockQuote(symbol);
    });
}

// make table sortable and searchable
$(scope).find('table').DataTable();
}

// disposes DOM events
protected unbindDomEvents() {
    var scope = this._container;
    $(".getQuote").off('click', scope);
    var table = $(scope).find('table').DataTable();
    table.destroy();
}
```

One of the user events observes clicks on the quote buttons. When the event is triggered, the following event handler is invoked:

```
private getStockQuote(symbol : string) {
    // navigate to route using route event
    this.triggerEvent(new AppEvent(
        "app.route",
        new Route("symbol", "quote", [symbol]),
        null));
}
```

As you can see, this event handler creates a new route and publishes an `app.route` event. This will cause the router to navigate to the quote action in the `SymbolController`: `export { MarketView };`

Implementing the market template

The template loaded and compiled by MarketView looks as follows:

```
<div class="panel panel-default fadeInUp animated">
  <div class="panel-body">
    <h2>{{market}}</h2>
    <table class="table table-responsive table-condensed">
      <thead>
        <tr>
          <th>Symbol</th>
          <th>Name</th>
          <th>Last Sale</th>
          <th>Market Capital</th>
          <th>IPO year</th>
          <th>Sector</th>
          <th>industry</th>
          <th>Quote</th>
        </tr>
      </thead>
      <tbody>
        {{#each items}}
          <tr>
            <td><span class="label label-default">{{Symbol}}</span></td>
            <td>{{Name}}</td>
            <td>{{LastSale}}</td>
            <td>{{MarketCap}}</td>
            <td>{{IPOyear}}</td>
            <td>{{Sector}}</td>
            <td>{{industry}}</td>
            <td>
              <button class="btn btn-primary btn-sm getQuote"
                     data-symbol="{{Symbol}}"
                     data-hidden="true">
                <span class="glyphicon glyphicon-stats" aria-
                      hidden="true"></span>
                Quote
              </button>
            </td>
          </tr>
        {{/each}}
      </tbody>
    </table>
  </div>
</div>
```

Implementing the symbol controller

Let's create a new file named `symbol_controller.ts` under the `app/controllers` directory. This file will contain a new controller named `SymbolController`. The implementation of this controller is largely similar to the implementation of the `MarketController` controller, so we are going to avoid going into too much detail.

The main difference between this controller and the previous controller is that the new controller uses two new models (`QuoteModel` and `ChartModel`) and two new views (`SymbolView` and `ChartView`):

```
/// <reference path="../../framework/interfaces"/>

import { Controller, AppEvent } from "../../framework/framework";
import { QuoteModel } from "../models/quote_model";
import { ChartModel } from "../models/chart_model";
import { SymbolView } from "../views/symbol_view";
import { ChartView } from "../views/chart_view";

class SymbolController extends Controller implements IController {
    private _quoteModel : IMModel;
    private _chartModel : IMModel;
    private _symbolView : IView;
    private _chartView : IView;

    constructor(mediator : IMediator) {
        super(mediator);
        this._quoteModel = new QuoteModel(mediator);
        this._chartModel = new ChartModel(mediator);
        this._symbolView = new SymbolView(mediator);
        this._chartView = new ChartView(mediator);
    }

    // initialize views/ models and start listening to controller
    // actions
    public initialize() : void {

        // subscribe to controller action events
        this.subscribeToEvents([
            new AppEvent("app.controller.symbol.quote", null, (e, symbol
                : string) => { this.quote(symbol); })
        ]);
    }
}
```

```

// initialize view and models events
this._quoteModel.initialize();
this._chartModel.initialize();
this._symbolView.initialize();
this._chartView.initialize();
}

// dispose views/models and stop listening to controller actions
public dispose() : void {

    // dispose the controller events
    this.unsubscribeToEvents();

    // dispose views and model events
    this._symbolView.dispose();
    this._quoteModel.dispose();
    this._chartView.dispose();
    this._chartModel.dispose();
}

```

It is also important to notice that the `quote` action passes the control to the `QuoteModel` model:

```

public quote(symbol : string) {
    this.triggerEvent(new AppEvent("app.model.quote.change",
        symbol, null));
}
export { SymbolController };

```

Implementing the quote model

Let's create a new file named `quote_model.ts` under the `app/models` directory. This is the third model that we have implemented so far. This means that you should be familiar with the basics already, but there are some minor additions in this particular model. The first thing that you will notice is that the web service is no longer a static file:

```

/// <reference path="../../framework/interfaces"/>

import { Model, AppEvent, ModelSettings } from "../../framework/
framework";

```

```

@ModelSettings("http://dev.markitondemand.com/Api/v2/Quote/jsonp")
class QuoteModel extends Model implements IModel {

    constructor(mediator : IMediator) {
        super(mediator);
    }

    // listen to model events
    public initialize() {
        this.subscribeToEvents([
            new AppEvent("app.model.quote.change", null, (e, args) => {
                this.onChange(args);
            })
        ]);
    }

    // dispose model events
    public dispose() {
        this.unsubscribeToEvents();
    }
}

```

The second thing that you should notice is that the `onChange` function invokes a new function (`formatModel`) when the promise returned by `getAsync` is fulfilled:

```

private onChange(args) : void {
    // format args
    var s = { symbol : args };
    this.getAsync("jsonp", s)
        .then((data) => {

            // format data
            var quote = this.formatModel(data);

            // pass control to the market view
            this.triggerEvent(new AppEvent("app.view.symbol.render",
                quote, null));
        })
        .catch((e) => {
            // pass control to the global error handler
            this.triggerEvent(new AppEvent("app.error", e, null));
        });
}

```

The new function just formats the response of the web services to be displayed in a user-friendly manner. We could have done this formatting inside the promise fulfillment callback. Using a separate function makes the code significantly cleaner.

```
private formatModel (data) {  
    data.Change = data.Change.toFixed(2);  
    data.ChangePercent = data.ChangePercent.toFixed(2);  
    data.Timestamp = new  
        Date(data.Timestamp).toLocaleDateString();  
    data.MarketCap = (data.MarketCap / 1000000).toFixed(2) + "M. ";  
    data.ChangePercentYTD = data.ChangePercentYTD.toFixed(2);  
    return { quote : data };  
}  
}  
export { QuoteModel };
```

Implementing the symbol view

Let's create a new file named `symbol_view.ts` under the `app/views` directory. The `SymbolView` view receives the stock data formatted by the `QuoteModel` model through the mediator using the `app.view.symbol.render` event:

```
/// <reference path="../../framework/interfaces"/>  
  
import { View, AppEvent, ViewSettings } from "../../framework/  
framework";  
  
@ViewSettings("./source/app/templates/symbol.hbs", "#outlet")  
class SymbolView extends View implements IView {  
  
    constructor(mediator : IMediator) {  
        super(mediator);  
    }  
}
```

This view is just like `MarketView`; it subscribes to some events using the `initialize` method, and later disposes of those events using the `dispose` method. The `SymbolView` view can also initialize and dispose of user events using the `bindDomEvents` and `unbindDomEvents` methods.

However, there is one significant difference between `SymbolView` and `MarketView`. After the promise returned by `renderAsync` has been fulfilled and the user events have been initialized, the execution flow is passed to another model via the `app.model.chart.change` event. At this point, the stock quote screen is visible but it is missing the chart.

```
initialize() : void {
  this.subscribeToEvents([
    new AppEvent("app.view.symbol.render", null, (e, model : any) => {
      this.renderAsync(model)
        .then((model) => {
          // set DOM events
          this.bindDomEvents(model);

          // pass control to chart View
          this.triggerEvent(new
            AppEvent("app.model.chart.change", model.quote.Symbol,
            null));
        })
        .catch((e) => {
          this.triggerEvent(new AppEvent("app.error", e,
            null));
        });
    )),
  ]);
}

public dispose() : void {
  this.unbindDomEvents();
  this.unsubscribeToEvents();
}

// initializes DOM events
protected bindDomEvents(model : any) {
  var scope = $(this._container);
  // set DOM events here
}

// disposes DOM events
protected unbindDomEvents() {
  var scope = this._container;
  // kill DOM events here
}

}

export { SymbolView };
```

Implementing the chart model

Let's create a new file named `chart_model.ts` under the `app/models` directory. This is the last model that we will implement:

```
/// <reference path="../../framework/interfaces"/>

import { Model, AppEvent, ModelSettings } from "../../framework/framework";

@ModelSettings("http://dev.markitondemand.com/Api/v2/InteractiveChart/
jsonp")
class ChartModel extends Model implements IModel {

    constructor(mediator : IMediator) {
        super(mediator);
    }

    // listen to model events
    public initialize() {
        this.subscribeToEvents([
            new AppEvent("app.model.chart.change", null, (e, args) => {
                this.onChange(args);
            })
        ]);
    }

    // dispose model events
    public dispose() {
        this.unsubscribeToEvents();
    }
}
```

This time, we will need to format both the request and the response. We need to encode the request parameter because the web service requires a group of settings that cannot be sent as parameters in the URL without encoding it first.

The `onChange` method uses the browser's `JSON.stringify` function to transform the required web service arguments (a JSON object) into a string. The string is then encoded using the browser's `encodeURIComponent` function so it can be used as a parameter in the URL.

The response is formatted using a method named `formatModel`:

```
private onChange(args) : void {  
  
    // format args (more info at http://dev.markitondemand.com/)  
    var p = {  
        Normalized : false,  
        NumberOfDays : 365,  
        DataPeriod : "Day",  
        Elements : [  
            { Symbol : args , Type : "price", Params : ["ohlc"] }  
        ]  
    };  
    var queryString = "parameters=" +  
        encodeURIComponent(JSON.stringify(p));  
  
    this.getAsync("jsonp", queryString)  
        .then((data) => {  
  
            // format data  
            var chartData = this.formatModel(args, data);  
  
            // pass control to the market view  
            this.triggerEvent(new AppEvent("app.view.chart.render",  
                chartData, null));  
        })  
        .catch((e) => {  
            // pass control to the global error handler  
            this.triggerEvent(new AppEvent("app.error", e, null));  
        });  
}
```

This function is used to format the response from `dev.markitondemand.com`, so it can be used by Highcharts with ease. Highcharts is a library that allows us to render graphs on the client side:

```
private formatModel(symbol, data) {  
    // more info at http://dev.markitondemand.com/  
    // and http://www.highcharts.com/demo/line-time-series  
    var chartData = {  
        title : symbol,  
        series : []  
    };
```

```

var series = [
  { name : "open", data :
    data.Elements[0].DataSeries.open.values },
  { name : "close", data :
    data.Elements[0].DataSeries.close.values },
  { name : "high", data :
    data.Elements[0].DataSeries.high.values },
  { name : "low", data :
    data.Elements[0].DataSeries.low.values }
];

for(var i = 0; i < series.length; i++) {
  var serie = {
    name: series[i].name,
    data: []
  }

  for(var j = 0; j < series[i].data.length; j++) {
    var val = series[i].data[j];
    var d = new Date(data.Dates[j]).getTime();
    serie.data.push([d, val]);
  }

  chartData.series.push(serie);
}
return chartData;
}
}

export { ChartModel };

```

Implementing the chart view

Let's create a new file named `chart_view.ts` under the `app/views` directory. This is the last view that we will implement. This view is almost identical to the previous ones, but there is one significant difference. As the chart is rendered by Highcharts and not Handlebars, we will avoid passing a template URL to the `ViewSettings` decorator:

```

/// <reference path="../../framework/interfaces"/>

import { View, AppEvent, ViewSettings } from "../../framework/
framework";

```

```
@ViewSettings(null, "#chart_container")
class ChartView extends View implements IView {

    constructor(mediator : IMediator) {
        super(mediator);
    }
}
```

The ChartView view is subscribed to the app.view.chart.render event. The event handler is invoked when the ChartModel model has been loaded and formatted, but since we don't need to render a Handlebars template, we will not invoke the renderAsync method here (as we did in all the previous views), and we will invoke a method named renderChart instead:

```
initialize() : void {
    this.subscribeToEvents([
        new AppEvent("app.view.chart.render", null, (e, model : any) =>
{
    this.renderChart(model);
    this.bindDomEvents(model);
}),
]);
}

public dispose() : void {
    this.unbindDomEvents();
    this.unsubscribeToEvents();
}

// initializes DOM events
protected bindDomEvents(model : any) {
    var scope = $(this._container);
    // set DOM events here
}

// disposes DOM events
protected unbindDomEvents() {
    var scope = this._container;
    // kill DOM events here
}
```

The renderChart method uses the Highcharts API (<http://api.highcharts.com/highcharts>) to transform the data returned by ChartModel into a nice looking interactive chart:

```
private renderChart(model) {
    $(this._container).highcharts({
        chart: {
            zoomType: 'x'
        },
        title: {
            text: model.title
        },
        subtitle: {
            text : 'Click and drag in the plot area to zoom in'
        },
        xAxis: {
            type: 'datetime'
        },
        yAxis: {
            title: {
                text: 'Price'
            }
        },
        legend: {
            enabled: true
        },
        tooltip: {
            shared: true,
            crosshairs: true
        },
        plotOptions: {
            area: {
                marker: {
                    radius: 0
                },
                lineWidth: 0.1,
                threshold: null
            }
        },
        series: model.series
    });
}
export { ChartView };
```

Testing the application

We can test this application using the same set of tools that we used in the previous chapters of this book. As you already know, in order to run our unit test, we need to create a Gulp task like the following one:

```
gulp.task("run-unit-test", function(cb) {  
    karma.start({  
        configFile : "karma.conf.js",  
        singleRun: true  
    }, cb);  
});
```

We have used the Karma test runner, and we need to set its configuration using the `karma.conf.js` file. The `karma.conf.js` file is almost identical to the one that we used in *Chapter 7, Application Testing*, and will not be included here for the sake of brevity.

We also need a task to run some end-to-end tests:

```
gulp.task('run-e2e-test', function() {  
    return gulp.src('')  
        .pipe(nightwatch({  
            configFile: 'nightwatch.json'  
        }));  
});
```

The `nightwatch.json` file is almost identical the one that we used in *Chapter 7, Application Testing*, and thus will not be included here.

Refer to the companion source code to see the content of `nightwatch.json` and the `karma.conf.js` file, as well as some examples of unit tests and E2E tests.

Preparing the application for a production release

Now that the application has been implemented and tested, we can prepare it for release in a production environment.

In this section, we will implement two Gulp tasks. The first task is used to compress the output JavaScript code. Compressing the JavaScript code will improve both the loading and execution performance of our application:

```
gulp.task("compress", function() {
    return gulp.src("bundled/source/bundle.js")
        .pipe(uglify({ preserveComments : false }))
        .pipe(gulp.dest("dist/"))
});
```

The second Gulp task that we will implement is used to add a copyright header. The task uses some of the fields from the npm configuration file (`package.json`) to generate a string, which contains the copyright details. The string is then added to the top of the compressed JavaScript file that was generated by the previous task:

```
gulp.task("header", function() {

    var pkg = require("package.json");

    var banner = ["/**",
        " * <%= pkg.name %> v.<%= pkg.version %> - <%= pkg.description %>",
        " * Copyright (c) 2015 <%= pkg.author %>",
        " * <%= pkg.license %>",
        " * <%= pkg.homepage %>",
        " */",
        ""].join("\n");

    return gulp.src("dist/bundle.js")
        .pipe(header(banner, { pkg : pkg } ))
        .pipe(gulp.dest("dist/"));
});
```

We could also create some extra Gulp tasks to improve the performance of our application further. For example, we could create a task to generate a cache manifest (a simple text file that lists the resources the browser should cache for offline access) to implement client-side caching.

Summary

In this chapter, we created an MVC application that allowed us to find out how the NASDAQ and NYSE stocks were doing on a particular day. This application is a single-page web application, and its architecture makes its components easy to extend, reuse, maintain, and test.

The application showcases many of the concepts that we covered in the previous chapters. We created an automated build, and we used many functions, classes, modules, and other core language features. We also used modules and worked with some asynchronous functions, and we used some decorators. The automated build performs some tasks that will help us to improve the application performance and ensures that it works correctly.

This application is not a very large JavaScript application. However, the application is large enough to showcase the ways in which TypeScript can help us develop complex applications that are ready to grow and adapt to changes with ease.

I hope you enjoyed this book and feel eager to learn more about TypeScript.

If you are up for a challenge and you would like to reinforce your TypeScript skills, try the following:

You can try to achieve 100 percent test coverage in the application that we have developed over the last two chapters. You can improve our custom SPA the framework and introduce features such as using an IoC container or using a unidirectional dataflow.

You can also visit the TodoMVC website (<http://todomvc.com/>) to find examples of integration between TypeScript and popular MV* frameworks, such as Ember.js or Backbone.js, to learn how to use a production-ready SPA framework.

Module 2

TypeScript Design Patterns

Boost your development efficiency by learning about design patterns in TypeScript

Module 2: TypeScript Design Patterns

Chapter 1: Tools and Frameworks

Installing the prerequisites	7
Installing Node.js	7
Installing TypeScript compiler	8
Choosing a handy editor	9
Visual Studio Code	9
Configuring Visual Studio Code	10
Opening a folder as a workspace	11
Configuring a minimum build task	12
Sublime Text with TypeScript plugin	13
Installing Package Control	14
Installing the TypeScript plugin	14
Other editor or IDE options	14
Atom with the TypeScript plugin	15
Visual Studio	15
WebStorm	16
Getting your hands on the workflow	16
Configuring a TypeScript project	16
Introduction to tsconfig.json	17
Compiler options	18
target	18
module	18
declaration	18
sourceMap	19
jsx	19
noEmitOnError	19
noEmitHelpers	19
noImplicitAny	19
experimentalDecorators*	19
emitDecoratorMetadata*	20
outDir	20
outFile	20
rootDir	20
preserveConstEnums	21
strictNullChecks	21
stripInternal*	21
isolatedModules	21
Adding source map support	21
Downloading declarations using typings	22
Installing typings	22

Downloading declaration files	23
Option “save”	24
Testing with Mocha and Istanbul	24
Mocha and Chai	24
Writing tests in JavaScript	25
Writing tests in TypeScript	25
Getting coverage information with Istanbul	27
Testing in real browsers with Karma	28
Creating a browser project	28
Installing Karma	30
Configuring and starting Karma	30
Integrating commands with npm	31
Why not other fancy build tools?	31
Summary	32
Chapter 2: The Challenge of Increasing Complexity	33
Implementing the basics	34
Creating the code base	34
Defining the initial structure of the data to be synchronized	35
Getting data by comparing timestamps	35
Two-way synchronizing	36
Things that went wrong while implementing the basics	37
Passing a data store from the server to the client does not make sense	37
Making the relationships clear	38
Growing features	39
Synchronizing multiple items	39
Simply replacing data type with an array	39
Server-centered synchronization	39
Synchronizing from the server to the client	40
Synchronizing from client to server	44
Synchronizing multiple types of data	49
Supporting multiple clients with incremental data	50
Updating the client side	51
Updating server side	55
Supporting more conflict merging	57
New data structures	57
Updating client side	58
Updating the server side	60
Things that go wrong while implementing everything	60
Piling up similar yet parallel processes	61
Data stores that are tremendously simplified	61
Getting things right	62
Finding abstraction	62
Implementing strategies	63

Wrapping stores	64
Summary	65
Chapter 3: Creational Design Patterns	66
Factory method	68
Participants	69
Pattern scope	69
Implementation	69
Consequences	72
Abstract Factory	73
Participants	74
Pattern scope	75
Implementation	75
Consequences	79
Builder	79
Participants	80
Pattern scope	81
Implementation	81
Consequences	86
Prototype	86
Singleton	87
Basic implementations	87
Conditional singletons	89
Summary	89
Chapter 4: Structural Design Patterns	90
Composite Pattern	90
Participants	92
Pattern scope	92
Implementation	92
Consequences	94
Decorator Pattern	95
Participants	96
Pattern scope	97
Implementation	97
Classical decorators	97
Decorators with ES-next syntax	100
Consequences	101
Adapter Pattern	101
Participants	103
Pattern scope	103

Implementation	103
Consequences	106
Bridge Pattern	106
Participants	106
Pattern scope	107
Implementation	107
Consequences	109
Façade Pattern	110
Participants	111
Pattern scope	112
Implementation	112
Consequences	114
Flyweight Pattern	114
Participants	115
Pattern scope	116
Implementation	116
Consequences	118
Proxy Pattern	118
Participants	119
Pattern scope	120
Implementation	120
Consequences	123
Summary	123
Chapter 5: Behavioral Design Patterns	124
Chain of Responsibility Pattern	124
Participants	127
Pattern scope	128
Implementation	128
Consequences	130
Command Pattern	130
Participants	132
Pattern scope	132
Implementation	133
Consequences	134
Memento Pattern	135
Participants	136
Pattern scope	136
Implementation	136
Consequences	138

Iterator Pattern	138
Participants	139
Pattern scope	139
Implementation	139
Simple array iterator	140
ES6 iterator	141
Consequences	143
Mediator Pattern	143
Participants	144
Pattern scope	145
Implementation	145
Consequences	147
Summary	148
Chapter 6: Behavioral Design Patterns: Continuous	149
Strategy Pattern	150
Participants	151
Pattern scope	152
Implementation	152
Consequences	154
State Pattern	154
Participants	155
Pattern scope	156
Implementation	156
Consequences	158
Template Method Pattern	158
Participants	159
Pattern scope	160
Implementation	160
Consequences	162
Observer Pattern	162
Participants	166
Pattern scope	167
Implementation	167
Consequences	169
Visitor Pattern	170
Participants	172
Pattern scope	173
Implementation	173
Consequences	176

Summary	176
Chapter 7: Patterns and Architectures in JavaScript and TypeScript	178
Promise-based web architecture	178
Promisifying existing modules or libraries	180
Views and controllers in Express	181
Abstraction of responses	184
Abstraction of permissions	186
Expected errors	187
Defining and throwing expected errors	188
Transforming errors	188
Modularizing project	189
Asynchronous patterns	191
Writing predictable code	191
Asynchronous creational patterns	193
Asynchronous middleware and hooks	194
Event-based stream parser	195
Summary	197
Chapter 8: SOLID Principles	198
Single responsibility principle	199
Example	199
Choosing an axis	200
Open-closed principle	201
Example	201
Abstraction in JavaScript and TypeScript	202
Refactor earlier	203
Liskov substitution principle	203
Example	204
The constraints of substitution	205
Interface segregation principle	205
Example	205
Proper granularity	207
Dependency inversion principle	207
Example	207
Separating layers	207
Summary	208
Chapter 9: The Road to Enterprise Application	209
Creating an application	210
Decision between SPA and “normal” web applications	210

Taking team collaboration into consideration	211
Building and testing projects	211
Static assets packaging with webpack	212
Introduction to webpack	212
Bundling JavaScript	212
Loading TypeScript	214
Splitting code	216
Loading other static assets	217
Adding TSLint to projects	217
Integrating webpack and tslint command with npm scripts	218
Version control	218
Git flow	219
Main branches	220
Supporting branches	220
Feature branches	220
Release branches	221
Hotfix branches	222
Summary of Git flow	222
Pull request based code review	223
Configuring branch permissions	223
Comments and modifications before merge	223
Testing before commits	224
Git hooks	224
Adding pre-commit hook automatically	224
Continuous integration	225
Connecting GitHub repository with Travis-CI	225
Deployment automation	226
Passive deployment based on Git server side hooks	227
Proactive deployment based on timers or notifications	228
Summary	228
Index	230

1

Tools and Frameworks

We could always use the help of real code to explain the design patterns we'll be discussing. In this chapter, we'll have a brief introduction to the tools and frameworks that you might need if you want to have some practice with complete working implementations of the contents of this book.

In this chapter, we'll cover the following topics:

- Installing Node.js and TypeScript compiler
- Popular editors or IDEs for TypeScript
- Configuring a TypeScript project
- A basic workflow that you might need to play with your own implementations of the design patterns in this book

Installing the prerequisites

The contents of this chapter are expected to work on all major and up-to-date desktop operating systems, including Windows, OS X, and Linux.

As Node.js is widely used as a runtime for server applications as well as frontend build tools, we are going to make it the main playground of code in this book.

TypeScript compiler, on the other hand, is the tool that compiles TypeScript source files into plain JavaScript. It's available on multiple platforms and runtimes, and in this book we'll be using the Node.js version.

Installing Node.js

Installing Node.js should be easy enough. But there's something we could do to minimize incompatibility over time and across different environments:

- **Version:** We'll be using Node.js 6 with npm 3 built-in in this book. (The major version of Node.js may increase rapidly over time, but we can expect minimum breaking changes directly related to our contents. Feel free to try a newer version if it's available.)
- **Path:** If you are installing Node.js without a package manager, make sure the environment variable `PATH` is properly configured.

Open a console (a command prompt or terminal, depending on your operating system) and make sure Node.js as well as the built-in package manager `npm` is working:

```
$ node -v  
6.x.x  
$ npm -v  
3.x.x
```

Installing TypeScript compiler

TypeScript compiler for Node.js is published as an npm package with command line interface. To install the compiler, we can simply use the `npm install` command:

```
$ npm install typescript -g
```

Option `-g` means a global installation, so that `tsc` will be available as a command. Now let's make sure the compiler works:

```
$ tsc -v  
Version 2.x.x
```



You may get a rough list of the options your TypeScript compiler provides with switch `-h`. Taking a look into these options may help you discover some useful features.

Before choosing an editor, let's print out the legendary phrase:

1. Save the following code to file `test.ts`:

```
function hello(name: string): void {
    console.log(`hello, ${name}!`);
}

hello('world');
```

2. Change the working directory of your console to the folder containing the created file, and compile it with `tsc`:

```
$ tsc test.ts
```

3. With luck, you should have the compiled JavaScript file as `test.js`. Execute it with Node.js to get the ceremony done:

```
$ node test.js
hello, world!
```

Here we go, on the road to retire your CTO.

Choosing a handy editor

A compiler without a good editor won't be enough (if you are not a believer of Notepad). Thanks to the efforts made by the TypeScript community, there are plenty of great editors and IDEs ready for TypeScript development.

However, the choice of an editor could be much about personal preferences. In this section, we'll talk about the installation and configuration of Visual Studio Code and Sublime Text. But other popular editors or IDEs for TypeScript will also be listed with brief introductions.

Visual Studio Code

Visual Studio Code is a free lightweight editor written in TypeScript. And it's an open source and cross-platform editor that already has TypeScript support built-in.

You can download Visual Studio Code from <https://code.visualstudio.com/> and the installation will probably take no more than 1 minute.

The following screenshot shows the debugging interface of Visual Studio Code with a TypeScript source file:

The screenshot displays the Visual Studio Code interface in DEBUG mode. The left pane shows the TypeScript code in 'index.ts':

```
index.ts src
1 function foo(...args: string[]): void {
2     let line = args.join(' ');
3     console.log(line);
4 }
5
6 foo("node", "index");
7 }
```

A yellow circular breakpoint icon is positioned next to the first line of code. The right pane contains the DEBUG tool bar with icons for play, step, and stop, followed by 'DEBUG', 'Launch', and settings. The DEBUG sidebar is open, showing the following sections:

- VARIABLES**:
 - Local
 - _i: 2
 - args: Array[2]
 - arguments: #<Object>
 - line: "node index.js"
 - this: #<Object>
- WATCH**: No items listed.
- CALL STACK**:
 - Paused on breakpoint.
 - foo index.ts 3
(anonymous function)
 - Module._compile module.js 398
 - Module_extensions.js
 - Module.load module.js 344
 - Module_load module.js 301
- BREAKPOINTS**: No items listed.

The bottom status bar shows 'Ln 7, Col 1' and other system information.

Configuring Visual Studio Code

As Code already has TypeScript support built-in, extra configurations are actually not required. But if the version of TypeScript compiler you use to compile the source code differs from what Code has built-in, it could result in un conformity between editing and compiling.

To stay away from the undesired issues this would bring, we need to configure TypeScript SDK used by Visual Studio Code manually:

1. Press **F1**, type `Open User Settings`, and enter. Visual Studio Code will open the settings JSON file by the side of a read-only JSON file containing all the default settings.
2. Add the field `typescript.tsdk` with the path of the `lib` folder under the TypeScript package we previously installed:
 1. Execute the command `npm root -g` in your console to get the root of global Node.js modules.
 2. Append the root path with `/typescript/lib` as the SDK path.



You can also have a TypeScript package installed locally with the project, and use the local TypeScript `lib` path for Visual Studio Code. (You will need to use the locally installed version for compiling as well.)

Opening a folder as a workspace

Visual Studio Code is a file- and folder-based editor, which means you can open a file or a folder and start work.

But you still need to properly configure the project to take the best advantage of Code. For TypeScript, the project file is `tsconfig.json`, which contains the description of source files and compiler options. Know little about `tsconfig.json`? Don't worry, we'll come to that later.

Here are some features of Visual Studio Code you might be interested in:

- **Tasks:** Basic task integration. You can build your project without leaving the editor.
- **Debugging:** Node.js debugging with source map support, which means you can debug Node.js applications written in TypeScript.
- **Git:** Basic Git integration. This makes comparing and committing changes easier.

Configuring a minimum build task

The default key binding for a build task is *Ctrl + Shift + B* or *cmd + Shift + B* on OS X. By pressing these keys, you will get a prompt notifying you that no task runner has been configured. Click **Configure Task Runner** and then select a TypeScript build task template (either with or without the watch mode enabled). A `tasks.json` file under the `.vscode` folder will be created automatically with content similar to the following:

```
{  
  "version": "0.1.0",  
  "command": "tsc",  
  "isShellCommand": true,  
  "args": ["-w", "-p", "."],  
  "showOutput": "silent",  
  "isWatching": true,  
  "problemMatcher": "$tsc-watch"  
}
```

Now create a `test.ts` file with some hello-world code and run the build task again. You can either press the shortcut we mentioned before or press `Ctrl/Cmd + P`, type `task tsc`, and enter.

If you were doing things correctly, you should be seeing the output `test.js` by the side of `test.ts`.

There are some useful configurations for tasking that can't be covered. You may find more information on the website of Visual Studio Code: <https://code.visualstudio.com/>.

From my perspective, Visual Studio Code delivers the best TypeScript development experience in the class of code editors. But if you are not a fan of it, TypeScript is also available with official support for Sublime Text.

Sublime Text with TypeScript plugin

Sublime Text is another popular lightweight editor around the field with amazing performance.

The following image shows how TypeScript IntelliSense works in Sublime Text:

A screenshot of the Sublime Text editor interface. The title bar shows "C:\Users\vilicvane\Desktop\typescript\src\index.ts • (typescript) - Sublime Text". The menu bar includes File, Edit, Selection, Find, Goto, Tools, Project, Preferences, and Help. On the left, a sidebar titled "FOLDERS" shows a project structure with "typescript", ".vscode", "out", and "src" folders. Inside "src", there are "index.ts", "typings", and "tsconfig.json" files. The main editor area has a dark theme and displays the following TypeScript code:

```
index.ts
1 function foo(...args: string[]): void {
2     let line = args.join(' ');
3     console.log(line);
4 }
5
6 foo();
7 foo(...args: string[]): void
    args:
        1/1
```

The code editor highlights the word "args" in orange, indicating it's a parameter name. A tooltip or dropdown menu is open over the word "args" at line 7, showing the type annotation "string[]: void" and the value "1/1". The status bar at the bottom left shows "Line 6, Column 5". The bottom right of the status bar shows "Tab Size: 4" and "TypeScript".

The TypeScript team has officially built a plugin for Sublime Text (version 3 preferred), and you can find a detailed introduction, including useful shortcuts, in their GitHub repository here: <https://github.com/Microsoft/TypeScript-Sublime-Plugin>.



There are still some issues with the TypeScript plugin for Sublime Text. It would be nice to know about them before you start writing TypeScript with Sublime Text.

Installing Package Control

Package Control is de facto package manager for Sublime Text, with which we'll install the TypeScript plugin.

If you don't have Package Control installed, perform the following steps:

1. Click **Preferences > Browse Packages...**, it opens the Sublime Text packages folder.
2. Browse up to the parent folder and then into the **Install Packages** folder, and download the file below into this folder: <https://packagecontrol.io/Package%2Control.sublime-package>
3. Restart Sublime Text and you should now have a working package manager.

Now we are only one step away from IntelliSense and refactoring with Sublime Text.

Installing the TypeScript plugin

With the help of Package Control, it's easy to install a plugin:

1. Open the Sublime Text editor; press Ctrl + Shift + P for Windows and Linux or Cmd + Shift + P for OS X.
2. Type **Install Package** in the command palette, select **Package Control: Install Package** and wait for it to load the plugin repositories.
3. Type **TypeScript** and select to install the official plugin.

Now we have TypeScript ready for Sublime Text, cheers!

Like Visual Studio Code, unmatched TypeScript versions between the plugin and compiler could lead to problems. To fix this, you can add the field "`"typescript_tsdk"`" with a path to the TypeScript `lib` in the **Settings – User** file.

Other editor or IDE options

Visual Studio Code and Sublime Text are recommended due to their ease of use and popularity respectively. But there are many great tools from the editor class to full-featured IDE.

Though we're not going through the setup and configuration of those tools, you might want to try them out yourself, especially if you are already working with some of them.

However, the configuration for different editors and IDEs (especially IDEs) could differ. It is recommended to use Visual Studio Code or Sublime Text for going through the workflow and examples in this book.

Atom with the TypeScript plugin

Atom is a cross-platform editor created by GitHub. It has a notable community with plenty of useful plugins, including `atom-typescript`. `atom-typescript` is the result of the hard work of Basarat Ali Syed, and it's used by my team before Visual Studio Code. It has many handy features that Visual Studio Code does not have yet, such as module path suggestion, compile on save, and so on.

Like Visual Studio Code, Atom is also an editor based on web technologies. Actually, the shell used by Visual Studio Code is exactly what's used by Atom: Electron, another popular project by GitHub, for building cross-platform desktop applications.

Atom is proud of being hackable, which means you can customize your own Atom editor pretty much as you want.

Then you may be wondering why we turned to Visual Studio Code. The main reason is that Visual Studio Code is being backed by the same company that develops TypeScript, and another reason might be the performance issue with Atom.

But anyway, Atom could be a great choice for a start.

Visual Studio

Visual Studio is one of the best IDEs in the market. And yet it has, of course, official TypeScript support.

Since Visual Studio 2013, a community version is provided for free to individual developers, small companies, and open source projects.

If you are looking for a powerful IDE of TypeScript on Windows, Visual Studio could be a wonderful choice. Though Visual Studio has built-in TypeScript support, do make sure it's up-to-date. And, usually, you might want to install the newest TypeScript tools for Visual Studio.

WebStorm

WebStorm is one of the most popular IDEs for JavaScript developers, and it has had an early adoption to TypeScript as well.

A downside of using WebStorm for TypeScript is that it is always one step slower catching up to the latest version compared to other major editors. Unlike editors that directly use the language service provided by the TypeScript project, WebStorm seems to have its own infrastructure for IntelliSense and refactoring. But, in return, it makes TypeScript support in WebStorm more customizable and consistent with other features it provides.

If you decide to use WebStorm as your TypeScript IDE, please make sure the version of supported TypeScript matches what you expect (usually the latest version).

Getting your hands on the workflow

After setting up your editor, we are ready to move to a workflow that you might use to practice throughout this book. It can also be used as the workflow for small TypeScript projects in your daily work.

In this workflow, we'll walk through these topics:

- What is a `tsconfig.json` file, and how can you configure a TypeScript project with it?
- TypeScript declaration files and the `typings` command-line tool
- How to write tests running under Mocha, and how to get coverage information using Istanbul
- How to test in browsers using Karma

Configuring a TypeScript project

The configurations of a TypeScript project can differ for a variety of reasons. But the goals remain clear: we need the editor as well as the compiler to recognize a project and its source files correctly. And `tsconfig.json` will do the job.

Introduction to tsconfig.json

A TypeScript project does not have to contain a `tsconfig.json` file. However, most editors rely on this file to recognize a TypeScript project with specified configurations and to provide related features.

A `tsconfig.json` file accepts three fields: `compilerOptions`, `files`, and `exclude`. For example, a simple `tsconfig.json` file could be like the following:

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "rootDir": "src",  
    "outDir": "out"  
  },  
  "exclude": [  
    "out",  
    "node_modules"  
  ]  
}
```

Or, if you prefer to manage the source files manually, it could be like this:

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "rootDir": "src",  
    "outDir": "out"  
  },  
  "files": [  
    "src/foo.ts",  
    "src/bar.ts"  
  ]  
}
```

Previously, when we used `tsc`, we needed to specify the source files explicitly. Now, with `tsconfig.json`, we can directly run `tsc` without arguments (or with `-w`/`--watch` if you want incremental compilation) in a folder that contains `tsconfig.json`.

Compiler options

As TypeScript is still evolving, its compiler options keep changing, with new features and updates. An invalid option may break the compilation or editor features for TypeScript. When reading these options, keep in mind that some of them might have been changed.

The following options are useful ones out of the list.

target

`target` specifies the expected version of JavaScript outputs. It could be `es5` (ECMAScript 5), `es6` (ECMAScript 6/2015), and so on.

Features (especially ECMAScript polyfills) that are available in different compilation targets vary. For example, before TypeScript 2.1, features such as `async/await` were available only when targeting ES6.

The good news is that Node.js 6 with the latest V8 engine has supported most ES6 features. And the latest browsers have also great ES6 support. So if you are developing a Node.js application or a browser application that's not required for backward compatibilities, you can have your configuration target ES6.

module

Before ES6, JavaScript had no standard module system. Varieties of module loaders are developed for different scenarios, such as `commonjs`, `amd`, `umd`, `system`, and so on.

If you are developing a Node.js application or an npm package, `commonjs` could be the value of this option. Actually, with the help of modern packaging tools such as `webpack` and `browserify`, `commonjs` could also be a nice choice for browser projects as well.

declaration

Enable this option to generate `.d.ts` declaration files along with JavaScript outputs. Declaration files could be useful as the type information source of a distributed library/framework; it could also be helpful for splitting a larger project to improve compiling performance and division cooperation.

sourceMap

By enabling this option, TypeScript compiler will emit source maps along with compiled JavaScript.

jsx

TypeScript provides built-in support for React JSX (`.tsx`) files. By specifying this option with value `react`, TypeScript compiler will compile `.tsx` files to plain JavaScript files. Or with value `preserve`, it will output `.jsx` files so you can post-process these files with other JSX compilers.

noEmitOnError

By default, TypeScript will emit outputs no matter whether type errors are found or not. If this is not what you want, you may set this option to `true`.

noEmitHelpers

When compiling a newer ECMAScript feature to a lower target version of JavaScript, TypeScript compiler will sometimes generate helper functions such as `__extends` (ES6 to lower versions), and `__awaiter` (ES7 to lower versions).

Due to certain reasons, you may want to write your own helper functions, and prevent TypeScript compiler from emitting these helpers.

noImplicitAny

As TypeScript is a superset of JavaScript, it allows variables and parameters to have no type notation. However, it could help to make sure everything is typed.

By enabling this option, TypeScript compiler will give errors if the type of a variable/parameter is not specified and cannot be inferred by its context.

experimentalDecorators*

As decorators, at the time of writing this book, has not yet reached a stable stage of the new ECMAScript standard, you need to enable this option to use decorators.

emitDecoratorMetadata*

Runtime type information could sometimes be useful, but TypeScript does not yet support reflection (maybe it never will). Luckily, we get decorator metadata that will help under certain scenarios.

By enabling this option, TypeScript will emit decorators along with a `Reflect.metadata()` decorator which contains the type information of the decorated target.

outDir

Usually, we do not want compiled files to be in the same folder of source code. By specifying `outDir`, you can tell the compiler where you would want the compiled JavaScript files to be.

outFile

For small browser projects, we might want to have all the outputs concatenated as a single file. By enabling this option, we can achieve that without extra build tools.

rootDir

The `rootDir` option is to specify the root of the source code. If omitted, the compiler would use the longest common path of source files. This might take seconds to understand.

For example, if we have two source files, `src/foo.ts` and `src/bar.ts`, and a `tsconfig.json` file in the same directory of the `src` folder, the TypeScript compiler will use `src` as the `rootDir`, so when emitting files to the `outDir` (let's say `out`), they will be `out/foo.js` and `out/bar.js`.

However, if we add another source file `test/test.ts` and compile again, we'll get those outputs located in `out/src/foo.js`, `out/src/bar.js`, and `out/test/test.js` respectively. When calculating the longest common path, declaration files are not involved as they have no output.

Usually, we don't need to specify `rootDir`, but it would be safer to have it configured.

preserveConstEnums

Enum is a useful tool provided by TypeScript. When compiled, it's in the form of an `Enum.member` expression. Constant enum, on the other hand, emits number literals directly, which means the `Enum` object is no longer necessary.

And thus TypeScript, by default, will remove the definitions of constant enums in the compiled JavaScript files.

By enabling this option, you can force the compiler to keep these definitions anyway.

strictNullChecks

TypeScript 2.1 makes it possible to explicitly declare a type with `undefined` or `null` as its subtype. And the compiler can now perform more thorough type checking for empty values if this option is enabled.

stripInternal*

When emitting declaration files, there could be something you'll need to use internally but without a better way to specify the accessibility. By commenting this code with `/** @internal */` (JSDoc annotation), TypeScript compiler then won't emit them to declaration files.

isolatedModules

By enabling this option, the compiler will unconditionally emit imports for unresolved files.



Options suffixed with * are experimental and might have already been removed when you are reading this book. For a more complete and up-to-date compiler options list, please check out <http://www.typescriptlang.org/docs/handbook/compiler-options.html>.

Adding source map support

Source maps can help a lot while debugging, no matter for a debugger or for error stack traces from a log.

To have source map support, we need to enable the `sourceMap` compiler option in `tsconfig.json`. Extra configurations might be necessary to make your debugger work with source maps.

For error stack traces, we can use the help of the `source-map-support` package:

```
$ npm install source-map-support --save
```

To put it into effect, you can import this package with its `register` submodule in your entry file:

```
import 'source-map-support/register';
```

Downloading declarations using typings

JavaScript has a large and booming ecosystem. As the bridge connecting TypeScript and other JavaScript libraries and frameworks, declaration files are playing a very important role.

With the help of declaration files, TypeScript developer can use existing JavaScript libraries with nearly the same experience as libraries written in TypeScript.

Thanks to the efforts of the TypeScript community, almost every popular JavaScript library or framework got its declaration files on a project called *DefinitelyTyped*. And there has already been a tool called `tsd` for declaration file management. But soon, people realized the limitation of a single huge repository for everything, as well as the issues `tsd` cannot solve nicely. Then `typings` is gently becoming the new tool for TypeScript declaration file management.

Installing typings

`typings` is just another Node.js package with a command-line interface like TypeScript compiler. To install `typings`, simply execute the following:

```
$ npm install typings -g
```

To make sure it has been installed correctly, you can now try the `typings` command with argument `--version`:

```
$ typings --version
1.x.x
```

Downloading declaration files

Create a basic Node.js project with a proper `tsconfig.json` (module option set as `commonjs`), and a `test.ts` file:

```
import * as express from 'express';
```

Without the necessary declaration files, the compiler would complain with **Cannot find module express**. And, actually, you can't even use Node.js APIs such as `process.exit()` or require Node.js modules, because TypeScript itself just does not know what Node.js is.

To begin with, we'll need to install declaration files of Node.js and Express:

```
$ typings install env~node --global  
$ typings install express
```

If everything goes fine, `typings` should've downloaded several declaration files and saved them to folder `typings`, including `node.d.ts`, `express.d.ts`, and so on. And I guess you've already noticed the dependency relationship existing on declaration files.

If this is not working for you and `typings` complains with **Unable to find “express” (“npm”) in the registry** then you might need to do it the hard way – to manually install Express declaration files and their dependencies using the following command:

 `$ typings install dt~<package-name> --global`

The reason for that is the community might still be moving from DefinitelyTyped to the `typings` registry. The prefix `dt~` tells `typings` to download declaration files from DefinitelyTyped, and `--global` option tells `typings` to save these declaration files as ambient modules (namely declarations with module name specified).

`typings` has several registries, and the default one is called `npm` (please understand this `npm` registry is not the `npm` package registry). So, if no registry is specified with `<source>` prefix or `--source` option, it will try to find declaration files from its `npm` registry. This means that `typings install express` is equivalent to `typings install npm~express` or `typings install express --source npm`.

While declaration files for `npm` packages are usually available on the `npm` registry, declaration files for the environment are usually available on the `env.` registry. As those declarations are usually global, a `--global` option is required for them to install correctly.

Option “save”

typings actually provides a `--save` option for saving the typing names and file sources to `typings.json`. However, in my opinion, this option is not practically useful.

It's great to have the most popular JavaScript libraries and frameworks typed, but these declaration files, especially declarations not frequently used, can be inaccurate, which means there's a fair chance that you will need to edit these files yourself.

It would be nice to contribute declarations, but it would also be more flexible to have `typings.m` managed by source control as part of the project code.

Testing with Mocha and Istanbul

Testing could be an important part of a project, which ensures feature consistency and discovers bugs earlier. It is common that a change made for one feature could break another working part of the project. A robust design could minimize the chance but we still need tests to make sure.

It could lead to an endless discussion about how tests should be written and there are interesting code design techniques such as **test-driven development (TDD)**; though there has been a lot of debates around it, it still worth knowing and may inspire you in certain ways.

Mocha and Chai

Mocha has been one of the most popular test frameworks for JavaScript, while Chai is a good choice as an assertion library. To make life easier, you may write tests for your own implementations of contents through this book using Mocha and Chai.

To install Mocha, simply run the following command, and it will add `mocha` as a global command-line tool just like `tsc` and `typings`:

```
$ npm install mocha -g
```

Chai, on the other hand, is used as a module of a project, and should be installed under the project folder as a development dependency:

```
$ npm install chai --save-dev
```

Chai supports `should` style assertion. By invoking `chai.should()`, it adds the `should` property to the prototype of `Object`, which means you can then write assertions such as the following:

```
'foo'.should.not.equal('bar');
'typescript'.should.have.length(10);
```

Writing tests in JavaScript

By executing the command `mocha`, it will automatically run tests inside the `test` folder. Before we start to write tests in TypeScript, let's try it out in plain JavaScript and make sure it's working.

Create a file `test/starter.js` and save it with the following code:

```
require('chai').should();

describe('some feature', () => {
  it('should pass', () => {
    'foo'.should.not.equal('bar');
  });

  it('should error', () => {
    () => {
      throw new Error();
    }).should.throw();
  });
});
```

Run `mocha` under the project folder and you should see all tests passing.

Writing tests in TypeScript

Tests written in TypeScript have to be compiled before being run; where to put those files could be a tricky question to answer.

Some people might want to separate tests with their own `tsconfig.json`:

```
src/tsconfig.json
test/tsconfig.json
```

They may also want to put output files somewhere reasonable:

```
out/app/
out/test/
```

However, this will increase the cost of build process management for small projects. So, if you do not mind having `src` in the paths of your compiled files, you can have only one `tsconfig.json` to get the job done:

```
src/  
test/  
tsconfig.json
```

The destinations will be as follows:

```
out/src/  
out/test/
```

Another option I personally prefer is to have tests inside of `src/test`, and use the `test` folder under the project root for Mocha configurations:

```
src/  
src/test/  
tsconfig.json
```

The destinations will be as follows:

```
out/  
out/test/
```

But, either way, we'll need to configure Mocha properly to do the following:

- Run tests under the `out/test` directory
- Configure the assertion library and other tools before starting to run tests

To achieve these, we can take advantage of the `mocha.opts` file instead of specifying command-line arguments every time. Mocha will combine lines in the `mocha.opts` file with other command-line arguments given while being loaded.

Create `test/mocha.opts` with the following lines:

```
--require ./test/mocha.js  
out/test/
```

As you might have guessed, the first line is to tell Mocha to require `./test/mocha.js` before starting to run actual tests. And the second line tells Mocha where these tests are located.

And, of course, we'll need to create `test/mocha.js` correspondingly:

```
require('chai').should();
```

Almost ready to write tests in TypeScript! But TypeScript compiler does not know how would function describe or it be like, so we need to download declaration files for Mocha:

```
$ typings install env~mocha --global
```

Now we can migrate the `test/starter.js` file to `src/test/starter.ts` with nearly no change, but removing the first line that enables the `should` style assertion of Chai, as we have already put it into `test/mocha.js`.

Compile and run; buy me a cup of coffee if it works. But it probably won't. We've talked about how TypeScript compiler determines the root of source files when explaining the `rootDir` compiler option. As we don't have any TypeScript files under the `src` folder (not including its subfolders), TypeScript compiler uses `src/test` as the `rootDir`. Thus the compiled test files are now under the `out` folder instead of the expected `out/test`.

To fix this, either explicitly specify `rootDir`, or just add the first non-test TypeScript file to the `src` folder.

Getting coverage information with Istanbul

Coverage could be important for measuring the quality of tests. However, it might take much effort to reach a number close to 100%, which could be a burden for developers. To balance efforts on tests and code that bring direct value to the product, there would go another debate.

Install Istanbul via `npm` just as with the other tools:

```
$ npm install istanbul -g
```

The subcommand for Istanbul to generate code coverage information is `istanbul cover`. It should be followed by a JavaScript file, but we need to make it work with Mocha, which is a command-line tool. Luckily, the entry of the Mocha command is, of course, a JavaScript file.

To make them work together, we'll need to install a local (instead of global) version of Mocha for the project:

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

After installation, we'll get the file `_mocha` under `node_modules/mocha/bin`, which is the JavaScript entry we were looking for. So now we can make Istanbul work:

```
$ istanbul cover node_modules/mocha/bin/_mocha
```

Then you should've got a folder named `coverage`, and within it the coverage report.

Reviewing the coverage report is important; it can help you decide whether you need to add tests for specific features and code branches.

Testing in real browsers with Karma

We've talked about testing with Mocha and Istanbul for Node.js applications. It is an important topic for testing code that runs in a browser as well.

Karma is a test runner for JavaScript that makes testing in real browsers on real devices much easier. It officially supports the Mocha, Jasmine, and JUnit testing frameworks, but it's also possible for Karma to work with any framework via a simple adapter.

Creating a browser project

A TypeScript application that runs in browsers can be quite different from a Node.js one. But if you know what the project should look like after the build, you should already have clues on how to configure that project.

To avoid introducing too many concepts and technologies not directly related, I will keep things as simple as possible:

- We're not going to use module loaders such as Require.js
- We're not going to touch the code packaging process

This means we'll go with separated output files that need to be put into an HTML file with a `script` tag manually. Here's the `tsconfig.json` we'll be playing with; notice that we no longer have the `module` option, set:

```
{
  "compilerOptions": {
    "target": "es5",
    "rootDir": "src",
    "outDir": "out"
  },
  "exclude": [
    "out",
```

```
        "node_modules"  
    ]  
}
```

Then let's create `package.json` and install packages `mocha` and `chai` with their declarations:

```
$ npm init  
$ npm install mocha chai --save-dev  
$ typings install env~mocha --global  
$ typings install chai
```

And to begin with, let's fill this project with some source code and tests.

Create `src/index.ts` with the following code:

```
function getLength(str: string): number {  
    return str.length;  
}
```

And create `src/test/test.ts` with some tests:

```
describe('get length', () => {  
    it('"abc" should have length 3', () => {  
        getLength('abc').should.equal(3);  
    });  
  
    it('"" should have length 0', () => {  
        getLength('').should.equal(0);  
    });  
});
```

Again, in order to make the `should` style assertion work, we'll need to call `chai.should()` before tests start. To do so, create file `test/mocha.js` just like we did in the Node.js project, though the code line slightly differs, as we no longer use modules:

```
chai.should();
```

Now compile these files with `tsc`, and we've got our project ready.

Installing Karma

Karma itself runs on Node.js, and is available as an npm package just like other Node.js tools we've been using. To install Karma, simply execute the `npm install` command in the project directory:

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

And, in our case, we are going to have Karma working with Mocha, Chai, and the browser Chrome, so we'll need to install related plugins:

```
$ npm install karma-mocha karma-chai karma-chrome-launcher --save-dev
```

Before we configure Karma, it is recommended to have `karma-cli` installed globally so that we can execute the `karma` command from the console directly:

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

Configuring and starting Karma

The configurations are to tell Karma about the testing frameworks and browsers you are going to use, as well as other related information such as source files and tests to run.

To create a Karma configuration file, execute `karma init` and answer its questions:

- **Testing framework:** Mocha
- **Require.js:** no
- **Browsers:** Chrome (add more if you like; be sure to install the corresponding launchers)
- **Source and test files:**
 - `test/mocha.js` (the file enables `should` style assertion)
 - `out/*.js` (source files)
 - `out/test/*.js` (test files)
- **Files to exclude:** empty
- **Watch for changes:** yes

Now you should see a `karma.conf.js` file under the project directory; open it with your editor and add '`chai`' to the list of option frameworks.

Almost there! Execute the command `karma start` and, if everything goes fine, you should have specified browsers opened with the testing results being logged in the console in seconds.

Integrating commands with npm

The npm provides a simple but useful way to define custom scripts that can be run with the `npm run` command. And it has another advantage – when `npm run` a custom script, it adds `node_modules/.bin` to the PATH. This makes it easier to manage project-related command-line tools.

For example, we've talked about Mocha and Istanbul. The prerequisite for having them as commands is to have them installed globally, which requires extra steps other than `npm install`. Now we can simply save them as development dependencies, and add custom scripts in `package.json`:

```
"scripts": {  
  "test": "mocha",  
  "cover": "istanbul cover node_modules/mocha/bin/_mocha"  
},  
"devDependencies": {  
  "mocha": "latest",  
  "istanbul": "latest"  
}
```

Now you can run `test` with `npm run test` (or simply `npm test`), and run `cover` with `npm run cover` without installing these packages globally.

Why not other fancy build tools?

You might be wondering: why don't we use a build system such as Gulp to set up our workflow? Actually, when I started to write this chapter, I did list Gulp as the tool we were going to use. Later, I realized it does not make much sense to use Gulp to build the implementations in most of the chapters in this book.

There is a message I want to deliver: *balance*.

Once, I had a discussion on *balance versus principles* with my boss. The disagreement was clear: he insisted on controllable principles over subjective balance, while I prefer contextual balance over fixed principles.

Actually, I agree with him, from the point of view of a team leader. A team is usually built up with developers at different levels; principles make it easier for a team to build high-quality products, while not everyone is able to find the right balance point.

However, when the role turns from a productive team member to a learner, it is important to learn and to feel the right balance point. And that's called experience.

Summary

The goal of this chapter was to introduce a basic workflow that could be used by the reader to implement the design patterns we'll be discussing.

We talked about the installation of TypeScript compiler that runs on Node.js, and had brief introductions to popular TypeScript editors and IDEs. Later, we spent quite a lot of pages walking through the tools and frameworks that could be used if the reader wants to have some practice with implementations of the patterns in this book.

With the help of these tools and frameworks, we've built a minimum workflow that includes creating, building, and testing a project. And talking about workflows, you must have noticed that they slightly differ among applications for different runtimes.

In the next chapter, we'll talk about what may go wrong and mess up the entire project when its complexity keeps growing. And we'll try to come up with specific patterns that can solve the problems this very project faces.

2

The Challenge of Increasing Complexity

The essence of a program is the combination of possible branches and automated selections based on certain conditions. When we write a program, we define what's going on in a branch, and under what condition this branch will be executed.

The number of branches usually grows quickly during the evolution of a project, as well as the number of conditions that determine whether a branch will be executed or not.

This is dangerous for human beings, who have limited brain capacities.

In this chapter, we are going to implement a data synchronizing service. Starting by implementing some very basic features, we'll keep adding stuff and see how things go.

The following topics will be covered:

- Designing a multi-device synchronizing strategy
- Useful JavaScript and TypeScript techniques and hints that are related, including objects as maps and the string literal type
- How the Strategy Pattern helps in a project

Implementing the basics

Before we start to write actual code, we need to define what this synchronizing strategy will be like. To keep the implementation from unnecessary distractions, the client will communicate with the server directly through function calls instead of using HTTP requests or Sockets. Also, we'll use **in-memory storage**, namely variables, to store data on both client and server sides.

Because we are not separating the client and server into two actual applications, and we are not actually using backend technologies, it does not require much Node.js experience to follow this chapter.

However, please keep in mind that even though we are omitting network and database requests, we hope the core logic of the final implementation could be applied to a real environment without being modified too much. So, when it comes to performance concerns, we still need to assume limited network resources, especially for data passing through the server and client, although the implementation is going to be synchronous instead of asynchronous. This is not supposed to happen in practice, but involving asynchronous operations will introduce much more code, as well as many more situations that need to be taken into consideration. But we will have some useful patterns on asynchronous programming in the coming chapters, and it would definitely help if you try to implement an asynchronous version of the synchronizing logic in this chapter.

A client, if without modifying what's been synchronized, stores a copy of all the data available on the server, and what we need to do is to provide a set of APIs that enable the client to keep its copy of data synchronized.

So, it is really simple at the beginning: comparing the last-modified timestamp. If the timestamp on the client is older than what's on the server, then update the copy of data along with new timestamp.

Creating the code base

Firstly, let's create `server.ts` and `client.ts` files containing the `Server` class and `Client` class respectively:

```
export class Server {  
    // ...  
}  
  
export class Client {  
    // ...
```

```
}
```

I prefer to create an `index.ts` file as the package entry, which handles what to export internally. In this case, let's export everything:

```
export * from './server';
export * from './client';
```

To import the `Server` and `Client` classes from a test file (assuming `src/test/test.ts`), we can use the following code to do so:

```
import { Server, Client } from '../';
```

Defining the initial structure of the data to be synchronized

Since we need to compare the timestamps from the client and server, we need to have a `timestamp` property on the data structure. I would like to have the data to synchronize as a string, so let's add a `DataStore` interface with a `timestamp` property to the `server.ts` file:

```
export interface DataStore {
  timestamp: number;
  data: string;
}
```

Getting data by comparing timestamps

Currently, the synchronizing strategy is one-way, from the server to the client. So what we need to do is simple: we compare the timestamps; if the server has the newer one, it responds with data and the server-side timestamp; otherwise, it responds with `undefined`:

```
class Server {
  store: DataStore = {
    timestamp: 0,
    data: ''
  };

  getData(clientTimestamp: number): DataStore {
    if (clientTimestamp < this.store.timestamp) {
      return this.store;
    } else {
      return undefined;
    }
  }
}
```

```
    }  
}  
}
```

Now we have provided a simple API for the client, and it's time to implement the client:

```
import { Server, DataStore } from './';  
  
export class Client {  
  store: DataStore = {  
    timestamp: 0,  
    data: undefined  
  };  
  constructor(  
    public server: Server  
  ) {}  
}
```



Prefixing a `constructor` parameter with access modifiers (including `public`, `private`, and `protected`) will create a property with the same name and corresponding accessibility. It will also assign the value automatically when the constructor is called.

Now we need to add a `synchronize` method to the `Client` class that does the job:

```
synchronize(): void {  
  let updatedStore = this.server.getData(this.store.timestamp);  
  if (updatedStore) {  
    this.store = updatedStore;  
  }  
}
```

That's easily done. However, are you already feeling somewhat awkward with what we've written?

Two-way synchronizing

Usually, when we talk about synchronization, we get updates from the server and push changes to the server as well. Now we are going to do the second part, pushing the changes if the client has newer data.

But first, we need to give the client the ability to update its data by adding an `update` method to the `Client` class:

```
update(data: string): void {
    this.store.data = data;
    this.store.timestamp = Date.now();
}
```

And we need the server to have the ability to receive data from the client as well. So we rename the `getData` method of the `Server` class as `synchronize` and make it satisfy the new job:

```
synchronize(clientDataStore: DataStore): DataStore {
    if (clientDataStore.timestamp > this.store.timestamp) {
        this.store = clientDataStore;
        return undefined;
    } else if (clientDataStore.timestamp < this.store.timestamp) {
        return this.store;
    } else {
        return undefined;
    }
}
```

Now we have the basic implementation of our synchronizing service. Later, we'll keep adding new things and make it capable of dealing with a variety of scenarios.

Things that went wrong while implementing the basics

Currently, what we've written is just too simple to be wrong. But there are still some semantic issues.

Passing a data store from the server to the client does not make sense

We used `DataStore` as the return type of the `synchronize` method on `Server`. But what we were actually passing through is not a data store, but information that involves data and its timestamp. The information object just *happened to* have the same properties as a data store *at this point in time*.

Also, it will be misleading to people who will later read your code (including yourself in the future). Most of the time, we are trying to eliminate redundancies. But that does not have to mean everything that looks the same. So let's make it two interfaces:

```
interface DataStore {  
    timestamp: number;  
    data: string;  
}  
  
interface DataSyncingInfo {  
    timestamp: number;  
    data: string;  
}
```

I would even prefer to create another instance, instead of directly returning `this.store`:

```
return {  
    timestamp: this.store.timestamp,  
    data: this.store.data  
};
```

However, if two pieces of code with different semantic meanings are doing the same thing from the perspective of code itself, you may consider extracting that part as a utility.

Making the relationships clear

Now we have two separated interfaces, `DataStore` and `DataSyncingInfo`, in `server.ts`. Obviously, `DataSyncingInfo` should be a shared interface between the server and the client, while `DataStore` happens to be the same on both sides, but it's not actually shared.

So what we are going to do is to create a separate `shared.d.ts` (it could also be `shared.ts` if it contains more than typings) that exports `DataSyncingInfo` and add another `DataStore` to `client.ts`.



Do not follow this blindly. Sometimes it is designed for the server and the client to have exactly the same stores. If that's the situation, the interface should be shared.

Growing features

What we've done so far is basically useless. But, from now on, we will start to add features and make it capable of fitting in practical needs, including the capability of synchronizing multiple data items with multiple clients, and merging conflicts.

Synchronizing multiple items

Ideally, the data we need to synchronize will have a lot of items contained. Directly changing the type of `data` to an array would work if there were only very limited number of these items.

Simply replacing data type with an array

Now let's change the type of the `data` property of `DataStore` and `DataSyncingInfo` interfaces to `string[]`. With the help of TypeScript, you will get errors for unmatched types this change would cause. Fix them by annotating the correct types.

But obviously, this is far from an efficient solution.

Server-centered synchronization

If the data store contains a lot of data, the ideal approach would be only updating items that are not up-to-date.

For example, we can create a timestamp for every single item and send these timestamps to the server, then let the server decide whether a specific data item is up-to-date. This is a viable approach for certain scenarios, such as checking updates for software extensions. It is okay to occasionally send even hundreds of timestamps with item IDs on a fast network, but we are going to use another approach for different scenarios, or I won't have much to write.

User data synchronization of offline apps on a mobile phone is what we are going to deal with, which means we need to try our best to avoid wasting network resources.



Here is an interesting question. What are the differences between user data synchronization and checking extension updates? Think about the size of data, issues with multiple devices, and more.

The reason why we thought about sending timestamps of all items is for the server to determine whether certain items need to be updated. However, is it necessary to have the timestamps of all data items stored on the client side?

What if we choose not to store the timestamp of data changing, but of data being synchronized with the server? Then we can get everything up-to-date by only sending the timestamp of the last successful synchronization. The server will then compare this timestamp with the last modified timestamps of all data items and decide how to respond.

As the title of this part suggests, the process is server-centered and relies on the server to generate the timestamps (though it does not have to, and practically should not, be the stamp of the actual time).

If you are getting confused about how these timestamps work, let's try again. The server will store the timestamps of the last time items were synchronized, and the client will store the timestamp of the last successful synchronization with the server. Thus, if no item on the server has a later timestamp than the client, then there's no change to the server data store after that timestamp. But if there are some changes, by comparing the timestamp of the client with the timestamps of server items, we'll know which items are newer.

Synchronizing from the server to the client

Now there seems to be quite a lot to change. Firstly, let's handle synchronizing data from server to client.

This is what's expected to happen on the server side:

- Add a timestamp and identity to every data item on the server
- Compare the client timestamp with every data item on the server

We don't need to actually compare the client timestamp with every item on server if those items have a sorted index. The performance would be acceptable using a database with a sorted index.

- Respond with items newer than what the client has as well as a new timestamp.

And here's what's expected to happen on the client side:

- Synchronize with the last timestamp sent to the server
- Update the local store with new data responded by the server
- Update the local timestamp of the last synchronization if it completes without error

Updating interfaces

First of all, we have now an updated data store on both sides. Starting with the server, the data store now contains an array of data items. So let's define the `ServerDataItem` interface and update `ServerDataStore` as well:

```
export interface ServerDataItem {  
    id: string;  
    timestamp: number;  
    value: string;  
}  
  
export interface ServerDataStore {  
    items: {  
        [id: string]: ServerDataItem;  
    };  
}
```

The `{ [id: string]: ServerDataItem }` type describes an object with `id` of type `string` as a key and has the value of type `ServerDataItem`. Thus, an item of type `ServerDataItem` can be accessed by `items['the-id']`.

And for the client, we now have different data items and a different store. The response contains only a subset of all data items, so we need IDs and a map with ID as the index to store the data:

```
export interface ClientDataItem {  
    id: string;  
    value: string;  
}  
  
export interface ClientDataStore {  
    timestamp: number;  
    items: {  
        [id: string]: ClientDataItem;  
    };  
}
```

Previously, the client and server were sharing the same `DataSyncingInfo`, but that's going to change. As we'll deal with server-to-client synchronizing first, we care only about the timestamp in a synchronizing request for now:

```
export interface SyncingRequest {  
    timestamp: number;  
}
```

As for the response from the server, it is expected to have an updated timestamp with data items that have changed compared to the request timestamp:

```
export interface SyncingResponse {  
    timestamp: number;  
    changes: {  
        [id: string]: string;  
    };  
}
```

I prefixed those interfaces with `Server` and `Client` for better differentiation. But it's not necessary if you are not exporting everything from `server.ts` and `client.ts` (in `index.ts`).

Updating the server side

With well-defined data structures, it should be pretty easy to achieve what we expected. To begin with, we have the `synchronize` method, which accepts a `SyncingRequest` and returns a `SyncingResponse`; and we need to have the updated timestamp as well:

```
synchronize(request: SyncingRequest): SyncingResponse {  
    let lastTimestamp = request.timestamp;  
    let now = Date.now();  
    let serverChanges: ServerChangeMap = Object.create(null);  
    return {  
        timestamp: now,  
        changes: serverChanges  
    };  
}
```

 For the `serverChanges` object, `{ }` (an object literal) might be the first thing (if not an ES6 Map) that comes to mind. But it's not absolutely safe to do so, because it would refuse `__proto__` as a key. The better choice would be `Object.create(null)`, which accepts all strings as its key.

Now we are going to add items that are newer than the client to `serverChanges`:

```
let items = this.store.items;

for (let id of Object.keys(items)) {
  let item = items[id];
  if (item.timestamp > lastTimestamp) {
    serverChanges[id] = item.value;
  }
}
```

Updating the client side

As we've changed the type of `items` under `ClientDataStore` to a map, we need to fix the initial value:

```
store: ClientDataStore = {
  timestamp: 0,
  items: Object.create(null)
};
```

Now let's update the `synchronize` method. Firstly, the client is going to send a request with a timestamp and get a response from the server:

```
synchronize(): void {
  let store = this.store;
  let response = this.server.synchronize({
    timestamp: store.timestamp
  });
}
```

Then we'll save the newer data items to the store:

```
let clientItems = store.items;
let serverChanges = response.changes;

for (let id of Object.keys(serverChanges)) {
  clientItems[id] = {
    id,
    value: serverChanges[id]
  };
}
```

Finally, update the timestamp of the last successful synchronization:

```
clientStore.timestamp = response.timestamp;
```



Updating the synchronization timestamp should be the last thing to do during a complete synchronization process. Make sure it's not stored earlier than data items, or you might have a broken offline copy if there's any errors or interruptions during synchronizing in the future.



To ensure that this works as expected, an operation with the same change information should give the same results even if it's applied multiple times.

Synchronizing from client to server

For a server-centered synchronizing process, most of the changes are made through clients. Consequently, we need to figure out how to organize these changes before sending them to the server.

One single client only cares about its own copy of data. What difference would this make when comparing to the process of synchronizing data from the server to clients? Well, think about why we need the timestamp of every data item on the server in the first place. We need them because we want to know which items are new compared to a specific client.

Now, for changes on a client: if they ever happen, they need to be synchronized to the server without requiring specific timestamps for comparison.

However, we might have more than one client with changes that need to be synchronized, which means that changes made later in time might actually get synchronized earlier, and thus we'll have to resolve conflicts. To achieve that, we need to add the last modified time back to every data item on the server and the changed items on the client.

I've mentioned that the timestamps stored on the server for finding out what needs to be synchronized to a client do not need to be (and better not be) an actual stamp of a physical time point. For example, it could be the count of synchronizations that happened between all clients and the server.

Updating the client side

To handle this efficiently, we may create a separated map with the IDs of the data items that have changed as keys and the last modified time as the value in `ClientDataStore`:

```
export interface ClientDataStore {
  timestamp: number;
  items: {
    [id: string]: ClientDataItem;
  };
}
```

```
changed: {
  [id: string]: number;
};

}
```

You may also want to initialize its value as `Object.create(null)`.

Now when we update an item in the client store, we add the last modified time to the changed map as well:

```
update(id: string, value: string): void {
  let store = this.store;
  store.items[id] = {
    id,
    value
  };
  store.changed[id] = Date.now();
}
```

A single timestamp in `SyncingRequest` certainly won't do the job any more; we need to add a place for the changed data, a map with data item ID as the index, and the changed information as the value:

```
export interface ClientChange {
  lastModifiedTime: number;
  value: string;
}

export interface SyncingRequest {
  timestamp: number;
  changes: {
    [id: string]: ClientChange;
  };
}
```

Here comes another problem. What if a change made to a client data item is done offline, with the system clock being at the wrong time? Obviously, we need some time calibration mechanisms. However, there's no way to make perfect calibration. We'll make some assumptions so we don't need to start another chapter for time calibration:

- The system clock of a client may be late or early compared to the server. But it ticks at a normal speed and won't jump between times.
- The request sent from a client reaches the server in a relatively short time.

With those assumptions, we can add those building blocks to the client-side synchronize method:

1. Add client-side changes to the synchronizing request (of course, before sending it to the server):

```
let clientItems = store.items;
let clientChanges: ClientChangeMap = Object.create(null);

let changedTimes = store.changed;

for (let id of Object.keys(changedTimes)) {
  clientChanges[id] = {
    lastModifiedTime: changedTimes[id],
    value: clientItems[id].value
  };
}
```

2. Synchronize changes to the server with the current time of the client's clock:

```
let response = this.server.synchronize({
  timestamp: store.timestamp,
  clientTime: Date.now(),
  changes: clientChanges
});
```

3. Clean the changes after a successful synchronization:

```
store.changed = Object.create(null);
```

Updating the server side

If the client is working as expected, it should send synchronizing requests with changes. It's time to enable the server to handling those changes from the client.

There are going to be two steps for the server-side synchronization process:

1. Apply the client changes to server data store.
2. Prepare the changes that need to be synchronized to the client.

First, we need to add `lastModifiedTime` to server-side data items, as we mentioned before:

```
export interface ServerDataItem {
  id: string;
  timestamp: number;
  lastModifiedTime: number;
```

```
    value: string;  
}
```

And we need to update the `synchronize` method:

```
let clientChanges = request.changes;  
let now = Date.now();  
  
for (let id of Object.keys(clientChanges)) {  
  let clientChange = clientChanges[id];  
  if (  
    hasOwnProperty.call(items, id) &&  
    items[id].lastModifiedTime > clientChange.lastModifiedTime  
  ) {  
    continue;  
  }  
  items[id] = {  
    id,  
    timestamp: now,  
    lastModifiedTime,  
    value: clientChange.value  
  };  
}  
}
```



We can actually use the `in` operator instead of `hasOwnProperty` here because the `items` object is created with `null` as its prototype. But a reference to `hasOwnProperty` will be your friend if you are using objects created by object literals, or in other ways, such as maps.

We already talked about resolving conflicts by comparing the last modified times. At the same time, we've made assumptions so we can calibrate the last modified times from the client easily by passing the client time to the server while synchronizing.

What we are going to do for calibration is to calculate the offset of the client time compared to the server time. And that's why we made the second assumption: the request needs to easily reach the server in a relatively short time. To calculate the offset, we can simply subtract the client time from the server time:

```
let clientTimeOffset = now - request.clientTime;
```



To make the time calibration more accurate, we would want the earliest timestamp after the request hits the server to be recorded as “now”. So in practice, you might want to record the timestamp of the request hitting the server before start processing everything. For example, for HTTP request, you may record the timestamp once the TCP connection gets established.

And now, the calibrated time of a client change is the sum of the original time and the offset. We can now decide whether to keep or ignore a change from the client by comparing the calibrated last modified time. It is possible for the calibrated time to be greater than the server time; you can choose either to use the server time as the maximum value or accept a small inaccuracy. Here, we will go the simple way:

```
let lastModifiedTime = Math.min(
  clientChange.lastModifiedTime + clientTimeOffset,
  now
);

if (
  hasOwnProperty.call(items, id) &&
  items[id].lastModifiedTime > lastModifiedTime
) {
  continue;
}
```

To make this actually work, we need to also exclude changes from the server that conflict with client changes in `SyncingResponse`. To do so, we need to know what the changes are that survive the conflict resolving process. A simple way is to exclude items with timestamp that equals `now`:

```
for (let id of Object.keys(items)) {
  let item = items[id];
  if (
    item.timestamp > lastTimestamp &&
    item.timestamp !== now
  ) {
    serverChanges[id] = item.value;
  }
}
```

So now we have implemented a complete synchronization logic with the ability to handle simple conflicts in practice.

Synchronizing multiple types of data

At this point, we've hard coded the data with the `string` type. But usually we will need to store varieties of data, such as numbers, booleans, objects, and so on.

If we were writing JavaScript, we would not actually need to change anything, as the implementation does not have anything to do with certain data types. In TypeScript, we don't need to do much either: just change the type of every related `value` to `any`. But that means you are losing type safety, which would definitely be okay if you are happy with that.

But taking my own preferences, I would like every variable, parameter, and property to be typed if it's possible. So we may still have a data item with `value` of type `any`:

```
export interface ClientDataItem {  
    id: string;  
    value: any;  
}
```

We can also have derived interfaces for specific data types:

```
export interface ClientStringDataItem extends ClientDataItem {  
    value: string;  
}  
  
export interface ClientNumberDataItem extends ClientDataItem {  
    value: number;  
}
```

But this does not seem to be good enough. Fortunately, TypeScript provides *generics*, so we can rewrite the preceding code as follows:

```
export interface ClientDataItem<T> {  
    id: string;  
    value: T;  
}
```

Assuming we have a store that accepts multiple types of data items – for example, `number` and `string` – we can declare it as follows with the help of the `union` type:

```
export interface ClientDataStore {  
    items: {  
        [id: string]: ClientDataItem<number | string>;  
    };  
}
```

If you remember that we are doing something for offline mobile apps, you might be questioning the long property names in changes such as `lastModifiedTime`. This is a fair question, and an easy fix is to use tuple types, maybe along with enums:

```
const enum ClientChangeIndex {  
    lastModifiedType,  
    value  
}  
  
type ClientChange<T> = [number, T];  
  
let change: ClientChange<string> = [0, 'foo'];  
let value = change[ClientChangeIndex.value];
```

You can apply less or more of the typing things we are talking about depending on your preferences. If you are not familiar with them yet, you can read more here: <http://www.typescriptlang.org/handbook>.

Supporting multiple clients with incremental data

Making the typing system happy with multiple data types is easy. But in the real world, we don't resolve conflicts of all data types by simply comparing the last modified times. An example is counting the daily active time of a user cross devices.

It's quite clear that we need to have every piece of active time in a day on multiple devices summed up. And this is how we are going to achieve that:

1. Accumulate active durations between synchronizations on the client.
2. Add a UID (unique identifier) to every piece of time before synchronizing with the server.
3. Increase the server-side value if the UID does not exist yet, and then add the UID to that data item.

But before we actually get our hands on those steps, we need a way to distinguish incremental data items from normal ones, for example, by adding a `type` property.

As our synchronizing strategy is server-centered, related information is only required for synchronizing requests and conflict merging. Synchronizing responses does not need to include the details of changes, but just merged values.



I will stop telling how to update every interface step by step as we are approaching the final structure. But if you have any problems with that, you can check out the complete code bundle for inspiration.

Updating the client side

First of all, we need the client to support incremental changes. And if you've thought about this, you might already be confused about where to put the extra information, such as UIDs.

This is because we were mixing up the concept *change* (noun) with *value*. It was not a problem before because, besides the last modified time, the value is what a change is about. We used a simple map to store the last modified times and kept the store clean from redundancy, which balanced well under that scenario.

But now we need to distinguish between these two concepts:

- **Value:** a value describes what a data item is in a static way
- **Change:** a change describes the information that may transform the value of a data item from one to another

We need to have a general type of changes as well as a new data structure for incremental changes with a numeric value:

```
type DataType = 'value' | 'increment';

interface ClientChange {
    type: DataType;
}

interface ClientValueChange<T> extends ClientChange {
    type: 'value';
    lastModifiedTime: number;
    value: T;
}

interface ClientIncrementChange extends ClientChange {
    type: 'increment';
    uid: string;
    increment: number;
}
```



We are using the `string` literal type here, which was introduced in TypeScript 1.8. To learn more, please refer to the TypeScript handbook as we mentioned before.

Similar changes to the data store structure should be made. And when we update an item on the client side, we need to apply the correct operations based on different data types:

```
update(id: string, type: 'increment', increment: number): void;
update<T>(id: string, type: 'value', value: T): void;
update<T>(id: string, type: DataType, value: T): void;
update<T>(id: string, type: DataType, value: T): void {
  let store = this.store;
  let items = store.items;
  let storedChanges = store.changes;
  if (type === 'value') {
    // ...
  } else if (type === 'increment') {
    // ...
  } else {
    throw new TypeError('Invalid data type');
  }
}
```

Use the following code for normal changes (while `type` equals '`'value'`':

```
let change: ClientValueChange<T> = {
  type: 'value',
  lastModifiedTime: Date.now(),
  value
};

storedChanges[id] = change;

if (hasOwnProperty.call(items, id)) {
  items[id].value = value;
} else {
  items[id] = {
    id,
    type,
    value
  };
}
```

For incremental changes, it takes a few more lines:

```
let storedChange = storedChanges[id] as ClientIncrementChange;

if (storedChange) {
    storedChange.increment += <any>value as number;
} else {
    storedChange = {
        type: 'increment',
        uid: Date.now().toString(),
        increment: <any>value as number
    };
    storedChanges[id] = storedChange;
}
```



It's my personal preference to use `<T>` for any casting and `as T` for non-any castings. Though it has been used in languages like C#, the `as` operator in TypeScript was originally introduced for compatibilities with XML tags in JSX. You can also write `<number><any>value` or `value as any as number` here if you like.

Don't forget to update the stored value. Just change `=` to `+=` comparing to updating normal data items:

```
if (hasOwnProperty.call(items, id)) {
    items[id].value += value;
} else {
    items[id] = {
        id,
        type,
        value
    };
}
```

That's not hard at all. But hey, we see branches.

We are writing branches all the time, but what are the differences between branches such as `if (type === 'foo') { ... }` and branches such as `if (item.timestamp > lastTimestamp) { ... }`? Let's keep this question in mind and move on.

With necessary information added by the `update` method, we can now update the `synchronize` method of the client. But there is a flaw in practical scenarios: a synchronizing request is sent to the server successfully, but the client failed to receive the response from the server. In this situation, when `update` is called after a failed synchronization, the increment is added to the might-be-synchronized change (identified by its UID), which will be ignored by the server in future synchronizations. To fix this, we'll

need to add a mark to all incremental changes that have started a synchronizing process, and avoid accumulating these changes. Thus, we need to create another change for the same data item.

This is actually a nice hint: as a change is about information that transforms a value from one to another, several changes pending synchronization might eventually be applied to one single data item:

```
interface ClientChangeList<T extends ClientChange> {
    type: DataType;
    changes: T[];
}

interface SyncingRequest {
    timestamp: number;
    changeLists: {
        [id: string]: ClientChangeList<ClientChange>;
    };
}

interface ClientIncrementChange extends ClientChange {
    type: 'increment';
    synced: boolean;
    uid: string;
    increment: number;
}
```

Now when we are trying to update an incremental data item, we need to get its last change from the change list (if any) and see whether it has ever been synchronized. If it has ever been involved in a synchronization, we create a new change instance. Otherwise, we'll just accumulate the `increment` property value of the last change on the client side:

```
let changeList = storedChangeLists[id];
let changes = changeList.changes;
let lastChange =
    changes[changes.length - 1] as ClientIncrementChange;

if (lastChange.synced) {
    changes.push({
        synced: false,
        uid: Date.now().toString(),
        increment: <any>value as number
    } as ClientIncrementChange);
} else {
    lastChange.increment += <any>value as number;
}
```

Or, if the change list does not exist yet, we'll need to set it up:

```
let changeList = {
  type: 'increment',
  changes: [
    {
      synced: false,
      uid: Date.now().toString(),
      increment: <any>value as number
    } as ClientIncrementChange
  ]
};

store.changeLists[id] = changeList;
```

We also need to update synchronize method to mark an incremental change as synced before starting the synchronization with the server. But the implementation is for you to do on your own.

Updating server side

Before we add the logic for handling incremental changes, we need to make server-side code adapt to the new data structure:

```
for (let id of Object.keys(clientChangeLists)) {
  let clientChangeList = clientChangeLists[id];
  let type = clientChangeList.type;
  let clientChanges = clientChangeList.changes;
  if (type === 'value') {
    // ...
  } else if (type === 'increment') {
    // ...
  } else {
    throw new TypeError('Invalid data type');
  }
}
```

The change list of a normal data item will always contain one and only one change. Thus we can easily migrate what we've written:

```
let clientChange = changes[0] as ClientValueChange<any>;
```

Now for incremental changes, we need to cumulatively apply possibly multiple changes in a single change list to a data item:

```
let item = items[id];
```

```

for (
  let clientChange
  of clientChanges as ClientIncrementChange[]
) {
  let {
    uid,
    increment
  } = clientChange;
  if (item.uids.indexOf(uid) < 0) {
    item.value += increment;
    item.uids.push(uid);
  }
}

```

But remember to take care of the timestamp or cases in which no item with a specified ID exists:

```

let item: ServerDataItem<any>;

if (hasOwnProperty.call(items, id)) {
  item = items[id];
  item.timestamp = now;
} else {
  item = items[id] = {
    id,
    type,
    timestamp: now,
    uids: [],
    value: 0
  };
}

```

Without knowing the current value of an incremental data item on the client, we cannot assure that the value is up to date. Previously, we decided whether to respond with a new value by comparing the timestamp with the timestamp of the current synchronization, but that does not work anymore for incremental changes.

A simple way to make this work is by deleting keys from `clientChangeLists` that still need to be synchronized to the client. And when preparing responses, it can skip IDs that are still in `clientChangeLists`:

```

if (
  item.timestamp > lastTimestamp &&
  !hasOwnProperty.call(clientChangeLists, id)
) {
  serverChanges[id] = item.value;
}

```

Remember to add `delete clientChangeLists[id]`; for normal data items that did not survive conflicts resolving as well.

Now we have implemented a synchronizing logic that can do quite a lot jobs for offline applications. Earlier, I raised a question about increasing branches that do not look good. But if you know your features are going to end there, or at least with limited changes, it's not a bad implementation, although we'll soon cross the balance point, as meeting 80% of the needs won't make us happy enough.

Supporting more conflict merging

Though we have met the needs of 80%, there is still a big chance that we might want some extra features. For example, we want the ratio of the days marked as available by the user in the current month, and the user should be able to add or remove days from the list. We can achieve that in different ways, and we'll choose a simple way, as usual.

We are going to support synchronizing a set with operations such as add and remove, and calculate the ratio on the client.

New data structures

To describe set changes, we need a new `ClientChange` type. When we are adding or removing an element from a set, we only care about the last operation to the same element. This means that the following:

1. If multiple operations are made to the same element, we only need to keep the last one.
2. A `time` property is required for resolving conflicts.

So here are the new types:

```
enum SetOperation {  
    add,  
    remove  
}  
  
interface ClientSetChange extends ClientChange {  
    element: number;  
    time: number;  
    operation: SetOperation;  
}
```

The set data stored on the server side is going to be a little different. We'll have a map with the element (in the form of a `string`) as key, and a structure with `operation` and `time` properties as the values:

```
interface ServerSetElementOperationInfo {  
    operation: SetOperation;  
    time: number;  
}
```

Now we have enough information to resolve conflicts from multiple clients. And we can generate the set by keys with a little help from the last operations done to the elements.

Updating client side

And now, the client-side `update` method gets a new part-time job: saving set changes just like value and incremental changes. We need to update the method signature for this new job (do not forget to add '`set`' to `DataType`):

```
update(  
    id: string,  
    type: 'set',  
    element: number,  
    operation: SetOperation  
) : void;  
update<T>(  
    id: string,  
    type: DataType,  
    value: T,  
    operation?: SetOperation  
) : void;
```

We also need to add another `else if`:

```
else if (type === 'set') {  
    let element = <any>value as number;  
    if (hasOwnProperty.call(storedChangeLists, id)) {  
        // ...  
    } else {  
        // ...  
    }  
}
```

If there are already operations made to this set, we need to find and remove that last operation to the target element (if any). Then append a new change with the latest operation:

```
let changeList = storedChangeLists[id];
let changes = changeList.changes as ClientSetChange[];

for (let i = 0; i < changes.length; i++) {
  let change = changes[i];
  if (change.element === element) {
    changes.splice(i, 1);
    break;
  }
}

changes.push({
  element,
  time: Date.now(),
  operation
});
```

If no change has been made since last successful synchronization, we'll need to create a new change list for the latest operation:

```
let changeList: ClientChangeList<ClientSetChange> = {
  type: 'set',
  changes: [
    {
      element,
      time: Date.now(),
      operation
    }
  ]
};

storedChangeLists[id] = changeList;
```

And again, do not forget to update the stored value. This is a little bit more than just assigning or accumulating the value, but it should still be quite easy to implement.

Updating the server side

Just like we've done with the client, we need to add a corresponding `else if` branch to merge changes of type 'set'. We are also deleting the ID from `clientChangeLists` regardless of whether there are newer changes for a simpler implementation:

```
else if (type === 'set') {
  let item: ServerDataItem<{
    [element: string]: ServerSetElementOperationInfo;
  }>;
  delete clientChangeLists[id];
}
```

The conflict resolving logic is quite similar to what we do to the conflicts of normal values. We just need to make comparisons to each element, and only keep the last operation.

And when preparing the response that will be synchronized to the client, we can generate the set by putting together elements with `add` as their last operations:

```
if (item.type === 'set') {
  let operationInfos: {
    [element: string]: ServerSetElementOperationInfo;
  } = item.value;
  serverChanges[id] = Object
    .keys(operationInfos)
    .filter(element =>
      operationInfos[element].operation ===
        SetOperation.add
    )
    .map(element => Number(element));
} else {
  serverChanges[id] = item.value;
}
```

Finally, we have a working mess (if it actually works). Cheers!

Things that go wrong while implementing everything

When we started to add features, things were actually fine, if you are not obsessive about pursuing the feeling of design. Then we sensed the code being a little awkward as we saw more and more nested branches.

So now it's time to answer the question, what are the differences between the two kinds of branch we wrote? My understanding of why I am feeling awkward about the `if (type === 'foo') { ... }` branch is that it's not strongly related to the context. Comparing timestamps, on the other hand, is a more natural part of a certain synchronizing process.

Again, I am not saying this is bad. But this gives us a hint about where we might start our surgery from when we start to lose control (due to our limited brain capacity, it's just a matter of complexity).

Piling up similar yet parallel processes

Most of the code in this chapter is to handle the process of synchronizing data between a client and a server. To get adapted to new features, we just kept adding new things into methods, such as `update` and `synchronize`.

You might have already found that most outlines of the logic can be, and should be, shared across multiple data types. But we didn't do that.

If we look into what's written, the duplication is actually minor judging from the aspect of code texts. Taking the `update` method of the client, for example, the logic of every branch seems to differ. If finding abstractions has not become your built-in reaction, you might just stop there. Or if you are not a fan of long functions, you might refactor the code by splitting it into small ones of the same class. That could make things a little better, but far from enough.

Data stores that are tremendously simplified

In the implementation, we were playing heavily and directly with ideal *in-memory* stores. It would be nice if we could have a wrapper for it, and make the real store interchangeable.

This might not be the case for this implementation as it is based on extremely ideal and simplified assumptions and requirements. But adding a wrapper could be a way to provide useful helpers.

Getting things right

So let's get out of the illusion of comparing code one character at a time and try to find an abstraction that can be applied to updating all of these data types. There are two key points of this abstraction that have already been mentioned in the previous section:

- A change contains the information that can transform the value of an item from one to another
- Multiple changes could be generated or applied to one data item during a single synchronization

Now, starting from changes, let's think about what happens when an `update` method of a client is called.

Finding abstraction

Take a closer look to the method `update` of client:

- For data of the '`value`' type, first we create the change, including a new value, and then update the change list to make the newly created change the only one. After that, we update the value of data item.
- For data of the '`increment`' type, we add a change including the increment in the change list; or if a change that has not be synchronized already exists, update the increment of the existing change. And then, we update the value of the data item.
- Finally, for data of the '`set`' type, we create a change reflecting the latest operation. After adding the new change to the change list, we also remove changes that are no longer necessary. Then we update the value of the data item.

Things are getting clear. Here is what's happening to these data types when `update` is called:

1. Create new change.
2. Merge the new change to the change list.
3. Apply the new change to the data item.

Now it's even better. Every step is different for different data types, but different steps share the same outline; what we need to do is to implement different strategies for different data types.

Implementing strategies

Doing all kind of changes with a single `update` function could be confusing. And before we move on, let's split it into three different methods: `update` for normal values, `increase` for incremental values, and `addTo/removeFrom` for sets.

Then we are going to create a new private method called `applyChange`, which will take the change created by other methods and continue with step 2 and step 3. It accepts a strategy object with two methods: `append` and `apply`:

```
interface ClientChangeStrategy<T extends ClientChange> {  
    append(list: ClientChangeList<T>, change: T): void;  
    apply(item: ClientDataItem<any>, change: T): void;  
}
```

For a normal data item, the strategy object could be as follows:

```
let strategy: ClientChangeStrategy<ClientValueChange<any>> = {  
    append(list, change) {  
        list.changes = [change];  
    },  
    apply(item, change) {  
        item.value = change.value;  
    }  
};
```

And for incremental data item, it takes a few more lines. First, the `append` method:

```
let changes = list.changes;  
let lastChange = changes[changes.length];  
  
if (!lastChange || lastChange.synced) {  
    changes.push(change);  
} else {  
    lastChange.increment += change.increment;  
}
```

The `append` method is followed by the `apply` method:

```
if (item.value === undefined) {  
    item.value = change.increment;  
} else {  
    item.value += change.increment;  
}
```

Now in the `applyChange` method, we need to take care of the creation of non-existing items and change lists, and invoke different `append` and `apply` methods based on different data types.

The same technique can be applied to other processes. Though detailed processes that apply to the client and the server differ, we can still write them together as modules.

Wrapping stores

We are going to make a lightweight wrapper around plain in-memory store objects with the ability to read and write, taking the server-side store as an example:

```
export class ServerStore {
  private items: {
    [id: string]: ServerDataItem<any>;
  } = Object.create(null);
}

export class Server {
  constructor(
    public store: ServerStore
  ) { }
}
```

To fit our requirements, we need to implement `get`, `set`, and `getAll` methods (or even better, a `find` method with conditions) for `ServerStore`:

```
get<T, TExtra extends ServerDataItemExtra>(id: string):
  ServerDataItem<T> & TExtra {
  return hasOwnProperty.call(this.items, id) ?
    this.items[id] as ServerDataItem<T> & TExtra : undefined;
}

set<T, TExtra extends ServerDataItemExtra>(
  id: string,
  item: ServerDataItem<T> & Textra
): void {
  this.items[id] = item;
}

getAll<T, TExtra extends ServerDataItemExtra>():
  (ServerDataItem<T> & TExtra)[] {
  let items = this.items;
  return Object
    .keys(items)
```

```
.map(id => items[id] as ServerDataItem<T> & TExtra);  
}
```

You may have noticed from the interfaces and generics that I've also torn down `ServerDataItem` into intersection types of the common part and extras.

Summary

In this chapter, we've been part of the evolution of a simplified yet reality-related project. Starting with a simple code base that couldn't be wrong, we added a lot of features and experienced the process of putting acceptable changes together and making the whole thing a mess.

We were always trying to write readable code by either naming things nicely or adding semantically necessary redundancies, but that won't help much as the complexity grows.

During the process, we've learned how offline synchronizing works. And with the help of the most common design patterns, such as the Strategy Pattern, we managed to split the project into small and controllable parts.

In the upcoming chapters, we'll catalog more useful design patterns with code examples in TypeScript, and try to apply those design patterns to specific issues.

3

Creational Design Patterns

Creational design patterns in object-oriented programming are design patterns that are to be applied during the instantiation of objects. In this chapter, we'll be talking about patterns in this category.

Consider we are building a rocket, which has payload and one or more stages:

```
class Payload {  
    weight: number;  
}  
  
class Engine {  
    thrust: number;  
}  
  
class Stage {  
    engines: Engine[];  
}
```

In old-fashioned JavaScript, there are two major approaches to building such a rocket:

- Constructor with `new` operator
- Factory function

For the first approach, things could be like this:

```
function Rocket() {  
    this.payload = {  
        name: 'cargo ship'  
    };  
    this.stages = [  
        {  
            engines: [  
                // ...  
            ]  
        }  
    ];  
}
```

```
        ]
    }
};

var rocket = new Rocket();
```

And for the second approach, it could be like this:

```
function buildRocket() {
  var rocket = {};
  rocket.payload = {
    name: 'cargo ship'
};
  rocket.stages = [
  {
    thrusters: [
      // ...
    ]
  }
];
  return rocket;
}

var rocket = buildRocket();
```

From a certain angle, they are doing pretty much the same thing, but semantically they differ a lot. The constructor approach suggests a strong association between the building process and the final product. The factory function, on the other hand, implies an interface of its product and claims the ability to build such a product.

However, neither of the preceding implementations provides the flexibility to modularly assemble rockets based on specific needs; this is what creational design patterns are about.

In this chapter, we'll cover the following creational patterns:

- **Factory method:** By using abstract methods of a factory instead of the constructor to build instances, this allows subclasses to change what's built by implementing or overriding these methods.
- **Abstract factory:** Defining the interface of compatible factories and their *products*. Thus by changing the factory passed, we can change the family of built products.
- **Builder:** Defining the *steps* of building complex objects, and changing what's built either by changing the sequence of steps, or using a different builder implementation.

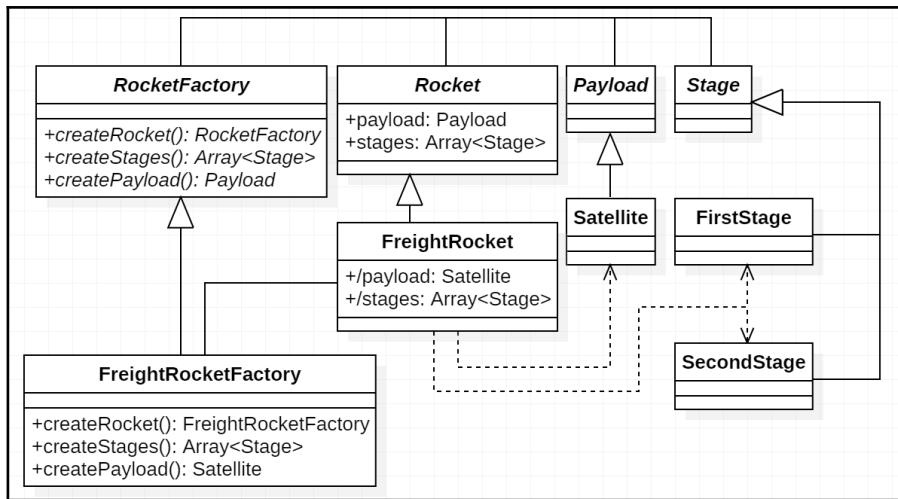
- **Prototype:** Creating objects by cloning parameterized prototypes. Thus by replacing these prototypes, we may build different products.
 - **Singleton:** Ensuring only one instance (under a certain scope) will be created.

It is interesting to see that even though the factory function approach to creating objects in JavaScript looks primitive, it does have parts in common with some patterns we are going to talk about (although applied to different scopes).

Factory method

Under some scenarios, a class cannot predict exactly what objects it will create, or its subclasses may want to create more specified versions of these objects. Then, the Factory Method Pattern can be applied.

The following picture shows the possible structure of the Factory Method Pattern applied to creating rockets:



A factory method is a method of a factory that builds objects. Take building rockets as an example; a factory method could be a method that builds either the entire rocket or a single component. One factory method might rely on other factory methods to build its target object. For example, if we have a `createRocket` method under the `Rocket` class, it would probably call factory methods like `createStages` and `createPayload` to get the necessary components.

The Factory Method Pattern provides some flexibility upon reasonable complexity. It allows extendable usage by implementing (or overriding) specific factory methods. Taking `createStages` method, for example, we can create a one-stage rocket or a two-stage rocket by providing different `createStages` method that return one or two stages respectively.

Participants

The participants of a typical Factory Method Pattern implementation include the following:

- Product: `Rocket`

Define an abstract class or an interface of a rocket that will be created as the product.

- Concrete product: `FreightRocket`

Implement a specific rocket product.

- Creator: `RocketFactory`

Define the optionally abstract factory class that creates products.

- Concrete creator: `FreightRocketFactory`

Implement or overrides specific factory methods to build products on demand.

Pattern scope

The Factory Method Pattern decouples `Rocket` from the constructor implementation and makes it possible for subclasses of a factory to change what's built accordingly. A concrete creator still cares about what exactly its components are and how they are built. But the implementation or overriding usually focuses more on each component, rather than the entire product.

Implementation

Let's begin with building a simple one-stage rocket that carries a 0-weight payload as the default implementation:

```
class RocketFactory {
```

```
buildRocket(): Rocket { }
createPayload(): Payload { }
createStages(): Stage[] { }
}
```

We start with creating components. We will simply return a payload with 0 weight for the factory method `createPayload` and one single stage with one single engine for the factory method `createStages`:

```
createPayload(): Payload {
    return new Payload(0);
}

createStages(): Stage[] {
    let engine = new Engine(1000);
    let stage = new Stage([engine]);
    return [stage];
}
```

After implementing methods to create the components of a rocket, we are going to put them together with the factory method `buildRocket`:

```
buildRocket(): Rocket {
    let rocket = new Rocket();
    let payload = this.createPayload();
    let stages = this.createStages();
    rocket.payload = payload;
    rocket.stages = stages;
    return rocket;
}
```

Now we have the blueprint of a simple rocket factory, yet with certain extensibilities. To build a rocket (that does nothing so far), we just need to instantiate this very factory and call its `buildRocket` method:

```
let rocketFactory = new RocketFactory();
let rocket = rocketFactory.buildRocket();
```

Next, we are going to build two-stage freight rockets that send satellites into orbit. Thus, there are some differences compared to the basic factory implementation.

First, we have a different payload, satellites, instead of a 0-weight placeholder:

```
class Satellite extends Payload {
    constructor(
        public id: number
    ) {
```

```
        super(200);  
    }  
}
```

Second, we now have two stages, probably with different specifications. The first stage is going to have four engines:

```
class FirstStage extends Stage {  
    constructor() {  
        super([  
            new Engine(1000),  
            new Engine(1000),  
            new Engine(1000),  
            new Engine(1000)  
        ]);  
    }  
}
```

While the second stage has only one:

```
class SecondStage extends Stage {  
    constructor() {  
        super([  
            new Engine(1000)  
        ]);  
    }  
}
```

Now we have what this new freight rocket would look like in mind, let's extend the factory:

```
type FreightRocketStages = [FirstStage, SecondStage];  
  
class FreightRocketFactory extends RocketFactory {  
    createPayload(): Satellite { }  
    createStages(): FreightRocketStages { }  
}
```



Here we are using the *type alias* of a *tuple* to represent the stages sequence of a freight rocket, namely the first and second stages. To find out more about type aliases, please refer to <https://www.typescriptlang.org/docs/handbook/advanced-types.html>.

As we added the `id` property to `Satellite`, we might need a counter for each instance of the factory, and then create every satellite with a unique ID:

```
nextSatelliteId = 0;  
  
createPayload(): Satellite {
```

```
    return new Satellite(this.nextSatelliteId++);
}
```

Let's move on and implement the `createStages` method that builds first and second stage of the rocket:

```
createStages(): FreightRocketStages {
    return [
        new FirstStage(),
        new SecondStage()
    ];
}
```

Comparing to the original implementation, you may have noticed that we've automatically decoupled specific stage building processes from assembling them into constructors of different stages. It is also possible to apply another creational pattern for the initiation of every stage if it helps.

Consequences

In the preceding implementation, the factory method `buildRocket` handles the outline of the building steps. We were lucky to have the freight rocket in the same structure as the very first rocket we had defined.

But that won't always happen. If we want to change the class of products (`Rocket`), we'll have to override the entire `buildRocket` with everything else but the class name. This looks frustrating but it can be solved, again, by decoupling the creation of a rocket instance from the building process:

```
buildRocket(): Rocket {
    let rocket = this.createRocket();
    let payload = this.createPayload();
    let stages = this.createStages();
    rocket.payload = payload;
    rocket.stages = stages;
    return rocket;
}

createRocket(): Rocket {
    return new Rocket();
}
```

Thus we can change the rocket class by overriding the `createRocket` method. However, the return type of the `buildRocket` of a subclass (for example, `FreightRocketFactory`)

is still Rocket instead of something like FreightRocket. But as the object created is actually an instance of FreightRocket, it is valid to cast the type by type assertion:

```
let rocket = FreightRocketFactory.buildRocket() as FreightRocket;
```

The trade-off is a little type safety, but that can be eliminated using generics. Unfortunately, in TypeScript what you get from a generic type argument is just a type without an actual value. This means that we may need another level of abstraction or other patterns that can use the help of type inference to make sure of everything.

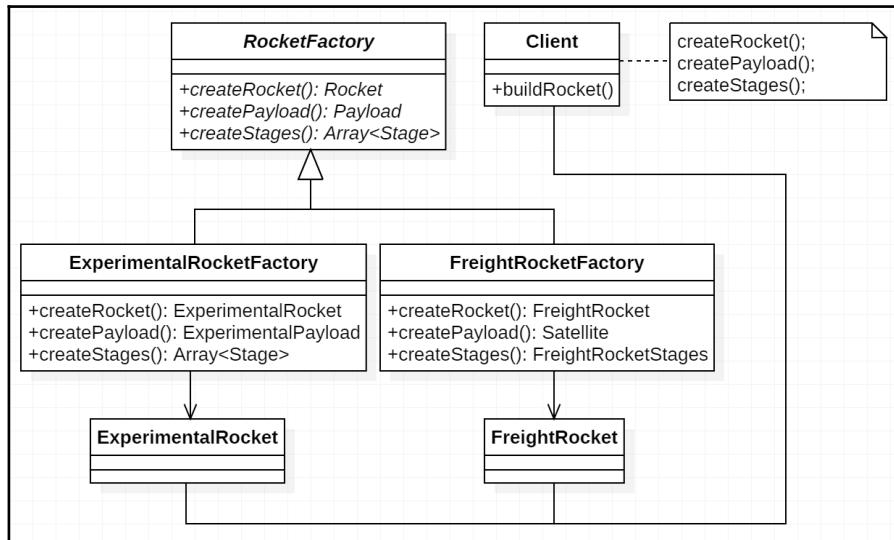
The former option would lead us to the Abstract Factory Pattern.



Type safety could be one reason to consider when choosing a pattern but usually, it will not be decisive. Please note we are not trying to switch a pattern for this single reason, but just exploring.

Abstract Factory

The Abstract Factory Pattern usually defines the interfaces of a collection of factory methods, without specifying concrete products. This allows an entire factory to be replaceable, in order to produce different products following the same production outline:



The details of the products (components) are omitted from the diagram, but do notice that these products belong to two parallel families: ExperimentalRocket and FreightRocket.

Different from the Factory Method Pattern, the Abstract Factory Pattern extracts another part called **client** that takes care of shaping the outline of the building process. This makes the factory part focused more on producing each component.

Participants

The participants of a typical Abstract Factory Pattern implementation include the following:

- **Abstract factory:** RocketFactory
 - Defines the *industrial standards* of a factory which provide interfaces for manufacturing components or complex products.
- **Concrete factory:** ExperimentalRocketFactory, FreightRocketFactory
 - Implements the interfaces defined by the abstract factory and builds concrete products.
- **Abstract products:** Rocket, Payload, Stage []
 - Define the interfaces of the products the factories are going to build.
- **Concrete products:** ExperimentalRocket/FreightRocket, ExperimentalPayload/Satellite, and so on.
 - Presents actual products that are manufactured by a concrete factory.
- **Client:**
 - Arranges the production process across factories (only if these factories conform to *industrial standards*).

Pattern scope

Abstract Factory Pattern makes the abstraction on top of different concrete factories. At the scope of a single factory or a single branch of factories, it just works like the Factory Method Pattern. However, the highlight of this pattern is to make a whole family of products interchangeable. A good example could be components of themes for a UI implementation.

Implementation

In the Abstract Factory Pattern, it is the client interacting with a concrete factory for building integral products. However, the concrete class of products is decoupled from the client during design time, while the client cares only about what a factory and its products look like instead of what exactly they are.

Let's start by simplifying related classes to interfaces:

```
interface Payload {  
    weight: number;  
}  
  
interface Stage {  
    engines: Engine[];  
}  
  
interface Rocket {  
    payload: Payload;  
    stages: Stage[];  
}
```

And of course the abstract factory itself is:

```
interface RocketFactory {  
    createRocket(): Rocket;  
    createPayload(): Payload;  
    createStages(): Stage[];  
}
```

The building steps are abstracted from the factory and put into the client, but we still need to implement it anyway:

```
class Client {  
    buildRocket(factory: RocketFactory): Rocket {  
        let rocket = factory.createRocket();  
        rocket.payload = factory.createPayload();  
        rocket.stages = factory.createStages();  
    }  
}
```

```
        return rocket;
    }
}
```

Now we have the same issue we previously had when we implemented the Factory Method Pattern. As different concrete factories build different rockets, the class of the product changes. However, now we have generics to the rescue.

First, we need a `RocketFactory` interface with a generic type parameter that describes a concrete rocket class:

```
interface RocketFactory<T extends Rocket> {
    createRocket(): T;
    createPayload(): Payload;
    createStages(): Stage[];
}
```

And second, update the `buildRocket` method of the client to support generic factories:

```
buildRocket<T extends Rocket>(
    factory: RocketFactory<T>
): T { }
```

Thus, with the help of the type system, we will have rocket type inferred based on the type of a concrete factory, starting with `ExperimentalRocket` and `ExperimentalRocketFactory`:

```
class ExperimentalRocket implements Rocket { }

class ExperimentalRocketFactory
implements RocketFactory<ExperimentalRocket> { }
```

If we call the `buildRocket` method of a client with an instance of `ExperimentalRocketFactory`, the return type will automatically be `ExperimentalRocket`:

```
let client = new Client();
let factory = new ExperimentalRocketFactory();
let rocket = client.buildRocket(factory);
```

Before we can complete the implementation of the `ExperimentalRocketFactory` object, we need to define concrete classes for the products of the family:

```
class ExperimentalPayload implements Payload {
    weight: number;
}
```

```
class ExperimentalRocketStage implements Stage {  
    engines: Engine[];  
}  
  
class ExperimentalRocket implements Rocket {  
    payload: ExperimentalPayload;  
    stages: [ExperimentalRocketStage];  
}
```



Trivial initializations of payload and stage are omitted for more compact content. The same kinds of omission may be applied if they are not necessary for this book.

And now we may define the factory methods of this concrete factory class:

```
class ExperimentalRocketFactory  
implements RocketFactory<ExperimentalRocket> {  
    createRocket(): ExperimentalRocket {  
        return new ExperimentalRocket();  
    }  
    createPayload(): ExperimentalPayload {  
        return new ExperimentalPayload();  
    }  
    createStages(): [ExperimentalRocketStage] {  
        return [new ExperimentalRocketStage()];  
    }  
}
```

Let's move on to another concrete factory that builds a freight rocket and products of its family, starting with the rocket components:

```
class Satellite implements Payload {  
    constructor(  
        public id: number,  
        public weight: number  
    ) {}  
}  
  
class FreightRocketFirstStage implements Stage {  
    engines: Engine[];  
}  
  
class FreightRocketSecondStage implements Stage {  
    engines: Engine[];  
}  
  
type FreightRocketStages =
```

```
[FreightRocketFirstStage, FreightRocketSecondStage];
```

Continue with the rocket itself:

```
class FreightRocket implements Rocket {  
    payload: Satellite;  
    stages: FreightRocketStages;  
}
```

With the structures or classes of the freight rocket family defined, we are ready to implement its factory:

```
class FreightRocketFactory  
implements RocketFactory<FreightRocket> {  
    nextSatelliteId = 0;  
    createRocket(): FreightRocket {  
        return new FreightRocket();  
    }  
    createPayload(): Satellite {  
        return new Satellite(this.nextSatelliteId++, 100);  
    }  
    createStages(): FreightRocketStages {  
        return [  
            new FreightRocketFirstStage(),  
            new FreightRocketSecondStage()  
        ];  
    }  
}
```

Now we once again have two families of rockets and their factories, and we can use the same client to build different rockets by passing different factories:

```
let client = new Client();  
  
let experimentalRocketFactory = new ExperimentalRocketFactory();  
let freightRocketFactory = new FreightRocketFactory();  
  
let experimentalRocket =  
    client.buildRocket(experimentalRocketFactory);  
  
let freightRocket = client.buildRocket(freightRocketFactory);
```

Consequences

The Abstract Factory Pattern makes it easy and smooth to change the entire family of products. This is the direct benefit brought by the factory level abstraction. As a consequence, it also brings other benefits, as well as some disadvantages at the same time.

On the one hand, it provides better compatibility within the products in a specific family. As the products built by a single factory are usually meant to work together, we can assume that they tend to cooperate more easily.

But on the other hand, it relies on a common outline of the building process, although for a well-abstracted building process, this won't always be an issue. We can also parameterize factory methods on both concrete factories and the client to make the process more flexible.

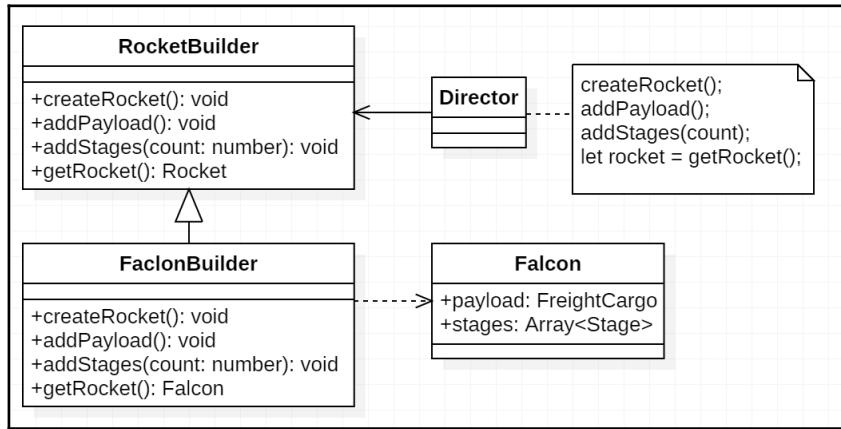
Of course, an abstract factory does not have to be a pure interface or an abstract class with no methods implemented. An implementation in practice should be decided based on detailed context.

Although the Abstract Factory Pattern and Factory Method Pattern have abstractions of different levels, what they encapsulate are similar. For building a product with multiple components, the factories split the products into components to gain flexibility. However, a fixed family of products and their internal components may not always satisfy the requirements, and thus we may consider the Builder Pattern as another option.

Builder

While Factory Patterns expose the internal components (such as the payload and stages of a rocket), the Builder Pattern encapsulates them by exposing only the building steps and provides the final products directly. At the same time, the Builder Pattern also encapsulates the internal structures of a product. This makes it possible for a more flexible abstraction and implementation of building complex objects.

The Builder Pattern also introduces a new role called **director**, as shown in the following diagram. It is quite like the client in the Abstract Factory Pattern, although it cares only about build steps or pipelines:



Now the only constraint from **RocketBuilder** that applies to a product of its subclass is the overall shape of a **Rocket**. This might not bring a lot of benefits with the **Rocket** interface we previously defined, which exposes some details of the rocket that the clients (by clients I mean those who want to send their satellites or other kinds of payload to space) may not care about that much. For these clients, what they want to know might just be which orbit the rocket is capable of sending their payloads to, rather than how many and what stages this rocket has.

Participants

The participants of a typical Builder Pattern implementation include the following:

- **Builder:** `RocketBuilder`

Defines the interface of a builder that builds products.

- **Concrete builder:** `FalconBuilder`

Implements methods that build parts of the products, and keeps track of the current building state.

- **Director**

Defines the steps and collaborates with builders to build products.

- **Final product:** Falcon

The product built by a builder.

Pattern scope

The Builder Pattern has a similar scope to the Abstract Factory Pattern, which extracts abstraction from a complete collection of operations that will finally initiate the products. Compared to the Abstract Factory Pattern, a builder in the Builder Pattern focuses more on the building steps and the association between those steps, while the Abstract Factory Pattern puts that part into the clients and makes its factory focus on producing components.

Implementation

As now we are assuming that stages are not the concern of the clients who want to buy rockets to carry their payloads, we can remove the `stages` property from the general Rocket interface:

```
interface Rocket {  
    payload: Payload;  
}
```

There is a rocket family called sounding rocket that sends probes to near space. And this means we don't even need to have the concept of stages. `SoundingRocket` is going to have only one engine property other than `payload` (which will be a `Probe`), and the only engine will be a `SolidRocketEngine`:

```
class Probe implements Payload {  
    weight: number;  
}  
  
class SolidRocketEngine extends Engine {}  
  
class SoundingRocket implements Rocket {  
    payload: Probe;  
    engine: SolidRocketEngine;  
}
```

But still we need rockets to send satellites, which usually use LiquidRocketEngine:

```
class LiquidRocketEngine extends Engine {  
    fuelLevel = 0;  
    refuel(level: number): void {  
        this.fuelLevel = level;  
    }  
}
```

And we might want to have the corresponding LiquidRocketStage abstract class that handles refuelling:

```
abstract class LiquidRocketStage implements Stage {  
    engines: LiquidRocketEngine[] = [];  
    refuel(level = 100): void {  
        for (let engine of this.engines) {  
            engine.refuel(level);  
        }  
    }  
}
```

Now we can update FreightRocketFirstStage and FreightRocketSecondStage as subclasses of LiquidRocketStage:

```
class FreightRocketFirstStage extends LiquidRocketStage {  
    constructor(thrust: number) {  
        super();  
        let enginesNumber = 4;  
        let singleEngineThrust = thrust / enginesNumber;  
        for (let i = 0; i < enginesNumber; i++) {  
            let engine =  
                new LiquidRocketEngine(singleEngineThrust);  
            this.engines.push(engine);  
        }  
    }  
}  
  
class FreightRocketSecondStage extends LiquidRocketStage {  
    constructor(thrust: number) {  
        super();  
        this.engines.push(new LiquidRocketEngine(thrust));  
    }  
}
```

The FreightRocket will remain the same as it was:

```
type FreightRocketStages =  
    [FreightRocketFirstStage, FreightRocketSecondStage];
```

```
class FreightRocket implements Rocket {  
    payload: Satellite;  
    stages = [] as FreightRocketStages;  
}
```

And, of course, there is the builder. This time, we are going to use an abstract class that has the builder partially implemented, with generics applied:

```
abstract class RocketBuilder<  
    TRocket extends Rocket,  
    TPayload extends Payload  
> {  
    createRocket(): void { }  
    addPayload(payload: TPayload): void { }  
    addStages(): void { }  
    refuelRocket(): void { }  
    abstract get rocket(): TRocket;  
}
```

 There's actually no abstract method in this abstract class. One of the reasons is that specific steps might be optional to certain builders. By implementing no-op methods, the subclasses can just leave the steps they don't care about empty.

Here is the implementation of the Director class:

```
class Director {  
    prepareRocket<  
        TRocket extends Rocket,  
        TPayload extends Payload  
>(  
        builder: RocketBuilder<TRocket, TPayload>,  
        payload: TPayload  
) : TRocket {  
        builder.createRocket();  
        builder.addPayload(payload);  
        builder.addStages();  
        builder.refuelRocket();  
        return builder.rocket;  
    }  
}
```

 Be cautious, without explicitly providing a building context, the builder instance relies on the building pipelines being queued (either synchronously or asynchronously). One way to avoid risk (especially with asynchronous operations) is to initialize a builder instance every time you prepare a rocket.

Now it's time to implement concrete builders, starting with `SoundingRocketBuilder`, which builds a `SoundingRocket` with only one `SolidRocketEngine`:

```
class SoundingRocketBuilder
extends RocketBuilder<SoundingRocket, Probe> {
    private buildingRocket: SoundingRocket;
    createRocket(): void {
        this.buildingRocket = new SoundingRocket();
    }
    addPayload(probe: Probe): void {
        this.buildingRocket.payload = probe;
    }
    addStages(): void {
        let payload = this.buildingRocket.payload;
        this.buildingRocket.engine =
            new SolidRocketEngine(payload.weight);
    }
    get rocket(): SoundingRocket {
        return this.buildingRocket;
    }
}
```

There are several notable things in this implementation:

- The `addStages` method relies on the previously added payload to add an engine with the correct thrust specification.
- The `refuel` method is not overridden (so it remains no-op) because a solid rocket engine does not need to be refueled.

We've sensed a little about the context provided by a builder, and it could have a significant influence on the result. For example, let's take a look at `FreightRocketBuilder`. It could be similar to `SoundingRocket` if we don't take the `addStages` and `refuel` methods into consideration:

```
class FreightRocketBuilder
extends RocketBuilder<FreightRocket, Satellite> {
    private buildingRocket: FreightRocket;
    createRocket(): void {
        this.buildingRocket = new FreightRocket();
    }
    addPayload(satellite: Satellite): void {
        this.buildingRocket.payload = satellite;
    }
    get rocket(): FreightRocket {
        return this.buildingRocket;
    }
}
```

```
}
```

Assume that a payload that weighs less than 1000 takes only one stage to send into space, while payloads weighing more take two or more stages:

```
addStages(): void {
    let rocket = this.buildingRocket;
    let payload = rocket.payload;
    let stages = rocket.stages;
    stages[0] = new FreightRocketFirstStage(payload.weight * 4);
    if (payload.weight >= FreightRocketBuilder.oneStageMax) {
        stages[1] = FreightRocketSecondStage(payload.weight);
    }
}

static oneStageMax = 1000;
```

When it comes to refueling, we can even decide how much to refuel based on the weight of the payloads:

```
refuel(): void {
    let rocket = this.buildingRocket;
    let payload = rocket.payload;
    let stages = rocket.stages;
    let oneMax = FreightRocketBuilder.oneStageMax;
    let twoMax = FreightRocketBuilder.twoStagesMax;
    let weight = payload.weight;
    stages[0].refuel(Math.min(weight, oneMax) / oneMax * 100);
    if (weight >= oneMax) {
        stages[1]
            .refuel((weight - oneMax) / (twoMax - oneMax) * 100);
    }
}

static oneStageMax = 1000;
static twoStagesMax = 2000;
```

Now we can prepare different rockets ready to launch, with different builders:

```
let director = new Director();

let soundingRocketBuilder = new SoundingRocketBuilder();
let probe = new Probe();
let soundingRocket
    = director.prepareRocket(soundingRocketBuilder, probe);

let freightRocketBuilder = new FreightRocketBuilder();
let satellite = new Satellite(0, 1200);
```

```
let freightRocket  
= director.prepareRocket(freightRocketBuilder, satellite);
```

Consequences

As the Builder Pattern takes greater control of the product structures and how the building steps influence each other, it provides the maximum flexibility by subclassing the builder itself, without changing the director (which plays a similar role to a client in the Abstract Factory Pattern).

Prototype

As JavaScript is a prototype-based programming language, you might be using prototype related patterns all the time without knowing it.

We've talked about an example in the Abstract Factory Pattern, and part of the code is like this:

```
class FreightRocketFactory  
implements RocketFactory<FreightRocket> {  
    createRocket(): FreightRocket {  
        return new FreightRocket();  
    }  
}
```

Sometimes we may need to add a subclass just for changing the class name while performing the same `new` operation. Instances of a single class usually share the same methods and properties, so we can `clone` one existing instance for new instances to be created. That is the concept of a prototype.

But in JavaScript, with the prototype concept built-in, `new Constructor()` does basically what a `clone` method would do. So actually a constructor can play the role of a concrete factory in some way:

```
interface Constructor<T> {  
    new (): T;  
}  
  
function createFancyObject<T>(constructor: Constructor<T>): T {  
    return new constructor();  
}
```

With this privilege, we can parameterize product or component classes as part of other patterns and make creation even more flexible.

There is something that could easily be ignored when talking about the Prototype Pattern in JavaScript: cloning with the state. With the `class` syntax sugar introduced in ES6, which hides the prototype modifications, we may occasionally forget that we can actually modify prototypes directly:

```
class Base {  
    state: number;  
}  
  
let base = new Base();  
base.state = 0;  
  
class Derived extends Base {}  
Derived.prototype = base;  
  
let derived = new Derived();
```

Now, the `derived` object will keep the `state` of the `base` object. This could be useful when you want to create copies of a specific instance, but keep in mind that properties in a prototype of these copies are not the *own properties* of these cloned objects.

Singleton

There are scenarios in which only one instance of the specific class should ever exist, and that leads to Singleton Pattern.

Basic implementations

The simplest singleton in JavaScript is an object literal; it provides a quick and cheap way to create a unique object:

```
const singleton = {  
    foo(): void {  
        console.log('bar');  
    }  
};
```

But sometimes we might want private variables:

```
const singleton = () => {
  let bar = 'bar';
  return {
    foo(): void {
      console.log(bar);
    }
  };
})();
```

Or we want to take the advantage of an anonymous constructor function or class expression in ES6:

```
const singleton = new class {
  private _bar = 'bar';
  foo(): void {
    console.log(this._bar);
  }
}();
```



Remember that the `private` modifier only has an effect at compile time, and simply disappears after being compiled to JavaScript (although of course its accessibility will be kept in `.d.ts`).

However, it is possible to have the requirements for creating new instances of “singletons” sometimes. Thus a normal class will still be helpful:

```
class Singleton {
  bar = 'bar';
  foo(): void {
    console.log(bar);
  }
  private static _default: Singleton;

  static get default(): Singleton {
    if (!Singleton._default) {
      Singleton._default = new Singleton();
    }
    return Singleton._default;
  }
}
```

Another benefit brought by this approach is lazy initialization: the object only gets initialized when it gets accessed the first time.

Conditional singletons

Sometimes we might want to get “singletons” based on certain conditions. For example, every country usually has only one capital city, thus a capital city could be treated as a singleton under the scope of the specific country.

The condition could also be the result of context rather than explicit arguments. Assuming we have a class `Environment` and its derived classes, `WindowsEnvironment` and `UnixEnvironment`, we would like to access the correct environment singleton across platforms by using `Environment.default` and apparently, a selection could be made by the `default` getter.

For more complex scenarios, we might want a registration-based implementation to make it extendable.

Summary

In this chapter, we've talked about several important creational design patterns including the Factory Method, Abstract Factory, Builder, Prototype, and Singleton.

Starting with the Factory Method Pattern, which provides flexibility with limited complexity, we also explored the Abstract Factory Pattern, the Builder Pattern and the Prototype Pattern, which share similar levels of abstraction but focus on different aspects. These patterns have more flexibility than the Factory Method Pattern, but are more complex at the same time. With the knowledge of the idea behind each of the patterns, we should be able to choose and apply a pattern accordingly.

While comparing the differences, we also found many things in common between different creational patterns. These patterns are unlikely to be isolated from others and some of them can even collaborate with or complete each other.

In the next chapter, we'll continue to discuss structural patterns that help to form large objects with complex structures.

4

Structural Design Patterns

While creational patterns play the part of flexibly creating objects, structural patterns, on the other hand, are patterns about composing objects. In this chapter, we are going to talk about structural patterns that fit different scenarios.

If we take a closer look at structural patterns, they can be divided into *structural class patterns* and *structural object patterns*. Structural class patterns are patterns that play with “interested parties” themselves, while structural object patterns are patterns that weave pieces together (like Composite Pattern). These two kinds of structural patterns complement each other to some degree.

Here are the patterns we'll walk through in this chapter:

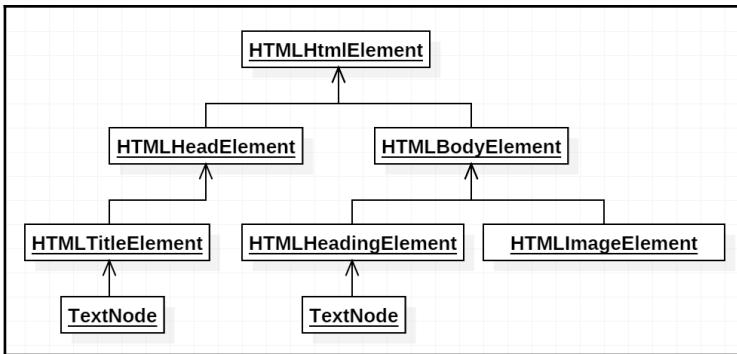
- **Composite:** Builds tree-like structures using primitive and composite objects. A good example would be the DOM tree that forms a complete page.
- **Decorator:** Adds functionality to classes or objects dynamically.
- **Adapter:** Provides a general interface and work with different adaptees by implementing different concrete adapters. Consider providing different database choices for a single content management system.
- **Bridge:** Decouples the abstraction from its implementation, and make both of them interchangeable.
- **Façade:** Provides a simplified interface for the combination of complex subsystems.
- **Flyweight:** Shares stateless objects that are being used many times to improve memory efficiency and performance.
- **Proxy:** Acts as the surrogate that takes extra responsibilities when accessing objects it manages.

Composite Pattern

Objects under the same class could vary from their properties or even specific subclasses, but a complex object can have more than just normal properties. Taking DOM elements, for example, all the elements are instances of class `Node`. These nodes form tree structures to represent different pages, but every node in these trees is complete and uniform compared to the node at the root:

```
<html>
  <head>
    <title>TypeScript</title>
  </head>
  <body>
    <h1>TypeScript</h1>
    <img />
  </body>
</html>
```

The preceding HTML represents a DOM structure like this:



All of the preceding objects are instances of `Node`, they implement the interface of a *component* in Composite Pattern. Some of these nodes like HTML elements (except for `HTMLImageElement`) in this example have child nodes (components) while others don't.

Participants

The participants of Composite Pattern implementation include:

- **Component:** Node

Defines the interface and implement the default behavior for objects of the composite. It should also include an interface to access and manage the child components of an instance, and optionally a reference to its parent.

- **Composite:** Includes some HTML elements, like `HTMLHeadElement` and `HTMLBodyElement`

Stores child components and implements related operations, and of course its own behaviors.

- **Leaf:** `TextNode`, `HTMLImageElement`

Defines behaviors of a primitive component.

- **Client:**

Manipulates the composite and its components.

Pattern scope

Composite Pattern applies when objects can and should be abstracted recursively as components that form tree structures. Usually, it would be a natural choice when a certain structure needs to be formed as a tree, such as trees of view components, abstract syntax trees, or trees that represent file structures.

Implementation

We are going to create a composite that represents simple file structures and has limited kinds of components.

First of all, let's import related node modules:

```
import * as Path from 'path';
import * as FS from 'fs';
```



Module path and fs are built-in modules of Node.js, please refer to Node.js documentation for more information: <https://nodejs.org/api/>.



It is my personal preference to have the first letter of a namespace (if it's not a function at the same time) in uppercase, which reduces the chance of conflicts with local variables. But a more popular naming style for namespace in JavaScript does not.

Now we need to make abstraction of the components, say FileSystemObject:

```
abstract class FileSystemObject {  
    constructor(  
        public path: string,  
        public parent?: FileSystemObject  
    ) {}  
  
    get basename(): string {  
        return Path.basename(this.path);  
    }  
}
```

We are using abstract class because we are not expecting to use FileSystemObject directly. An optional parent property is defined to allow us to visit the upper component of a specific object. And the basename property is added as a helper for getting the basename of the path.

The FileSystemObject is expected to have subclasses, FolderObject and FileObject. For FolderObject, which is a composite that may contain other folders and files, we are going to add an items property (getter) that returns other FileSystemObject it contains:

```
class FolderObject extends FileSystemObject {  
    items: FileSystemObject[];  
  
    constructor(path: string, parent?: FileSystemObject) {  
        super(path, parent);  
    }  
}
```

We can initialize the items property in the constructor with actual files and folders existing at given path:

```
this.items = FS  
.readdirSync(this.path)  
.map(path => {
```

```
let stats = FS.statSync(path);

if (stats.isFile()) {
    return new FileObject(path, this);
} else if (stats.isDirectory()) {
    return new FolderObject(path, this);
} else {
    throw new Error('Not supported');
}
});
```

You may have noticed we are forming `items` with different kinds of objects, and we are also passing `this` as the parent of newly created child components.

And for `FileObject`, we'll add a simple `readAll` method that reads all bytes of the file:

```
class FileObject extends FileSystemObject {
    readAll(): Buffer {
        return FS.readFileSync(this.path);
    }
}
```

Currently, we are reading the child items inside a folder from the actual filesystem when a folder object gets initiated. This might not be necessary if we want to access this structure on demand. We may actually create a getter that calls `readdir` only when it's accessed, thus the object would act like a proxy to the real filesystem.

Consequences

Both the primitive object and composite object in Composite Pattern share the component interface, which makes it easy for developers to build a composite structure with fewer things to remember.

It also enables the possibility of using markup languages like XML and HTML to represent a really complex object with extreme flexibility. Composite Pattern can also make the rendering easier by having components rendered recursively.

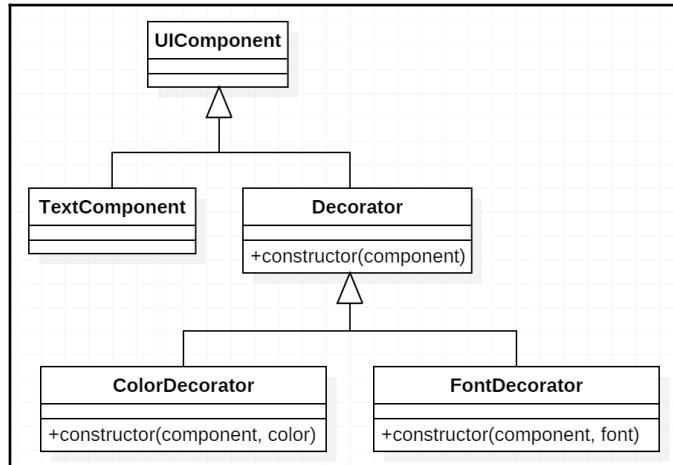
As most components are compatible with having child components or being child components of their parents themselves, we can easily create new components that work great with existing ones.

Decorator Pattern

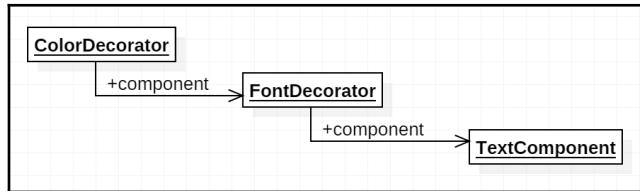
Decorator Pattern adds new functionality to an object dynamically, usually without compromising the original features. The word decorator in Decorator Pattern does share something with the word decorator in the ES-next decorator syntax, but they are not exactly the same. Classical Decorator Pattern as a phrase would differ even more.

The classical Decorator Pattern works with a composite, and the brief idea is to create decorators as components that do the decorating work. As composite objects are usually processed recursively, the decorator components would get processed automatically. So it becomes your choice to decide what it does.

The inheritance hierarchy could be like the following structure shown:



The decorators are applied recursively like this:



There are two prerequisites for the decorators to work correctly: the awareness of context or object that a decorator is decorating, and the ability of the decorators being applied. The Composite Pattern can easily create structures that satisfy those two prerequisites:

- The decorator knows what it decorates as the `component` property
- The decorator gets applied when it is rendered recursively

However, it doesn't really need to take a structure like a composite to gain the benefits from Decorator Pattern in JavaScript. As JavaScript is a dynamic language, if you can get your decorators called, you may add whatever you want to an object.

Taking method `log` under `console` object as an example, if we want a timestamp before every log, we can simply replace the `log` function with a wrapper that has the timestamp prefixed:

```
const _log = console.log;
console.log = function () {
    let timestamp = `[$new Date().toTimeString()]`;
    return _log.apply(this, [timestamp, ...arguments]);
};
```

Certainly, this example has little to do with the classical Decorator Pattern, but it enables a different way for this pattern to be done in JavaScript. Especially with the help of new decorator syntax:

```
class Target {
    @decorator
    method() {
        // ...
    }
}
```



TypeScript provides the decorator syntax transformation as an experimental feature. To learn more about decorator syntax, please check out the following link: <http://www.typescriptlang.org/docs/handbook/decorators.html>.

Participants

The participants of classical Decorator Pattern implementation include:

- **Component:** `UIComponent`

Defines the interface of the objects that can be decorated.

- **ConcreteComponent:** TextComponent

Defines additional functionalities of the concrete component.

- **Decorator:** Decorator

Defines a reference to the component to be decorated, and manages the context. Conforms the interface of a component with proper behaviors.

- **ConcreteDecorator:** ColorDecorator, FontDecorator

Defines additional features and exposes API if necessary.

Pattern scope

Decorator Pattern usually cares about objects, but as JavaScript is prototype-based, decorators would work well with the classes of objects through their prototypes.

The classical implementation of Decorator Pattern could have much in common with other patterns we are going to talk about later, while the function one seems to share less.

Implementation

In this part, we'll talk about two implementations of Decorator Pattern. The first one would be classical Decorator Pattern that decorates the target by wrapping with new classes that conform to the interface of `UIComponent`. The second one would be decorators written in new decorator syntax that processes target objects.

Classical decorators

Let's get started by defining the outline of objects to be decorated. First, we'll have the `UIComponent` as an abstract class, defining its abstract function `draw`:

```
abstract class UIComponent {  
    abstract draw(): void;  
}
```

Then a TextComponent that extends the `UIComponent`, as well as its text contents of class `Text`:

```
class Text {  
    content: string;  
  
    setColor(color: string): void {}  
    setFont(font: string): void {}  
  
    draw(): void {}  
}  
  
class TextComponent extends UIComponent {  
    texts: Text[];  
  
    draw(): void {  
        for (let text of this.texts) {  
            text.draw();  
        }  
    }  
}
```

What's next is to define the interface of decorators to decorate objects that are instances of class `TextComponent`:

```
class Decorator extends UIComponent {  
    constructor(  
        public component: TextComponent  
    ) {  
        super();  
    }  
  
    get texts(): Text[] {  
        return this.component.texts;  
    }  
  
    draw(): void {  
        this.component.draw();  
    }  
}
```

Now we have everything for concrete decorators. In this example, `ColorDecorator` and `FontDecorator` look similar:

```
class ColorDecorator extends Decorator {  
    constructor(  
        component: TextComponent,  
        public color: string
```

```

) {
    super(component);
}

draw(): void {
    for (let text of this.texts) {
        text.setColor(this.color);
    }

    super.draw();
}

class FontDecorator extends Decorator {
    constructor(
        component: TextComponent,
        public font: string
    ) {
        super(component);
    }

    draw(): void {
        for (let text of this.texts) {
            text.setFont(this.font);
        }

        super.draw();
    }
}

```



In the implementation just described, `this.texts` in `draw` method calls the getter defined on class `Decorator`. As this in that context would ideally be an instance of class `ColorDecorator` or `FontDecorator`; the `texts` it accesses would finally be the array in its `component` property.



This could be even more interesting or confusing if we have nested decorators like we will soon. Try to draw a schematic diagram if it confuses you later.

Now it's time to actually assemble them:

```

let decoratedComponent = new ColorDecorator(
    new FontDecorator(
        new TextComponent(),
        'sans-serif'
    ),
)

```

```
'black'  
);
```

The order of nesting decorators does not matter in this example. As either `ColorDecorator` or `FontDecorator` is a valid `UIComponent`, they can be easily dropped in and replace previous `TextComponent`.

Decorators with ES-next syntax

There is a limitation with classical Decorator Pattern that can be pointed out directly via its nesting form of decorating. That applies to ES-next decorators as well. Take a look at the following example:

```
class Foo {  
  @prefix  
  @suffix  
  getContent(): string {  
    return '...';  
  }  
}
```



What follows the `@` character is an expression that evaluates to a decorator. While a decorator is a function that processes target objects, we usually use higher-order functions to parameterize a decorator.

We now have two decorators `prefix` and `suffix` decorating the `getContent` method. It seems that they are just parallel at first glance, but if we are going to add a prefix and suffix onto the content returned, like what the name suggests, the procedure would actually be recursive rather than parallel just like the classical implementation.

To make decorators cooperate with others as we'd expect, we need to handle things carefully:

```
function prefix(  
  target: Object,  
  name: string,  
  descriptor: PropertyDescriptor  
) : PropertyDescriptor {  
  let method = descriptor.value as Function;  
  
  if (typeof method !== 'function') {  
    throw new Error('Expecting decorating a method');  
  }
```

```
return {
  value: function () {
    return '[prefix] ' + method.apply(this, arguments);
  },
  enumerable: descriptor.enumerable,
  configurable: descriptor.configurable,
  writable: descriptor.writable
};
}
```



In current ECMAScript decorator proposal, when decorating a method or property (usually with getter or setter), you will have the third argument passed in as the property descriptor.



Check out the following link for more information about property descriptors: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.

The `suffix` decorator would be just like the `prefix` decorator. So I'll save the code lines here.

Consequences

The key to the Decorator Pattern is being able to add functionalities dynamically, and decorators are usually expected to play nice with each other. Those expectations of Decorator Pattern make it really flexible to form a customized object. However, it would be hard for certain types of decorators to actually work well together.

Consider decorating an object with multiple decorators just like the second example of implementation, would the decorating order matter? Or should the decorating order matter?

A properly written decorator should always work no matter where it is in the decorators list. And it's usually *preferred* that the decorated target behaves almost the same with decorators decorated in different orders.

Adapter Pattern

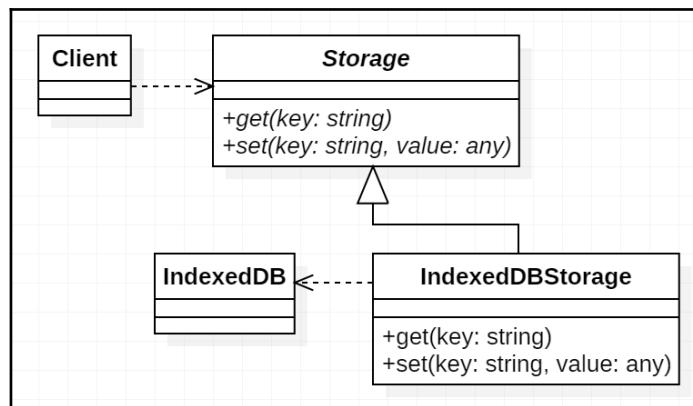
Adapter Pattern connects existing classes or objects with another existing client. It makes classes that are not designed to work together possible to cooperate with each other.

An adapter could be either a *class* adapter or an *object* adapter. A class adapter extends the adaptee class and exposes extra APIs that would work with the client. An object adapter, on the other hand, does not extend the adaptee class. Instead, it stores the adaptee as a dependency.

The class adapter is useful when you need to access protected methods or properties of the adaptee class. However, it also has some restrictions when it comes to the JavaScript world:

- The adaptee class needs to be extendable
- If the client target is an abstract class other than pure interface, you can't extend the adaptee class and the client target with the same adapter class without a *mixin*
- A single class with two sets of methods and properties could be confusing

Due to those limitations, we are going to talk more about object adapters. Taking browser-side storage for example, we'll assume we have a client working with storage objects that have both methods `get` and `set` with correct signatures (for example, a storage that stores data online through AJAX). Now we want the client to work with IndexedDB for faster response and offline usage; we'll need to create an adapter for IndexedDB that gets and sets data:



We are going to use Promise for receiving results or errors of asynchronous operations. See the following link for more information if you are not yet familiar with Promise: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Participants

The participants of Adapter Pattern include:

- **Target:** Storage

Defines the interface of existing targets that works with client

- **Adaptee:** IndexedDB

The implementation that is not designed to work with the client

- **Adapter:** IndexedDBStorage

Conforms the interface of target and interacts with adaptee

- **Client.**

Manipulates the target

Pattern scope

Adapter Pattern can be applied when the existing client class is not designed to work with the existing adaptees. It focuses on the unique *adapter* part when applying to different combinations of clients and adaptees.

Implementation

Start with the Storage interface:

```
interface Storage {  
    get<T>(key: string): Promise<T>;  
    set<T>(key: string, value: T): Promise<void>;  
}
```



We defined the `get` method with generic, so that if we neither specify the generic type, nor cast the value type of a returned Promise, the type of the value would be `{ }`. This would probably fail following type checking.

With the help of examples found on MDN, we can now set up the IndexedDB adapter. Visit [IndexedDBStorage: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB).

The creation of IndexedDB instances is asynchronous. We could put the opening operation inside a get or set method so the database can be opened on demand. But for now, let's make it easier by creating an instance of `IndexedDBStorage` that has a database instance which is already opened.

However, constructors usually don't have asynchronous code. Even if they do, it cannot apply changes to the instance before completing the construction. Fortunately, Factory Method Pattern works well with asynchronous initiation:

```
class IndexedDBStorage implements Storage {
    constructor(
        public db: IDBDatabase,
        public storeName = 'default'
    ) { }

    open(name: string): Promise<IndexedDBStorage> {
        return new Promise<IndexedDBStorage>(
            (resolve, reject) => {
                let request = indexedDB.open(name);
                // ...
            });
    }
}
```

Inside the Promise resolver of method `open`, we'll get the asynchronous work done:

```
let request = indexedDB.open(name);

request.onsuccess = event => {
    let db = request.result as IDBDatabase;
    let storage = new IndexedDBStorage(db);
    resolve(storage);
};

request.onerror = event => {
    reject(request.error);
};
```

Now when we are accessing an instance of `IndexedDBStorage`, we can assume it has an opened database and is ready to make queries. To make changes or to get values from the database, we need to create a transaction. Here's how:

```
get<T>(key: string): Promise<T> {
    return new Promise<T>((resolve, reject) => {
        let transaction = this.db.transaction(this.storeName);
        let store = transaction.objectStore(this.storeName);

        let request = store.get(key);

        request.onsuccess = event => {
            resolve(request.result);
        };

        request.onerror = event => {
            reject(request.error);
        };
    });
}
```

Method `set` is similar. But while the transaction is by default read-only, we need to explicitly specify '`readwrite`' mode.

```
set<T>(key: string, value: T): Promise<void> {
    return new Promise<void>((resolve, reject) => {
        let transaction =
            this.db.transaction(this.storeName, 'readwrite');
        let store = transaction.objectStore(this.storeName);

        let request = store.put(value, key);

        request.onsuccess = event => {
            resolve();
        };

        request.onerror = event => {
            reject(request.error);
        };
    });
}
```

And now we can have a drop-in replacement for the previous storage used by the client.

Consequences

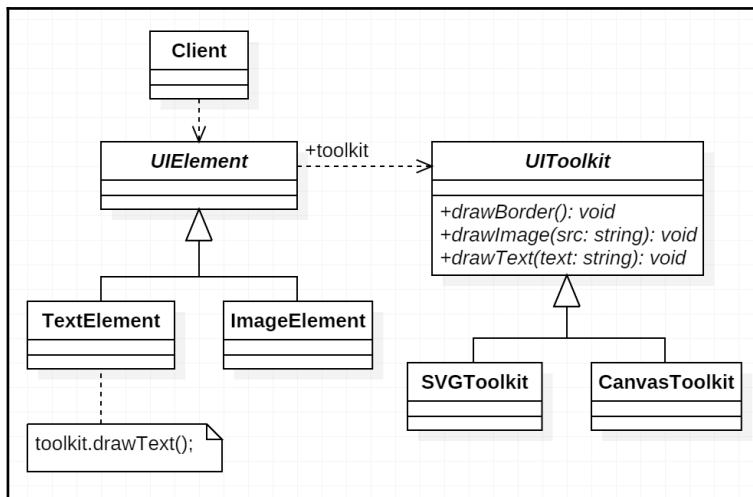
By applying Adapter Pattern, we can fill the gap between classes that originally would not work together. In this situation, Adapter Pattern is quite a straightforward solution that might come to mind.

But in other scenarios like a debugger *adapter* for debugging extensions of an IDE, the implementation of Adapter Pattern could be more challenging.

Bridge Pattern

Bridge Pattern decouples the abstraction manipulated by clients from functional implementations and makes it possible to add or replace these abstractions and implementations easily.

Take a set of *cross-API* UI elements as an example:



We have the abstraction **UIElement** that can access different implementations of **UIToolkit** for creating different UI based on either SVG or canvas. In the preceding structure, the *bridge* is the connection between **UIElement** and **UIToolkit**.

Participants

The participants of Bridge Pattern include:

- **Abstraction:** UIElement

Defines the interface of objects to be manipulated by the client and stores the reference to its implementer.

- **Refined abstraction:** TextElement, ImageElement

Extends abstraction with specialized behaviors.

- **Implementer:** UIToolkit

Defines the interface of a general implementer that will eventually carry out the operations defined in abstractions. The implementer usually cares only about basic operations while the abstraction will handle high-level operations.

- **Concrete implementer:** SVGToolkit, CanvasToolkit

Implements the implementer interface and manipulates low-level APIs.

Pattern scope

Although having abstraction and implementer decoupled provides Bridge Pattern with the ability to work with several abstractions and implementers, most of the time, bridge patterns work only with a single implementer.

If you take a closer look, you will find Bridge Pattern is extremely similar to Adapter Pattern. However, while Adapter Pattern tries to make existing classes cooperate and focuses on the adapters part, Bridge Pattern foresees the divergences and provides a well-thought-out and universal interface for its abstractions that play the part of adapters.

Implementation

A working implementation could be non-trivial in the example we are talking about. But we can still sketch out the skeleton easily.

Start with implementer `UIToolkit` and abstraction `UIElement` that are directly related to the bridge concept:

```
interface UIToolkit {  
    drawBorder(): void;  
    drawImage(src: string): void;  
    drawText(text: string): void;  
}  
  
abstract class UIElement {  
    constructor(  
        public toolkit: UIToolkit  
    ) {}  
  
    abstract render(): void;  
}
```

And now we can extend `UIElement` for refined abstractions with different behaviors. First the `TextElement` class:

```
class TextElement extends UIElement {  
    constructor(  
        public text: string,  
        toolkit: UIToolkit  
    ) {  
        super(toolkit);  
    }  
  
    render(): void {  
        this.toolkit.drawText(this.text);  
    }  
}
```

And the `ImageElement` class with similar code:

```
class ImageElement extends UIElement {  
    constructor(  
        public src: string,  
        toolkit: UIToolkit  
    ) {  
        super(toolkit);  
    }  
  
    render(): void {  
        this.toolkit.drawImage(this.src);  
    }  
}
```

By creating concrete `UIKit` subclasses, we can manage to make everything together with the client. But as it could lead to hard work we would not want to touch now, we'll skip it by using a variable pointing to `undefined` in this example:

```
let toolkit: UIKit;

let imageElement = new ImageElement('foo.jpg', toolkit);
let textElement = new TextElement('bar', toolkit);

imageElement.render();
textElement.render();
```

In the real world, the render part could also be a heavy lift. But as it's coded at a relatively higher-level, it tortures you in a different way.

Consequences

Despite having completely different names for the abstraction (`UIElement`) in the example above and the adapter interface (`Storage`), they play similar roles in a static combination.

However, as we mentioned in the pattern scope section, the intentions of Bridge Pattern and Adapter Pattern differ.

By decoupling the abstraction and implementer, Bridge Pattern brings great extensibility to the system. The client does not need to know about the implementation details, and this helps to build more stable systems as it forms a healthier dependency structure.

Another bonus that might be brought by Bridge Pattern is that, with a properly configured build process, it can reduce compilation time as the compiler does not need to know information on the other end of the bridge when changes are made to a refined abstraction or concrete implementer.

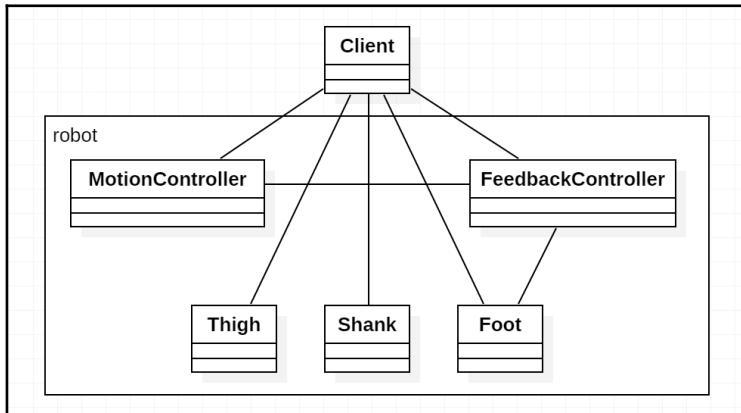
Façade Pattern

The Façade Pattern organizes subsystems and provides a unified higher-level interface. An example that might be familiar to you is a modular system. In JavaScript (and of course TypeScript), people use modules to organize code. A modular system makes projects easier to maintain, as a clean project structure can help reveal the interconnections among different parts of the project.

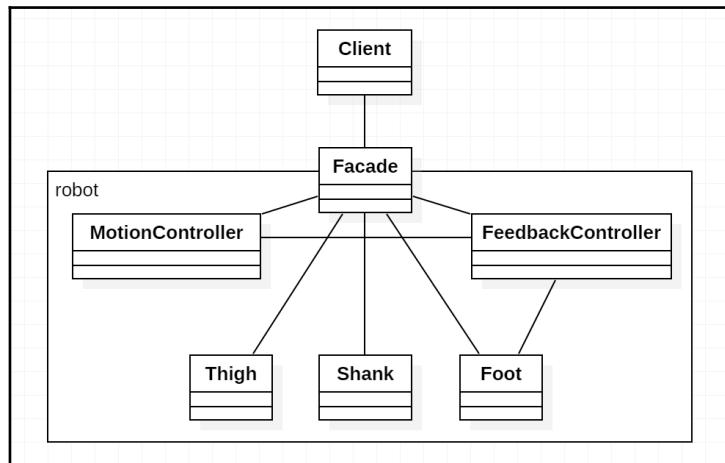
It is common that one project gets referenced by others, but obviously the project that references other projects doesn't and shouldn't care much about the inner structures of its dependencies. Thus a façade can be introduced for a dependency project to provide a higher-level API and expose what really matters to its dependents.

Take a robot as an example. People who build a robot and its components will need to control every part separately and let them cooperate at the same time. However, people who want to use this robot would only need to send simple commands like “walk” and “jump”.

For the most flexible usage, the robot “SDK” can provide classes like MotionController, FeedbackController, Thigh, Shank, Foot and so on. Possibly like the following image shows:



But certainly, most of the people who want to control or program this robot do not want to know as many details as this. What they really want is not a fancy tool box with *everything* inbox, but just an integral robot that follows their commands. Thus the robot “SDK” can actually provide a façade that controls the inner pieces and exposes much simpler APIs:



Unfortunately, Façade Pattern leaves us an open question of how to design the façade API and subsystems. Answering this question properly is not easy work.

Participants

The participants of a Façade Pattern are relatively simple when it comes to their categories:

- **Façade:** Robot

Defines a set of higher-level interfaces, and makes subsystems cooperate.

- **Subsystems:** MotionController, FeedbackController, Thigh, Shank and Foot

Implements their own functionalities and communicates internally with other subsystems if necessary. Subsystems are dependencies of a façade, and they do not depend on the façade.

Pattern scope

Facades usually act as junctions that connect a higher-level system and its subsystems. The key to the Façade Pattern is to draw a line between what a dependent should or shouldn't care about its dependencies.

Implementation

Consider putting up a robot with its left and right legs, we can actually add another abstraction layer called `Leg` that manages `Thigh`, `Shank`, and `Foot`. If we are going to separate motion and feedback controllers to different legs respectively, we may also add those two as part of the `Leg`:

```
class Leg {  
    thigh: Thigh;  
    shank: Shank;  
    foot: Foot;  
  
    motionController: MotionController;  
    feedbackController: FeedbackController;  
}
```

Before we add more details to `Leg`, let's first define `MotionController` and `FeedbackController`.

The `MotionController` is supposed to control a whole leg based on a value or a set of values. Here we are simplifying that as a single angle for not being distracted by this impossible robot:

```
class MotionController {  
    constructor(  
        public leg: Leg  
    ) {}  
  
    setAngle(angle: number): void {  
        let {  
            thigh,  
            shank,  
            foot  
        } = this.leg;  
  
        // ...  
    }  
}
```

And the FeedbackController is supposed to be an instance of EventEmitter that reports the state changes or useful events:

```
import { EventEmitter } from 'events';

class FeedbackController extends EventEmitter {
  constructor() {
    public foot: Foot
  }
  super();
}
}
```

Now we can make class Leg relatively complete:

```
class Leg {
  thigh = new Thigh();
  shank = new Shank();
  foot = new Foot();

  motionController: MotionController;
  feedbackController: FeedbackController;

  constructor() {
    this.motionController =
      new MotionController(this);
    this.feedbackController =
      new FeedbackController(this.foot);

    this.feedbackController.on('touch', () => {
      // ...
    });
  }
}
```

Let's put two legs together to sketch the skeleton of a robot:

```
class Robot {
  leftLegMotion: MotionController;
  rightLegMotion: MotionController;

  leftFootFeedback: FeedbackController;
  rightFootFeedback: FeedbackController;

  walk(steps: number): void { }
  jump(strength: number): void { }
}
```

I'm omitting the definition of classes `Thigh`, `Shank`, and `Foot` as we are not actually going to walk the robot. Now for a user that only wants to walk or jump a robot via simple API, they can make it via the `Robot` object that has everything connected.

Consequences

Façade Pattern loosens the coupling between client and subsystems. Though it does not decouple them completely as you will probably still need to work with objects defined in subsystems.

Façades usually forward operations from client to proper subsystems or even do heavy work to make them work together.

With the help of Façade Pattern, the system and the relationship and structure within the system can stay clean and intuitive.

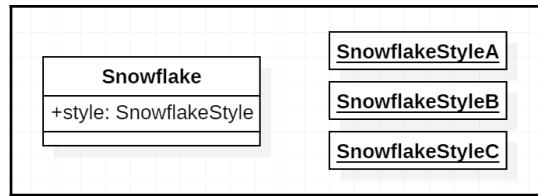
Flyweight Pattern

A flyweight in Flyweight Pattern is a stateless object that can be shared across objects or maybe classes many times. Obviously, that suggests Flyweight Pattern is a pattern about memory efficiency and maybe performance if the construction of objects is expensive.

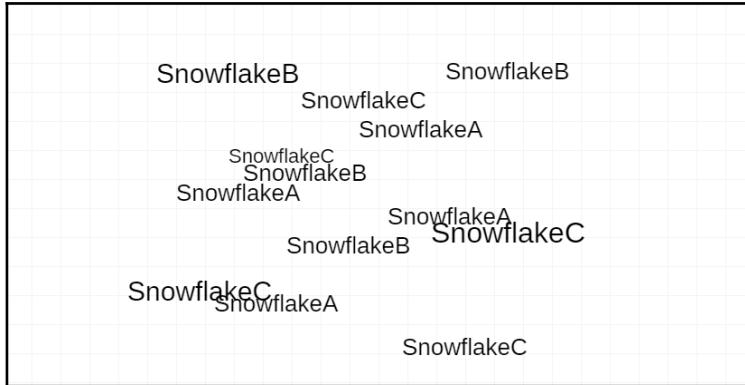
Taking drawing snowflakes as an example. Despite real snowflakes being different to each other, when we are trying to draw them onto canvas, we usually have a limited number of styles. However, by adding *properties* like sizes and transformations, we can create a beautiful snow scene with limited snowflake styles.

As a flyweight is stateless, ideally it allows multiple operations simultaneously. You might need to be cautious when working with multi-thread stuff. Fortunately, JavaScript is usually single-threaded and avoids this issue if all related code is synchronous. You will still need to take care in detailed scenarios if your code is working asynchronously.

Assume we have some flyweights of class Snowflake:



When it snows, it would look like this:



In the image above, snowflakes in different styles are the result of rendering with different properties.

It's common that we would have styles and image resources being loaded dynamically, thus we could use a `FlyweightFactory` for creating and managing flyweight objects.

Participants

The simplest implementation of Flyweight Pattern has the following participants:

- **Flyweight:** `Snowflake`

Defines the class of flyweight objects.

- **Flyweight factory:** `FlyweightFactory`

Creates and manages flyweight objects.

- Client.

Stores states of targets and uses flyweight objects to manipulate these targets.

With these participants, we assume that the manipulation could be accomplished through flyweights with different states. It would also be helpful sometimes to have concrete flyweight class allowing customized behaviors.

Pattern scope

Flyweight Pattern is a result of efforts to improving memory efficiency and performance. The implementation cares about having the instances being stateless, and it is usually the client that manages detailed states for different targets.

Implementation

What makes Flyweight Pattern useful in the snowflake example is that a snowflake with the same style usually shares the same image. The image is what consumes time to load and occupies notable memory.

We are starting with a fake `Image` class that pretends to load images:

```
class Image {  
    constructor(url: string) {}  
}
```

The `Snowflake` class in our example has only a single `image` property, and that is a property that will be shared by many snowflakes to be drawn. As the instance is now stateless, parameters from context are required for rendering:

```
class Snowflake {  
    image: Image;  
  
    constructor(  
        public style: string  
    ) {  
        let url = style + '.png';  
        this.image = new Image(url);  
    }  
  
    render(x: number, y: number, angle: number): void {  
        // ...  
    }  
}
```

```
}
```

The flyweights are managed by a factory for easier accessing. We'll have a `SnowflakeFactory` that caches created snowflake objects with certain styles:

```
const hasOwnProperty = Object.prototype.hasOwnProperty;

class SnowflakeFactory {
  cache: {
    [style: string]: Snowflake;
  } = {};
  get(style: string): Snowflake {
    let cache = this.cache;
    let snowflake: Snowflake;
    if (hasOwnProperty.call(cache, style)) {
      snowflake = cache[style];
    } else {
      snowflake = new Snowflake(style);
      cache[style] = snowflake;
    }
    return snowflake;
  }
}
```

With building blocks ready, we'll implement the client (`Sky`) that snows:

```
const SNOW_STYLES = ['A', 'B', 'C'];

class Sky {
  constructor(
    public width: number,
    public height: number
  ) { }

  snow(factory: SnowflakeFactory, count: number) { }
}
```

We are going to fill the sky with random snowflakes at random positions. Before that let's create a helper function that generates a number between 0 and a max value given:

```
function getRandomInteger(max: number): number {
  return Math.floor(Math.random() * max);
}
```

And then complete method `snow` of `Sky`:

```
  snow(factory: SnowflakeFactory, count: number) {
    let stylesCount = SNOW_STYLES.length;

    for (let i = 0; i < count; i++) {
      let style = SNOW_STYLES[getRandomInteger(stylesCount)];
      let snowflake = factory.get(style);

      let x = getRandomInteger(this.width);
      let y = getRandomInteger(this.height);

      let angle = getRandomInteger(60);

      snowflake.render(x, y, angle);
    }
  }
```

Now we may have thousands of snowflakes in the sky but with only three instances of `Snowflake` created. You can continue this example by storing the state of snowflakes and animating the snowing.

Consequences

Flyweight Pattern reduces the total number of objects involved in a system. As a direct result, it may save quite a lot memory. This saving becomes more significant when the flyweights get used by the client that processes a large number of targets.

Flyweight Pattern also brings extra logic into the system. When to use or not to use this pattern is again a balancing game between development efficiency and runtime efficiency from this point of view. Though most of the time, if there's not a good reason, we go with development efficiency.

Proxy Pattern

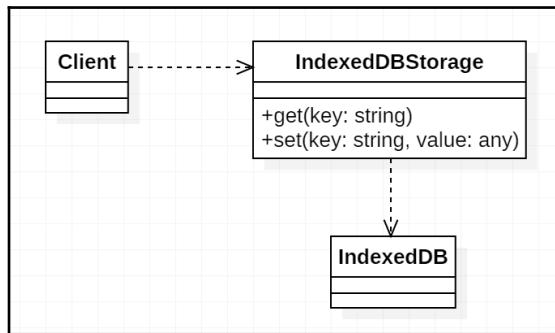
Proxy Pattern applies when the program needs to know about or to intervene the behavior of accessing objects. There are several detailed scenarios in Proxy Pattern, and we can distinguish those scenarios by their different purposes:

- **Remote proxy:** A proxy with interface to manipulate remote objects, such as data items on a remote server

- **Virtual proxy:** A proxy that manages expensive objects which need to be loaded on demand
- **Protection proxy:** A proxy that controls access to target objects, typically it verifies permissions and validates values
- **Smart proxy:** A proxy that does additional operations when accessing target objects

In the section of Adapter Pattern, we used factory method `open` that creates an object asynchronously. As a trade-off, we had to let the client wait before the object gets created.

With Proxy Pattern, we could now `open` database on demand and create storage instances synchronously.



A proxy is usually dedicated to object or objects with known methods and properties. But with the new `Proxy` API provided in ES6, we can get more interesting things done by getting to know what methods or properties are being accessed. Please refer to the following link for more information: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

Participants

The participants of Proxy Pattern include:

- **Proxy:** `IndexedDBStorage`

Defines interface and implements operations to manage access to the subject.

- **Subject:** IndexedDB

The subject to be accessed by proxy.

- **Client:** Accesses subject via proxy.

Pattern scope

Despite having a similar structure to Adapter Pattern, the key of Proxy Pattern is to intervene the access to target objects rather than to adapt an incompatible interface. Sometimes it might change the result of a specific method or the value of a certain property, but that is probably for falling back or exception handling purposes.

Implementation

There are two differences we'll have in this implementation compared to the example for pure Adapter Pattern. First, we'll create the `IndexedDBStorage` instance with a constructor, and have the database opened on demand. Second, we are going to add a useless permission checking for methods `get` and `set`.

Now when we call the method `get` or `set`, the database could either have been opened or not. Promise is a great choice for representing a value that might either be pending or settled. Consider this example:

```
let ready = new Promise<string>(resolve => {
  setTimeout(() => {
    resolve('biu~');
  }, Math.random() * 1000);
});

setTimeout(() => {
  ready.then(text => {
    console.log(text);
  });
}, 999);
```

It's hard to tell whether Promise `ready` is fulfilled when the second timeout fires. But the overall behavior is easy to predict: it will log the '`'biu~'` text in around 1 second. By replacing the Promise variable `ready` with a method or getter, it would be able to start the asynchronous operation only when needed.

So let's start the refactoring of class `IndexedDBStorage` with the getter that creates the Promise of the database to be opened:

```
private dbPromise: Promise<IDBDatabase>;  
  
constructor(  
    public name: string,  
    public storeName = 'default'  
) { }  
  
private get dbReady(): Promise<IDBDatabase> {  
    if (!this.dbPromise) {  
        this.dbPromise =  
            new Promise<IDBDatabase>((resolve, reject) => {  
                let request = indexedDB.open(name);  
  
                request.onsuccess = event => {  
                    resolve(request.result);  
                };  
  
                request.onerror = event => {  
                    reject(request.error);  
                };  
            });  
    }  
  
    return this.dbPromise;  
}
```

Now the first time we access property `dbReady`, it will open the database and create a Promise that will be fulfilled with the database being opened. To make this work with methods `get` and `set`, we just need to wrap what we've implemented into a `then` method following the `dbReady` Promise.

First for method `get`:

```
get<T>(key: string): Promise<T> {  
    return this  
        .dbReady  
        .then(db => new Promise<T>((resolve, reject) => {  
            let transaction = db.transaction(this.storeName);  
            let store = transaction.objectStore(this.storeName);  
  
            let request = store.get(key);  
  
            request.onsuccess = event => {  
                resolve(request.result);  
            };  
        }));  
}
```

```

    });
    request.onerror = event => {
      reject(request.error);
    };
  }));
}

```

And followed by updated method set:

```

set<T>(key: string, value: T): Promise<void> {
  return this
    .dbReady
    .then(db => new Promise<void>((resolve, reject) => {
      let transaction = db
        .transaction(this.storeName, 'readwrite');
      let store = transaction.objectStore(this.storeName);

      let request = store.put(value, key);

      request.onsuccess = event => {
        resolve();
      };

      request.onerror = event => {
        reject(request.error);
      };
    }));
}

```

Now we finally have the `IndexedDBStorage` property that can do a real drop-in replacement for the client that supports the interface. We are also going to add simple permission checking with a plain object that describes the permission of read and write:

```

interface Permission {
  write: boolean;
  read: boolean;
}

```

Then we will add permission checking for method `get` and `set` separately:

```

get<T>(key: string): Promise<T> {
  if (!this.permission.read) {
    return Promise.reject<T>(new Error('Permission denied'));
  }

  // ...
}

```

```
set<T>(key: string, value: T): Promise<void> {
  if (!this.permission.write) {
    return Promise.reject(new Error('Permission denied'));
  }

  // ...
}
```

You may recall Decorator Pattern when you are thinking about the permission checking part, and decorators could be used to simplify the lines written. Try to use decorator syntax to implement this permission checking yourself.

Consequences

The implementation of Proxy Pattern can usually be treated as the encapsulation of the operations to specific objects or targets. It is easy to have the encapsulation augmented without extra burden on the client.

For example, a working online database proxy could do much more than just acting like a plain surrogate. It may cache data and changes locally, or synchronize on schedule without the client being aware.

Summary

In this chapter, we learned about structural design patterns including Composite, Decorator, Adapter, Bridge, Façade, Flyweight, and Proxy. Again we found some of these patterns are highly inter related and even similar to each other to some degree.

For example, we mixed Composite Pattern with Decorator Pattern, Adapter Pattern with Proxy Pattern, compared Adapter Pattern and Bridge Pattern. During the journey of exploring, we sometimes found it was just a natural result to have our code end in a pattern that's similar to what we've listed if we took writing *better code* into consideration.

Taking Adapter Pattern and Bridge Pattern as an example, when we are trying to make two classes cooperate, it comes out with Adapter Pattern and when we are planning on connecting with different classes in advance, it goes with Bridge Pattern. There are no actual lines between each pattern and the applications of those patterns, though the techniques behind patterns could usually be useful.

In the next chapter, we are going to talk about behavioral patterns that help to form algorithms and assign the responsibilities.

5

Behavioral Design Patterns

As the name suggests, behavioral design patterns are patterns about how objects or classes interact with each other. The implementation of behavioral design patterns usually requires certain data structures to support the interaction in a system. However, behavioral patterns and structural patterns focus on different aspects when applied. As a result, you might find patterns in the category of behavioral design patterns usually have simpler or more straightforward structures compared to structural design patterns.

In this chapter, we are going to talk about some of the following common behavioral patterns:

- **Chain of Responsibility:** Organizes behaviors with different scopes
- **Command:** Exposes commands from the internal with encapsulated context
- **Memento:** Provides an approach for managing states outside of their owners without exposing detailed implementations
- **Iterator:** Provides a universal interface for traversing
- **Mediator:** It groups coupling and logically related objects and makes interconnections cleaner in a system that manages many objects

Chain of Responsibility Pattern

There are many scenarios under which we might want to apply certain actions that can fall back from a detailed scope to a more general one.

A nice example would be the help information of a GUI application: when a user requests help information for a certain part of the user interface, it is expected to show information as specific as possible. This can be done with different implementations, and the most intuitive one for a web developer could be events bubbling.

Consider a DOM structure like this:

```
<div class="outer">
  <div class="inner">
    <span class="origin"></span>
  </div>
</div>
```

If a user clicks on the `span.origin` element, a `click` event would start bubbling from the `span` element to the document root (if `useCapture` is `false`):

```
$('.origin').click(event => {
  console.log('Click on `span.origin`.');
});

$('.outer').click(event => {
  console.log('Click on `div.outer`.');
});
```

By default, it will trigger both event listeners added in the preceding code. To stop the propagation as soon as an event gets handled, we can call its `stopPropagation` method:

```
$('.origin').click(event => {
  console.log('Click on `span.origin`.');
  event.stopPropagation();
});

$('.outer').click(event => {
  console.log('Click on `div.outer`.');
});
```

Though a `click` event is not exactly the same as the help information request, with the support of custom events, it's quite easy to handle help information with necessary detailed or general information in the same chain.

Another important implementation of the Chain of Responsibility Pattern is related to error handling. A primitive example for this could be using `try...catch`. Consider code like this: we have three functions: `foo`, `bar`, and `biu`, `foo` is called by `bar` while `bar` is called by `biu`:

```
function foo() {
  // throw some errors.
}

function bar() {
  foo();
}
```

```
function biu() {
  bar();
}

biu();
```

Inside both functions `bar` and `biu`, we can do some error catching. Assuming function `foo` throws two kinds of errors:

```
function foo() {
  let value = Math.random();

  if (value < 0.5) {
    throw new Error('Awesome error');
  } else if (value < 0.8) {
    throw new TypeError('Awesome type error');
  }
}
```

In function `bar` we would like to handle the `TypeError` and leave other errors throwing on:

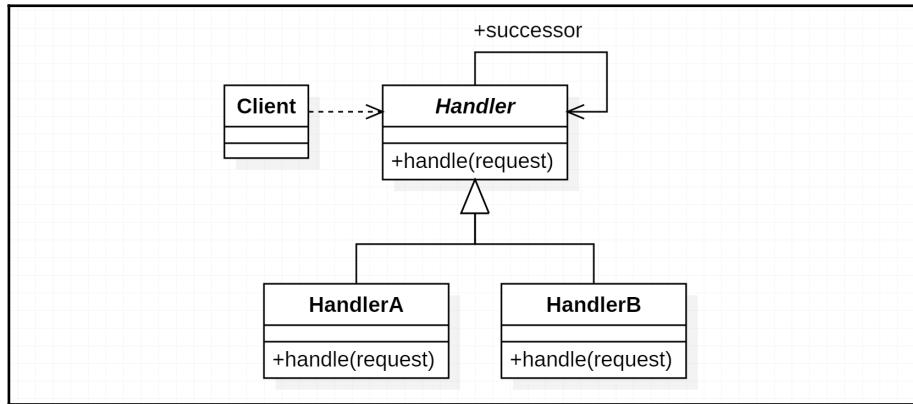
```
function bar() {
  try {
    foo();
  } catch (error) {
    if (error instanceof TypeError) {
      console.log('Some type error occurs', error);
    } else {
      throw error;
    }
  }
}
```

And in function `biu`, we would like to add more general handling that catches all the errors so that the program will not crash:

```
function biu() {
  try {
    bar();
  } catch (error) {
    console.log('Some error occurs', error);
  }
}
```

So by using `try...catch` statements, you may have been using the Chain of Responsibility Pattern constantly without paying any attention to it. Just like you may have been using other well-known design patterns all the time.

If we abstract the structure of Chain of Responsibility Pattern into objects, we could have something as illustrated in the figure:



Participants

The participants of the Chain of Responsibility Pattern include:

- **Handler:** Defines the interface of the handler with successor and method to handle requests. This is done implicitly with classes like `EventEmitter` and `try...catch` syntax.
- **Concrete handler:** `EventListener`, `catch` block and `HandlerA/HandlerB` in the class version. Defines handlers in the form of callbacks, code blocks and classes that handle requests.
- **Client:** Initiates the requests that go through the chain.

Pattern scope

The Chain of Responsibility Pattern itself could be applied to many different scopes in a program. It requires a multi-level chain to work, but this chain could be in different forms. We've been playing with events as well as `try...catch` statements that have structural levels, this pattern could also be applied to scenarios that have logical levels.

Consider objects marked with different scopes using string:

```
let objectA = {  
    scope: 'user.installation.package'  
};  
  
let objectB = {  
    scope: 'user.installation'  
};
```

Now we have two objects with related scopes specified by string, and by adding filters to these scope strings, we can apply operations from specific ones to general ones.

Implementation

In this part, we are going to implement the class version we've mentioned at the end of the introduction to the Chain of Responsibility Pattern. Consider requests that could either ask for help information or feedback prompts:

```
type RequestType = 'help' | 'feedback';  
  
interface Request {  
    type: RequestType;  
}
```

 We are using string literal type here with union type. It is a pretty useful feature provided in TypeScript that plays well with existing JavaScript coding styles.

See the following link for more information: <http://www.typescriptlang.org/docs/handbook/advanced-types.html>.

One of the key processes for this pattern is going through the handlers' chain and finding out the most specific handler that's available for the request. There are several ways to achieve this: by recursively invoking the `handle` method of a successor, or having a separate logic walking through the handler successor chain until the request is confirmed as handled.

The logic walking through the chain in the second way requires the acknowledgment of whether a request has been properly handled. This can be done either by a state indicator on the request object or by the return value of the `handle` method.

We'll go with the recursive implementation in this part. Firstly, we want the default handling behavior of a handler to be forwarding requests to its successor if any:

```
class Handler {  
    private successor: Handler;  
  
    handle(request: Request): void {  
        if (this.successor) {  
            this.successor.handle(request);  
        }  
    }  
}
```

And now for `HelpHandler`, it handles help requests but forwards others:

```
class HelpHandler extends Handler {  
    handle(request: Request): void {  
        if (request.type === 'help') {  
            // Show help information.  
        } else {  
            super.handle(request);  
        }  
    }  
}
```

The code for `FeedbackHandler` is similar:

```
class FeedbackHandler extends Handler {  
    handle(request: Request): void {  
        if (request.type === 'feedback') {  
            // Prompt for feedback.  
        } else {  
            super.handle(request);  
        }  
    }  
}
```

Thus, a chain of handlers could be made up in some way. And if a request got in this chain, it would be passed on until a handler recognizes and handles it. However, it is not necessary to have all requests *handled* after processing them. The handlers can always pass a request on whether this request gets processed by this handler or not.

Consequences

The Chain of Responsibility Pattern decouples the connection between objects that issue the requests and logic that handles those requests. The sender assumes that its requests could, but not necessarily, be properly handled without knowing the details. For some implementations, it is also very easy to add new responsibilities to a specific handler on the chain. This provides notable flexibility for handling requests.

Besides the examples we've been talking about, there is another important mutation of `try...catch` that can be treated in the Chain of Responsibility Pattern – Promise. Within a smaller scope, the chain could be represented as:

```
promise
  .catch(TypeError, reason => {
    // handles TypeError.
  })
  .catch(ReferenceError, reason => {
    // handles ReferenceError.
  })
  .catch(reason => {
    // handles other errors.
  });
}
```



The standard `catch` method on an ES Promise object does not provide the overload that accepts an error type as a parameter, but many implementations do.

In a larger scope, this chain would usually appear when the code is playing with third-party libraries. A common usage would be converting errors produced by other libraries to errors known to the current project. We'll talk more about error handling of asynchronous code later in this book.

Command Pattern

Command Pattern involves encapsulating operations as executable commands and could either be in the form of objects or functions in JavaScript. It is common that we may want to make operations rely on certain context and states that are not accessible for the invokers. By storing those pieces of information with a command and passing it out, this situation could be properly handled.

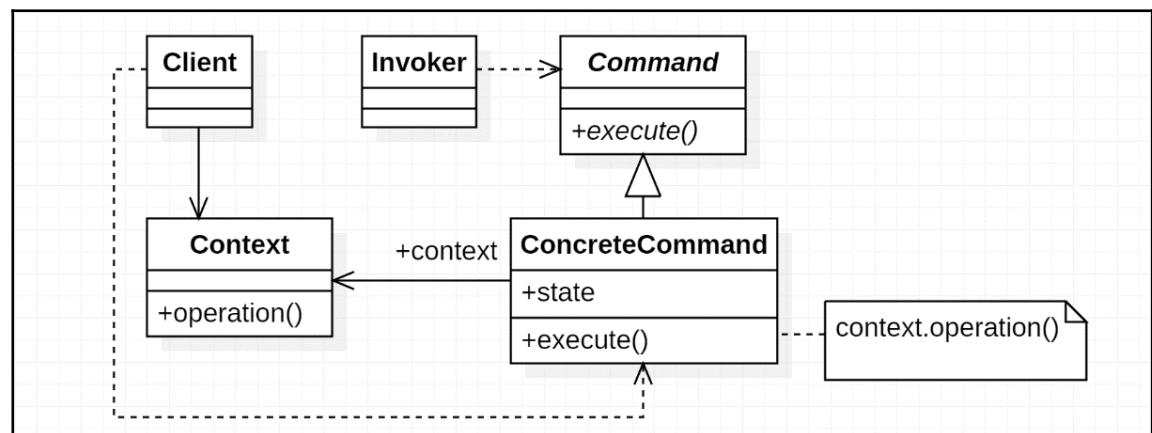
Consider an extremely simple example: we want to provide a function called `wait`, which returns a `cancel` handler:

```
function wait() {  
  let $layer = $('.wait-layer');  
  $layer.show();  
  return () => {  
    $layer.hide();  
  };  
}  
  
let cancel = wait();  
  
setTimeout(() => cancel(), 1000);
```

The `cancel` handler in the preceding code is just a command we were talking about. It stores the context (`$layer`) using closure and is passed out as the return value of function `wait`.

Closure in JavaScript provides a really simple way to store command context and states, however, the direct disadvantage would be compromised flexibility between context/states and command functions because closure is lexically determined and cannot be changed at runtime. This would be okay if the command is only expected to be invoked with fixed context and states, but for more complex situations, we might need to construct them as objects with a proper data structure.

The following diagram shows the overall relations between participants of Command Pattern:



By properly splitting apart context and states with the command object, Command Pattern could also play well with Flyweight Pattern if you wanted to reuse command objects multiple times.

Other common extensions based on Command Pattern include undo support and macros with multiple commands. We are going to play with those later in the implementation part.

Participants

The participants of Command Pattern include:

- **Command:** Defines the general interface of commands passing around, it could be a function signature if the commands are in the form of functions.
- **Concrete command:** Defines the specific behaviors and related data structure. It could also be a function that matches the signature declared as `Command`. The `cancel` handler in the very first example is a concrete command.
- **Context:** The context or receiver that the command is associated with. In the first example, it is the `$layer`.
- **Client:** Creates concrete commands and their contexts.
- **Invoker:** Executes concrete commands.

Pattern scope

Command Pattern suggests two separate parts in a single application or a larger system: *client* and *invoker*. In the simplified example `wait` and `cancel`, it could be hard to distinguish the difference between those parts. But the line is clear: *client* knows or controls the context of commands to be executed with, while *invoker* does not have access or does not need to care about that information.

The key to the Command Pattern is the separation and bridging between those two parts through commands that store context and states.

Implementation

It's common for an editor to expose commands for third-party extensions to modify the text content. Consider a `TextContext` that contains information about the text file being edited and an abstract `TextCommand` class associated with that context:

```
class TextContext {  
    content = 'text content';  
}  
  
abstract class TextCommand {  
    constructor(  
        public context: TextContext  
    ) {}  
  
    abstract execute(...args: any[]): void;  
}
```

Certainly, `TextContext` could contain much more information like language, encoding, and so on. You can add them in your own implementation for more functionality. Now we are going to create two commands: `ReplaceCommand` and `InsertCommand`.

```
class ReplaceCommand extends TextCommand {  
    execute(index: number, length: number, text: string): void {  
        let content = this.context.content;  
  
        this.context.content =  
            content.substr(0, index) +  
            text +  
            content.substr(index + length);  
    }  
}  
  
class InsertCommand extends TextCommand {  
    execute(index: number, text: string): void {  
        let content = this.context.content;  
  
        this.context.content =  
            content.substr(0, index) +  
            text +  
            content.substr(index);  
    }  
}
```

Those two commands share similar logic and actually `InsertCommand` can be treated as a subset of `ReplaceCommand`. Or if we have a new delete command, then replace command can be treated as the combination of delete and insert commands.

Now let's assemble those commands with the client and invoker:

```
class Client {  
    private context = new TextContext();  
  
    replaceCommand = new ReplaceCommand(this.context);  
    insertCommand = new InsertCommand(this.context);  
}  
  
let client = new Client();  
  
$('.replace-button').click(() => {  
    client.replaceCommand.execute(0, 4, 'the');  
});  
  
$('.insert-button').click(() => {  
    client.insertCommand.execute(0, 'awesome');  
});
```

If we go further, we can actually have a command that executes other commands. Namely, we can have macro commands. Though the preceding example alone does not make it necessary to create a macro command, there would be scenarios where macro commands help. As those commands are already associated with their contexts, a macro command usually does not need to have an explicit context:

```
interface TextCommandInfo {  
    command: TextCommand,  
    args: any[];  
}  
  
class MacroTextCommand {  
    constructor(  
        public infos: TextCommandInfo[]  
    ) {}  
  
    execute(): void {  
        for (let info of this.infos) {  
            info.command.execute(...info.args);  
        }  
    }  
}
```

Consequences

Command Pattern decouples the client (who knows or controls context) and the invoker (who has no access to or does not care about detailed context).

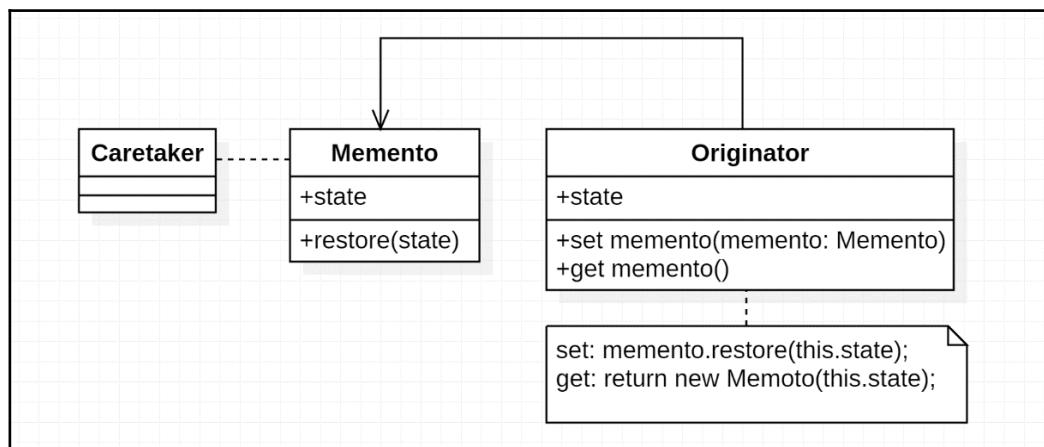
It plays well with Composite Pattern. Consider the example of macro commands we mentioned above: a macro command can have other macro commands as its components, thus we make it a composite command.

Another important case of Command Pattern is adding support for undo operations. A direct approach is to add the `undo` method to every command. When an undo operation is requested, invoke the `undo` method of commands in reverse order, and we can pray that every command would be undone correctly. However, this approach relies heavily on a flawless implementation of the `undo` method as every mistake will accumulate. To implement more stable undo support, redundant information or snapshots could be stored.

Memento Pattern

We've talked about an undo support implementation in the previous section on the Command Pattern, and found it was not easy to implement the mechanism purely based on reversing all the operations. However, if we take snapshots of objects as their history, we may manage to avoid accumulating mistakes and make the system more stable. But then we have a problem: we need to store the states of objects while the states are encapsulated with objects themselves.

Memento Pattern helps in this situation. While a memento carries the state of an object at a certain time point, it also controls the process of setting the state back to an object. This makes the internal state implementation hidden from the undo mechanism in the following example:



We have the instances of the memento controlling the state restoration in the preceding structure. It can also be controlled by the caretaker, namely the undo mechanism, for simple state restoring cases.

Participants

The participants of Memento Pattern include:

- **Memento:** Stores the state of an object and defines method `restore` or other APIs for restoring the states to specific objects
- **Originator:** Deals with objects that need to have their internal states stored
- **Caretaker:** Manages mementos without intervening with what's inside

Pattern scope

Memento Pattern mainly does two things: it prevents the caretaker from knowing the internal state implementation and decouples the state retrieving and restoring process from states managed by the `Caretaker` or `Originator`.

When the state retrieving and restoring processes are simple, having separated mementos does not help much if you are already keeping the decoupling idea in mind.

Implementation

Start with an empty `State` interface and `Memento` class. As we do not want `Caretaker` to know the details about state inside an `Originator` or `Memento`, we would like to make `state` property of `Memento` private. Having restoration logic inside `Memento` does also help with this, and thus we need method `restore`. So that we don't need to expose a public interface for reading state inside a memento.

And as an object assignment in JavaScript assigns only its reference, we would like to do a quick copy for the states (assuming state objects are single-level):

```
interface State { }

class Memento {
    private state: State;

    constructor(state: State) {
        this.state = Object.assign({} as State, state);
```

```
    }

    restore(state: State): void {
        Object.assign(state, this.state);
    }
}
```

For Originator we use a getter and a setter for creating and restoring specific mementos:

```
class Originator {
    state: State;

    get memento(): Memento {
        return new Memento(this.state);
    }

    set memento(memento: Memento) {
        memento.restore(this.state);
    }
}
```

Now the Caretaker would manage the history accumulated with mementos:

```
class Caretaker {
    originator: Originator;
    history: Memento[] = [];

    save(): void {
        this.history.push(this.originator.memento);
    }

    restore(): void {
        this.originator.memento = this.history.shift();
    }
}
```

In some implementations of Memento Pattern, a `getState` method is provided for instances of `Originator` to read state from a memento. But to prevent classes other than `Originator` from accessing the `state` property, it may rely on language features like a *friend modifier* to restrict the access (which is not yet available in TypeScript).

Consequences

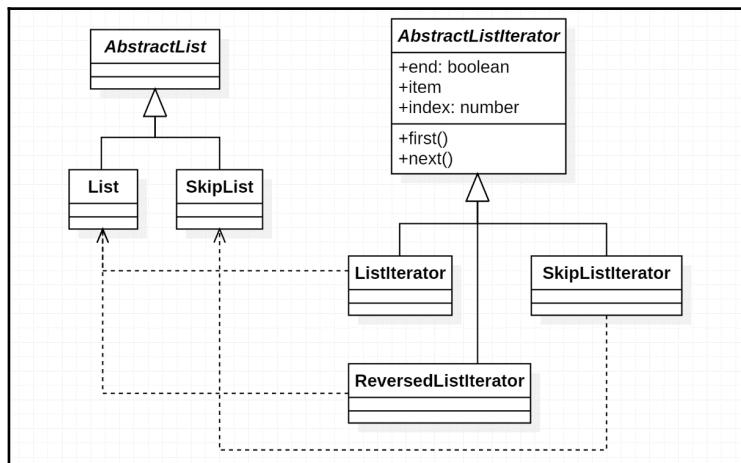
Memento Pattern makes it easier for a caretaker to manage the states of originators and it becomes possible to extend state retrieving and restoring. However, a perfect implementation that seals everything might rely on language features as we've mentioned before. Using mementos could also bring a performance cost as they usually contain redundant information in trade of stability.

Iterator Pattern

Iterator Pattern provides a universal interface for accessing internal elements of an aggregate without exposing the underlying data structure. A typical iterator contains the following methods or getters:

- `first()`: moves the cursor to the *first* element in the aggregates
- `next()`: moves the cursor to the *next* element
- `end`: a getter that returns a Boolean indicates whether the cursor is at the end
- `item`: a getter that returns the element at the position of the current cursor
- `index`: a getter that returns the index of the element at the current cursor

Iterators for aggregates with different interfaces or underlying structures usually end with different implementations as shown in the following figure:



Though the client does not have to worry about the structure of an aggregate, an iterator would certainly need to. Assuming we have everything we need to build an iterator, there could be a variety of ways for creating one. The factory method is widely used when creating iterators, or a *factory getter* if no parameter is required.

Starting with ES6, syntax sugar `for...of` is added and works for all objects with property `Symbol.iterator`. This makes it even easier and more comfortable for developers to work with customized lists and other classes that can be iterated.

Participants

The participants of Iterator Pattern include:

- **Iterator:** `AbstractListIterator`

Defines the universal iterator interface that is going to transverse different aggregates.

- **Concrete iterator:** `ListIterator`, `SkipListIterator` and `ReversedListIterator`

Implements specific iterator that transverses and keeps track of a specific aggregate.

- **Aggregate:** `AbstractList`

Defines a basic interface of aggregates that iterators are going to work with.

- **Concrete aggregate:** `List` and `SkipList`

Defines the data structure and factory method/getter for creating associated iterators.

Pattern scope

Iterator Pattern provides a unified interface for traversing aggregates. In a system that doesn't rely on iterators, the main functionality provided by iterators could be easily taken over by simple helpers. However, the reusability of those helpers could be reduced as the system grows.

Implementation

In this part, we are going to implement a straightforward array iterator, as well as an ES6 iterator.

Simple array iterator

Let's start by creating an iterator for a JavaScript array, which should be extremely easy.

Firstly, the universal interface:

```
interface Iterator<T> {  
    first(): void;  
    next(): void;  
    end: boolean;  
    item: T;  
    index: number;  
}
```



Please notice that the TypeScript declaration for ES6 has already declared an interface called `Iterator`. Consider putting the code in this part into a namespace or module to avoid conflicts.

And the implementation of a simple array iterator could be:

```
class ArrayIterator<T> implements Iterator<T> {  
    index = 0;  
  
    constructor(  
        public array: T[]  
    ) { }  
  
    first(): void {  
        this.index = 0;  
    }  
  
    next(): void {  
        this.index++;  
    }  
  
    get end(): boolean {  
        return this.index >= this.array.length;  
    }  
  
    get item(): T {  
        return this.array[this.index];  
    }  
}
```

```
}
```

Now we need to extend the prototype of native `Array` to add an `iterator` getter:

```
Object.defineProperty(Array.prototype, 'iterator', {
  get() {
    return new ArrayIterator(this);
  }
});
```

To make `iterator` a valid property of the `Array` instance, we also need to extend the interface of `Array`:

```
interface Array<T> {
  iterator: IteratorPattern.Iterator<T>;
}
```



This should be written outside the namespace under the global scope. Or if you are in a module or ambient module, you might want to try `declare global { ... }` for adding new properties to existing global interfaces.

ES6 iterator

ES6 provides syntax sugar for `...of` and other helpers for *iterable* objects, namely the objects that have implemented the `Iterable` interface of the following:

```
interface IteratorResult<T> {
  done: boolean;
  value: T;
}

interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}

interface Iterable<T> {
  [Symbol.iterator](): Iterator<T>;
}
```

Assume we have a class with the following structure:

```
class SomeData<T> {
    array: T[];
}
```

And we would like to make it iterable. More specifically, we would like to make it iterates reversely. As the `Iterable` interface suggests, we just need to add a method with a special name `Symbol.iterator` for creating an `Iterator`. Let's call the iterator `SomeIterator`:

```
class SomeIterator<T> implements Iterator<T> {
    index: number;

    constructor(
        public array: T[]
    ) {
        this.index = array.length - 1;
    }

    next(): IteratorResult<T> {
        if (this.index <= this.array.length) {
            return {
                value: undefined,
                done: true
            };
        } else {
            return {
                value: this.array[this.index--],
                done: false
            }
        }
    }
}
```

And then define the `iterator` method:

```
class SomeData<T> {
    array: T[];

    [Symbol.iterator]() {
        return new SomeIterator<T>(this.array);
    }
}
```

Now we would have `SomeData` that works with `for...of`.



Iterators also play well with generators; see the following link for more examples: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols.

Consequences

Iterator Pattern decouples iteration usage from the data structure that is being iterated. The direct benefit of this is enabling an interchangeable data class that may have completely different internal structures, like an array and binary tree. Also, one data structure can be iterated via different iterators with different traversal mechanisms and results in different orders and efficiencies.

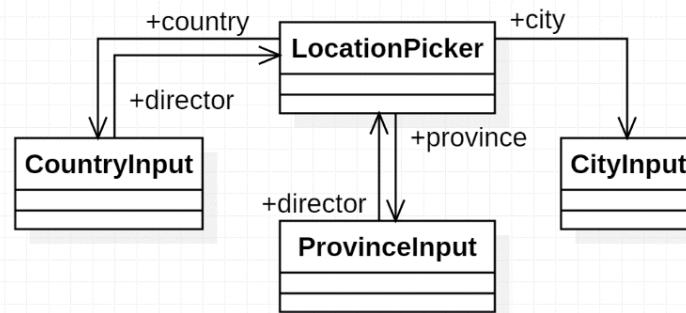
A unified iterator interface in one system could also help the developer from being confused when facing different aggregates. As we mentioned previously, some language like your beloved JavaScript provides a language level abstraction for iterators and makes life even easier.

Mediator Pattern

The connections between UI components and related objects could be extremely complex. Object-oriented programming distributes functionalities among objects. This makes coding easier with cleaner and more intuitive logic; however, it does not ensure the reusability and sometimes makes it difficult to understand if you look at the code again after some days (you may still understand every single operation but would be confused about the interconnections if the network becomes really intricate).

Consider a page for editing user profile. There are standalone inputs like nickname and tagline, as well as inputs that are related to each other. Taking location selection for example, there could easily be a tree-level location and the options available in lower levels are determined by the selection of higher levels. However, if those objects are managed directly by a single huge controller, it will result in a page that has limited reusability. The code formed under this situation would also tend to have a hierarchy that's less clean for people to understand.

Mediator Pattern tries to solve this problem by separating coupling elements and objects as groups, and adding a *director* between a group of elements and other objects as shown in the following figure:



Those objects form a mediator with their colleagues that can interact with other objects as a single object. With proper encapsulation, the mediator will have better reusability as it has just the right size and properly divided functionality. In the world of web front end development, there are concepts or implementations that fit Mediator Pattern well, like *Web Component* and *React*.

Participants

The participants of Mediator Pattern include:

- **Mediator:**

Usually, the abstraction or skeleton predefined by a framework. Defines the interface that colleagues in a mediator communicate through.

- **Concrete mediator:** LocationPicker

Manages the colleagues and makes them cooperate, providing a higher level interface for objects outside.

- **Colleague classes:** CountryInput, ProvinceInput, CityInput

Defines references to their mediator and notifies its changes to the mediator and accepts modifications issued by the mediator.

Pattern scope

Mediator Pattern could connect many parts of a project, but does not have direct or enormous impact on the outline. Most of the credit is given because of increased usability and cleaner interconnections introduced by mediators. However, along with a nice overall architecture, Mediator Pattern can help a lot with refined code quality, and make the project easier to maintain.

Implementation

Using libraries like React would make it very easy to implement Mediator Pattern, but for now, we are going with a relatively primitive way and handle changes by hand. Let's think about the result we want for a `LocationPicker` we've discussed, and hopefully, it includes country, province and city fields:

```
interface LocationResult {  
    country: string;  
    province: string;  
    city: string;  
}
```

And now we can sketch the overall structure of class `LocationPicker`:

```
class LocationPicker {  
    $country = $(document.createElement('select'));  
    $province = $(document.createElement('select'));  
    $city = $(document.createElement('select'));  
  
    $element = $(document.createElement('div'))  
        .append(this.$country)  
        .append(this.$province)  
        .append(this.$city);  
  
    get value(): LocationResult {  
        return {  
            country: this.$country.val(),  
            province: this.$province.val(),  
            city: this.$city.val()  
        };  
    }  
}
```

Before we can tell how the colleagues are going to cooperate, we would like to add a helper method `setOptions` for updating options in a `select` element:

```
private static setOptions(  
    $select: JQuery,  
    values: string[]  
): void {  
    $select.empty();  
  
    let $options = values.map(value => {  
        return $(document.createElement('option'))  
            .text(value)  
            .val(value);  
    });  
  
    $select.append($options);  
}
```

I personally tend to have methods that do not depend on a specific instance static methods and this applies to methods `getCountries`, `getProvincesByCountry`, and `getCitiesByCountryAndProvince` that simply return a list by the information given as function arguments (though we are not going to actually implement that part):

```
private static getCountries(): string[] {  
    return ['-'].concat([/* countries */]);  
}  
  
private static getProvincesByCountry(country: string): string[] {  
    return ['-'].concat([/* provinces */]);  
}  
  
private static getCitiesByCountryAndProvince(  
    country: string,  
    province: string  
): string[] {  
    return ['-'].concat([/* cities */]);  
}
```

Now we may add methods for updating options in the `select` elements:

```
updateProvinceOptions(): void {  
    let country: string = this.$country.val();  
  
    let provinces = LocationPicker.getProvincesByCountry(country);  
    LocationPicker.setOptions(this.$province, provinces);  
  
    this.$city.val('-');
```

```
}
```

```
updateCityOptions(): void {
  let country: string = this.$country.val();
  let province: string = this.$province.val();

  let cities = LocationPicker
    .getCitiesByCountryAndProvince(country, province);
  LocationPicker.setOptions(this.$city, cities);
}
```

Finally, weave those colleagues together and add listeners to the change events:

```
constructor() {
  LocationPicker
    .setOptions(this.$country, LocationPicker.getCountries());
  LocationPicker.setOptions(this.$province, ['-']);
  LocationPicker.setOptions(this.$city, ['-']);

  this.$country.change(() => {
    this.updateProvinceOptions();
  });

  this.$province.change(() => {
    this.updateCityOptions();
  });
}
```

Consequences

Mediator Pattern, like many other design patterns, downgrades a level-100 problem into two level-10 problems and solves them separately. A well-designed mediator usually has a proper size and usually tends to be reused in the future. For example, we might not want to put nickname input together with the country, province, and city inputs as this combination doesn't tend to occur in other situations (which means they are not strongly related).

As the project evolves, a mediator may grow to a size that's not efficient anymore. So a properly designed mediator should also take the dimension of time into consideration.

Summary

In this chapter, we talked about some common behavioral patterns for different scopes and different scenarios. Chain of Responsibility Pattern and Command Pattern can apply to a relatively wide range of scopes, while other patterns mentioned in this chapter usually care more about the scope with objects and classes directly related.

Behavioral patterns we've talked about in this chapter are less like each other compared to creational patterns and structural patterns we previously walked through. Some of the behavioral patterns could compete with others, but many of them could cooperate. For example, we talked about Command Pattern with Memento Pattern to implement undo support. Many others may cooperate in parallel and do their own part.

In the next chapter, we'll continue talking about other behavioral design patterns that are useful and widely used.

6

Behavioral Design Patterns: Continuous

In the previous chapter, we've already talked about some of the behavioral design patterns. We'll be continuing with more patterns in this category in this chapter, including: Strategy Pattern, State Pattern, Template Method Pattern, Observer Pattern, and Visitor Pattern.

Many of these patterns share the same idea: unify the shape and vary the details. Here is a quick overview:

- **Strategy Pattern** and **Template Pattern**: Defines the same outline of algorithms
- **State Pattern**: Provides different behavior for objects in different states with the same interface
- **Observer Pattern**: Provides a unified process of handling subject changes and notifying observers
- **Visitor Pattern**: Does similar jobs as Strategy Pattern sometimes, but avoids an over complex interface that might be required for Strategy Pattern to handle objects in many different types

Patterns that will be discussed in this chapter could be applied in different scopes just as many patterns in other categories.

Strategy Pattern

It's common that a program has similar outlines for processing different targets with different detailed algorithms. Strategy Pattern encapsulates those algorithms and makes them interchangeable within the shared outline.

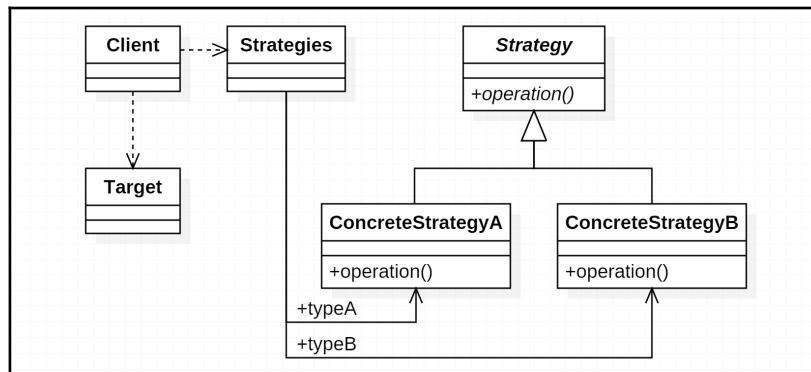
Consider conflicting merging processes of data synchronization, which we talked about in Chapter 2, *The Challenge of Increasing Complexity*. Before refactoring, the code was like this:

```
if (type === 'value') {  
    // ...  
} else if (type === 'increment') {  
    // ...  
} else if (type === 'set') {  
    // ...  
}
```

But later we found out that we could actually extract the same outlines from different phases of the synchronization process, and encapsulate them as different strategies. After refactoring, the outline of the code became as follows:

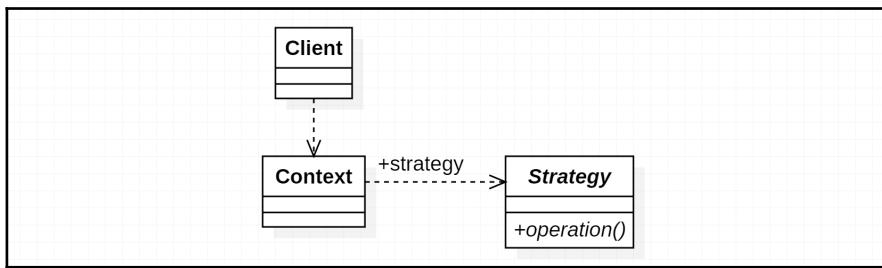
```
let strategy = strategies[type];  
strategy.operation();
```

We get a lot of ways to compose and organize those strategy objects or classes sometimes in JavaScript. A possible structure for Strategy Pattern could be:



In this structure, the client is responsible for fetching specific strategies from the table and applying operations of the current phase.

Another structure is using contextual objects and letting them control their own strategies:



Thus the client needs only to link a specific context with the corresponding strategy.

Participants

We've mentioned two possible structures for Strategy Pattern, so let's discuss the participants separately. For the first structure, the participants include the following:

- **Strategy**

Defines the interface of strategy objects or classes.

- **Concrete strategy:** `ConcreteStrategyA` and `ConcreteStrategyB`

Implements concrete strategy operations defined by the `Strategy` interface.

- **Strategy manager:** `Strategies`

Defines a data structure to manage strategy objects. In the example, it's just a simple hash table that uses data type names as keys and strategy objects as values. It could be more complex on demand: for example, with matching patterns or conditions.

- **Target**

The target to apply algorithms defined in strategy objects.

- **Client**

Makes targets and strategies cooperate.

The participants of the second structure include the following:

- **Strategy and concrete strategy**

The same as in the preceding section.

- **Context**

Defines a reference to the strategy object applied. Provides related methods or property getters for clients to operate.

- **Client**

Manages context objects.

Pattern scope

Strategy Pattern is usually applied to scopes with small or medium sizes. It provides a way to encapsulate algorithms and makes those algorithms easier to manage under the same outline. Strategy Pattern can also be the core of an entire solution sometimes, and a good example is the synchronization implementation we've been playing with. In this case, Strategy Pattern builds the bridge of plugins and makes the system extendable. But most of the time, the fundamental work done by Strategy Pattern is decoupling concrete strategies, contexts, or targets.

Implementation

The implementation starts with defining the interfaces of objects we'll be playing with. We have two target types in string literal type 'a' and 'b'. Targets of type 'a' have a `result` property with type `string`, while targets of type 'b' have a `value` property with type `number`.

The interfaces we'll have look, are like:

```
type TargetType = 'a' | 'b';

interface Target {
    type: TargetType;
}

interface TargetA extends Target {
    type: 'a';
}
```

```
    result: string;
}

interface TargetB extends Target {
  type: 'b';
  value: number;
}

interface Strategy<TTarget extends Target> {
  operationX(target: TTarget): void;
  operationY(target: TTarget): void;
}
```

Now we'll define the concrete strategy objects without a constructor:

```
let strategyA: Strategy<TargetA> = {
  operationX(target) {
    target.result = target.result + target.result;
  },
  operationY(target) {
    target.result = target
      .result
      .substr(Math.floor(target.result.length / 2));
  }
};

let strategyB: Strategy<TargetB> = {
  operationX(target) {
    target.value = target.value * 2;
  },
  operationY(target) {
    target.value = Math.floor(target.value / 2);
  }
};
```

To make it easier for a client to fetch those strategies, we'll put them into a hash table:

```
let strategies: {
  [type: string]: Strategy<Target>
} = {
  a: strategyA,
  b: strategyB
};
```

And now we can make them work with targets in different types:

```
let targets: Target[] = [
  { type: 'a' },
  { type: 'a' },
  { type: 'b' }
];

for (let target of targets) {
  let strategy = strategies[target.type];

  strategy.operationX(target);
  strategy.operationY(target);
}
```

Consequences

Strategy Pattern makes the foreseeable addition of algorithms for contexts or targets under new categories easier. It also makes the outline of a process even cleaner by hiding trivial branches of behaviors selection.

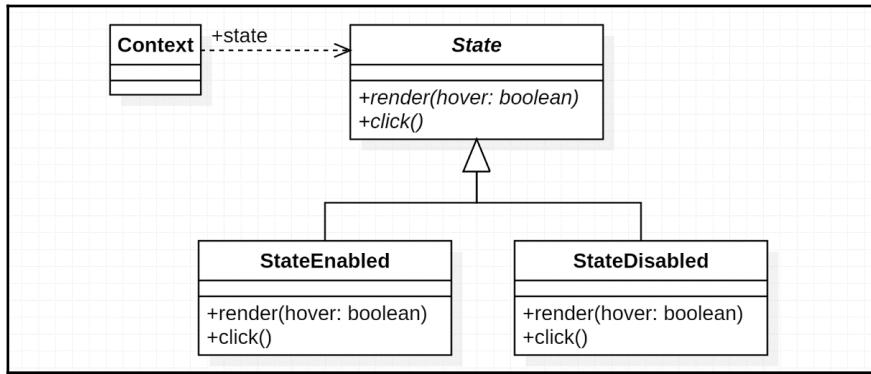
However, the abstraction of algorithms defined by the `Strategy` interface may keep growing while we are trying to add more strategies and satisfy their requirements of parameters. This could be a problem for a Strategy Pattern with clients that are managing targets and strategies. But for the other structures which the references of strategy objects are stored by contexts themselves, we can manage to trade-off the interchangeability. This would result in Visitor Pattern, which we are going to talk about later in this chapter.

And as we've mentioned before, Strategy Pattern can also provide notable extensibility if an extendable strategy manager is available or the client of contexts is designed to.

State Pattern

It's possible for some objects to behave completely differently when they are in different states. Let's think about an easy example first. Consider rendering and interacting with a custom button in two states: enabled and disabled. When the button is enabled, it lights up and changes its style to active on a mouse hover, and of course, it handles clicks; when disabled, it dims and no longer cares about mouse events.

We may think of an abstraction with two operations: `render` (with a parameter that indicates whether the mouse is hovering) and `click`; along with two states: *enabled* and *disabled*. We can even divide deeper and have state *active*, but that won't be necessary in our case.



And now we can have `StateEnabled` with both `render` and `click` methods implemented, while having `StateDisabled` with only `render` method implemented because it does not care about the `hover` parameter. In this example, we are expecting every method of the states being callable. So we can have the abstract class `State` with empty `render` and `click` methods.

Participants

The participants of State Pattern include the following:

- **State**

Defines the interface of state objects that are being switched to internally.

- **Concrete state:** `StateEnabled` and `StateDisabled`

Implements the `State` interface with behavior corresponding to a specific state of the context. May have an optional reference back to its context.

- **Context**

Manages references to different states, and makes operations defined on the active one.

Pattern scope

State Pattern usually applies to the code of scopes with the size of a feature. It does not specify whom to transfer the state of context: it could be either the context itself, the state methods, or code that controls context.

Implementation

Start with the `State` interface (it could also be an abstract class if there are operations or logic to share):

```
interface State {  
    render(hover: boolean): void;  
    click(): void;  
}
```

With the `State` interface defined, we can move to `Context` and sketch its outline:

```
class Context {  
    $element: JQuery;  
  
    state: State;  
  
    private render(hover: boolean): void {  
        this.state.render(hover);  
    }  
  
    private click(): void {  
        this.state.click();  
    }  
    onclick(): void {  
        console.log('I am clicked.');//  
    }  
}
```

Now we are going to have the two states, `StateEnabled` and `StateDisabled` implemented. First, let's address `StateEnabled`, it cares about hover status and handles click event:

```
class StateEnabled implements State {  
    constructor(  
        public context: Context  
    ) {}  
  
    render(hover: boolean): void {
```

```

    this
      .context
      .$element
      .removeClass('disabled')
      .toggleClass('hover', hover);
}

click(): void {
  this.context.onclick();
}
}

```

Next, for `StateDisabled` it just ignores `hover` parameter and does nothing when `click` event emits:

```

class StateDisabled implements State {
  constructor(
    public context: Context
  ) { }

  render(): void {
    this
      .context
      .$element
      .addClass('disabled')
      .removeClass('hover');
  }

  click(): void {
    // Do nothing.
  }
}

```

Now we have classes of states `enabled` and `disabled` ready. As the instances of those classes are associated with the context, we need to initialize every state when a new `Context` is initiated:

```

class Context {
  ...

  private stateEnabled = new StateEnabled(this);
  private stateDisabled = new StateDisabled(this);

  state: State = this.stateEnabled;
  ...
}

```

It is possible to use flyweights by passing context in when invoking every operation on the active state as well.

Now let's finish the Context by listening to and forwarding proper events:

```
constructor() {
  this
  .$element
  .hover(
    () => this.render(true),
    () => this.render(false)
  )
  .click(() => this.click());
}

this.render(false);
}
```

Consequences

State Pattern reduces conditional branches in potentially multiple methods of context objects. As a trade-off, extra state objects are introduced, though it usually won't be a big problem.

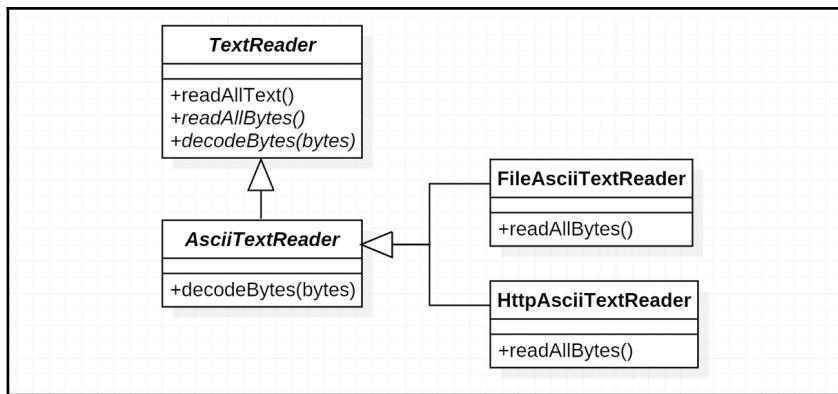
The context object in State Pattern usually delegates operations and forwards them to the current state object. Thus operations defined by a concrete state may have access to the context itself. This makes reusing state objects possible with flyweights.

Template Method Pattern

When we are talking about subclassing or inheriting, the building is usually built from the bottom up. Subclasses inherit the basis and then provide more. However, it could be useful to reverse the structure sometimes as well.

Consider Strategy Pattern which defines the outline of a process and has interchangeable algorithms as strategies. If we apply this structure under the hierarchy of classes, we will have Template Method Pattern.

A template method is an abstract method (optionally with default implementation) and acts as a placeholder under the outline of a larger process. Subclasses override or implement related methods to modify or complete the behaviors. Imaging the skeleton of a `TextReader`, we are expecting its subclasses to handle text files from different storage media, detect different encodings and read all the text. We may consider a structure like this:



The `TextReader` in this example has a method `readAllText` that reads all text from a resource by two steps: reading all bytes from the resource (`readAllBytes`), and then decoding those bytes with certain encoding (`decodeBytes`).

The structure also suggests the possibility of sharing implementations among concrete classes that implement template methods. We may create an abstract class `AsciiTextReader` that extends `TextReader` and implements method `decodeBytes`. And build concrete classes `FileAsciiTextReader` and `HttpAsciiTextReader` that extend `AsciiTextReader` and implement method `readAllBytes` to handle resources on different storage media.

Participants

The participants of Template Method Pattern include the following:

- **Abstract class:** `TextReader`

Defines the signatures of template methods, as well as the outline of algorithms that weave everything together.

- **Concrete classes:** AsciiTextReader, FileAsciiTextReader and HttpAsciiTextReader

Implements template methods defined in abstract classes. Typical concrete classes are FileAsciiTextReader and HttpAsciiTextReader in this example. However, compared to being abstract, *defining the outline of algorithms* weighs more in the categorization.

Pattern scope

Template Method Pattern is usually applied in a relatively small scope. It provides an extendable way to implement features and avoid redundancy from the upper structure of a series of algorithms.

Implementation

There are two levels of the inheriting hierarchy: the AsciiTextReader will subclass TextReader as another abstract class. It implements method decodeBytes but leaves readAllBytes to its subclasses. Starting with the TextReader:

```
abstract class TextReader {  
    async readAllText(): Promise<string> {  
        let bytes = await this.readAllBytes();  
        let text = this.decodeBytes(bytes);  
  
        return text;  
    }  
  
    abstract async readAllBytes(): Promise<Buffer>;  
  
    abstract decodeBytes(bytes: Buffer): string;  
}
```

We are using Promises with `async` and `await` which are coming to ECMAScript next. Please refer to the following links for more information:
<https://github.com/Microsoft/TypeScript/issues/1664>
<https://tc39.github.io/ecmascript-asyncawait/>



And now let's subclass TextReader as AsciiTextReader which still remains abstract:

```
abstract class AsciiTextReader extends TextReader {  
    decodeBytes(bytes: Buffer): string {  
        return bytes.toString('ascii');  
    }  
}
```

For FileAsciiTextReader, we'll need to import filesystem (fs) module of Node.js to perform file reading:

```
import * as FS from 'fs';  
  
class FileAsciiTextReader extends AsciiTextReader {  
    constructor(  
        public path: string  
    ) {  
        super();  
    }  
  
    async readAllBytes(): Promise<Buffer> {  
        return new Promise<Buffer>((resolve, reject) => {  
            FS.readFile(this.path, (error, bytes) => {  
                if (error) {  
                    reject(error);  
                } else {  
                    resolve(bytes);  
                }  
            });  
        });  
    }  
}
```

For HttpAsciiTextReader, we are going to use a popular package request to send HTTP requests:

```
import * as request from 'request';  
  
class HttpAsciiTextReader extends AsciiTextReader {  
    constructor(  
        public url: string  
    ) {  
        super();  
    }  
  
    async readAllBytes(): Promise<Buffer> {  
        return new Promise<Buffer>((resolve, reject) => {  
            request(this.url, {
```

```
        encoding: null
    }, (error, bytes, body) => {
        if (error) {
            reject(error);
        } else {
            resolve(body);
        }
    });
});
}
}
```



Both concrete reader implementations pass resolver functions to the Promise constructor for converting asynchronous Node.js style callbacks to Promises. For more information, read more about the Promise constructor : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Consequences

Compared to Strategy Pattern, Template Method Pattern provides convenience for building objects with the same outline of algorithms outside of the existing system. This makes Template Method Pattern a useful way to build tooling classes instead of fixed processes built-in.

But Template Method Pattern has less runtime flexibility as it does not have a *manager*. It also relies on the client who's using those objects to do the work. And as the implementation of Template Method Pattern relies on subclassing, it could easily result in hierarchies that have a similar code on different branches. Though this could be optimized by using techniques like *mixin*.

Observer Pattern

Observer Pattern is an important Pattern backed by an important idea in software engineering. And it is usually a key part of MVC architecture and its variants as well.

If you have ever written an application with a rich user interface without a framework like Angular or a solution with React, you might probably have struggled with changing class names and other properties of UI elements. More specifically, the code that controls those properties of the same group of elements lies every branch related to the elements in related event listeners, just to keep the elements being correctly updated.

Consider a “Do” button of which the `disabled` property should be determined by the status of a WebSocket connection to a server and whether the currently active item is done. Every time the status of either the connection or the active item gets updated, we’ll need to update the button correspondingly. The most “handy” way could be two somewhat identical groups of code being put in two event listeners. But in this way, the amount of similar code would just keep growing as more relevant objects get involved.

The problem in this “Do” button example is that, the behavior of code that’s controlling the button is driven by primitive events. The heavy load of managing the connections and behaviors among different events is directly taken by the developer who’s writing that code. And unfortunately, the complexity in this case, grows exponentially, which means it could easily exceed our brain capacity. Writing code this way might result in more bugs and make maintaining much likely to introduce new bugs.

But the beautiful thing is, we can find the factors that multiply and output the desired result, and the reference for dividing those factors are groups of related states. Still speaking of the “Do” button example, what the button cares about is: connection status and the active item status (assuming they are booleans `connected` and `loaded`). We can have the code written as two parts: one part that changes those states, and another part that updates the button:

```
let button = document.getElementById('do-button');

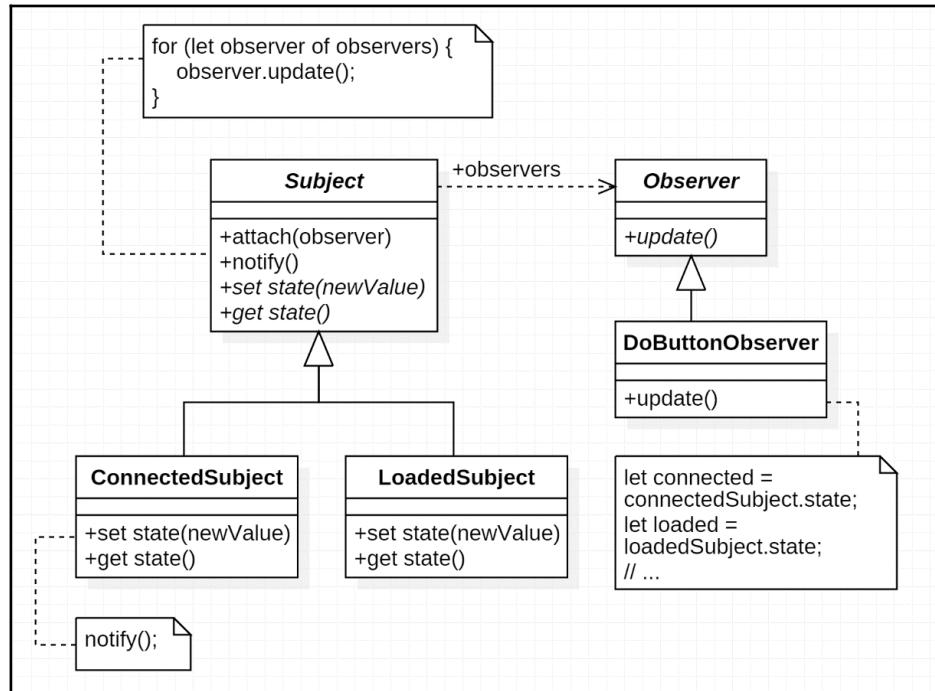
let connected = false;
let loaded = false;

function updateButton() {
    let disabled = !connected && !loaded;
    button.disabled = disabled;
}

connection.on('statuschange', event => {
    connected = event.connected;
    updateButton();
});

activeItem.on('statuschange', event => {
    loaded = event.loaded;
    updateButton();
});
```

The preceding sample code already has the embryo of Observer Pattern: the subjects (states connected and loaded) and the observer (updateButton function), though we still need to call updateButton manually every time any related state changes. An improved structure could look like the following figure:



But just like the example we've been talking about, observers in many situations care about more than one state. It could be less satisfying to have subjects attach observers separately.

A solution to this could be multi-state subjects, to achieve that, we can form a composite subject that contains sub-subjects. If a subject receives a `notify` call, it wakes up its observers and at the same time notifies its parent. Thus the observer can attach one composite subject for notifications of changes that happen to multiple states.

However, the process of creating the composite itself could still be annoying. In dynamic programming languages like JavaScript, we may have a state manager that contains specific states handling notifications and attaching observers directly with implicit creations of subjects:

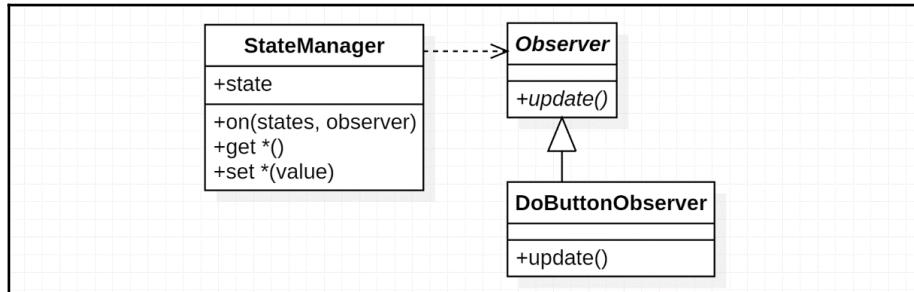
```
let stateManager = new StateManager({
  connected: false,
  loaded: false,
  foo: 'abc',
  bar: 123
});

stateManager.on(['connected', 'loaded'], () => {
  let disabled =
    !stateManager.connected && !stateManager.loaded;
  button.disabled = disabled;
});
```



In many MV* frameworks, the states to be observed are analyzed automatically from related expressions by built-in parsers or similar mechanisms.

And now the structure gets even simpler:



Participants

We've talked about the basic structure of Observer Pattern with subjects and observers, and a variant with implicit subjects. The participants of the basic structure include the following:

- **Subject**

Subject to be observed. Defines methods to attach or notify observers. A subject could also be a composite that contains sub-subjects, which allows multiple states to be observed with the same interface.

- **Concrete subject:** `ConnectedSubject` and `LoadedSubject`

Contains state related to the subject, and implements methods or properties to get and set their state.

- **Observer**

Defines the interface of an object that reacts when an observation notifies. In JavaScript, it could also be an interface (or signature) of a function.

- **Concrete observer:** `DoButtonObserver`

Defines the action that reacts to the notifications of subjects being observed. Could be a callback function that matches the signature defined.

In the variant version, the participants include the following:

- **State manager**

Manages a complex, possibly multi-level state object containing multiple states. Defines the interface to attach observers with subjects, and notifies those observers when a subject changes.

- **Concrete subject**

Keys to specific states. For example, string "connected" may represent state `stateManager.connected`, while string "foo.bar" may represent state `stateManager.foo.bar`.

Observer and *concrete observer* are basically the same as described in the former structure. But observers are now notified by the state manager instead of subject objects.

Pattern scope

Observer Pattern is a pattern that may easily structure half of the project. In MV* architectures, Observer Pattern can decouple the view from business logic. The concept of view can be applied to other scenarios related to displaying information as well.

Implementation

Both of the structures we've mentioned should not be hard to implement, though more details should be put into consideration for production code. We'll go with the second implementation that has a central state manager.



To simplify the implementation, we will use `get` and `set` methods to access specific states by their keys. But many frameworks available might handle those through getters and setters, or other mechanisms.



To learn about how frameworks like Angular handle states changing, please read their documentation or source code if necessary.

We are going to have `StateManager` inherit `EventEmitter`, so we don't need to care much about issues like multiple listeners. But as we are accepting multiple state keys as subjects, an overload to method `on` will be added. Thus the outline of `StateManager` would be as follows:

```
type Observer = () => void;

class StateManager extends EventEmitter{
  constructor(
    private state: any
  ) {
    super();
  }

  set(key: string, value: any): void { }

  get(key: string): any { }

  on(state: string, listener: Observer): this;
  on(states: string[], listener: Observer): this;
  on(states: string | string[], listener: Observer): this { }
}
```



You might have noticed that method `on` has the return type `this`, which may keep referring to the type of current instance. Type `this` is very helpful for chaining methods.

The keys will be "foo" and "foo.bar", we need to split a key as separate identifiers for accessing the value from the `state` object. Let's have a private `_get` method that takes an array of identifiers as input:

```
private _get(identifiers: string[]): any {
  let node = this.state;

  for (let identifier of identifiers) {
    node = node[identifier];
  }

  return node;
}
```

Now we can implement method `get` upon `_get`:

```
get(key: string): any {
  let identifiers = key.split('.');
  return this._get(identifiers);
}
```

For method `set`, we can get the parent object of the last identifier of property to be set, so things work like this:

```
set(key: string, value: any): void {
  let identifiers = key.split('.');
  let lastIndex = identifiers.length - 1;

  let node = this._get(identifiers.slice(0, lastIndex));
  node[identifiers[lastIndex]] = value;
}
```

But there's one more thing, we need to notify observers that are observing a certain subject:

```
set(key: string, value: any): void {
  let identifiers = key.split('.');
  let lastIndex = identifiers.length - 1;

  let node = this._get(identifiers.slice(0, lastIndex));
  node[identifiers[lastIndex]] = value;

  for (let i = identifiers.length; i > 0; i--) {
```

```
    let key = identifiers.slice(0, i).join('.');
    this.emit(key);
}
}
```

When we're done with the notifying part, let's add an overload for method `on` to support multiple keys:

```
on(state: string, listener: Observer): this;
on(states: string[], listener: Observer): this;
on(states: string | string[], listener: Observer): this {
  if (typeof states === 'string') {
    super.on(states, listener);
  } else {
    for (let state of states) {
      super.on(state, listener);
    }
  }
}

return this;
}
```

Problem solved. Now we have a state manager that will work for simple scenarios.

Consequences

Observer Pattern decouples subjects with observers. While an observer may be observing multiple states in subjects at the same time, it usually does not care about which state triggers the notification. As a result, the observer may make *unnecessary* updates that actually do nothing to – for example – the view.

However, the impact on performance could be negligible most of the time, not even need to mention the benefits it brings.

By splitting view and logic apart, Observer Pattern may reduce possible branches significantly. This will help eliminate bugs caused at the coupling part between view and logic. Thus, by properly applying Observer Pattern, the project will be made much more robust and easier to maintain.

However, there are some details we still need care about:

1. The observer that updates the state could cause circular invocation.
2. For more complex data structures like collections, it might be expensive to re-render everything. Observers in this scenario may need more information about the change to only perform necessary updates. View implementations like React do this in another way; they introduce a concept called **Virtual DOM**. By updating and diffing the virtual DOM before re-rendering the actual DOM (which could usually be the bottleneck of performance), it provides a relatively general solution for different data structures.

Visitor Pattern

Visitor Pattern provides a uniformed interface for *visiting* different data or objects while allowing detailed operations in concrete visitors to vary. Visitor Pattern is usually used with composites, and it is widely used for walking through data structures like **abstract syntax tree (AST)**. But to make it easier for those who are not familiar with compiler stuff, we will provide a simpler example.

Consider a DOM-like tree containing multiple elements to render:

```
[  
  Text {  
    content: "Hello, "  
  },  
  BoldText {  
    content: "TypeScript"  
  },  
  Text {  
    content: "! Popular editors:\n"  
  },  
  UnorderedList {  
    items: [  
      ListItem {  
        content: "Visual Studio Code"  
      },  
      ListItem {  
        content: "Visual Studio"  
      },  
      ListItem {  
        content: "WebStorm"  
      }  
    ]  
  }]
```

]

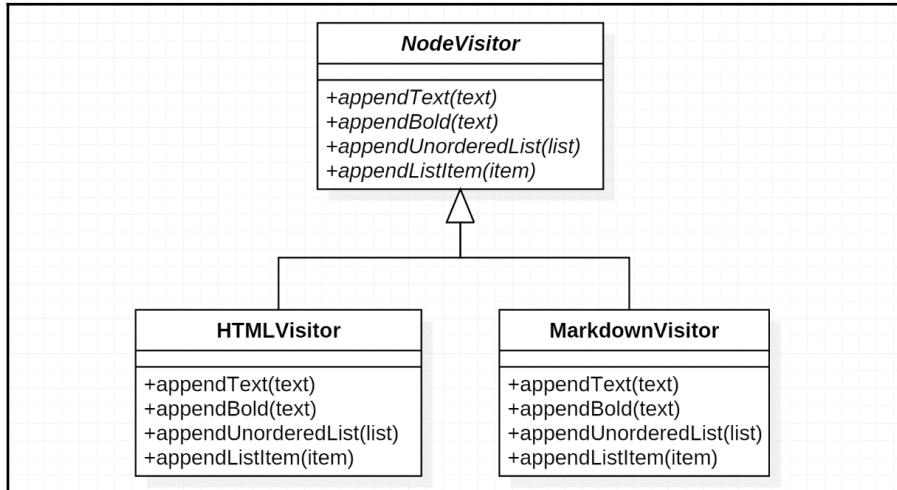
The rendering result in HTML would look like this:

```
Hello, <b>TypeScript</b>! Popular editors:  
<ul>  
<li>Visual Studio Code</li>  
<li>Visual Studio</li>  
<li>WebStorm</li>  
</ul>
```

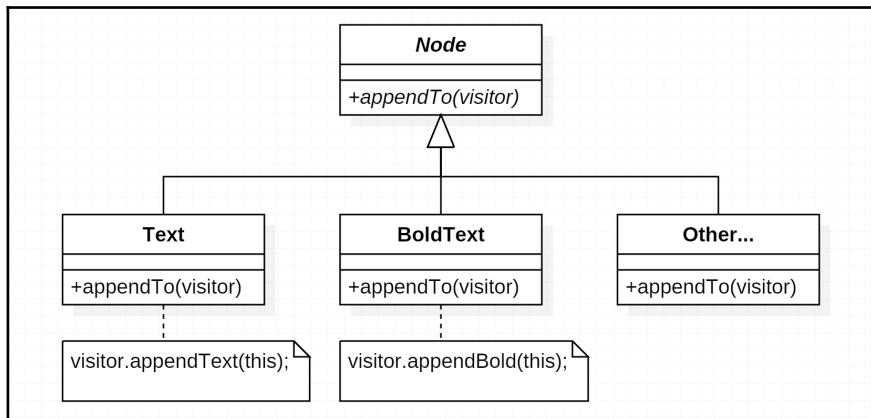
While in Markdown, it would look like this:

```
Hello, **TypeScript**! Popular editors:  
- Visual Studio Code  
- Visual Studio  
- WebStorm
```

Visitor Pattern allows operations in the same category to be coded in the same place. We'll have concrete visitors, `HTMLVisitor` and `MarkdownVisitor` that take the responsibilities of transforming different nodes by visiting them respectively and recursively. The nodes being visited have a method `accept` for accepting a visitor to perform the transformation. An overall structure of Visitor Pattern could be split into two parts, the first part is the visitor abstraction and its concrete subclasses:



The second part is the abstraction and concrete subclasses of nodes to be visited:



Participants

The participants of Visitor Pattern include the following:

- **Visitor:** `NodeVisitor`

Defines the interface of operations corresponding to each element class. In languages with static types and method overloading, the method names can be unified. But as it takes extra runtime checking in JavaScript, we'll use different method names to distinguish them. The operation methods are usually named after `visit`, but here we use `append` as its more related to the context.

- **Concrete visitor:** `HTMLVisitor` and `MarkdownVisitor`

Implements every operation of the concrete visitor, and handles internal states if any.

- **Element:** `Node`

Defines the interface of the element accepting the visitor instance. The method is usually named `accept`, though here we are using `appendTo` for a better matching with the context. Elements could themselves be composites and pass visitors on with their child elements.

- **Concrete element:** Text, BoldText, UnorderedList and ListItem

Implements accept method and calls the method from the visitor instance corresponding to the element instance itself.

- **Client:**

Enumerates elements and applies visitors to them.

Pattern scope

Visitor Pattern can form a large feature inside a system. For some programs under certain categories, it may also form the core architecture. For example, *Babel* uses Visitor Pattern for AST transforming and a plugin for Babel is actually a visitor that can visit and transform elements it cares about.

Implementation

We are going to implement `HTMLVisitor` and `MarkdownVisitor` which may transform nodes to text, as we've talked about. Start with the upper abstraction:

```
interface Node {  
    appendTo(visitor: NodeVisitor): void;  
}  
  
interface NodeVisitor {  
    appendText(text: Text): void;  
    appendBold(text: BoldText): void;  
    appendUnorderedList(list: UnorderedList): void;  
    appendListItem(item: ListItem): void;  
}
```

Continue with concrete nodes that do similar things, `Text` and `BoldText`:

```
class Text implements Node {  
    constructor(  
        public content: string  
    ) {}  
  
    appendTo(visitor: NodeVisitor): void {  
        visitor.appendText(this);  
    }  
}
```

```
class BoldText implements Node {
    constructor(
        public content: string
    ) { }

    appendTo(visitor: NodeVisitor): void {
        visitor.appendBold(this);
    }
}
```

And list stuff:

```
class UnorderedList implements Node {
    constructor(
        public items: ListItem[]
    ) { }

    appendTo(visitor: NodeVisitor): void {
        visitor.appendUnorderedList(this);
    }
}

class ListItem implements Node {
    constructor(
        public content: string
    ) { }

    appendTo(visitor: NodeVisitor): void {
        visitor.appendListItem(this);
    }
}
```

Now we have the elements of a structure to be visited, we'll begin to implement concrete visitors. Those visitors will have an `output` property for the transformed string.

`HTMLVisitor` goes first:

```
class HTMLVisitor implements NodeVisitor {
    output = '';

    appendText(text: Text) {
        this.output += text.content;
    }

    appendBold(text: BoldText) {
        this.output += `<b>${text.content}</b>`;
    }

    appendUnorderedList(list: UnorderedList) {
```

```

    this.output += '<ul>';

    for (let item of list.items) {
        item.appendTo(this);
    }

    this.output += '</ul>';
}

appendListItem(item: ListItem) {
    this.output += `<li>${item.content}</li>`;
}
}

```

Pay attention to the loop inside `appendUnorderedList`, it handles visiting of its own list items.

A similar structure applies to `MarkdownVisitor`:

```

class MarkdownVisitor implements NodeVisitor {
    output = '';

    appendText(text: Text) {
        this.output += text.content;
    }

    appendBold(text: BoldText) {
        this.output += `**${text.content}**`;
    }

    appendUnorderedList(list: UnorderedList) {
        this.output += '\n';

        for (let item of list.items) {
            item.appendTo(this);
        }
    }

    appendListItem(item: ListItem) {
        this.output += `- ${item.content}\n`;
    }
}

```

Now the infrastructures are ready, let's create the tree-like structure we've been imagining since the beginning:

```

let nodes = [
    new Text('Hello, '),

```

```
new BoldText('TypeScript'),  
new Text('! Popular editors:\n'),  
new UnorderedList([  
    new ListItem('Visual Studio Code'),  
    new ListItem('Visual Studio'),  
    new ListItem('WebStorm')  
])  
];
```

And finally, build the outputs with visitors:

```
let htmlVisitor = new HTMLVisitor();  
let markdownVisitor = new MarkdownVisitor();  
  
for (let node of nodes) {  
    node.appendTo(htmlVisitor);  
    node.appendTo(markdownVisitor);  
}  
  
console.log(htmlVisitor.output);  
console.log(markdownVisitor.output);
```

Consequences

Both Strategy Pattern and Visitor Pattern could be applied to scenarios of processing objects. But Strategy Pattern relies on clients to handle all related arguments and contexts, this makes it hard to come out with an exquisite abstraction if the expected behaviors of different objects differ a lot. Visitor Pattern solves this problem by decoupling visit actions and operations to be performed.

By passing different visitors, Visitor Pattern can apply different operations to objects without changing other code although it usually means adding new elements and would result in adding related operations to an abstract visitor and all of its concrete subclasses.

Visitors like the `NodeVisitor` in the previous example may store state itself (in that example, we stored the output of transformed nodes) and more advanced operations can be applied based on the state accumulated. For example, it's possible to determine what has been appended to the output, and thus we can apply different behaviors with the node currently being visited.

However, to complete certain operations, extra public methods may need to be exposed from the elements.

Summary

In this chapter, we've talked about other behavior design patterns as complements to the former chapter, including Strategy, State, Template Method, Observer and Visitor Pattern.

Strategy Pattern is so common and useful that it may appear in a project several times, with different forms. And you might not know you were using Observer Pattern with implementation in a daily framework.

After walking through those patterns, you might find there are many ideas in common behind each pattern. It is worth thinking what's behind them and even letting the outline go in your mind.

In the next chapter, we'll continue with some handy patterns related to JavaScript and TypeScript, and important scenarios of those languages.

7

Patterns and Architectures in JavaScript and TypeScript

In the previous four chapters, we've walked through common and classical design patterns and discussed some of their variants in JavaScript or TypeScript. In this chapter, we'll continue with some architecture and patterns closely related to the language and their common applications. We don't have many pages to expand and certainly cannot cover everything in a single chapter, so please take it as an appetizer and feel free to explore more.

Many topics in this chapter are related to asynchronous programming. We'll start with a web architecture for Node.js that's based on Promise. This is a larger topic that has interesting ideas involved, including abstractions of responses and permissions, as well as error handling tips. Then we'll talk about how to organize modules with **ECMAScript (ES)** module syntax. And this chapter will end with several useful asynchronous techniques.

Overall, we'll have the following topics covered in this chapter:

- Architecture and techniques related to Promise
- Abstraction of responses and permissions in a web application
- Modularizing a project to scale
- Other useful asynchronous techniques

Again, due to the limited length, some of the related code is aggressively simplified and nothing more than the idea itself can be applied practically.



Promise-based web architecture

To have a better understanding of the differences between Promises and traditional callbacks, consider an asynchronous task like this:

```
function process(callback) {
  stepOne((error, resultOne) => {
    if (error) {
      callback(error);
      return;
    }

    stepTwo(resultOne, (error, resultTwo) => {
      if (error) {
        callback(error);
        return;
      }

      callback(undefined, resultTwo + 1);
    });
  });
}
```

If we write preceding above in Promise style, it would be as follows:

```
function process() {
  return stepOne()
    .then(result => stepTwo(result))
    .then(result => result + 1);
}
```

As in the preceding example, Promise makes it easy and *natural* to write asynchronous operations with a flat chain instead of nested callbacks. But the most exciting thing about Promise might be the benefits it brings to error handling. In a Promise-based architecture, throwing an error can be safe and pleasant. You don't have to explicitly handle errors when chaining asynchronous operations, and this makes mistakes less likely to happen.

With the growing usage with ES6 compatible runtimes, Promise is already there out of the box. And we actually have plenty of polyfills for Promises (including my *ThenFail* written in TypeScript), as people who write JavaScript roughly refer to the same group of people who created wheels.

Promises work well with other Promises:

- A *Promises/A+*-compatible implementation should work with other *Promises/A+*-compatible implementations
- Promises work best in a Promise-based architecture

If you are new to Promise, you might be complaining about using Promises with a callback-based project. Using asynchronous helpers such as `Promise.each` (non-standard) provided by Promise libraries is a common reason for people to try out Promise, but it turns out they have better alternatives (for a callback-based project) such as the popular `async` library.

The reason that makes you decide to switch should not be these helpers (as there are a lot of them for old-school callbacks as well), but an easier way to handle errors or to take advantage of the ES `async/await` feature, which is based on Promise.

Promisifying existing modules or libraries

Though Promises do best in a Promise-based architecture, it is still possible to begin using Promise with a smaller scope by promisifying existing modules or libraries.

Let's take Node.js style callbacks as an example:

```
import * as FS from 'fs';

FS.readFile('some-file.txt', 'utf-8', (error, text) => {
  if (error) {
    console.error(error);
    return;
  }

  console.log('Content:', text);
});
```

You may expect a promisified version of the `readFile` function to look like the following:

```
FS
.readFile('some-file.txt', 'utf-8')
.then(text => {
  console.log('Content:', text);
})
.catch(reason => {
  console.error(reason);
});
```

The implementation of the promisified function `readFile` can be easy:

```
function readFile(path: string, options: any): Promise<string> {
  return new Promise((resolve, reject) => {
    FS.readFile(path, options, (error, result) => {
      if (error) {
        reject(error);
      } else {
        resolve(result);
      }
    });
  });
}
```



I am using the type `any` here for parameter options to reduce the size of the code example, but I would suggest not using `any` whenever possible in practice.

There are libraries that are able to promisify methods automatically. Though, unfortunately, you might need to write declaration files yourself for the promisified methods if there are no promisified version available.

Views and controllers in Express

Many of us may have already worked with frameworks such as **Express**. And this is how we render a view or response with JSON in Express:

```
import * as Path from 'path';
import * as express from 'express';

let app = express();

app.set('engine', 'hbs');
app.set('views', Path.join(__dirname, '../views'));

app.get('/page', (req, res) => {
  res.render('page', {
    title: 'Hello, Express!',
    content: '...'
  });
});

app.get('/data', (req, res) => {
  res.json({
```

```
    version: '0.0.0',
    items: []
  });
});

app.listen(1337);
```

We will usually separate controllers from the routing configuration:

```
import { Request, Response } from 'express';

export function page(req: Request, res: Response): void {
  res.render('page', {
    title: 'Hello, Express!',
    content: '...'
  });
}
```

Thus we may have a better idea of existing routes, and have controllers managed more easily. Furthermore, automated routing could be introduced so that we don't always need to update routing manually:

```
import * as glob from 'glob';

let controllersDir = Path.join(__dirname, 'controllers');

let controllerPaths = glob.sync('**/*.js', {
  cwd: controllersDir
});

for (let path of controllerPaths) {
  let controller = require(Path.join(controllersDir, path));
  let urlPath = path.replace(/\//g, '/').replace(/\.js$/,'');

  for (let actionPerformed of Object.keys(controller)) {
    app.get(
      `/${urlPath}/${actionName}`,
      controller[actionName]
    );
  }
}
```

The implementation above is certainly too simple to cover daily use, but it shows a rough idea of how automated routing could work: via conventions based on file structures.

Now, if we are working with asynchronous code written in Promises, an action in the controller could be like the following:

```
export function foo(req: Request, res: Response): void {
  Promise
    .all([
      Post.getContent(),
      Post.getComments()
    ])
    .then(([post, comments]) => {
      res.render('foo', {
        post,
        comments
      });
    });
}
```



We are destructuring an array within a parameter. `Promise.all` returns a Promise of an array with elements corresponding to the values of the resolvable passed in. (A resolvable means a normal value or a Promise-like object that may resolve to a normal value.)

But that's not enough; we still need to handle errors properly, or in some Promise implementations, the preceding code may fail in silence because the Promise chain is not handled by a rejection handler (which is terrible). In Express, when an error occurs, you should call `next` (the third argument passed into the callback) with the error object:

```
import { Request, Response, NextFunction } from 'express';

export function foo(
  req: Request,
  res: Response,
  next: NextFunction
): void {
  Promise
    // ...
    .catch(reason => next(reason));
}
```

Now, we are fine with the correctness of this approach, but that's simply not how Promises work. Explicit error handling with callbacks could be eliminated in the scope of controllers, and the easiest way is to return the Promise chain and hand over to code that was previously doing routing logic. So the controller could be written like this:

```
export function foo(req: Request, res: Response) {
  return Promise
    .all([

```

```

    Post.getContent(),
    Post.getComments()
  ])
  .then(([post, comments]) => {
    res.render('foo', {
      post,
      comments
    });
  });
}

```

But, could we make it even better?

Abstraction of responses

We've already been returning a Promise to tell whether an error occurs. So now the returned Promise indicates the status of the response: success or failure. But why we are still calling `res.render()` for rendering the view? The returned promise object could be the response itself rather than just an error indicator.

Think about the controller again:

```

export class Response { }

export class PageResponse extends Response {
  constructor(view: string, data: any) { }
}

export function foo(req: Request) {
  return Promise
    .all([
      Post.getContent(),
      Post.getComments()
    ])
    .then(([post, comments]) => {
      return new PageResponse('foo', {
        post,
        comments
      });
    });
}

```

The response object returned could vary for different response outputs. For example, it could be either a `PageResponse` like it is in the preceding example, a `JSONResponse`, a `StreamResponse`, or even a simple Redirection.

As, in most cases, `PageResponse` or `JSONResponse` is applied, and the view of a `PageResponse` can usually be implied by the controller path and action name, it is useful to have those two responses automatically generated from a plain data object with a proper view to render with:

```
export function foo(req: Request) {
  return Promise
    .all([
      Post.getContent(),
      Post.getComments()
    ])
    .then(([post, comments]) => {
      return {
        post,
        comments
      };
    });
}
```

And that's how a Promise-based controller should respond. With this idea, let's update the routing code with the abstraction of responses. Previously, we were passing controller actions directly as Express request handlers. Now we need to do some wrapping up with the actions by resolving the return value, and applying operations based on the resolved result:

1. If it fulfills and it's an instance of `Response`, apply it to the `res` object passed in by Express.
2. If it fulfills and it's a plain object, construct a `PageResponse` or a `JSONResponse` if no view found and apply it to the `res` object.
3. If it rejects, call the `next` function with the reason.

Previously, it was like this:

```
app.get(`/${urlPath}/${actionName}`, controller[actionName]);
```

Now it gets a few more lines:

```
let action = controller[actionName];

app.get(`/${urlPath}/${actionName}`, (req, res, next) => {
  Promise
    .resolve(action(req))
    .then(result => {
      if (result instanceof Response) {
        result.applyTo(res);
      }
    })
    .catch(next);
});
```

```

    } else if (existsView(actionName)) {
        new PageResponse(actionName, result).applyTo(res);
    } else {
        new JSONResponse(result).applyTo(res);
    }
}
.catch(reason => next(reason));
});

```

However, so far we can handle only GET requests as we hardcoded `app.get()` in our router implementation. The poor view-matching logic can hardly be used in practice either. We need to make the actions configurable, and ES decorators could do nice work here:

```

export default class Controller {
    @get({
        view: 'custom-view-path'
    })
    foo(req: Request) {
        return {
            title: 'Action foo',
            content: 'Content of action foo'
        };
    }
}

```

I'll leave the implementation to you, and feel free to make it awesome.

Abstraction of permissions

Permissions play an important role in a project, especially in systems that have different user groups, for example, a forum. The abstraction of permissions should be extendable to satisfy changing requirements, and it should be easy to use as well.

Here, we are going to talk about the abstraction of permission in the level of controller actions. Consider the legibility of performing one or more actions as a *privilege*. The permission of a user may consist of several privileges and usually most users at the same level would have the same set of privileges. So we may have a larger concept, namely *groups*.

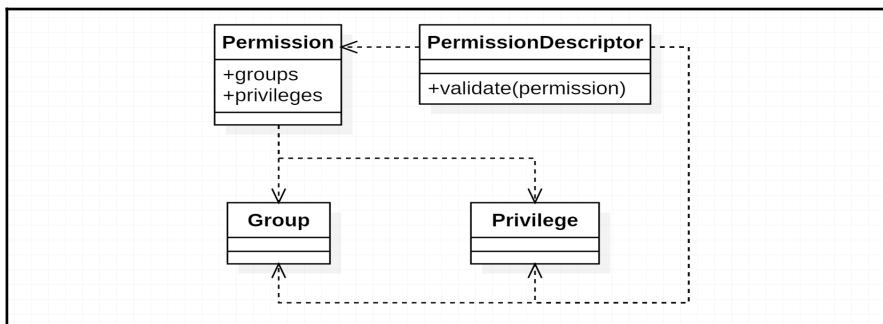
The abstraction could either work based on both groups and privileges or based on privileges only (groups are then just aliases to sets of privileges):

- Abstractions that validate based on privileges and groups at the same time is easier to build. You do not need to create a large list of which actions can be performed for a certain group of users; granular privileges are only required

when necessary.

- Abstractions that validate based on privileges have better control and more flexibility for describing the permission. For example, you can remove a small set of privileges from the permission of a user easily.

However, both approaches have similar upper-level abstractions and differ mostly in implementation. The general structure of the permission abstractions we've talked about is as follows:



The participants include the following:

- **Privilege**: Describes detailed privileges corresponding to specific actions
- **Group**: Defines a set of privileges
- **Permission**: Describes what a user is capable of doing; consists of groups the user belongs to and privileges the user has
- **Permission descriptor**: Describes how the permission of a user would be sufficient; consists of *possible* groups and privileges

Expected errors

A great concern wiped away by using Promises is that we do not need to worry about throwing an error in a callback would crash the application most of the time. The error will flow through the Promises chain and, if not caught, will be handled by our router. Errors can be roughly divided into expected errors and unexpected errors. Expected errors are usually caused by incorrect input or foreseeable exceptions, and unexpected errors are usually caused by bugs or other libraries the project relies on.

For expected errors, we usually want to give user-friendly responses with readable error messages and codes, so that users can help themselves to find solutions or report to us with useful context. For unexpected errors, we would also want reasonable responses (usually messages described as unknown errors), a detailed server-side log (including the real error name, message, stack information, and so on), and even alarms for getting the team notified as soon as possible.

Defining and throwing expected errors

The router will need to handle different types of errors, and an easy way to achieve that is to subclass a universal `ExpectedError` class and throw its instances out:

```
import ExtendableError from 'extendable-error';

class ExpectedError extends ExtendableError {
  constructor(
    message: string,
    public code: number
  ) {
    super(message);
  }
}
```



The `extendable-error` is a package of mine that handles stack trace and the `message` property. You can directly extend the `Error` class as well.

Thus, when receiving an expected error, we can safely output its message as part of the response. And if it's not an instance of `ExpectedError`, we can then output predefined unknown error messages and have detailed error information logged.

Transforming errors

Some errors, such as those caused by unstable networks or remote services, are expected; we may want to catch those errors and throw them out again as expected errors. But it is rather trivial to actually do that. A centralized error-transforming process can then be applied to reduce the efforts required to manage those errors.

The transforming process includes two parts: filtering (or matching) and transforming. There are many approaches to filter errors, such as the following:

- **Filter by error class:** Many third-party libraries throw errors of certain classes. Taking Sequelize (a popular Node.js ORM) as an example, it throws `DatabaseError`, `ConnectionError`, `ValidationError`, and so on. By filtering errors by checking whether they are instances of a certain error class, we may easily pick up target errors from the pile.
- **Filter by string or regular expression:** Sometimes a library might be throwing errors that are instances of an `Error` class itself instead of its subclasses; this makes those errors harder to distinguish from others. In this situation, we may filter those errors by their message, with keywords or regular expressions.
- **Filter by scope:** It's possible that instances of the same error class with the same error message should result in different responses. One of the reasons might be that the operation that throws a certain error is at a lower level, but is being used by upper structures within different scopes. Thus, a `scope` mark could be added for those errors and make them easier to be filtered.

There could be more ways to filter errors, and they are usually able to cooperate as well. By properly applying those filters and transforming errors, we can reduce noise for analyzing what's going on within a system and locate problems faster if they show up.

Modularizing project

Before ES6, there were a lot of module solutions for JavaScript that worked. The two most famous of them are AMD and commonjs. AMD is designed for asynchronous module loading, which is mostly applied in browsers, while commonjs does module loading synchronously, and that's the way the Node.js module system works.

To make it work asynchronously, writing an AMD module takes more characters. And due to the popularity of tools such as browserify and webpack, commonjs becomes popular even for browser projects.

The proper granularity of internal modules could help a project keep its structure healthy. Consider a project structure like this:

```
project
├── controllers
└── core
    └── index.ts
```

```
|-product
  index.ts
  order.ts
  shipping.ts

|-user
  index.ts
  account.ts
  statistics.ts

|-helpers
|-models
|-utils
|-views
```

Assume we are writing a controller file that's going to import a module defined by the core/product/order.ts file. Previously, with the commonjs require style, we would want to write the following:

```
const Order = require('../core/product/order');
```

Now, with the new ES import syntax, it would be as follows:

```
import * as Order from '../core/product/order';
```

Wait, isn't that essentially the same? Sort of. But you may have noticed several index.ts files I've put into folders. Now, in the file core/product/index.ts, we can have the following:

```
import * as Order from './order';
import * as Shipping from './shipping';

export { Order, Shipping }
```

Alternatively, we could have the following:

```
export * from './order';
export * from './shipping';
```

What's the difference? The ideas behind those two approaches of re-exporting modules can vary. The first style works better when we treat Order and Shipping as namespaces, under which the entity names may not be easy to distinguish from one group to another. With this style, the files are the natural boundaries of building those namespaces. The second style weakens the namespace property of two files and uses them as tools to organize objects and classes under the same larger category.

A good thing about using those files as namespaces is that multiple-level re-exporting is fine while weakening namespaces makes it harder to understand different identifier names as the number of re-exporting levels grows.

Asynchronous patterns

When we are writing JavaScript with network or file system I/O, there is a 95% chance that we are doing it asynchronously. However, an asynchronous code may tremendously decrease the determinability at the dimension of time. But we are so lucky that JavaScript is usually single-threaded; this makes it possible for us to write predictable code without mechanisms such as locks most of the time.

Writing predictable code

The predictable code relies on predictable tools (if you are using any). Consider a helper like this:

```
type Callback = () => void;

let isReady = false;
let callbacks: Callback[] = [];

setTimeout(() => {
  callbacks.forEach(callback => callback());
  callbacks = undefined;
}, 100);
export function ready(callback: Callback): void {
  if (!callbacks) {
    callback();
  } else {
    callbacks.push(callback);
  }
}
```

This module exports a `ready` function, which will invoke the callbacks passed in when “ready”. It will assure that callbacks will be called even if added after that. However, you cannot say for sure whether the callback will be called in the current event loop:

```
import { ready } from './foo';

let i = 0;

ready(() => {
```

```
    console.log(i);
});

i++;
```

In the preceding example, `i` could either be 0 or 1 when the callback gets called. Again, this is not wrong, or even bad, it just makes the code less predictable. When someone else reads this piece of code, he or she will need to consider two possibilities of how this program would run. To avoid this issue, we can simply wrap up the synchronous invocation with `setImmediate` (it may fallback to `setTimeout` in older browsers):

```
export function ready(callback: Callback): void {
  if (!callbacks) {
    setImmediate(() => callback());
  } else {
    callbacks.push(callback);
  }
}
```

Writing predictable code is actually more than writing predictable asynchronous code. The highlighted line above can also be written as `setImmediate(callback)`, but that would make people who read your code think twice: how will `callback` get called and what are the arguments?

Consider the line of code below:

```
let results = ['1', '2', '3'].map(parseInt);
```

What's the value of the array `results`? Certainly not `[1, 2, 3]`. Because the callback passed to the method `map` receives several arguments: value of current item, index of current item, and the whole array, while the function `parseInt` accepts two arguments: string to parse, and radix. So `results` are actually the results of the following snippet:

```
[parseInt('1', 0), parseInt('2', 1), parseInt('3', 2)];
```

However, it is actually okay to write `setImmediate(callback)` directly, as the APIs of those functions (including `setTimeout`, `setInterval`, `process.nextTick`, and so on) are designed to be used in this way. And it is fair to assume people who are going to maintain this project know that as well. But for other asynchronous functions whose signatures are not well known, it is recommended to call them with explicit arguments.

Asynchronous creational patterns

We talked about many creational patterns in Chapter 3, *Creational Design Patterns*. While a constructor cannot be asynchronous, some of those patterns may have problems applying to asynchronous scenarios. But others need only slight modifications for asynchronous use.

In Chapter 4, *Structural Design Patterns* we walked through the Adapter Pattern with a storage example that opens the database and creates a storage object asynchronously:

```
class Storage {  
    private constructor() {}  
  
    open(): Promise<Storage> {  
        return openDatabase()  
            .then(db => new Storage(db))  
    }  
}
```

And in the Proxy Pattern, we made the storage object immediately available from its constructor. When a method of the object is called, it waits for the initialization to complete and finishes the operation:

```
class Storage {  
    private dbPromise: Promise<IDBDatabase>;  
  
    get dbReady(): Promise<IDBDatabase> {  
        if (this.dbPromise) {  
            return this.dbPromise;  
        }  
        // ...  
    }  
  
    get<T>(): Promise<T> {  
        return this  
            .dbReady  
            .then(db => {  
                // ...  
            });  
    }  
}
```

A drawback of this approach is that all members that rely on initialization have to be asynchronous, though most of the time they just are asynchronous.

Asynchronous middleware and hooks

The concept of middleware is widely used in frameworks such as Express. Middleware usually processes its target in serial. In Express, middleware is applied roughly in the order it is added while there are not different phases. Some other frameworks, however, provide hooks for different phases in time. For example, there are hooks that will be triggered *before install*, *after install*, *after uninstall*, and so on.



The middleware mechanism of Express is actually a variant of the Chain of Responsibility Pattern. And depending on the specific middleware to be used, it can act more or less like hooks instead of a responsibility chain.

The reasons to implement middleware or hooks vary. They may include the following:

- **Extensibility:** Most of the time, they are applied due to the requirement of extensibility. New rules and processes could be easily added by new middleware or hooks.
- **Decoupling interactions with business logic:** A module that should only care about business logic could need potential interactions with an interface. For example, we might expect to be able to either enter or update credentials while processing an operation, without restarting everything. Thus we can create a middleware or a hook, so that we don't need to have them tightly coupled.

The implementation of asynchronous middleware could be interesting. Take the Promise version as an example:

```
type Middleware = (host: Host) => Promise<void>;  
  
class Host {  
    middlewares: Middleware[] = [];  
  
    start(): Promise<void> {  
        return this  
            .middlewares  
            .reduce((promise, middleware) => {  
                return promise.then(() => middleware(this));  
            }, Promise.resolve());  
    }  
}
```

Here, we're using `reduce` to do the trick. We passed in a Promise fulfilled with undefined as the initial value, and chained it with the result of `middleware(this)`. And this is actually how the `Promise.each` helper is implemented in many Promise libraries.

Event-based stream parser

When creating an application relies on socket, we usually need a lightweight “protocol” for the client and server to communicate. Unlike XHR that already handles everything, by using socket, you will need to define the boundaries so data won’t be mixed up.

Data transferred through a socket might be concatenated or split, but TCP connection ensures the order and correctness of bytes gets transferred. Consider a tiny protocol that consists of only two parts: a 4-byte unsigned integer followed by a JSON string with byte length that matches the 4-byte unsigned integer.

For example, for JSON "`{ }`", the data packet would be as follows:

```
Buffer <00 00 00 02 7b 7d>
```

To build such a data packet, we just need to convert the JSON string to `Buffer` (with encoding such as `utf-8`, which is default encoding for Node.js), and then prepend its length:

```
function buildPacket(data: any): Buffer {
  let json = JSON.stringify(data);
  let jsonBuffer = new Buffer(json);

  let packet = new Buffer(4 + jsonBuffer.length);

  packet.writeUInt32BE(jsonBuffer.length, 0);
  jsonBuffer.copy(packet, 4, 0);

  return packet;
}
```

A socket client emits a `data` event when it receives new buffers. Assume we are going to send the following JSON strings:

```
// 00 00 00 02 7b 7d
{}

// 00 00 00 0f 7b 22 6b 65 79 22 3a 22 76 61 6c 75 65 22 7d
{"key": "value"}
```

We may be receiving them like this:

- Get two buffers separately; each of them is a complete packet with length and JSON bytes

- Get one single buffer with two buffers concatenated
- Get two, or more than two, buffers; at least one of the previously sent packets gets split into several ones.

The entire process is happening asynchronously. But just like the socket client emits a `data` event, the parser can just emit its own `data` event when a complete packet gets parsed. The parser for parsing our tiny protocol may have only two states, corresponding to header (JSON byte length) and body (JSON bytes), and the emitting of the `data` event happens after successfully parsing the body:

```
class Parser extends EventEmitter {  
    private buffer = new Buffer(0);  
    private state = State.header;  
  
    append(buffer: Buffer): void {  
        this.buffer = Buffer.concat([this.buffer, buffer]);  
        this.parse();  
    }  
  
    private parse(): void { }  
  
    private parseHeader(): boolean { }  
  
    private parseBody(): boolean { }  
}
```

Due to the limitation of length, I'm not going to put the complete implementation of the parser here. For the complete code, please refer to the file `src/event-based-parser.ts` in the code bundle of *Chapter 7, Patterns and Architectures in JavaScript and TypeScript*.

Thus the use of such a parser could be as follows:

```
import * as Net from 'net';  
  
let parser = new Parser();  
let client = Net.connect(port);  
  
client.on('data', (data: Buffer) => {  
    parser.append(data);  
});  
  
parser.on('data', (data: any) => {  
    console.log('Data received:', data);  
});
```

Summary

In this chapter, we discussed some interesting ideas and an architecture formed by those ideas. Most of the topics focus on a small scope and do their own job, but there are also ideas about putting a whole system together.

The code that implements techniques such as expected error and the approach to managing modules in a project is not hard to apply. But with proper application, it can bring notable convenience to the entire project.

However, as I have already mentioned at the beginning of this chapter, there are too many beautiful things in JavaScript and TypeScript to be covered or even mentioned in a single chapter. Please don't stop here, and keep exploring.

Many patterns and architectures are the result of some fundamental principles in software engineering. Those principles might not always be applicable in every scenario, but they may help when you feel confused. In the next chapter, we are going to talk about SOLID principles in object-oriented design and find out how those principles may help form a useful pattern.

8

SOLID Principles

SOLID Principles are well-known Object-Oriented Design (OOD) principles summarized by Uncle Bob (Robert C. Martin). The word SOLID comes from the initials of the five principles it refers to, including **Single responsibility principle**, **Open-closed principle**, **Liskov substitution principle**, **Interface segregation principle** and **Dependency inversion principle**. Those principles are closely related to each other, and can be a great guidance in practice.

Here is a widely used summary of SOLID principles from Uncle Bob:

- **Single responsibility principle:** A class should have one, and only one, reason to change
- **Open-closed principle:** You should be able to extend a classes behavior, without modifying it
- **Liskov substitution principle:** Derived classes must be substitutable for their base classes
- **Interface segregation principle:** Make fine-grained interfaces that are client specific
- **Dependency inversion principle:** Depend on abstractions, not on concretions

In this chapter, we will walk through them and find out how those principles can help form a design that *smells* nice.

But before we proceed, I want to mention that a few of the reasons why those principles exist might be related to the age in which they were raised, the languages and their building or distributing process people were working with, and even computing resources. When being applied to JavaScript and TypeScript projects nowadays, some of the details may not be necessary. Think more about what problems those principles want to prevent people from getting into, rather than the literal descriptions of how a principle should be followed.

Single responsibility principle

The single responsibility principle declares that a class should have one, and only one reason to change. And the definition of the word *reason* in this sentence is important.

Example

Consider a `Command` class that is designed to work with both command-line interface and graphical user interface:

```
class Command {  
    environment: Environment;  
  
    print(items: ListItem[]) {  
        let stdout = this.environment.stdout;  
        stdout.write('Items:\n');  
        for (let item of items) {  
            stdout.write(item.text + '\n');  
        }  
    }  
    render(items: ListItem[]) {  
        let element = <List items={items}></List>;  
        this.environment.render(element);  
    }  
    execute() { }  
}
```

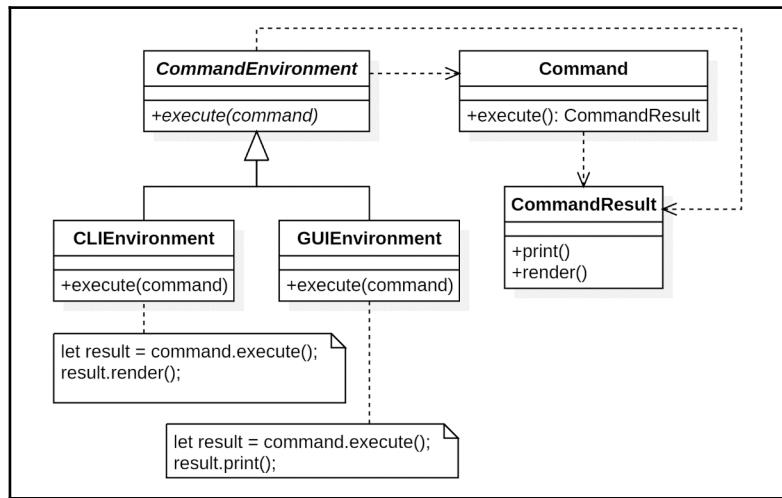
To make this actually work, `execute` method would need to handle both the command execution and result displaying:

```
class Command {  
    . . .  
    execute() {  
        let items = ...;  
        if (this.environment.type === 'cli') {  
            this.print(items);  
        } else {  
            this.render(items);  
        }  
    }  
}
```

In this example, there are two reasons for changes:

1. How a command gets executed.
2. How the result of a command gets displayed in different environments.

Those reasons lead to changes in different dimensions and violate the single responsibility principle. This might result in a messy situation over time. A better solution is to have those two responsibilities separated and managed by the CommandEnvironment:



Does this look familiar to you? Because it is a variant of the Visitor Pattern. Now it is the environment that executes a specific command and handles its result based on a concrete environment class.

Choosing an axis

You might be thinking, doesn't **CommandResult** violate the single responsibility principle by having the abilities to display content in a different environment? Yes, and no. When the axis of this reason is set to displaying content, it does not; but if the axis is set to displaying in a specific environment, it does. But take the overall structure into consideration, the result of a command is expected to be an output that can adapt to a different environment. And thus the reason is one-dimensional and confirms the principle.

Open-closed principle

The open-closed principle declares that you should be able to extend a class' behavior, without modifying it. This principle is raised by Bertrand Meyer in 1988:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

A program depends on all the entities it uses, that means changing the already-being-used part of those entities may just crash the entire program. So the idea of the open-closed principle is straightforward: we'd better have entities that never change in any way other than extending itself.

That means once a test is written and passing, ideally, it should never be changed for newly added features (and it needs to keep passing, of course). Again, ideally.

Example

Consider an API hub that handles HTTP requests to and responses from the server. We are going to have several files written as modules, including `http-client.ts`, `hub.ts` and `app.ts` (but we won't actually write `http-client.ts` in this example, you will need to use some imagination).

Save the code below as file `hub.ts`.

```
import { HttpClient, HttpResponseMessage } from './http-client';

export function update(): Promise<HttpResponseMessage> {
    let client = new HttpClient();
    return client.get('/api/update');
}
```

And save the code below as file `app.ts`.

```
import Hub from './hub';

Hub
    .update()
    .then(response => JSON.stringify(response.text))
    .then(result => {
        console.log(result);
    });
}
```

Bravely done! Now we have `app.ts` badly coupled with `http-client.ts`. And if we want to adapt this API hub to something like WebSocket, BANG.

So how can we create entities that are open for extension, but closed for modification? The key is a *stable abstraction that adapts*. Consider the storage and client example we took with Adapter Pattern in Chapter 4, *Structural Design Patterns* we had a `Storage` interface that isolates implementation of database operations from the client. And assuming that the interface is well-designed to meet upcoming feature requirements, it is possible that it will never change or just need to be extended during the life cycle of the program.

Abstraction in JavaScript and TypeScript

Guess what, our beloved JavaScript does not have an interface, and it is dynamically typed. We were not even able to actually write an interface. However, we could still write down documentation about the abstraction and create new concrete implementations just by obeying that description.

But TypeScript offers interface, and we can certainly take advantage of it. Consider the `CommandResult` class in the previous section. We were writing it as a concrete class, but it may have subclasses that override the `print` or `render` method for customized output. However, the type system in TypeScript cares only about the shape of a type. That means, while you are declaring an entity with type `CommandResult`, the entity does not need to be an instance of `CommandResult`: any object with a compatible type (namely has methods `print` and `render` with proper signatures in this case) will do the job.

For example, the following code is valid:

```
let environment: Environment;

let command: Command = {
  environment,
  print(items) { },
  render(items) { },
  execute() { }
};
```

Refactor earlier

I double stressed that the open-closed principle can only be perfectly followed under ideal scenarios. That can be a result of two reasons:

1. *Not all entities in a system can be open to extension and closed to modification at the same time.* There will always be changes that need to break the closure of existing entities to complete their functionalities. When we are designing the interfaces, we need different strategies for creating stable closures for different foreseeable situations. But this requires notable experience and no one can do it perfectly.
2. *None of us is too good at designing a program that lasts long and stays healthy forever.* Even with thorough consideration, abstractions designed at the beginning can be choppy facing the changing requirements.

So when we are expecting the entities to be closed for modification, it does not mean that we should just stand there and watch it being closed. Instead, when things are still under control, we should refactor and *keep the abstraction in the status of being open to extension and closed to modification* at the time point of refactoring.

Liskov substitution principle

The open-closed principle is the essential principle of keeping code maintainable and reusable. And the key to the open-closed principle is abstraction with polymorphism. Behaviors like implementing interfaces, or extending classes make polymorphic *shapes*, but that might not be enough.

The Liskov substitution principle declares that derived classes must be substitutable for their base classes. Or in the words of Barbara Liskov, who raised this principle:

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Never mind. Let's try another one: *any foreseeable usage of the instance of a class should be working with the instances of its derived classes.*

Example

And here we go with a straightforward violation example. Consider Noodles and InstantNoodles (a subclass of Noodles) to be cooked:

```
function cookNoodles(noodles: Noodles) {  
    if (noodles instanceof InstantNoodles) {  
        cookWithBoiledWaterAndBowl(noodles);  
    } else {  
        cookWithWaterAndBoiler(noodles);  
    }  
}
```

Now if we want to have some fried noodles... The `cookNoodles` function does not seem to be capable of handling that. Clearly, this violates the Liskov substitution principle, though it does not mean that it's a bad design.

Let's consider another example written by Uncle Bob in his article talking about this principle. We are creating class `Square` which is a subclass of `Rectangle`, but instead of adding new features, it adds a constraint to `Rectangle`: the width and height of a square should always be equal to each other. Assume we have a `Rectangle` class that allows its width and height to be set:

```
class Rectangle {  
    constructor()  
    private _width: number;  
    private _height: number;  
} {}  
set width(value: number) {  
    this._width = value;  
}  
set height(value: number) {  
    this._height = value;  
}  
}
```

Now we have a problem with its subclass `Square`, because it gets `width` and `height` setters from `Rectangle` while it shouldn't. We can certainly override those setters and make both of them update width and height simultaneously. But in some situations, the client might just not want that, because doing so will make the program harder to be predicted.

The `Square` and `Rectangle` example violates the Liskov substitution principle. Not because we didn't find a good way to inherit, but because `Square` does not conform the behavior of `Rectangle` and should not be a subclass of it at the beginning.

The constraints of substitution

Type is an important part in a programming language, even in JavaScript. But having the same *shape*, being on the same hierarchy does not mean they can be the substitution of another without some pain. More than just the *shape*, the complete behavior is what really matters for implementations that hold to the Liskov substitution principle.

Interface segregation principle

We've already discussed the important role played by abstractions in object-oriented design. The abstractions and their derived classes without separation usually come up with hierarchical tree structures. That means when you choose to create a branch, you create a parallel abstraction to all of those on another branch.

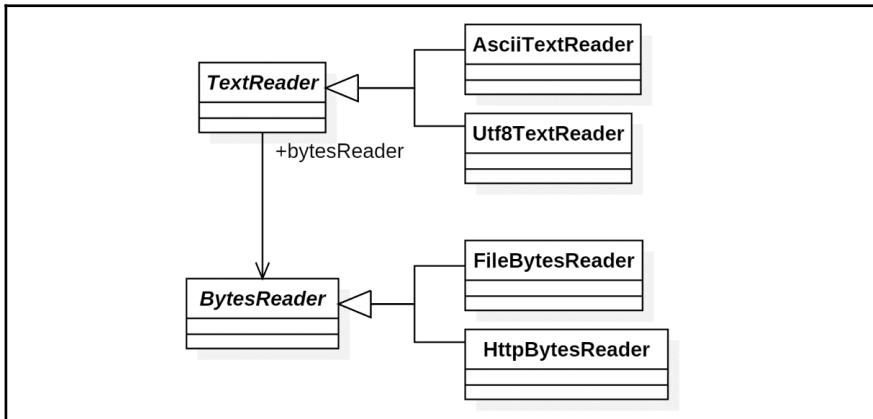
For a family of classes with only one level of inheritance, this is not a problem: because it is just what you want to have those classes derived from. But for a hierarchy with greater depth, it could be.

Example

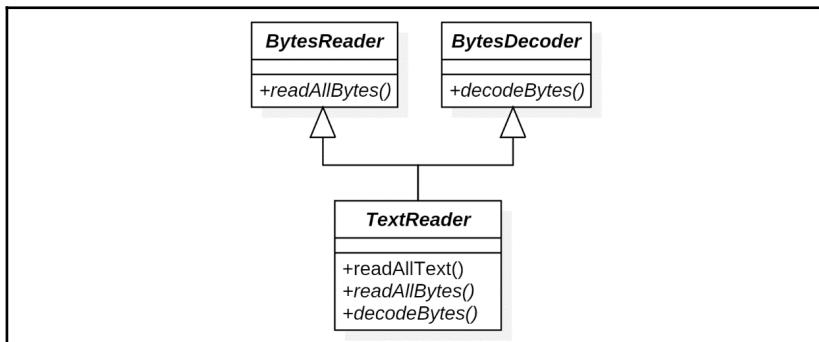
Consider the `TextReader` example we took with Template Method Pattern in Chapter 6, *Behavioral Design Patterns: Continuous* we had `FileAsciiTextReader` and `HttpAsciiTextReader` derived from `AsciiTextReader`. But what if we want to have other readers that understand UTF-8 encoding?

To achieve that goal, we have two common options: separate the interface into two for different objects that cooperate, or separate the interface into two then get them implemented by a single class.

For the first case, we can refactor the code with two abstractions, BytesReader and TextReader:



And for the second case, we can separate method `readAllBytes` and `decodeBytes` onto two interfaces, for example, BytesReader and BytesDecoder. Thus we may implement them separately and use techniques like mixin to put them together:



An interesting point about this example is that `TextReader` above itself is an abstract class. To make this mixin actually work, we need to create a concrete class of `TextReader` (without actually implementing `readAllBytes` and `decodeBytes`), and then mixin two concrete classes of `BytesReader` and `BytesDecoder`.

Proper granularity

It is said that by creating smaller interfaces, we can avoid a client from using big classes with features that it never needs. This may cause unnecessary usage of resources, but in practice, that usually won't be a problem. The most important part of the interface segregation principle is still about keeping code maintainable and reusable.

Then the question comes out again, how small should an interface be? I don't think I have a simple answer for that. But I am sure that being too small might not help.

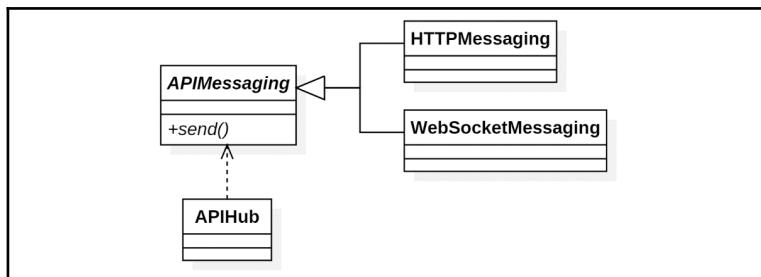
Dependency inversion principle

When we talk about dependencies, the natural sense is about dependencies from bottom to top, just like how buildings are built. But unlike a building that stands for tens of years with little change, software keeps changing during its life cycle. Every change costs, more or less.

The dependency inversion principle declares that entities should depend on abstractions, not on concretions. Higher level code should not depend directly on low-level implementations, instead, it should depend on abstractions that *lead to* those implementations. And this is why things are *inverse*.

Example

Still taking the HTTP client and API hub as an example, which obviously violates the dependency inversion principle, taking the foreseeable application into consideration, what the API hub should depend on is a messaging mechanism bridging client and server, but not bare HTTP client. This means we should have an abstraction layer of messaging before the concrete implementation of HTTP client:



Separating layers

Compared to other principles discussed in this chapter, the dependency inversion principle cares more about the scope of modules or packages. As the abstraction might usually be more stable than concrete implementations, by following dependency inversion principle, we can minimize the impact from low-level changes to higher level behaviors.

But for JavaScript (or TypeScript) projects as the language is dynamically typed, this principle is more about an idea of guidance that leads to a stable abstraction between different layers of code implementation.

Originally, an important benefit of following this principle is that, if modules or packages are relatively larger, separating them by abstraction could save a lot of time in compilation. But for JavaScript, we don't have to worry about that; and for TypeScript, we don't have to recompile the entire project for making changes to separated modules either.

Summary

In this chapter, we walked through the well-known SOLID principles with simple examples. Sometimes, following those principles could lead us to a useful design pattern. And we also found that those principles are strongly bound to each other. Usually violating one of them may indicate other violations.

Those principles could be extremely helpful for OOD, but could also be overkill if they are applied without proper adaptions. A well-designed system should have those principles confirmed just right, or it might harm.

In the next chapter, instead of theories, we'll have more time with a complete workflow with testing and continuous integration involved.

9

The Road to Enterprise Application

After walking through common design patterns, we have now the basis of code designing. However, software engineering is more about writing beautiful code. While we are trying to keep the code healthy and robust, we still have a lot to do to keep the project and the team healthy, robust, and ready to scale. In this chapter, we'll talk about popular elements in the workflow of web applications, and how to design a workflow that fits your team.

The first part would be setting up the build steps of our demo project. We'll quickly walk through how to build frontend projects with *webpack*, one of the most popular packaging tools these days. And we'll configure tests, code linter, and then set up continuous integration.

There are plenty of nice choices when it comes to workflow integration. Personally, I prefer Team Foundation Server for private projects or a combination of GitHub and Travis-CI for open-source projects. While Team Foundation Server (or Visual Studio Team Services as its cloud-based version) provides a one-stop solution for the entire application life cycle, the combination of GitHub and Travis-CI is more popular in the JavaScript community. In this chapter, we are going to use the services provided by GitHub and Travis-CI for our workflow.

Here are what we are going to walk through:

- Packaging frontend assets with *webpack*.
- Setting up tests and linter.
- Getting our hands on a Git flow branching model and other Git-related workflow.
- Connecting a GitHub repository with Travis-CI.
- A peek into automated deployment.

Creating an application

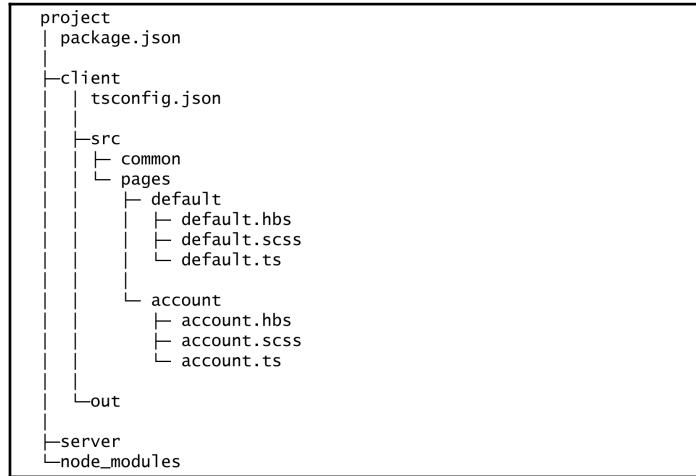
We've talked about creating TypeScript applications for both frontend and backend projects in the [Chapter 1, Tools and Frameworks](#). And now we are going to create an application that contains two TypeScript projects at the same time.

Decision between SPA and “normal” web applications

Applications for different purposes result in different choices. SPA (single page application) usually delivers a better user experience after being loaded, but it can also lead to trade-offs on SEO and may rely on more complex MV* frameworks like Angular.

One solution to build SEO-friendly SPA is to build a universal (or isomorphic) application that runs the *same* code on both frontend and backend, but that could introduce even more complexity. Or a reverse proxy could be configured to render automatically generated pages with the help of tools like *Phantom*.

In this demo project, we'll choose a more traditional web application with multiple pages to build. And here's the file structure of the client project:



Taking team collaboration into consideration

Before we actually start creating a real-world application, we need to come up with a reasonable application structure. A proper application structure is more than something under which the code compiles and runs. It should be a result, taking how your team members work together into consideration.

For example, a naming convention is involved in this demo client structure shown earlier: page assets are named after page names instead of their types (for example, `style.scss`) or names like `index.ts`. And the consideration behind this convention is making it more friendly for file navigation by the keyboard.

Of course, this consideration is valid only if a significant number of developers in your team are cool with keyboard navigation. Other than operation preferences, the experiences and backgrounds of a team should be seriously considered as well:

- Should the “full-stack” mode be enabled for your team?
- Should the “full-stack” mode be enabled for every engineer in your team?
- How should you divide work between frontend and backend?

Usually, it's not necessary and not efficient to limit the access of a frontend engineer to client-side development. If it's possible, frontend engineers could take over the controller layer of the backend and leave hardcore business models and logic to engineers that focus more on the backend.

We are having the client and server-side projects in the same repository for an easier integration during development. But it does not mean everything in the frontend or backend code base should be in this single repository. Instead, multiple modules could be extracted and maintained by different developers in practice. For example, you can have database models and business logic models separated from the controllers on the backend.

Building and testing projects

We have already talked about building and testing TypeScript projects at the beginning of this book. In this section, we will go a little bit further for frontend projects, including the basis of using Webpack to load static assets as well as **code linting**.

Static assets packaging with webpack

Modularizing helps code keep a healthy structure and makes it maintainable. However, it could lead to performance issues if development-time code written in *small* modules are directly deployed without bundling for production usage. So static assets packaging becomes a serious topic of frontend engineering.

Back to the old days, packaging JavaScript files was just about *uglifying* source code and concatenating files together. The project might be modularized as well, but in a *global* way. Then we have libraries like Require.js, with modules no longer automatically exposing themselves to the global scope.

But as I have mentioned, having the client download module files separately is not ideal for performance; soon we had tools like browserify, and later, webpack – one of the most popular frontend packaging tools these days.

Introduction to webpack

Webpack is an integrated packaging tool dedicated (at least at the beginning) to frontend projects. It is designed to package not only JavaScript, but also other static assets in a frontend project. Webpack provides built-in support for both **asynchronous module definition (AMD)** and commonjs, and can load ES6 or other types of resources via plugins.



ES6 module support will get built-in for webpack 2.0, but by the time this chapter is written, you still need plugins like `babel-loader` or `ts-loader` to make it work. And of course we are going to use `ts-loader` later.

To install webpack via `npm`, execute the following command:

```
$ npm install webpack -g
```

Bundling JavaScript

Before we actually use webpack to load TypeScript files, we'll have a quick walk through of bundling JavaScript.

First, let's create the file `index.js` under the directory `client/src/` with the following code inside:

```
var Foo = require('./foo');

Foo.test();
```

Then create the file `foo.js` in the same folder with the following content:

```
exports.test = function test() {
  console.log('Hello, Webpack!');
};
```

Now we can have them bundled as a single file using the webpack command-line interface:

```
$ webpack ./client/src/index.js ./client/out/bundle.js
```

By viewing the `bundle.js` file generated by webpack, you will see that the contents of both `index.js` and `foo.js` have been wrapped into that single file, together with the bootstrap code of webpack. Of course, we would prefer not to type those file paths in the command line every time, but to use a configuration file instead.

Webpack provides configuration file support in the form of JavaScript files, which makes it more flexible to generate necessary data like bundle entries automatically. Let's create a simple configuration file that does what the previous command did.

Create file `client/webpack.config.js` with the following lines:

```
'use strict';

const Path = require('path');

module.exports = {
  entry: './src/index',
  output: {
    path: Path.join(__dirname, 'out'),
    filename: 'bundle.js'
  }
};
```

These are the two things to mention:

1. The value of the `entry` field is not the filename, but the *module id* (most of the time this is unresolved) instead. This means that you can have the `.js` extension omitted, but have to prefix it with `./` or `../` by default when referencing a file.
2. The output path is required to be absolute. Building an absolute path with `__dirname` ensures it works properly if we are not executing webpack under the same directory as the configuration file.

Loading TypeScript

Now we are going to load and transpile our beloved TypeScript using the webpack plugin `ts-loader`. Before updating the configuration, let's install the necessary npm packages:

```
$ npm install typescript ts-loader --save-dev
```

If things go well, you should have the TypeScript compiler as well as the `ts-loader` plugin installed locally. We may also want to rename and update the files `index.js` and `foo.js` to TypeScript files.

Rename `index.js` to `index.ts` and update the module importing syntax:

```
import * as Foo from './foo';  
  
Foo.test();
```

Rename `foo.js` to `foo.ts` and update the module exporting syntax:

```
export function test() {  
  console.log('Hello, Webpack!');  
}
```

Of course, we would want to add the `tsconfig.json` file for those TypeScript files (in the folder `client`):

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs"  
  },  
  "exclude": [  
    "out",  
    "node_modules"  
  ]  
}
```



The compiler option `outDir` is omitted here because it is managed in the webpack configuration file.

To make webpack work with TypeScript via `ts-loader`, we'll need to tell webpack some information in the configuration file:

1. Webpack will need to resolve files with `.ts` extensions. Webpack has a default extensions list to resolve, including '' (empty string), `'.webpack.js'`, `'.web.js'`, and `'.js'`. We need to add `'.ts'` to this list for it to recognize TypeScript files.
2. Webpack will need to have `ts-loader` loading `.ts` modules because it does not compile TypeScript itself.

And here is the updated `webpack.config.js`:

```
'use strict';

const Path = require('path');

module.exports = {
  entry: './src/index',
  output: {
    path: Path.join(__dirname, 'bld'),
    filename: 'bundle.js'
  },
  resolve: {
    extensions: ['', '.webpack.js', '.web.js', '.ts', '.js']
  },
  module: {
    loaders: [
      { test: /\.ts$/, loader: 'ts-loader' }
    ]
  }
};
```

Now execute the command `webpack` under the `client` folder again, we should get the compiled and bundled output as expected.

During development, we can enable *transpile mode* (corresponding to the compiler option `isolatedModules`) of TypeScript to have better performance on compiling changing files. But it means we'll need to rely on an IDE or an editor to provide error hints. And remember to make another compilation with transpile mode disabled after debugging to ensure things still work.

To enable transpile mode, add a `ts` field (defined by the `ts-loader` plugin) with `transpileOnly` set to `true`:

```
module.exports = {
  ...
  ts: {
    transpileOnly: true
  }
};
```

Splitting code

To take the advantage of code caching across pages, we might want to split the packaged modules as common pieces. The webpack provides a built-in plugin called `CommonsChunkPlugin` that can pick out common modules and have them packed separately.

For example, if we create another file called `bar.ts` that imports `foo.ts` just like `index.ts` does, `foo.ts` can be treated as a common chunk and be packed separately:

```
module.exports = {
  entry: ['./src/index', './src/bar'],
  ...
  plugins: [
    new Webpack.optimize.CommonsChunkPlugin({
      name: 'common',
      filename: 'common.js'
    })
  ]
};
```

For multi-page applications, it is common to have different pages with different entry scripts. Instead of manually updating the `entry` field in the configuration file, we can take advantage of it being JavaScript and generate proper entries automatically. To do so, we might want the help of the npm package `glob` for matching page entries:

```
$ npm install glob --save-dev
```

And then update the webpack configuration file:

```
const glob = require('glob');

module.exports = {
  entry: glob
  .sync('./src/pages/**/*.ts')
```

```
.filter(path =>
  Path.basename(path, '.ts') ===
  Path.basename(Path.dirname(path))
),
...
};
```

Splitting the code can be rather a complex topic for deep dive, so we'll stop here and let you explore.

Loading other static assets

As we've mentioned, webpack can also be used to load other static assets like stylesheet and its extensions. For example, you can use the combination of `style-loader`, `css-loader` and `sass-loader/less-loader` to load `.sass/.less` files.

The configuration is similar to `ts-loader` so we'll not spend extra pages for their introductions. For more information, refer to the following URLs:

- Embedded stylesheets in webpack: <https://webpack.github.io/docs/stylesheets.html>
- SASS loader for webpack: <https://github.com/jtangelder/sass-loader>
- LESS loader for webpack: <https://github.com/webpack/less-loader>

Adding TSLint to projects

A consistent code style is an important factor of code quality, and linters are our best friends when it comes to code styles (and they also helps with common mistakes). For TypeScript linting, TSLint is currently the simplest choice.

The installation and configuration of TSLint are easy. To begin with, let's install `tslint` as a global command:

```
$ npm install tslint -g
```

And then we need to initialize a configuration file using the following command under the project root directory:

```
$ tslint --init
```

TSLint will then generate a default configuration file named `tslint.json`, and you may customize it based on your own preferences. And now we can use it to lint our TypeScript source code:

```
$ tslint */src/**/*.ts
```

Integrating webpack and tslint command with npm scripts

As we've mentioned before, an advantage of using npm scripts is that they can handle local packages with executables properly by adding `node_modules/.bin` to PATH. And to make our application easier to build and test for other developers, we can have `webpack` and `tslint` installed as development dependencies and add related scripts to `package.json`:

```
"scripts": {  
  "build-client": "cd client && webpack",  
  "build-server": "tsc --project server",  
  "build": "npm run build-client && npm run build-server",  
  "lint": "tslint ./src/**/*.ts",  
  "test-client": "cd client && mocha",  
  "test-server": "cd server && mocha",  
  "test": "npm run lint && npm run test-client && npm run test-server"  
}
```

Version control

Thinking back to my senior high school days, I knew nothing about version control tools. The best thing I could do was to create a daily archive of my code on a USB disk. And yes I did lose one!

Nowadays, with the boom of version control tools like Git and the availabilities of multiple free services like GitHub and Visual Studio Team Services, managing code with version control tools has become a daily basis for every developer.

As the most popular version control tool, Git has already been playing an important role in your work or personal projects. In this section, we'll talk about popular practices of using Git in a team.



Note that I am assuming that you already have the basic knowledge of Git, and know how to make operations like `init`, `commit`, `push`, `pull` and `merge`. If not, please get hands on and try to understand those operations before continue.



Check out this quick tutorial at: <https://try.github.io/>.

Git flow

Version control plays an important a role and it does not only influence the source code management process but also shapes the entire workflow of product development and delivery. Thus a *successful* branching model becomes a serious choice.

Git flow is a collection of Git extensions that provides high-level repository operations for a branching model raised by Vincent Driessen. The name *Git flow* usually refers to the branching model as well.

In this branching model, there are two main branches: `master` and `develop`, as well as three different types of supporting branches: `feature`, `hotfix`, and `release`.

With the help of Git flow extensions, we can easily apply this branching model without having to remember and type detailed sequences of commands. To install, please check out the installation guide of Git flow at: <https://github.com/nvie/gitflow/wiki/Installation>.

Before we can use Git flow to create and merge branches, we'll need to make an initialization:

```
$ git flow init -d
```

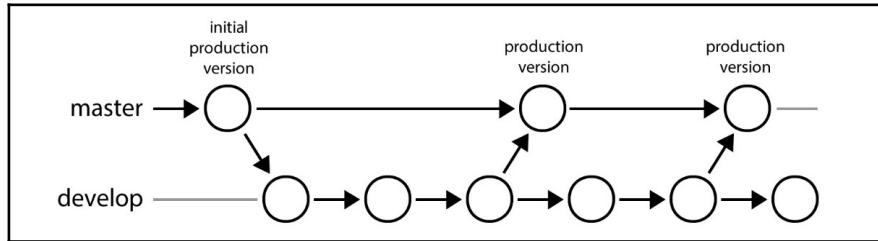


Here `-d` stands for using default branch naming conventions. If you would like to customize, you may omit the `-d` option and answer the questions about `git flow init` command.

This will create `master` and `develop` branches (if not present) and save Git flow-related configuration to the local repository.

Main branches

The branching model defines two main branches: `master` and `develop`. Those two branches exist in the lifetime of the current repository:



The graph in the preceding shows a simplified relationship between `develop` and `master` branches.



- **Branch master:** The `HEAD` of `master` branch should always contain production-ready source code. It means that no daily development is done on `master` branch in this branching model, and only commits that are fully tested and can be performed with a fast-forward should be merged into this branch.
- **Branch develop:** The `HEAD` of `develop` branch should contain delivered development source code. Changes to `develop` branch will finally be merged into `master`, but usually not directly. We'll come to that later when we talk about release branches.

Supporting branches

There are three types of supporting branches in the branching model of Git flow: `feature`, `hotfix`, and `release`. What they roughly do has already been suggested by their names, and we'll have more details to follow.

Feature branches

A feature branch has only direct interactions with the `develop` branch, which means it checks out from a `develop` branch and merges back to a `develop` branch. The feature branches might be the simplest type of branches out of the three.

To create a feature branch with Git flow, simply execute the following command:

```
$ git flow feature start <feature-name>
```

Now Git flow will automatically checkout a new branch named after `feature/<feature-name>`, and you are ready to start development and commit changes occasionally.

After completing feature development, Git flow can automatically merge things back to the `develop` branch by the following command:

```
$ git flow feature finish <feature-name>
```

A feature branch is usually started by the developer who is assigned to the development of that very feature and is merged by the developer him or herself, or the owners of the `develop` branch (for example, if code review is required).

Release branches

In a single iteration of a product, after finishing the development of features, we usually need a stage for fully testing everything, fixing bugs, and actually getting it ready to be released. And work for this stage will be done on release branches.

Unlike feature branches, a repository usually has only one active release branch at a time, and it is usually created by the owner of the repository. When the development branch is reaching a state of release and a thorough test is about to begin, we can then create a release branch using the following command:

```
$ git flow release start <version>
```

From now on, bug fixes that are going to be released in this iteration should be merged or committed to branch `release/<version>` and changes to the current `release` branch can be merged back to the `develop` branch anytime.

If the test goes well and important bugs have been fixed, we can then finish this release and put it online:

```
$ git flow release finish <version>
```

After executing this command, Git flow will merge the current release branch to both master and develop branches. So in a standard Git flow branching model, the develop branch will not be merged into the master directly, though after finishing a release, the content on develop and master branches could be identical (if no more changes are made to the develop branch during the releasing stage).



Finishing the current release usually means the end of the iteration, and the decision should be made with serious consideration.

Hotfix branches

Unfortunately, there's a phenomenon in the world of developers: bugs are always harder to find before the code goes live. After releasing, if serious bugs were found, we would have to use hotfixes to make things right.

A hotfix branch works kind of like a release branch but lasts shorter (because you would probably want it merged as soon as possible). Unlike feature branches being checked out from develop branch, a hotfix branch is checked out from master. And after getting things done, it should be merged back to both master and develop branches, just like a release branch does.

To create a hotfix branch, similarly you can execute the following command:

```
$ git flow hotfix start <hotfix-name>
```

And to finish, execute the following command:

```
$ git flow hotfix finish <hotfix-name>
```

Summary of Git flow

The most valuable idea in Git flow beside the branching model itself is, in my opinion, the clear outline of one iteration. You may not need to follow every step mentioned thus far to use Git flow, but just make it fit your work. For example, for small features that can be done in a single commit, you might not actually need a feature branch. But conversely, Git flow might not bring much value if the iteration itself gets chaotic.

Pull request based code review

Code review could be a very important joint of team cooperation. It ensures acceptable quality of the code itself and helps newcomers correct their misunderstanding of the project and accumulate experiences rapidly without taking a wrong path.

If you have tried to contribute code to open-source projects on GitHub, you must be familiar with pull requests or PR. There are actually tools or IDEs with code reviewing workflow built-in. But with GitHub and other self-hosted services like GitLab, we can get it done smoothly without relying on specific tools.

Configuring branch permissions

Restrictions on accessing specific branches like `master` and `develop` are not technically necessary. But without those restrictions, developers can easily skip code reviewing because they are just able to do so. In services provided by the Visual Studio Team Foundation Server, we may add a custom check in policy to force code review. But in lighter services like GitHub and GitLab, it might be harder to have similar functionality.

The easiest way might be to have developers who are more qualified and familiar with the current project have the permissions for writing the `develop` branch, and restrict code reviewing in this group verbally. For other developers working on this project, pull requests are now forced for getting changes they merged.



GitHub requires an organization account to specify push permissions for branches. Besides this, GitHub provides a status API and can add restrictions to merging so that only branches with a valid status can get merged.

Comments and modifications before merge

A great thing about those popular Git services is that the reviewer and maybe other colleagues of yours may comment on your pull requests or even specific lines of code to raise their concerns or suggestions. And accordingly, you can make modifications to the active pull request and make things a little bit closer to perfect.

Furthermore, references between issues and pull requests are shown in the conversation. This along with the comments and modification records makes the context of current pull requests clear and traceable.

Testing before commits

Ideally, we would expect every commit we make to pass tests and code linting. But because we are human, we can easily forget about running tests before committing changes. And then, if we have already set up continuous integration (we'll come to that shortly) of this project, pushing the changes would make it red. And if your colleague has set up a CI light with an alarm, you would make it flash and sound out.

To avoid breaking the build constantly, you might want to add a pre-commit hook to your local repository.

Git hooks

Git provides varieties of hooks corresponding to specific phases of an operation or an event. After initializing a Git repository, Git will create hook samples under the directory `.git/hooks`.

Now let's create the file `pre-commit` under the directory `.git/hooks` with the following content:

```
#!/bin/sh  
npm run test
```



The hook file does not have to be a bash file, and it can just be any executable. For example, if you want like to work with a Node.js hook, you can update the shebang as `#!/usr/bin/env node` and then write the hook in JavaScript.

And now Git will run tests before every commit of changes.

Adding pre-commit hook automatically

Adding hooks manually to the local repository could be trivial, but luckily we have npm packages like `pre-commit` that will add pre-commit hooks automatically when it's installed (as you usually might need to run `npm install` anyway).

To use the `pre-commit` package, just install it as a development dependency:

```
$ npm install pre-commit --save-dev
```

It will read your package.json and execute npm scripts listed with the field pre-commit or precommit:

```
{  
  ..  
  "script": {  
    "test": "istanbul cover ..."  
  },  
  "pre-commit": ["test"]  
}
```



At the time of writing, npm package pre-commit uses symbolic links to create Git hook, which requires administrator privileges on Windows. But failing to create a symbolic link won't stop the npm install command from completing. So if you are using Windows, you probably might want to ensure pre-commit is properly installed.

Continuous integration

The **continuous integration** (CI) refers to a practice of integrating multiple parts of a project or solution together regularly. Depending on the size of the project, the integration could be taken for every single change or on a timed schedule.

The main goal of continuous integration is to avoid integration issues, and it also enforces the discipline of frequent automated testing, this helps to find bugs earlier and prevents the degeneration of functionalities.

There are many solutions or services with continuous integration support. For example, self-hosted services like TFS and Jenkins, or cloud-based services like Visual Studio Team Services, Travis-CI, and AppVeyor. We are going to walk through the basic configuration of Travis-CI with our demo project.

Connecting GitHub repository with Travis-CI

We are going to use GitHub as the Git service behind continuous integration. First of all, let's get our GitHub repository and Travis-CI settings ready:

1. Create a correspondent repository as origin and push the local repository to GitHub:

```
$ git remote add origin https://github.com/<username>/<repo>.git
```

```
$ git push -u origin master
```

2. Sign into Travis-CI with your GitHub account at: <https://travis-ci.org/auth>.
3. Go to the account page, find the project we are working with, and then flick the repository switch on.

Now the only thing we need to make the continuous integration setup work is a proper Travis-CI configuration file. Travis-CI has built-in support for many languages and runtimes. It provides multiple versions of Node.js and makes it extremely easy to test Node.js projects.

Create the file `.travis.yml` in the root of project with the following content:

```
language: node_js
node_js:
  - "4"
  - "6"
before_script:
  - npm run build
```

This configuration file tells Travis-CI to test with both Node.js v4 and v6, and execute the command `npm run build` before testing (it will run the `npm test` command automatically).

Almost ready! Now add and commit the new `.travis.yml` file and push it to `origin`. If everything goes well, we should see Travis-CI start the build of this project shortly.



You might be seeing building status badges everywhere nowadays, and it's easy to add one to the `README.md` of your own project. In the project page on Travis-CI, you should see a badge next to the project name. Copy its URL and add it to the `README.md` as an image:

```
![building status] (https://api.travis-ci.org/<username>/<repo>.svg)
```

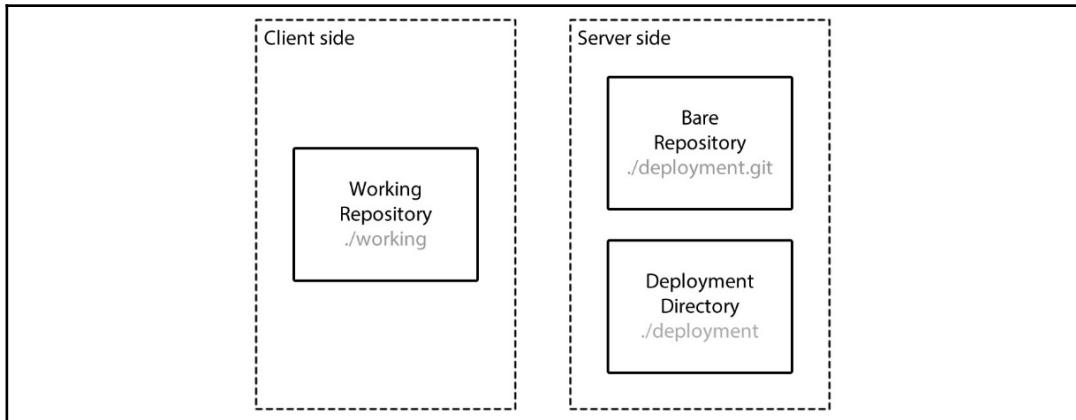
Deployment automation

Rather than a version control tool, Git is also popular for relatively simple deployment automation. And in this section, we'll get our hands on and configure automated deployment based on Git.

Passive deployment based on Git server side hooks

The idea of passive deployment is simple: when a client pushes commits to the bare repository on the server, a `post-receive` hook of Git will be triggered. And thus we can add scripts checking out changes and start deployment.

The elements involved in the Git deployment solution on both the client and server sides includes:



To make this mechanism work, we need to perform the following steps:

1. Create a bare repository on the server with the following command:

```
$ mkdir deployment.git
$ cd deployment.git
$ git init --bare
```



A bare repository usually has the extension `.git` and can be treated as a centralized place for sharing purposes. Unlike normal repositories, a bare repository does not have the working copy of source files, and its structure is quite similar to what's inside a `.git` directory of a normal repository.

2. Add `deployment.git` as a remote repository of our project, and try to push the `master` branch to the `deployment.git` repository:

```
$ cd ../demo-project
$ git remote add deployment ../deployment.git
$ git push -u deployment master
```



We are adding a local bare repository as the remote repository in this example. Extra steps might be required to create real remote repositories.

3. Add a post-receive hook for the `deployment.git` repository. We've already worked with the client side Git hook `pre-commit`, and the server side hooks work the same way.

But when it comes to a serious production deployment, how to write the hook could be a hard question to answer. For example, how do we minimize the impact of deploying new builds?

If we have set up our application with high availability load balancing, it might not be a big issue to have one of them offline for minutes. But certainly not all of them in this case. So here are some basic requirements of the deploy scripts on both the client and server sides:

- The deployment should be proceeded in a certain sequence
- The deployment should stop running services gently

And we can do better by:

- Building outside of the previous deployment directory
- Only trying to stop running services after the newly deployed application is ready to start immediately

Proactive deployment based on timers or notifications

Instead of using Git hooks, we can have other tools pull and build the application automatically as well. In this way, we no longer need the client to push changes to servers separately. And instead, the program on the server will pull changes from a remote repository and complete deployment.

A notification mechanism is preferred to avoid frequent fetching though, and there are already tools like PM2 that have automated deployment built-in. You can also consider building up your own using hooks provided by cloud-based or self-hosted Git services.

Summary

In this final chapter, we built the outline of a complete workflow starting with building and testing to continuous integration and automated deployment. We've covered some popular services or tools and provide other options for readers to discover and explore.

Among the varieties of choice, you might agree that the most appropriate workflow for your team is the workflow that fits the best. Taking people rather than technologies alone into consideration is an important part of software engineering, and it is also the key to keeping the team efficient (and happy, perhaps).

The sad thing about a team, or a crowd of people is that usually only a few of them can keep the passion burning. We've talked about finding the balance point, but that is what we still need to practice. And in most of the cases, expecting every one of your team to find the right point is just unreasonable. When it comes to team projects, we'd better have rules that can be validated automatically instead of conventions that are not testable.

After reading this book, I hope the reader gets the outlines of the build steps, workflow, and of course knowledge of common design patterns. But rather than the cold explanations of different terms and patterns, there are more important ideas I wanted to deliver:

- We as humans are dull, and should always keep our work divided as controllable pieces, instead of acting like a genius. And that's also why we need to *design* software to make our lives easier.
- And we are also unreliable, especially at a scale of some mass (like a team).
- As a learner, always try to understand the reason behind a conclusion or mechanism behind a phenomenon.

Module 3

TypeScript Blueprints

Build exciting end-to-end applications with TypeScript

Module 3: TypeScript Blueprints

Chapter 1: TypeScript 2.0 Fundamentals

What is TypeScript?	1
Quick example	2
Transpiling	3
Type checking	4
Learning modern JavaScript	4
let and const	5
Classes	6
Arrow functions	6
Function arguments	7
Array spread	8
Destructuring	8
Template strings	8
New classes	9
Type checking	9
Primitive types	9
Defining types	10
Undefined and null	11
Type annotations	11
Summary	12

Chapter 2: A Weather Forecast Widget with Angular 2

Using modules	14
Setting up the project	15
Directory structure	15
Configuring TypeScript	16
Building the system	16
The HTML file	18
Creating the first component	20
The template	21
Testing	21
Interactions	22
One-way variable binding	23
Event listeners	24
Adding conditions to the template	25

Directives	25
The template tag	25
Modifying the about template	26
Using the component in other components	26
Showing a forecast	27
Using the API	27
Typing the API	28
Creating the forecast component	28
Templates	30
Downloading the forecast	32
Adding @Output	35
The main component	40
Using our other components	40
Two-way bindings	40
Listening to our event	41
Geolocation API	41
Component sources	41
Summary	44
Chapter 3: Note-Taking App with a Server	45
Setting up the project structure	46
Directories	46
Configuring the build tool	46
Type definitions	48
Getting started with NodeJS	49
Asynchronous code	49
Callback approach for asynchronous code	50
Disadvantages of callbacks	51
The database	52
Wrapping functions in promises	52
Connecting to the database	53
Querying the database	54
Understanding the structural type system	55
Generics	55
Typing the API	58
Adding authentication	58
Implementing users in the database	60
Adding users to the database	61
Testing the API	62
Adding CRUD operations	63
Implementing the handlers	64

Request handling	66
Writing the client side	66
Creating the login form	68
Creating a menu	70
The note editor	71
The main component	71
Error handler	73
Running the application	75
Summary	75
Chapter 4: Real-Time Chat	76
Setting up the project	77
Configuring gulp	78
Getting started with React	79
Creating a component with JSX	79
Adding props and state to a component	80
Creating the menu	81
Testing the application	84
Writing the server	84
Connections	84
Typing the API	84
Accepting connections	85
Storing recent messages	86
Handling a session	87
Implementing a chat message session	88
Connecting to the server	90
Automatic reconnecting	90
Sending a message to the server	92
Writing the event handler	93
Creating the chat room	94
Two-way bindings	94
Stateless functional components	95
Running the application	96
Comparing React and Angular	96
Templates and JSX	96
Libraries or frameworks	97
Summary	98
Chapter 5: Native QR Scanner App	99
Getting started with NativeScript	100

Creating the project structure	101
Adding TypeScript	102
Creating a Hello World page	103
Creating the main view	105
Adding a details view	109
Scanning QR codes	113
Type definitions	113
Implementation	113
Testing on a device	115
Adding persistent storage	116
Styling the app	117
Comparing NativeScript to alternatives	119
Summary	120
Chapter 6: Advanced Programming in TypeScript	121
Using type guards	121
Narrowing	122
Narrowing any	123
Combining type guards	123
More accurate type guards	124
Assignments	125
Checking null and undefined	126
Guard against null and undefined	126
The never type	127
Creating tagged union types	127
Comparing performance of algorithms	128
Big-Oh notation	129
Optimizing algorithms	130
Binary search	131
Built-in functions	132
Summary	133

Chapter 7: Spreadsheet Applications with Functional Programming	134
Setting up the project	135
Functional programming	137
Calculating a factorial	138
Using data types for expressions	139
Creating data types	139
Traversing data types	141
Validating an expression	143

Calculating expressions	145
Parsing an expression	147
Creating core parsers	147
Running parsers in a sequence	148
Parsing a number	151
Order of operations	152
Defining the sheet	155
Calculating all fields	156
Using the Flux architecture	158
Defining the state	158
Creating the store and dispatcher	159
Creating actions	160
Adding a column or a row	161
Changing the title	162
Showing the input popup	163
Testing actions	165
Writing the view	166
Rendering the grid	166
Rendering a field	168
Showing the popup	170
Adding styles	172
Gluing everything together	174
Advantages of Flux	175
Going cross-platform	175
Summary	176
Chapter 8: Pac Man in HTML5	177
Setting up the project	178
Using the HTML5 canvas	179
Saving and restoring the state	180
Designing the framework	181
Creating pictures	182
Wrapping other pictures	185
Creating events	187
Binding everything together	188
Drawing on the canvas	190
Adding utility functions	192
Creating the models	193
Using enums	194
Storing the level	195

Creating the default level	196
Creating the state	199
Drawing the view	200
Handling events	203
Working with key codes	203
Creating the time handler	205
Running the game	209
Adding a menu	210
Changing the model	210
Rendering the menu	212
Handling events	213
Modifying the time handler	214
Summary	215
Chapter 9: Playing Tic-Tac-Toe against an AI	216
Creating the project structure	218
Configure TypeScript	218
Adding utility functions	219
Creating the models	219
Showing the grid	220
Creating operations on the grid	221
Creating the grid	225
Adding tests	226
Random testing	228
Implementing the AI using Minimax	230
Implementing Minimax in TypeScript	231
Optimizing the algorithm	232
Creating the interface	233
Handling interaction	233
Creating players	236
Testing the AI	238
Testing with a random player	239
Summary	240
Chapter 10: Migrate JavaScript to TypeScript	241
Gradually migrating to TypeScript	241
Adding TypeScript	242
Configuring TypeScript	242
Configuring the build tool	244
Acquiring type definitions	245

Testing the project	245
Migrating each file	245
Converting to ES modules	245
Correcting types	247
Adding type guards and casts	248
Using modern syntax	249
Adding types	250
Refactoring the project	250
Enable strict checks	251
Summary	251
Index	252

1

TypeScript 2.0 Fundamentals

In Chapters 2 through 5, we will learn a few frameworks to create (web) applications with TypeScript. First you need some basic knowledge of **TypeScript 2.0**. If you have used TypeScript previously, then you can skim over this chapter, or use it as a reference while reading the other chapters. If you have not used TypeScript yet, then this chapter will teach you the fundamentals of TypeScript.

What is TypeScript?

The TypeScript language looks like JavaScript; it is JavaScript with type annotations added to it. The TypeScript compiler has two main features: it is a **transpiler** and a **type checker**. A transpiler is a special form of compiler that outputs source code. In case of the TypeScript compiler, TypeScript source code is compiled to JavaScript code. A type checker searches for contradictions in your code. For instance, if you assign a string to a variable, and then use it as a number, you will get a type error.

The compiler can figure out some types without type annotations; for others you have to add type annotations. An additional advantage of these types is that they can also be used in editors. An editor can provide completions and refactoring based on the type information. Editors such as Visual Studio Code and Atom (with a plugin, namely atom-typescript) provide such features.

Quick example

The following example code shows some basic TypeScript usage. If you understand this code, you have enough knowledge for the next chapters. This example code creates an input box in which you can enter a name. When you click on the button, you will see a personalized greeting:

```
class Hello {  
    private element: HTMLDivElement;  
    private elementInput: HTMLInputElement;  
    private elementText: HTMLDivElement;  
    constructor(defaultName: string) {  
        this.element = document.createElement("div");  
        this.elementInput = document.createElement("input");  
        this.elementText = document.createElement("div");  
        const elementButton = document.createElement("button");  
  
        elementButton.textContent = "Greet";  
  
        this.element.appendChild(this.elementInput);  
        this.element.appendChild(elementButton);  
        this.element.appendChild(this.elementText);  
  
        this.elementInput.value = defaultName;  
        this.greet();  
  
        elementButton.addEventListener("click",  
            () => this.greet()  
        );  
    }  
  
    show(parent: HTMLElement) {  
        parent.appendChild(this.element);  
    }  
  
    greet() {  
        this.elementText.textContent = `Hello,  
        ${this.elementInput.value}!`;  
    }  
}  
  
const hello = new Hello("World");  
hello.show(document.body);
```

The preceding code creates a class, `Hello`. The class has three properties that contain an HTML element. We create these elements in the constructor. TypeScript has different types for all HTML elements and `document.createElement` gives the corresponding element type. If you replace `div` with `span` (on the first line of the constructor), you would get a type error saying that type `HTMLSpanElement` is not assignable to type `HTMLDivElement`. The class has two functions: one to add the element to the HTML page and one to update the greeting based on the entered name.

It is not necessary to specify types for all variables. The types of the variables `elementButton` and `hello` can be inferred by the compiler.

You can see this example in action by creating a new directory and saving the file as `scripts.ts`. In `index.html`, you must add the following code:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <script src="scripts.js"></script>
  </body>
</html>
```

The TypeScript compiler runs on NodeJS, which can be installed from <https://nodejs.org>. Afterward, you can install the TypeScript compiler by running `npm install typescript -g` in a console/terminal. You can compile the source file by running `tsc scripts.ts`. This will create the `scripts.js` file. Open `index.html` in a browser to see the result.

The next sections explain the basics of TypeScript in more detail. After reading those sections, you should understand this example fully.

Transpiling

The compiler transpiles TypeScript to JavaScript. It does the following transformations on your source code:

- Remove all type annotations
- Compile new JavaScript features for old versions of JavaScript
- Compile TypeScript features that are not standard JavaScript

We can see the preceding three transformations in action in the next example:

```
enum Direction {  
    Left,  
    Right,  
    Up,  
    Down  
}  
let x: Direction = Direction.Left;
```

TypeScript compiles this to the following:

```
var Direction;  
(function (Direction) {  
    Direction[Direction["Left"] = 0] = "Left";  
    Direction[Direction["Right"] = 1] = "Right";  
    Direction[Direction["Up"] = 2] = "Up";  
    Direction[Direction["Down"] = 3] = "Down";  
})(Direction || (Direction = {}));  
var x = Direction.Left;
```

In the last line, you can see that the type annotation was removed. You can also see that `let` was replaced by `var`, since `let` is not supported in older versions of JavaScript. The `enum` declaration, which is not standard JavaScript, was transpiled to normal JavaScript.

Type checking

The most important feature of TypeScript is type checking. For instance, for the following code, it will report that you cannot assign a number to a string:

```
let x: string = 4;
```

In the next sections, you will learn the new features of the latest JavaScript versions. Afterward, we will discuss the basics of the type checker.

Learning modern JavaScript

JavaScript has different versions. Some of these are **ES3**, **ES5**, **ES2015** (also known as **ES6**), and **ES2016**. Recent versions are named after the year in which they were introduced.

Depending on the environment for which you write code, some features might be or might not be supported. TypeScript can compile new features of JavaScript to an older version of JavaScript. That is not possible with all features, however.

Recent web browsers support ES5 and they are working on ES2015.

We will first take a look at the constructs that can be transpiled to older versions.

let and const

ES2015 has introduced `let` and `const`. These keywords are alternatives to `var`. These prevent issues with scoping, as `let` and `const` are block scoped instead of function scoped. You can use such variables only within the block in which they were created. It is not allowed to use such variables outside of that block or before its definition. The following example illustrates some dangerous behavior that could be prevented with `let` and `const`:

```
alert(x.substring(1, 2));
var x = "lorem";
for (var i = 0; i < 10; i++) {
    setTimeout(function() {
        alert(i);
    }, 10 * i);
}
```

The first two lines give no error, as a variable declared with `var` can be used before its definition. With `let` or `const`, you will get an error, as expected.

The second part shows 10 message boxes saying 10. We would expect 10 messages saying 0, 1, 2, and so on up to 9. But, when the callback is executed and `alert` is called, `i` is already 10, so you see 10 messages saying 10.

When you change the `var` keywords to `let`, you will get an error in the first line and the messages work as expected. The variable `i` is bound to the loop body. For each iteration, it will have a different value. The for loop is transpiled as follows:

```
var _loop_1 = function(i) {
    setTimeout(function () {
        alert(i);
    }, 10 * i);
};
for (var i = 0; i < 10; i++) {
    _loop_1(i);
}
```

A variable declared with `const` cannot be reassigned, and a variable with `let` can be reassigned. If you reassign a `const` variable, you get a compile error.

Classes

As of ES2015, you can create classes easily. In older versions, you could simulate classes to a certain extent. TypeScript transpiles a class declaration to the old way to simulate a class:

```
class Person {  
    age: number;  
    constructor(public name: string) {  
    }  
    greet() {  
        console.log("Hello, " + this.name);  
    }  
}  
  
const person = new Person("World");  
person.age = 35;  
person.greet();
```

This example is transpiled to the following:

```
var Person = (function () {  
    function Person(name) {  
        this.name = name;  
    }  
    Person.prototype.greet = function () {  
        console.log("Hello, " + this.name);  
    };  
    return Person;  
}());  
var person = new Person("World");  
person.age = 35;  
person.greet();
```

When you prefix an argument of the constructor with public or private, it is added as a property of the class. Other properties must be declared in the body of the class. This is not per the JavaScript specification, but needed with TypeScript for type information.

Arrow functions

ES6 introduced a new way to create functions. **Arrow functions** are function expressions defined using `=>`. Such function looks like the following:

```
(x: number, y: boolean): string => {  
    statements  
}
```

The function expression starts with an argument list, followed by an optional return type, the arrow (`=>`), and then a block with statements. If the function has only one argument without type annotation and no return type annotation, you may omit the parenthesis: `x => { ... }`. If the body contains only one return statement, without any other statements, you can simplify it to `(x: number, y: number) => expression`. A function with one argument and only a return statement can be simplified to `x => expression`.

Besides the short syntax, arrow functions have one other major difference with normal functions. Arrow functions share the value of `this` and the position where it was defined; `this` is lexically bound. Previously, you would store the value of `this` in a variable called `_this` or `self`, or you would fix the value using `.bind(this)`. With arrow functions, that is not required any more.

Function arguments

It is possible to add a default value to an argument:

```
function sum(a = 0, b = 0, c = 0) {
    return a + b + c;
}
sum(10, 5);
```

When you call this function with less than three arguments, it will set the other arguments to 0. TypeScript will automatically infer the types of `a`, `b`, and `c` based on their default values, so you do not have to add a type annotation there.

You can also define an optional argument without a default value: `function a(x?: number) {}`. The argument will then be undefined when it is not provided. This is not standard JavaScript, but only available in TypeScript.

The `sum` function can be defined even better, with a **rest argument**. At the end of a function, you can add a rest argument:

```
function sum(...xs: number[]) {
    let total = 0;
    for (let i = 0; i < xs.length; i++) total += xs[i];
    return total;
}
sum(10, 5, 2, 1);
```

Array spread

It is easier to create arrays in ES6. You can create an array literal (with brackets), in which you use another array. In the following example, you can see how you can add an item to a list and how you can concatenate two lists:

```
const a = [0, 1, 2];
const b = [...a, 3];
const c = [...a, ...b];
```

A similar feature for object literals will probably be added to JavaScript too.

Destructuring

With **destructuring**, you can easily create variables for properties of an object or elements of an array:

```
const a = { x: 1, y: 2, z: 3 };
const b = [4, 5, 6];

const { x, y, z } = a;
const [u, v, w] = b;
```

The preceding is transpiled to the following:

```
var a = { x: 1, y: 2, z: 3 };
var b = [4, 5, 6];

var x = a.x, y = a.y, z = a.z;
var u = b[0], v = b[1], w = b[2];
```

You can use destructuring in an assignment, variable declaration, or argument of a function header.

Template strings

With template strings, you can easily create a string with expressions in it. If you would write "Hello, " + name + "!", you can now write Hello \${ name }!.

New classes

ES2015 has introduced some new classes, including Map, Set, WeakMap, WeakSet, and Promise. In modern browsers, these classes are already available. For other environments, TypeScript does not automatically add a fallback for these classes. Instead, you should use a **polyfill**, such as es6-shim. Most browsers already support these classes, so in most cases, you do not need a polyfill. You can find information on browser support at <http://caniuse.com>.

Type checking

The compiler will check the types of your code. It has several primitive types and you can define new types yourself. Based on these types, the compiler will warn when a value of a type is used in an invalid manner. That could be using a string for multiplication or using a property of an object that does not exist. The following code would show these errors:

```
let x = "foo";  
x * 2;  
x.bar();
```

TypeScript has a special type, called `any`, that allows everything; you can assign every value to it and you will never get type errors. The type `any` can be used if you do not have an exact type (yet), for instance, because it is a complex type or if it is from a library that was not written in TypeScript. This means that the following code gives no compile errors:

```
let x: any = "foo";  
x * 2;  
x.bar();
```

In the next sections, we will discover these types and learn how the compiler finds these types.

Primitive types

TypeScript has several primitive types, which are listed in the following table:

Name	Values	Example
boolean	true, false	let x: boolean = true;
string	Any string literal	let x: string = "foo";

number	Any number, including Infinity, -Infinity, and NaN	let x: number = 42; let y: number = NaN;
Literal types	Literal types can only contain one value	let x: "foo" = "foo";
void	Only used for a function that does not return a value	function a(): void { }
never	No values	
any	All values	let x: any = "foo"; let y: any = true;

Defining types

You can define your own types in various ways:

Kind	Meaning	Example
Object type	Represents an object, with the specified properties. Properties marked with ? are optional. Objects can also have an indexer (for example, like an array), or call signatures. Object types can be defined inline, with a class or with an interface declaration.	let x: { a: boolean, b: string, c?: number, [i: number]: string }; x = { a: true, b: "foo" }; x[0] = "foo";
Union type	A value is assignable to a union type if it is assignable to one of the specified types. In the example, it should be a string or a number.	let x: string number; x = "foo"; x = 42;
Intersection type	A value is assignable to an intersection type if it is assignable to all specified types.	let x: { a: string } & { b: number } = { a: "foo", b: 42 };
Enum type	A special number type, with several values declared. The declared members get a value automatically, but you can also specify a value.	enum E { X, Y = 100 } let a: E = E.X;

Function type	Represents a function with the specified arguments and return type. Optional and rest arguments can also be specified.	let f: (x: string, y?: boolean) => number; let g: (...xs: number[]) => number;
Tuple type	Multiple values are placed in one, as an array.	let x: [string, number]; x = ["foo", 42];

Undefined and null

By default, `undefined` and `null` can be assigned to every type. Thus, the compiler cannot give you a warning when a value can possibly be `undefined` or `null`. TypeScript 2.0 has introduced a new mode, called `strictNullChecks`, which adds two new types: `undefined` and `null`. With that mode, you do get warnings in such cases. We will discover that mode in Chapter 6, *Advanced Programming in TypeScript*.

Type annotations

TypeScript can infer some types. This means that the TypeScript compiler knows the type, without a type annotation. If a type cannot be inferred, it will default to `any`. In such a case, or in case the inferred type is not correct, you have to specify the types yourself. The common declarations that you can annotate are given in the following table:

Location	Can it be inferred?	Examples
Variable declaration	Yes, based on initializer	let a: number; let b = 1;
Function argument	Yes, based on default value (second example) or when passing the function to a typed variable or function (third example)	function a(x: number) {} function b(x = 1) {} [1, 2].map(x => x * 2) ;
Function return type	Yes, based on return statements in body	function a(): number {} (): number => {} function c() { return 1; }

Class member	Yes, based on default value	<pre>class A { x: number; y = 0; }</pre>
Interface member	No	<pre>interface A { x: number; }</pre>

You can set the compiler option `noImplicitAny` to get compiler errors when a type could not be inferred and falls back to `any`. It is advised to use that option always, unless you are migrating a JavaScript codebase to TypeScript. You can read about such migration in Chapter 10, *Migrate JavaScript to TypeScript*.

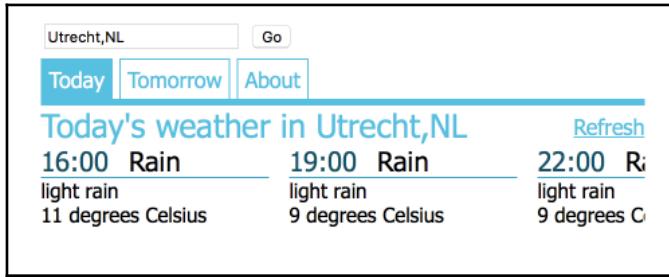
Summary

In this chapter, you discovered the basics of TypeScript. You should now be familiar with the principles of TypeScript and you should understand the code example at the beginning of the chapter. You now have the knowledge to start with the next chapters, in which you will learn two major web frameworks, Angular 2 and React. We will start with Angular 2 in Chapter 2, *A Weather Forecast Widget with Angular 2*.

2

A Weather Forecast Widget with Angular 2

In this chapter, we'll create a simple application that shows us the weather forecast. The framework we use, Angular 2, is a new framework written by Google in TypeScript. The application will show the weather of the current day and the next. In the following screenshot, you can see the result. We will explore some key concepts of Angular, such as data binding and directives.



We will build the application in the following steps:

- Using modules
- Setting up the project
- Creating the first component
- Adding conditions to the template
- Showing a forecast
- Creating the forecast components
- The main component

Using modules

We will use **modules** in all applications in this book. Modules (also called external modules and ES2015 modules) are a concept of separating code in multiple files. Every file is a module. Within these modules, you can use variables, functions, and classes (members) exported by other modules and you can make some members visible for other modules. To use other modules, you must import them, and to make members visible, you need to export them. The following example will show some basic usage:

```
// x.ts
import { one, add, Lorem } from './y';
console.log(add(one, 2));

var lorem = new Lorem();
console.log(lorem.name);

// y.ts
export var one = 1;
export function add(a: number, b: number) {
    return a + b;
}
export class Lorem {
    name = "ipsum";
}
```

You can **export** declarations by prefixing them with the `export` keyword or by prefixing them with `export default`. A default export should be imported differently though we will not use such an export as it can be confusing. There are various ways to **import** a file. We have seen the variant that is used most times, `import { a, b, c } from './d'`. The dot and slash mean that the `d.ts` file is located in the same directory. You can use `./x/y` and `../z` to reference a file in a subdirectory or a parent directory. A reference that does not start with a dot can be used to import a library, such as Angular. Another import variant is `import * as e from './d'`. This will import all exports from `d.ts`. These are available as `e.a, e.b, e` is an object that contains all exports.

To keep code readable and maintainable, it is advisable to use multiple small files instead of one big file.

Setting up the project

We will quickly set up the project before we can start writing. We will use npm to manage our dependencies and gulp to build our project. These tools are built on NodeJS, so it should be installed from nodejs.org.

First of all, we must create a new directory in which we will place all files. We must create a package.json file used by npm:

```
{  
  "name": "weather-widget",  
  "version": "1.0.0",  
  "private": true,  
  "description": ""  
}
```

The package.json file contains information about the project, such as the name, version, and a description. These fields are used by npm when you publish a project on the registry on NPM, which contains a lot of open source projects. We will not publish it there. We set the private field to true, so we cannot accidentally publish it.

Directory structure

We will separate the TypeScript sources from the other files. The TypeScript files will be added in the lib directory. Static files, such as HTML and CSS, will be located in the static directory. This directory can be uploaded to a webserver. The compiled sources will be written to static/scripts. We first install Angular and some requirements of Angular with npm. In a terminal, we run the following command in the root directory of the project:

```
npm install angular2 rxjs es6-shim reflect-metadata zone.js --save
```

The console might show some warnings about unmet peer dependencies. These will probably be caused by a minor version mismatch between Angular and one of its dependencies. You can ignore these warnings.

Configuring TypeScript

TypeScript can be configured using a `tsconfig.json` file. We will place that file in the `lib` directory, as all our files are located there. We specify the `experimentalDecorators` and `emitDecoratorMetadata` options, as these are necessary for Angular:

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "lib": ["es2015", "dom"]  
  }  
}
```

The `target` option specifies the version of JavaScript of the generated code. Current browsers support `es5`. TypeScript will compile newer JavaScript features, such as classes, to an `es5` equivalent. With the `lib` option, we can specify the version of the JavaScript library. We use the libraries from `es2015`, the version after `es5`. Since these libraries might not be available in all browsers, we will add a polyfill for these features later on. We also include the libraries for the **DOM**, which contains functions such as `document.createElement` and `document.getElementById`.

Building the system

With `gulp`, it is easy to compile a program in multiple steps. For most webapps, multiple steps are needed: compiling TypeScript, bundling modules, and finally minifying all code. In this application, we need to do all of these steps.

Gulp streams source files through a series of plugins. These plugins can (just like `gulp` itself) be installed using `npm`:

```
npm install gulp --global  
npm install gulp gulp-typescript gulp-sourcemaps gulp-uglify small --save-dev
```



The `--global` flag will install the dependency globally such that you can call `gulp` from a terminal. The `--save-dev` flag will add the dependency to the `devDependencies` (development dependencies) section of the `package.json` file. Use `--save` to add a runtime dependency.

We use the following plugins for gulp:

- The `gulp-typescript` plugin compiles TypeScript to JavaScript
- The `gulp-uglify` plugin can minify JavaScript files
- The `small` plugin can bundle external modules
- The `gulp-sourcemaps` plugin improves the debugging experience with source maps

We will create two tasks, one that compiles the sources to a development build and another that can create a release build. The development build will have source maps and will not be minified, whereas the release build will be minified without source maps. Minifying takes some time so we do not do that on the debug task. Creating source maps in the release task is possible too, but generating the source map is slow so we will not do that.

We write these tasks in `gulpfile.js` in the root of the project. The second task is the easiest to write, as it only uses one plugin. The task will look like this:

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

gulp.task('release', ['compile'], function() {
  return gulp.src('static/scripts/scripts.js')
    .pipe(uglify())
    .pipe(gulp.dest('static/scripts'));
});
```

The `gulp.task` call will register a task named `release`, which will take `static/scripts/scripts.js` (which will be created by the `compile` task), run `uglify` (a tool that minifies JavaScript) on it, and then save it in the same directory again. This task depends on the `compile` task, meaning that the `compile` task will be run before this one.

The first task, `compile`, is more complicated. The task will transpile TypeScript, and bundle the files with the external libraries.

First, we must load some plugins:

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

var typescript = require('gulp-typescript');var small =
require('small').gulp;var sourcemaps = require('gulp-sourcemaps');

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

We load the configuration of TypeScript in the `tsconfig.json` file:

```
var tsProject = typescript.createProject('lib/tsconfig.json');
```

Now, we can finally write the task. First, we load all sources and compile them using the TypeScript compiler. After that, we bundle these files (including Angular, stored under `node_modules`, using `small`):

```
gulp.task('compile', function() {
    return gulp.src('lib/**/*.ts')
        .pipe(sourcemaps.init())
        .pipe(typescript(tsProject))
        .pipe(small('index.js', {
            externalResolve: ['node_modules'],
            globalModules: {
                "crypto": {
                    standalone: "undefined"
                }
            }
        }))
        .pipe(sourcemaps.write('.'))
        .pipe(gulp.dest('static/scripts'));
});

gulp.task('release', ['compile'], function() {
    return gulp.src('static/scripts/scripts.js')
        .pipe(uglify())
        .pipe(gulp.dest('static/scripts'));
});

gulp.task('default', ['compile']);
```

This task compiles our project and saves the result as `static/scripts/scripts.js`. The `sourcemaps.init()` and `sourcemaps.write('.')` functions handle the creation of source maps, which will improve the debugging experience.

The HTML file

The main file of our application is the HTML file, `static/index.html`. This file will reference our (compiled) scripts and stylesheet:

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Weather</title>
        <link rel="stylesheet" href="style.css" />
    </head>
```

```
<body>
  <div id="wrapper">
    <weather-widget>Loading..</weather-widget>
  </div>
  <script src="scripts/index.js" type="text/javascript"></script>
</body>
</html>
```

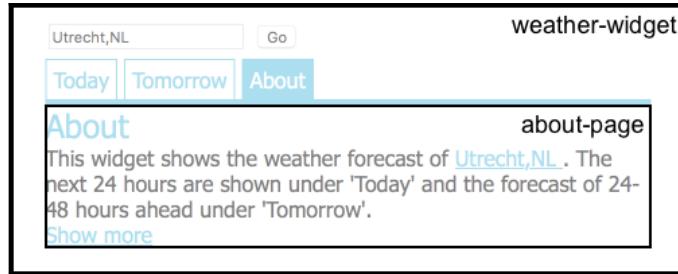
The `weather-widget` tag will be initialized by Angular. We will add some fancy styles in `static/style.css`:

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  font-weight: 100;
}
h1, h2, h3 {
  font-weight: 100;
  margin: 0;
  padding: 0;
  color: #57BEDE;
}
#wrapper {
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  width: 450px;
  margin: 10% auto;
}
a:link, a:visited {
  color: #57BEDE;
  text-decoration: underline;
}
a:hover, a:active {
  color: #44A4C2;
}
.clearfix {
  clear: both;
}
```

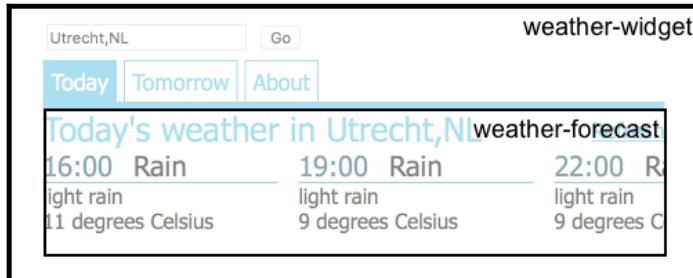
Creating the first component

Angular is based on components. Components are built with other components and normal HTML tags. Our application will have three components: the forecast page, the about page, and the whole widget. The widget itself, which is referenced in the HTML page, will use the other two widgets.

The widget will show the **About** page in the third tab, as you can see in the following screenshot:



The forecast component is shown in the first tab of the following screenshot. We will create the forecast and the widget later in this chapter.



The template

A component is a class decorated with some metadata. **Decorators** are functions that can modify a class or decorate it with some metadata. A simple component that does not have any interaction will look like this:

```
import { Component } from "angular2/core";  
  
@Component({  
  selector: "about-page",  
  template: `  
    <h2>About</h2>  
    This widget shows the weather forecast of Utrecht.  
    The next 24 hours are shown under 'Today' and the forecast of  
    24-48 hours ahead under 'Tomorrow'.  
`  
})  
export class About {  
}
```



As a convention, you can always choose selector names with a dash (-). You can then identify components by the dash. Normal HTML tags will never have names with a dash.

This component will be the about page selector of our application. We will modify it in the next sessions. We will use one file per component, so we save this as lib/about.ts.

Testing

We can test the component by calling the bootstrap function. We create a new file, lib/index.ts, which will start the application:

```
import "zone.js";  
import "rxjs";  
import "reflect-metadata";  
import "es6-shim";  
import { bootstrap } from "angular2/platform/browser";  
import { About } from "./about";  
  
bootstrap(About).catch(err => console.error(err));
```



The `.catch` section will show errors in the console. If you do not include that call, you will not see those errors and that can be pretty frustrating.

We must change the `weather-widget` tag in `static/index.html` to an `about-page` tag. Now, we can run `gulp` and open `index.html` in a browser to see the results.

At the time of writing this, when you run this command, you get an error when saying that the type definition of `zone.js` is incorrect. You can ignore this error as it is a bug of `zone.js`.

Test early



It's always a good idea to test during development. If you test after writing a lot of code, you will discover issues late, and it will take more work to repair them. Every time that you want to test the project, you must first run `gulp` and then open or refresh `index.html`.

Interactions

We can add an interaction inside the class body. We must use bindings to connect the template to definitions in the body. There are three different bindings:

- One-way variable binding
- One-way event listener
- Two-way binding

A one-way binding will connect the class body and template in one direction. In case of a variable, changes of the variable will update the template, but the template cannot update the variable. A template can only send an event to the class. In case of a two-way binding, a change of the variable changes the template and a change in the template will change the variable. This is useful for the value of an input element, for example. We will take a look at one-way bindings in the next section.

One-way variable binding

In the first attempt of the about page, the location (Utrecht) is hardcoded. In the final application, we want to choose our own location. The first step we will take is to add a property to the class that contains the location. Using a one-way binding, we will reference that value in the template. A one-way variable binding is denoted with brackets inside attributes and double curly brackets inside text:

```
import { Component } from "angular2/core";  
  
@Component({  
  selector: "about-page",  
  template: `  
    <h2>About</h2>  
    This widget shows the weather forecast of  
    <a [href]="'https://maps.google.com/?q=' + encodedLocation">  
      {{ location }}  
    </a>  
    The next 24 hours are shown under 'Today' and the forecast  
    24-48 hours ahead under 'Tomorrow'.  
  `}  
}  
  
export class About {  
  location = "Utrecht";  
  
  get encodedLocation() {  
    return encodeURIComponent(this.location);  
  }  
}
```



At the time of writing this, templates aren't checked by TypeScript. Make sure that you write the correct names of the variables. Variables should not be prefixed by `this.`, like you would do in class methods.

You can add an expression in such bindings. In this example, the binding of the `href` attribute does string concatenation. However, the subset of expressions is limited. You can add more complex code inside getters in the class, as done with `encodedLocation`.



You can also use a different getter, which would encode the location and concatenate it with the Google Maps URL.

Event listeners

Event bindings can connect an event emitter of a tag or component to a method of a function. Such binding is denoted with parenthesis in the template. We will add a show-more button to our application:

```
import { Component } from "angular2/core";  
  
@Component({  
  selector: "about-page",  
  template: `  
    <h2>About</h2>  
    This widget shows the weather forecast of  
    <a [href]="'https://maps.google.com/?q=' + encodedLocation">  
      {{ location }}  
    </a>.  
    The next 24 hours are shown under 'Today' and the forecast  
    24-48 hours ahead under 'Tomorrow'.  
    <br />  
    <a href="javascript:;" (click)="show()">Show more</a>  
    <a href="javascript:;" (click)="hide()">Show less</a>  
  `)  
})  
export class About {  
  location = "Utrecht";  
  collapsed = true;  
  show() {  
    this.collapsed = false;  
  }  
  hide()  
  {  
    this.collapsed = true;  
  }  
  
  get encodedLocation() {  
    return encodeURIComponent(this.location);  
  }  
}
```

The `show()` or `hide()` function will be called when one of the show or hide links is clicked on.

Adding conditions to the template

The event handler in the previous section sets the property collapsed to false but that does not modify the template. In normal code, we would have written `if (this.collapsed) { ... }`. In templates, we cannot use that, but we can use `ngIf`.

Directives

A directive is an extension to normal HTML tags and attributes. It can define custom behavior. A custom component, such as the **About** page, can be seen as a directive too. The `ngIf` condition is a built-in directive in Angular. It is a custom attribute that displays the content if the specified value is true.

The template tag

If a piece of a component needs to be shown a variable an amount of times, you can wrap it in a `template` tag. Using the `ngIf` (or `ngFor`) directive, you can control how often it is shown (in case of `ngIf`, once or zero times). The `template` tag will look like this:

```
<template [ngIf]="collapsed">
  <div>Content</div>
</template>
```

You can abbreviate this as follows:

```
<div *ngIf="collapsed">Content</div>
```

It is advised to use the abbreviated style, but it's good to remember that it is shorthand for the `template` tag.

Modifying the about template

Since `ngIf` is a built-in directive, it doesn't have to be imported. Custom directives need to be imported. We will see an example of using custom components later in this chapter. In the template, we can use `*ngIf` now. The template will thus look like this:

```
template: `<h2>About</h2>
This widget shows the weather forecast of
<a [href]="'https://maps.google.com/?q=' + encodedLocation">
  {{ location }}
</a>.
The next 24 hours are shown under 'Today' and the forecast
24-48 hours ahead under 'Tomorrow'.
<br />
<a *ngIf="collapsed" href="javascript:;" (click)="show()">Show
more</a>
<div *ngIf="!collapsed">
The forecast uses data from <a
href="http://openweathermap.org">Open Weather Map</a>.
<br />
<a href="javascript:;" (click)="hide()">Hide</a>
</div>
`)
```

The class body does not have to be changed. As you can see, you can use expressions in the `*ngIf` bindings, which is not surprising as it is a shorthand for one-way variable bindings.

Using the component in other components

We can use the `about-page` component in other components, as if it was a normal HTML tag. But the component is still boring, as it will always say that it shows the weather broadcast of Utrecht. We can mark the `location` property as an input. After that, `location` is an attribute that we can set from other components. It is even possible to bind it as a one-way binding. The `Input` decorator, which we are using here, needs to be imported just like `Component`:

```
import { Component, Input } from "angular2/core";

@Component ({
  ...
})
export class About {
```

```
@Input()
location: string = "Utrecht";
collapsed = true;
show() {
    this.collapsed = false;
}
hide() {
    this.collapsed = true;
}

get encodedLocation() {
    return encodeURIComponent(this.location);
}
```

Showing a forecast

We still have not shown a forecast yet. We will use data from open weather map (<http://www.openweathermap.org>). You can create an account on their website. With your account, you can request an API token. You need the token to request the forecast. A free account is limited to 60 requests per second and 50,000 requests per day.

We save the API token in a separate file, `lib/config.ts`:

```
export const openWeatherMapKey = "your-token-here";
export const apiURL = "http://api.openweathermap.org/data/2.5/";
```



Add constants to a separate file

When you add constants in separate configuration files, you can easily change them and your code is more readable. This gives you better maintainable code.

Using the API

We will create a new file, `lib/api.ts`, that will simplify downloading data from open weather map. The API uses URLs such as

`http://api.openweathermap.org/data/2.5/forecast?mode=json&q=Utrecht,NL&appid=your-token-here`. We will create a function that will build the full URL out of `forecast?mode=json&q=Utrecht,NL`. The function must check whether the path already contains a question mark. If so, it must add `&appid=`, otherwise `?appid=`:

```
import { openWeatherMapKey, apiURL } from "./config";
```

```
export function getUrl(path: string) {
  let url = apiURL + path;
  if (path.indexOf(?) === -1) {
    url += "?";
  } else {
    url += "&";
  }
  url += "appid=" + openWeatherMapKey;
  return url;
}
```

Write small functions

Small functions are easy to reuse. This reduces the amount of code you need to write. The same applies to components—small components are easy to reuse.

Typing the API

You can open the URL in the previous section to get a look at the data you get. We will write an interface for the part of the API that we will use:

```
export interface ForecastResponse {
  city: {
    name: string;
    country: string;
  };
  list: ForecastItem[];
}

export interface ForecastItem {
  dt: number;
  main: {
    temp: number
  };
  weather: {
    main: string,
    description: string
  };
}
```

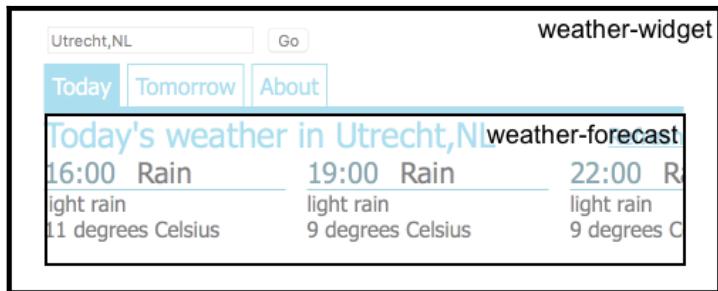
JSDoc comments

You can add documentation for interfaces and their properties by adding a JSDoc comment before it:

```
/** Documentation here */
```

Creating the forecast component

As a quick recap, the forecast widget will look like this:



What properties does the class need? The template will need forecast data of the current day or the next day. The component can show the weather of **Today** and **Tomorrow**, so we will also need a property for that. For fetching the forecast, we also need the location. To show the loading state in the template, we will also store that in the class. This will result in the following class, in `lib/forecast.ts`:

```
import { Component, Input } from "angular2/core";
import { ForecastResponse } from "./api";

export interface ForecastData {
  date: string;
  temperature: number;
  main: string;
  description: string;
}

enum State {
  Loading,
  Refreshing,
  Loaded,
  Error
}

@Component({
  selector: "weather-forecast",
  template: `...
})

export class Forecast {
  temperatureUnit = "degrees Celsius";

  @Input()
}
```

```
tomorrow = false;  
@Input()  
location = "Utrecht";  
  
data: ForecastData[] = [];  
  
state = State.Loading;  
}
```



Testing

You can test this component by adjusting the tag in `index.html` and bootstrapping the right component in `index.ts`. Run `gulp` to compile the sources and open the web browser.

Templates

The template uses the `ngFor` directive to iterate over the `data` array:

```
import { Component, Input } from "angular2/core";  
import { ForecastResponse } from "./api";  
  
...  
  
@Component ({  
  selector: "weather-forecast",  
  template: `  
    <span *ngIf="loading" class="state">Loading...</span>  
    <span *ngIf="refreshing" class="state">Refreshing...</span>  
    <a *ngIf="loaded || error" href="javascript:;" (click)="load()"  
      class="state">Refresh</a>  
    <h2>{{ tomorrow ? 'Tomorrow' : 'Today' }}'s weather in {{  
      location }}</h2>  
    <div *ngIf="error">Failed to load data.</div>  
    <ul>  
      <li *ngFor="#item of data">  
        <div class="item-date">{{ item.date }}</div>  
        <div class="item-main">{{ item.main }}</div>  
        <div class="item-description">{{ item.description }}</div>  
        <div class="item-temperature">  
          {{ item.temperature }} {{ temperatureUnit }}  
        </div>  
      </li>  
    </ul>  
    <div class="clearfix"></div>  
`,
```

Using the `styles` property, we can add nice CSS styles, as shown here:

```
  styles: [
    `.state {
      float: right;
      margin-top: 6px;
    }
    ul {
      margin: 0;
      padding: 0 0 15px;
      list-style: none;
      width: 100%;
      overflow-x: scroll;
      white-space: nowrap;
    }
    li {
      display: inline-block;
      margin-right: 15px;
      width: 170px;
      white-space: initial;
    }
    .item-date {
      font-size: 15pt;
      color: #165366;
      margin-right: 10px;
      display: inline-block;
    }
    .item-main {
      font-size: 15pt;
      display: inline-block;
    }
    .item-description {
      border-top: 1px solid #44A4C2;
      width: 100%;
      font-size: 11pt;
    }
    .item-temperature {
      font-size: 11pt;
    }
  ]
})
```

In the `class body`, we add the getters which we used in the template:

```
export class Forecast {
  ...
  state = State.Loading;
  get loading() {
```

```
    return this.state === State.Loading;
}
get refreshing() {
    return this.state === State.Refreshing;
}
get loaded() {
    return this.state === State.Loaded;
}
get error() {
    return this.state === State.Error;
}
...
}
```



Enums

Enums are just numbers with names attached to them. It's more readable to write `State.Loaded` than `2`, but they mean the same in this context.

As you can see, the syntax of `ngFor` is `*ngFor="#variable of array"`. The enum cannot be referenced from the template, so we need to add getters in the body of the class.

Downloading the forecast

To download data from the Internet in Angular, we need to get the HTTP service. We need to set the `viewProviders` section for that:

```
import { Component, Input } from "angular2/core";
import { Http, Response, HTTP_PROVIDERS } from "angular2/http";
import { getUrl, ForecastResponse } from "./api";

...
@Component({
  selector: "weather-forecast",
  viewProviders: [HTTP_PROVIDERS],
  template: `...`,
  styles: [...]
})
export class Forecast {
  constructor(private http: Http) {
    ...
  }
}
```

Angular will inject the `Http` service into the constructor.



By including `private` or `public` before an argument of the constructor, that argument will become a property of the class, initialized by the value of the argument.

We will now implement the `load` function, which will try to download the forecast on the specified location. The function can also use coordinates as a location, written as `Coordinates lat lon`, where `lat` and `lon` are the coordinates as shown here:

```
private load() {
  let path = "forecast?mode=json&";
  const start = "coordinate ";
  if (this.location &&
      this.location.substring(0,
                            start.length).toLowerCase() === start) {
    const coordinate = this.location.split(" ");
    path += `lat=${parseFloat(coordinate[1])}&lon=${
      parseFloat(coordinate[2])}`;
  } else {
    path += "q=" + this.location;
  }

  this.state = this.state === State.Loaded ?
    State.Refreshing : State.Loading;
  this.http.get(getUrl(path))
    .map(response => response.json())
    .subscribe(res =>
      this.update(<ForecastResponse> res), ()
      => this.showError());
};
```

Three kinds of variables

You can define variables with `const`, `let`, and `var`. A variable declared with `const` cannot be modified. Variables declared with `const` or `let` are block-scoped and cannot be used before their definition. A variable declared with `var` is function scoped and can be used before its definition. Such variable can give unexpected behavior, so it's advised to use `const` or `let`.



The function will first calculate the **URL**, then set the state and finally fetch the data and get returns an observable. An observable, comparable to a promise, is something that contains a value that can change later on. Like with arrays, you can map an observable to a different observable. `Subscribe` registers a callback, which is called when the observable is changed.

This observable changes only once, when the data is loaded. If something goes wrong, the second callback will be called.



Lambda expressions (inline functions)

The fat arrow (`=>`) creates a new function. It's almost equal to a function defined with the function keyword (`function () { return ... }`), but it is scoped lexically, which means that `this` refers to the value of `this` outside the function. `x => expression` is a shorthand for `(x) => { return expression; }`. TypeScript will automatically infer the type of the argument based on the signature of `map` and `subscribe`.

As you can see, this function uses the `update` and `showError` functions. The `update` function stores the results of the open weather map API, and `showError` is a small function that sets the state to `State.Error`. Since temperatures of the API are expressed in Kelvin, we must subtract 273 to get the value in Celsius:

```
fullData: ForecastData[] = [];
data: ForecastData[] = [];

private formatDate(date: Date) {
    return date.getHours() + ":" +
        date.getMinutes() + ":" +
        date.getSeconds();
}

private update(data: ForecastResponse) {
    if (!data.list) {
        this.showError();
        return;
    }

    this.fullData = data.list.map(item => ({
        date: this.formatDate(new Date(item.dt * 1000)),
        temperature: Math.round(item.main.temp - 273),
        main: item.weather[0].main,
        description: item.weather[0].description
    }));
    this.filterData();
    this.state = State.Loaded;
}

private showError() {
    this.data = [];
    this.state = State.Error;
}

private filterData() {
    const start = this.tomorrow ? 8 : 0;
```

```
        this.data = this.fullData.slice(start, start + 8);  
    }  
}
```

The `filterData` method will filter the forecast based on whether we want to see the forecast of today or tomorrow. Open weather map has one forecast per 3 hours, so 8 per day. The `slice` function will return a section of the array. `fullData` will contain the full forecast, so we can easily show the forecast of tomorrow, if we have already shown today.



Change detection

Angular will automatically reload the template when some property is changed, there's no need to invalidate anything (as C# developers might expect). This is called change detection.

We also want to refresh data when the location is changed. If tomorrow is changed, we do not need to download any data, because we can just use a different section of the `fullData` array. To do that, we will use getters and setters. In the setter, we can detect changes:

```
private _tomorrow = false;  
@Input()  
set tomorrow(value) {  
    if (this._tomorrow === value) return;  
    this._tomorrow = value;  
    this.filterData();  
}  
get tomorrow() {  
    return this._tomorrow;  
}  
  
private _location: string;  
@Input()  
set location(value) {  
    if (this._location === value) return;  
    this._location = value;  
    this.state = State.Loading;  
    this.data = [];  
    this.load();  
}  
get location() {  
    return this._location;  
}
```

Adding @Output

The response of Open weather map contains the name of the city. We can use this to simulate completion later on. We will create an event emitter. Other components can listen to the event and update the location when the event is triggered. The whole code will look like this with final changes highlighted:

```
import { Component, Input, Output, EventEmitter } from "angular2/core";
import { Http, Response, HTTP_PROVIDERS } from "angular2/http";
import { getUrl, ForecastResponse } from "./api";

interface ForecastData {
  date: string;
  temperature: number;
  main: string;
  description: string;
}

enum State {
  Loading,
  Refreshing,
  Loaded,
  Error
}

@Component({
  selector: "weather-forecast",
  viewProviders: [HTTP_PROVIDERS],
  template: `
    <span *ngIf="loading" class="state">Loading...</span>
    <span *ngIf="refreshing" class="state">Refreshing...</span>
    <a *ngIf="loaded || error" href="javascript:;" (click)="load()" class="state">Refresh</a>
    <h2>{{ tomorrow ? 'Tomorrow' : 'Today' }}'s weather in {{ location }}</h2>
    <div *ngIf="error">Failed to load data.</div>
    <ul>
      <li *ngFor="#item of data">
        <div class="item-date">{{ item.date }}</div>
        <div class="item-main">{{ item.main }}</div>
        <div class="item-description">{{ item.description }}</div>
        <div class="item-temperature">
          {{ item.temperature }} {{ temperatureUnit }}
        </div>
      </li>
    </ul>
    <div class="clearfix;"></div>
```

```
    ``
  styles: [
    `.state {
      float: right;
      margin-top: 6px;
    }
    ul {
      margin: 0;
      padding: 0 0 15px;
      list-style: none;
      width: 100%;
      overflow-x: scroll;
      white-space: nowrap;
    }
    li {
      display: inline-block;
      margin-right: 15px;
      width: 170px;
      white-space: initial;
    }
    .item-date {
      font-size: 15pt;
      color: #165366;
      margin-right: 10px;
      display: inline-block;
    }
    .item-main {
      font-size: 15pt;
      display: inline-block;
    }
    .item-description {
      border-top: 1px solid #44A4C2;
      width: 100%;
      font-size: 11pt;
    }
    .item-temperature {
      font-size: 11pt;
    }
  }`]
})
export class Forecast {
  constructor(private http: Http) {
  }
  temperatureUnit = "degrees Celsius";
  private _tomorrow = false;
```

```

@Input()
set tomorrow(value) {
    if (this._tomorrow === value) return;
    this._tomorrow = value;
    this.filterData();
}
get tomorrow() {
    return this._tomorrow;
}

private _location: string;
@Input()
set location(value) {
    if (this._location === value) return;
    this._location = value;
    this.state = State.Loading;
    this.data = [];
    this.load();
}
get location() {
    return this._location;
}

fullData: ForecastData[] = [];
data: ForecastData[] = [];

state = State.Loading;
get loading() {
    return this.state === State.Loading;
}
get refreshing() {
    return this.state === State.Refreshing;
}
get loaded() {
    return this.state === State.Loaded;
}
get error() {
    return this.state === State.Error;
}

@Output()
correctLocation = new EventEmitter<string>(true);

private formatDate(date: Date) {
    return date.getHours() + ":" + date.getMinutes() +
date.getSeconds();
}
private update(data: ForecastResponse) {

```

```

if (!data.list) {
    this.showError();
    return;
}

const location = data.city.name + ", " + data.city.country;
if (this._location !== location) {
    this._location = location;
    this.correctLocation.next(location);
}

this.fullData = data.list.map(item => ({
    date: this.formatDate(new Date(item.dt * 1000)),
    temperature: Math.round(item.main.temp - 273),
    main: item.weather[0].main,
    description: item.weather[0].description
}));
this.filterData();
this.state = State.Loaded;
}
private showError() {
    this.data = [];
    this.state = State.Error;
}
private filterData() {
    const start = this.tomorrow ? 8 : 0;
    this.data = this.fullData.slice(start, start + 8);
}

private load() {
    let path = "forecast?mode=json&";
    const start = "coordinate ";
    if (this.location&&this.location.substring(0,
start.length).toLowerCase() === start) {
        const coordinate = this.location.split(" ");
        path += `lat=${parseFloat(coordinate[1])}&lon=${
parseFloat(coordinate[2])}`;
    } else {
        path += "q=" + this.location;
    }

    this.state = this.state === State.Loaded ? State.Refreshing
State.Loading;
    this.http.get(getUrl(path))
        .map(response => response.json())
        .subscribe(res => this.update(<ForecastResponse> res),
() =>
this.showError());
}

```

}



The generic (type argument) in `new EventEmitter<string>()` means that the contents of an event will be a string. If the generic is not specified, it defaults to `{}`, an empty object type, which means that there is no content. In this case, we want to send the new location, which is a string.

The main component

As you can see in the screenshot in the introduction of this chapter, this component should have a textbox, a button, and three tabs. Under the tabs, these component will show the forecast or the [About page](#).

Using our other components

We can use our components that we have already written by adding them to the `directives` section and using their tag names in the template.

Two-way bindings

To get the value of the input box, we need two-way bindings. We can use the `ngModel` directive for that. The syntax combines the syntaxes of the two one-way bindings: `[(ngModel)]="property"`. The directive is again a built-in one, so we don't have to import it.

Using this two-way binding, we can automatically update the weather widget after every key press. That would cause a lot of requests to the server, and especially on slow connections, that's not desired.

To prevent these issues, we will add two separate properties. The property `location` will contain the content of the input and `activeLocation` will contain the location, which is being shown.

Listening to our event

We can listen to our event, just like we did with other events. We can access the event content with `$event`. Such a listener will look like `(correctLocation) = "correctLocation($event)"`. When the server responds with the forecast, it also provides the name of the location. If the user had a small typo in the name, the response will correct that. This event will be fired in such a case and the name will be corrected in the input box.

Geolocation API

Because our forecast widget supports coordinates, we can use the geolocation API to set the initial location. That API can give the coordinates where the device is located (roughly). Later on, we will use this API to set the widget to the current location when the page loads as shown here:

```
navigator.geolocation.getCurrentPosition(position => {
  const location = `Coordinate ${ position.coords.latitude } ${ position.coords.longitude }`;
  this.location = location;
  this.activeLocation = location;
});
```

Template string

Template strings, not to be confused with Angular templates, are strings wrapped in backticks (`). These strings can be multiline and can contain expressions between `${}` and `}``.



Component sources

As usual, we start by importing Angular. We also have to import the two components we have already written. We use an enumeration again to store the state of the component:

```
import { Component } from "angular2/core";
import { Forecast } from "./forecast";
import { About } from "./about";

enum State {
  Today,
  Tomorrow,
  About
}
```

The template will use the two-way binding on the `input` element:

```
@Component({
  selector: "weather-widget",
  directives: [Forecast, About],
  template: `
    <input [(ngModel)]="location" (keyup.enter)="clickGo()"
(blur)="clickGo()" />
    <button (click)="clickGo()">Go</button>
    <div class="tabs">
      <a href="javascript:;" [class.selected]="selectedTab === 0"
(click)="selectTab(0)">Today</a>
      <a href="javascript:;" [class.selected]="selectedTab === 1"
(click)="selectTab(1)">Tomorrow</a>
      <a href="javascript:;" [class.selected]="selectedTab === 2"
(click)="selectTab(2)">About</a>
    </div>
    <div class="content" [class.is-dirty]="isDirty" *ngIf="selectedTab
=== 0 || selectedTab === 1">
      <weather-forecast [location]="activeLocation"
[tomorrow]="selectedTab === 1"
(correctLocation)="correctLocation($event)" />
    </div>
    <div class="content" *ngIf="selectedTab === 2">
      <about-page [location]="activeLocation" />
    </div>
  `,
```

Binding to `class.selected` means that the element will have the `selected` class if the bound value is true. After the template, we can add some styles as shown here:

```
styles: [
  `.tabs > a {
    display: inline-block;
    padding: 5px;
    margin-top: 5px;
    border: 1px solid #57BEDE;
    border-bottom: 0px none;
    text-decoration: none;
  }
  .tabs>a.selected {
    background-color: #57BEDE;
    color: #fff;
  }
  .content {
    border-top: 5px solid #57BEDE;
  }
  .is-dirty {
```

```
    opacity: 0.4;
    background-color: #ddd;
  }`  
]  
})
```

In the constructor, we can use the geolocation API to get the coordinates of the current position:

```
export class Widget {  
  constructor() {  
    navigator.geolocation.getCurrentPosition(position => {  
      const location = `Coordinate ${ position.coords.latitude }  
${ position.coords.longitude }`;  
      this.location = location;  
      this.activeLocation = location;  
    });  
  }  
  
  location: string = "Utrecht,NL";  
  activeLocation: string = "Utrecht,NL";  
  get isDirty() {  
    return this.location !== this.activeLocation;  
  }  
  
  clickGo() {  
    this.activeLocation = this.location;  
  }  
  correctLocation(location: string) {  
    if (!this.isDirty) this.location = location;  
    this.activeLocation = location;  
  }  
  
  selectedTab = 0;  
  selectTab(index: number) {  
    this.selectedTab = index;  
  }  
}
```

Summary

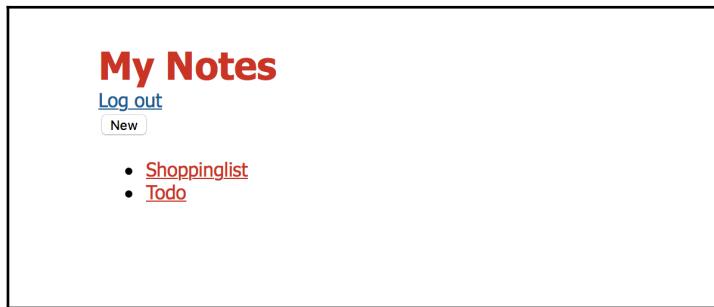
In this chapter, we created an application with Angular 2. We explored Angular 2 and used its directives and bindings in our components. We also used an online API. You should now be able to build small Angular applications. In the next chapter, we will build a more complex application in Angular, which will also use its own server.

3

Note-Taking App with a Server

In this chapter, we will create a client-server app. The client will be written using **Angular 2** and the server will be written using **NodeJS** and MongoDB. We can use TypeScript on both sides and we will see how we can reuse code between them.

The application can be used to take notes. We will implement a login page and basic **Create, Read, Update, and Delete (CRUD)** operations for the notes.



In this chapter, we will cover the following topics:

- Setting up the project structure
- Getting started with NodeJS
- Understanding the structural type system
- Adding authentication
- Testing the API
- Adding CRUD operations
- Writing the client side
- Running the application

Setting up the project structure

First, we have to setup the project. The difference with the previous chapter is that we now have to build two applications—the client side and the server side. This causes some differences with the previous setup.

Directories

We will again place our TypeScript sources in the `lib` directory. In that directory, we will create four subdirectories: `client`, `server`, `shared`, and `typings`. The `lib/client` directory will contain the client-side application and the `lib/server` directory will contain the server code. Codes that can be used by both the server and the client will go in `lib/shared`. Last but not least, `lib/typings` will contain type definitions for some dependencies, including NodeJS.

Configuring the build tool

In `lib`, we create a `tsconfig.json` file that will contain some configuration for TypeScript. We want to compile the server-side code to `es2015`, so we can use some new features of TypeScript and JavaScript. The client side, however, must be compiled to `es5` for browser support. In the `tsconfig` file, we will specify `es2015` as target and override it in the `gulp` file. We can also specify the version of the default library that we want to use. We need `es2015` and `dom`. The first contains the recent classes and functions from JavaScript, such as `Map` and `Object.assign`:

```
{  
  "compilerOptions": {  
    "target": "es6",  
    "module": "commonjs",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "lib": ["es2015", "dom"]  
  }  
}
```

The `lib` option will only make the types for new classes and functions available. At runtime, these might not be present. We include a polyfill, `es6-shim`, to make sure that these will always be available.

The `gulp` file, located in the root of the project, is comparable to the configuration of the previous chapter. We can install all necessary dependencies, including runtime dependencies, using `npm`:

```
npm init
npm install gulp gulp-typescript small gulp-sourcemaps merge2 gulp-
concat gulp-uglify --save-dev
npm install angular2 es6-shim rxjs phaethon --save
```

You can again set the `private` property in `package.json` so that you don't accidentally upload your project to `npm`. In `gulpfile.js`, we can now load all dependencies:

```
var gulp = require("gulp");
var typescript = require("gulp-typescript");
var small = require("small").gulp;
var sourcemaps = require("gulp-sourcemaps");
var merge = require("merge2");
var concat = require("gulp-concat");
var uglify = require("gulp-uglify");
```

We will create two TypeScript projects: one for the server and one for the client side. In the second project, we will override the target to `es5`:

```
var tsServer = typescript.createProject("lib/tsconfig.json");
var tsClient = typescript.createProject("lib/tsconfig.json", {
  target: "es5"
});
```

Now we can use almost the same task as in the previous chapter. The sources must be loaded from `lib/client` instead of `lib`, and `lib/shared` should be included too:

```
gulp.task("compile-client", function() {
  return gulp.src(["lib/client/**/*.ts", "lib/shared/**/*.ts"], {
    base: "lib"
  })
  .pipe(sourcemaps.init())
  .pipe(typescript(tsClient))
  .pipe(small('client/index.js', {
    outputFileName: { standalone: "scripts.js" },
    externalResolve: ['node_modules'],
    globalModules: {
      "crypto": {
        standalone: "undefined"
      }
    }
  })
  .pipe(gulp.dest("lib/client"))
});
```

```
        }
    })
    .pipe(sourcemaps.write('.'))
    .pipe(gulp.dest('static/scripts'));
});
```

The compilation of the server-side code is simpler, as the code doesn't have to be bundled. NodeJS has a built-in module loader:

```
gulp.task("compile-server", function() {
    return gulp.src(["lib/server/**/*.ts", "lib/shared/**/*.ts"], { base:
    "lib" })
    .pipe(sourcemaps.init())
    .pipe(typescript(tsServer))
    .pipe(sourcemaps.write("."))
    .pipe(gulp.dest("dist"));
});
```

We add the release and default tasks that can build the release and debug tasks:

```
gulp.task("release", ["compile-client", "compile-server"], function() {
    return gulp.src("static/scripts/scripts.js")
    .pipe(uglify())
    .pipe(gulp.dest("static/scripts"));
});

gulp.task("default", ["compile-client", "compile-server"]);
```

The tasks can be started using `gulp` or `gulp release`.

Type definitions

Before a library can be used in TypeScript, you have to have type definitions for it. These are stored in `.d.ts` files. For some packages, these are automatically installed. For example, we used Angular in the previous chapter and we didn't install the definitions manually.

Packages distributed on npm can include their type definitions in the same package. When you download such a package, the `typings` come along. Unfortunately, not all packages do this. As of TypeScript 2.0, it is possible to download typings for these packages on npm too. For instance, the typings for `mongodb` are published in the `@types/mongodb` package.

You can install types for a lot of packages this way. Types for NodeJS itself are available in `@types/node`. Run these commands in the root directory:

```
npm install @types/node --save
npm install @types/mongodb --save
```

The compiler will automatically find the types for `mongodb` when you import it. Since we will not explicitly import NodeJS in the code, the compiler will not find it. We must add it to our `tsconfig` file.

```
{  
  "compilerOptions": {  
    "target": "es6",  
    "module": "commonjs",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "lib": ["es2015", "dom"],  
    "types": ["node"]  
  }  
}
```

The compiler can now use all type definitions.

Getting started with NodeJS

In the previous chapter, we used NodeJS, as gulp uses it. Node can be used for a server and for a command line tool. In this chapter, we will build a server and in Chapter 9, *Playing Tic-Tac-Toe against an AI*, we will create a command line application. If you haven't installed Node yet, you can download it from nodejs.org.

We will first create a simple server. We will use **Phaethon**, a package for Node that makes it easy to build a server in NodeJS. Phaethon includes type definitions, so we can use it immediately. We create a file `lib/server/index.ts` and add the following:

```
import { Server } from "phaethon";  
const server = new Server();  
server.listener = request => new phaethon.ServerResponse("Hello");  
server.listenHttp(8800);
```

We can run this server using the following command:

```
gulp && node dist/server
```

When you open `localhost:8800` in a web browser, the listener callback will be called and you will see **Hello** in the browser.

Asynchronous code

A server doesn't do all the work itself. It will also delegate some tasks. For instance, it might need to download a webpage or fetch something from a database. Such a task will not give a result immediately. In the meantime, the server could do something else. This style of programming is called asynchronous or nonblocking, as the order of execution is not fixed and such task does not block the rest of the application.

Imagine we have a task that will download a webpage. The synchronous variant would look like the following:

```
function download() {  
    return ...;  
}  
  
function demo() {  
    // Before download  
    try {  
        const result = download();  
        const result2 = download();  
        // Download completed  
    } catch (error) {  
        // Error  
    }  
}
```

Callback approach for asynchronous code

In a webserver, this would prevent the server from handling other requests. The task blocks the whole server. That is, of course, not what we want. The simplest asynchronous approach uses callbacks. The first argument of the callback will contain an error if something went wrong and the second argument will contain the result if there is a result:

```
function download(callback: (error: any, result: string) => void) {  
    ...  
}  
  
function demo() {  
    // Before download  
    download((error, result) => {  
        if (error) {  
            // Error  
        } else {  
            // Download completed  
            download((error2, result2) => {  
                //...  
            })  
        }  
    })  
}
```

```
    if (error2) {
    } else {
        // Download 2 completed
    }
});
```

});

});

Disadvantages of callbacks

The disadvantage of this is that when you have a lot of callbacks, you have to nest callbacks in callbacks, which is called callback hell. In ES6, a new class was introduced, that acts like an abstraction of such a task. It is called a promise. Such a value promises that there will be a result now or later on. The promise can be resolved, which means that the result is ready. The promise can also be rejected, which means that there was some error:

```
function download(): Promise<string> {
    ...
}

function demo() {
    // Before download
    download().then(result => {
        // Download completed

        return download();
    }).then(result2 => {
        // Second download completed
    });
}
```

As you can see, the preceding code is more readable than the callbacks code. It's also easier to chain tasks since you can return another promise in the `then` section of a promise.

However, the synchronous code is still more readable. ES7 has introduced **async** functions. These functions are syntactic sugar around promises. Instead of calling `then` on a promise, you can **await** it and write code as if it were synchronous.



At the time of writing, `async` functions can only be compiled to ES6. TypeScript 2.1 will introduce support for ES5 too.

```
function download(): Promise<string> {
    ...
}
```

```
}

async function demo() {
  try {
    const result = await download();
    const result2 = await download();
  } catch (error) {
  }
}
```

As you can see, this is almost the same as the code we started with. This gives the best of both worlds: it results in readable and performant code.



Do not forget the `async` keyword in the function header. If you want to annotate the function with a return type, write `Promise<T>` instead of `T`.

The database

A lot of programmers use MongoDB in combination with NodeJS. You can install MongoDB from www.mongodb.org. MongoDB can be started using the following command in the project root:

```
mongod --dbpath ./data
```

You can keep the preceding command running in one terminal window and run NodeJS in another terminal window later on.

Wrapping functions in promises

We will run the database on the same computer as the server and we will name the database `notes`. This yields the URL `mongodb://localhost/notes`, which we need to connect to the database. We have already installed the definitions with `tsd`. MongoDB exposes an API based on callbacks. We will wrap these in promises, as we will use `async/await` later on. Wrapping a function in a promise will look like the following:

```
function wrapped() {
  return new Promise<string>((resolve, reject) => {
    originalFunction((error, result) => {
      if (error) {
        reject(error);
      } else {
```

```
        resolve(result);
    }
});
});
}
```

The `Promise` constructor takes a callback function. This function can call the `resolve` callback if everything succeeded or call the `reject` function if something failed.

Connecting to the database

We add the following in `lib/server/database.ts`. First we must connect to the database. Instead of rejecting when the connection failed, we will throw the error. This way the server will quit if it can't connect to the database:

```
import { MongoClient, Db, Collection } from "mongodb";

const databaseUrl = "mongodb://localhost:27017/notes";
const database = new Promise<Db>(resolve => {
    MongoClient.connect(databaseUrl, (error, db) => {
        if (error) {
            throw error;
        }
        resolve(db);
    })
});
```



Usually, you would reject the promise in case of an error. Here, we throw the error and crash the server. In this case it is better since the server cannot do anything without a database connection.

The database contains two collections (tables): `users` and `notes`. Since we can only access these after the connection to the database has succeeded, these should also be placed in a `Promise`. Since `database` already is a `Promise`, we can use `async/await`:

```
async function getCollection(name: string) {
    const db = await database;
    return db.collection(name);
}
export const users = getCollection("users");
export const notes = getCollection("notes");
```

The `users` and `notes` variables have the type `Promise<Collection>`.

We can now write a function that will insert an item into a collection and return a promise. Since this promise doesn't have a resulting value, we will type it as `Promise<void>`:

```
export function insert(table: Promise<Collection>, item: any) {
  const collection = await table;
  return new Promise<void>((resolve, reject) => {
    collection.insertOne(item, (error) => {
      if (error) {
        reject(error);
      } else {
        resolve();
      }
    });
  });
}
```

Querying the database

To query the database, we will use the function `find`. MongoDB returns a cursor object, which allows you to stream all results. If you have a big application, and queries that return a lot of results, this can improve the performance of your application. Instead of streaming the results, we can also buffer them in an array with the `toArray` function:

```
export function find(table: Promise<Collection>, query: any) {
  const collection = await table;
  return new Promise<U[]>((resolve, reject) => {
    collection.find(query, (error, cursor) => {
      if (error) {
        reject(error);
      } else {
        cursor.toArray((error, results) => {
          if (error) {
            reject(error);
          } else {
            resolve(results);
          }
        });
      }
    });
  });
}
```

We will add `update` and `remove` functions later on.

Understanding the structural type system

TypeScript uses a structural type system. What that means can be easily demonstrated using the following example:

```
class Person {  
    name: string;  
}  
class City {  
    name: string;  
}  
const x: City = new Person();
```

In languages like C#, this would not compile. These languages use a nominal type system. Based on the name, a `Person` is not a `City`. TypeScript uses a structural type system. Based on the structure of `Person` and `City`, these types are equal, as they both have a `name` property. This fits well in the dynamic nature of JavaScript. It can, however, lead to some unexpected behavior, as the following would compile:

```
class Foo {  
}  
const f: Foo = 42;
```

Since `Foo` does not have any properties, every value would be assignable to it. In cases where the structural behavior is not desired, you can add a **brand**, a property that adds type safety but does not exist at runtime:

```
class Foo {  
    __fooBrand: void;  
}  
const f: Foo = 42;
```

Now the last line will give an error, as expected.

Generics

The typings for MongoDB don't use generics or type arguments. Given that we already have to add a tiny wrapper around it, we can also easily add generics to that wrapper. We will create a new type for the data store that has generics:

```
export interface Table<T> extends Collection {  
    __tableBrand: T;  
}
```

If you didn't include the brand, `Table<A>` would be structurally identical to `Table`, which we do not want. We can now load the collections with the correct types. We use the `User` and `Note` types here. We will create these interfaces later on:

```
import { User } from "./user";
import { Note } from "./note";

async function getCollection<U>(name: string) {
    const db = await database;
    return <Table<U>> db.collection(name);
}

export const users = getCollection<User>("users");
export const notes = getCollection<Note>("notes");
```

With generics, the `insert` function will look like the following:

```
export function insert<U>(table: Table<U>, item: U) {
    return new Promise<void>((resolve, reject) => {
        table.insertOne(item, (error) => {
            if (error) {
                reject(error);
            } else {
                resolve();
            }
        });
    });
}
```

For `find`, we want the query to be a supertype of the table content. In other words, you want to query on some properties of the content of the table. Support for this was added in TypeScript 1.8:

```
export function find<U extends V, V>(table: Table<U>, query: V) {
    return new Promise<U[]>((resolve, reject) => {
        table.find(query, (error, result) => {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
        });
    });
}
```

We will also write wrappers for update and remove. Together these functions can do the **CRUD** operations: create, read, update, and delete:

```
export function update<U extends V, V>(table: Table<U>, query: V, newItem: U) {
  return new Promise<void>((resolve, reject) => {
    table.update(query, newItem, (error) => {
      if (error) {
        reject(error);
      } else {
        resolve();
      }
    });
  });
}

export function remove<U extends V, V>(table: Table<U>, query: V) {
  return new Promise<void>((resolve, reject) => {
    table.remove(query, (error) => {
      if (error) {
        reject(error);
      } else {
        resolve();
      }
    });
  });
}
```

In lib/server/user.ts, we will create the User model. For MongoDB, such types should have an `_id` property. The database will use that property to identify instances of the models:

```
import { ObjectId } from "mongodb";
export interface User {
  _id: ObjectId;
  username: string;
  passwordHash: string;
}
```

And in lib/server/note.ts, we add the Note model:

```
import { ObjectId } from "mongodb";
export interface Note {
  _id: ObjectId;
  userId: string;
  content: string;
}
```

Typing the API

In `lib/shared/api.ts`, we will add some typings for the API. On the server side, we can check that the response has the right type:

```
export interface LoginResult {  
    ok: boolean;  
    message?: string;  
}  
export interface MenuResult {  
    items: MenuItem[];  
}  
export interface MenuItem {  
    id: string;  
    title: string;  
}  
export interface ItemResult {  
    id: string;  
    content: string;  
}
```

We will now implement the functions that return these types.

Adding authentication

In `lib/server/index.ts`, we will first add sessions. A session is a place to store data, which is persistent for a client on the server. On the client side, a cookie will be saved, which contains an identifier of the session. If a request contains a valid cookie with such an identifier, you will get the same session object. Otherwise, a new session will be created:

```
import { Server, ServerRequest, ServerResponse, ServerError, StatusCode,  
SessionStore } from "phaethon";  
import { ObjectId } from "mongodb";  
import { User, login, logout } from "./user";  
import * as note from "./note";
```

With `import { ... }`, we can import a set of entities from another file. With `import * as ...`, we import the whole file as an object. The following two snippets are equivalent:

```
import * as foo from "./foo"; foo.bar(); import { bar }  
from "./foo"; bar();
```



We define the type of the content of the session as follows:

```
export interface Session {  
    userId: ObjectId;  
}  
  
const server = new Server();
```

The sessions will be stored in a `SessionStore`. The lifetime of a session is $60 * 60 * 24$ seconds or one day:

```
const sessionStore = new SessionStore<Session>("session-id", () => ({  
    userId: undefined }), 60 * 60 * 24, 1024);  
server.listener = sessionStore.wrapListener(async (request, session) =>  
{  
    const response = await handleRequest(request, session.data);  
    if (response instanceof ServerResponse) {  
        return response;  
    } else {  
        const serverResponse = new  
ServerResponse(JSON.stringify(response));  
        return serverResponse;  
    }  
});  
server.listenHttp(8800);
```

`JSON.stringify` will convert an object to a string. Such a string can easily be converted back to an object on the client side. In *Chapter 2, Weather Forecast Widget*, the responses of the weather API were also formatted as JSON strings.

In `handleRequest`, all requests will be sent to a handler based on their path:

```
async function handleRequest(request: ServerRequest, session: Session):  
Promise<ServerResponse | Object> {  
    const path = request.path.toLowerCase();  
  
    if (path === "/api/login") return login(request, session);  
    if (path === "/api/logout") return logout(request, session);  
    throw new ServerError(StatusCode.ClientErrorNotFound);  
}
```

Implementing users in the database

Now we can implement authentication in `user.ts`. For safety, we won't store plain passwords in our database. Instead we **hash** them. A hash is a manipulation of an input, in a way that you cannot find the input based on the hash. When someone wants to log in, the password is hashed and compared with the hashed password from the database. Using the built-in module `crypto`, this can easily be done:

```
import * as crypto from "crypto";
function getPasswordHash(username: string, password: string): string {
    return crypto.createHash("sha256").update(password.length + "-" +
username + "-" + password).digest("hex");
}
```

The logout handler is easy to write. We must remove the `userId` of the session as follows:

```
export function logout(request: ServerRequest, session: Session): LoginResult {
    session.userId = undefined;
    return { ok: true };
}
```

As you can see, we are using the `LoginResult` interface that we wrote previously. The `login` function will use the `async/await` syntax. The function expects that the `username` and `password` are available in the URL query. If they are not available, `validate.expect` will throw an error, which will be displayed as a `Bad Request` error:

```
export async function login(request: ServerRequest, session: Session): Promise<LoginResult> {
    const username = validate.expect(request.query["username"], validate.isString);
    const password = validate.expect(request.query["password"], validate.isString);
    const passwordHash = getPasswordHash(username, password);

    const results = await find(users, { username, passwordHash });
    if (results.length === 0) {
        return { ok: false, message: "Username or password incorrect" };
    }
    const user = results[0];
    session.userId = user._id;
    return { ok: true };
}
```

Adding users to the database

To add some users to the database, we must add some code and run the server once with it. In a real-world application, you would probably want to add a register form. That is comparable to adding a note, which we will do later on in this chapter.

We will also add two helper functions that we can use in `note.ts` to check whether the user is logged in:

```
import * as crypto from "crypto";
import { ServerRequest, ServerResponse, ServerError, StatusCode, validate } from "phaethon";
import { Session } from "./index";
import { LoginResult } from "../shared/api";
import { users, find, insert } from "./database";

export interface User {
  _id: string;
  username: string;
  passwordHash: string;
}

function getPasswordHash(username: string, password: string): string {
  return crypto.createHash("sha256").update(password.length + "-" + username + "-" + password).digest("hex");
}

insert(users, {
  _id: undefined,
  username: "lorem",
  passwordHash: getPasswordHash("lorem", "ipsum")
});
insert(users, {
  _id: undefined,
  username: "foo",
  passwordHash: getPasswordHash("foo", "bar")
});

export async function login(request: ServerRequest, session: Session): Promise<LoginResult> {
  const username = validate.expect(
    request.query["username"], validate.isString);
  const password = validate.expect(
    request.query["password"], validate.isString);
  const passwordHash = getPasswordHash(username, password);

  const results = await find(users, { username, passwordHash });
}
```

```
if (results.length === 0) {
  return { ok: false, message: "Username or password incorrect" };
}
const user = results[0];
session.userId = user._id;
return { ok: true };
}
export function logout(request: ServerRequest, session: Session): LoginResult {
  session.userId = undefined;
  return { ok: true };
}
export async function getUser(session: Session) {
  if (session.userId === undefined) return undefined;
  const results = await find(users, { _id: session.userId });
  return results[0];
}
export async function getUserOrError(session: Session) {
  const user = await getUser(session);
  if (user === undefined) {
    throw new ServerError(StatusCode.ClientErrorUnauthorized);
  }
  return user;
}
```

Run the server once and remove the two `insert` calls afterward.

Testing the API

We can start the server by running the following command:

```
gulp && node --harmony_destructuring dist/server
```

In a web browser, you can open

`localhost:8800/api/login?username=lorem&password=ipsum` to test the code. You can change the parameters to test how a wrong username or password behaves.

For debugging, you can add `console.log("...");` calls in your code.

Adding CRUD operations

Most servers handle CRUD operations primarily. Our server must handle five different requests: list all notes of the current user, find a specific note, insert a new note, update a note, and remove a note.

First, we add a helper function that can be used on the server side and the client side. In `lib/shared/note.ts`, we add a function that returns the title of a note—the first line, if available, or “Untitled”:

```
export function getTitle(content: string) {
  const lineEnd = content.indexOf("\n");
  if (content === "" || lineEnd === 0) {
    return "Untitled";
  }
  if (lineEnd === -1) {
    // Note contains one line
    return content;
  }
  // Get first line
  return content.substring(0, lineEnd);
}
```

We write the CRUD functions in `lib/server/note.ts`. We start with imports and the `Note` definition:

```
import { ServerRequest, ServerResponse, ServerError, StatusCode, validate } from "phaethon";
import { ObjectId } from "mongodb";
import { Session } from "./index";
import { getUserOrError } from "./user";
import { Note } from "./note";
import { getTitle } from "../shared/note";
import { MenuResult, ItemResult } from "../shared/api";
import * as database from "./database";

export interface Note {
  _id: string;
  userId: string;
  content: string;
}
```

Implementing the handlers

Now we can implement the `list` function. Using the helper functions we wrote previously, we can easily write the following function:

```
export async function list(request: ServerRequest, session: Session): Promise<MenuResult> {
    const user = await getUserOrError(session);
    const results = await database.find(
        database.notes, { userId: user._id });
    const items = results.map(note => ({
        id: note._id.toHexString(),
        title: getTitle(note.content)
    }));
    return { items };
}
```

With `toHexString`, an `ObjectID` can be converted to a string. It can be converted back using `new ObjectID(...)`. The `map` function transforms an array with a specific callback.

In the `find` function, we must search for a note based on a specific ID:

```
export async function find(request: ServerRequest, session: Session): Promise<ItemResult> {
    const user = await getUserOrError(session);
    const id = validate.expect(
        request.query["id"], validate.isString());
    const notes = await database.find(database.notes,
        { _id: new ObjectID(id), userId: user._id });
    if (notes.length === 0) {
        throw new ServerError(StatusCode.ClientErrorNotFound);
    }
    const note = notes[0];
    return {
        id: note._id.toHexString(),
        content: note.content
    };
}
```

Do not forget to add the `userId` in the query. Otherwise, a hacker could find notes of a different user without knowing his/her password.



The `insert`, `update`, and `remove` functions can be implemented as follows. In `insert`, we set `_id` to `undefined`, as MongoDB will add a unique ID itself:

```
export async function insert(request: ServerRequest, session: Session): Promise<ItemResult> {
    const user = await getUserOrError(session);
    const content = validate.expect(
        request.query["content"], validate.isString);
    const note: Note = {
        _id: undefined,
        userId: user._id,
        content
    };
    await database.insert(database.notes, note);
    return {
        id: note._id.toHexString(),
        content: note.content
    };
}

export async function update(request: ServerRequest, session: Session): Promise<ItemResult> {
    const user = await getUserOrError(session);
    const id = validate.expect(
        request.query["id"], validate.isString);
    const content = validate.expect(
        request.query["content"], validate.isString);
    const note: Note = {
        _id: new ObjectId(id),
        userId: user._id,
        content
    };
    await database.update(database.notes,
        { _id: new ObjectId(id), userId: user._id }, note);
    return {
        id: note._id.toHexString(),
        content: note.content
    };
}

export async function remove(request: ServerRequest, session: Session) {
    const user = await getUserOrError(session);
    const id = validate.expect(
        request.query["id"], validate.isString);
    await database.remove(database.notes,
        { _id: new ObjectId(id), userId: user._id });
    return {};
}
```

Request handling

In lib/server/index.ts, we must add references to these functions in handleRequest:

```
async function handleRequest(request: ServerRequest, session: Session): Promise<ServerResponse | Object> {
    const path = request.path.toLowerCase();

    if (path === "/api/login")
        return login(request, session);
    if (path === "/api/logout")
        return logout(request, session);
    if (path === "/api/note/list")
        return note.list(request, session);
    if (path === "/api/note/insert")
        return note.insert(request, session);
    if (path === "/api/note/update")
        return note.update(request, session);
    if (path === "/api/note/remove")
        return note.remove(request, session);
    if (path === "/api/note/find")
        return note.find(request, session);
    throw new ServerError(HttpStatus.ClientErrorNotFound);
}
```

Writing the client side

Just like the weather widget, we will write the client side of the note application with Angular 2. When the application starts, it will try to download the list of notes. If the user is not logged in, we will get an Unauthorized error (status code 401) and show the login form. Otherwise, we can show the menu with all notes, a logout button, and a button to create a new note. When clicking on a note, that note is downloaded and the user can edit it in the note editor. If the user clicks on the **new** button, the user can write the new note in the (same) note editor.

The server uses a cookie to manage the session, so we do not have to do that manually on the client side.

We start with almost the same HTML file saved as static/index.html:

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>My Notes</title>
```

```
<link rel="stylesheet" href="style.css" />
</head>
<body>
  <div id="wrapper">
    <note-application>Loading..</note-application>
  </div>
  <script type="text/javascript">
    var global = window;
  </script>
  <script src="scripts/scripts.js"
type="text/javascript"></script>
</body>
</html>
```

In static/style.css, we add some styles as follows:

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  font-weight: 100;
}
h1, h2, h3 {
  margin: 0 0;
  padding: 0 0;
  color: #C93524;
}
h2 {
  margin: 0 0;
  padding: 0 0;
  color: #1C5C91;
}
#wrapper {
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  width: 450px;
  margin: 10% auto;
}
a:link, a:visited {
  color: #1C5C91;
  text-decoration: underline;
}
a:hover, a:active {
  color: #3B6282;
}
li >a:link, li >a:visited {
  color: #C93524;
  text-decoration: underline;
```

```
}

li >a:hover, li >a:active {
  color: #AD4236;
}

label {
  display: block;
}
```

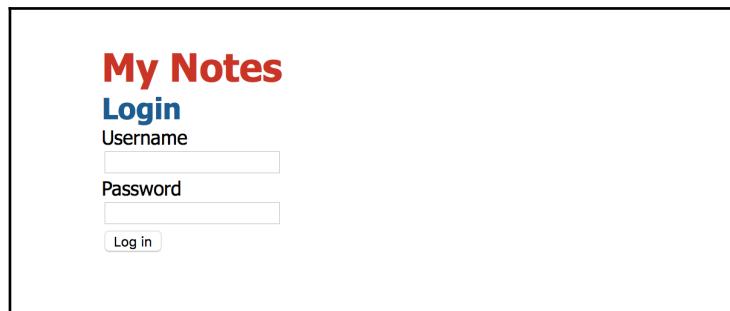
In `lib/client/api.ts`, we create a function, `getUrl`, that will simplify API access. With this function, we can write `getUrl("login", { username: "lorem", password: "ipsum" })` instead of `"login?username=lorem&password=ipsum"`. The function also takes the escaping of characters, such as an ampersand, into account:

```
export const baseUrl = "/api/";

export function getUrl(method: string, query: { [key: string]: string }) {
  let url = baseUrl + method;
  let separator = "?";
  for (const key of Object.keys(query)) {
    url += separator + encodeURIComponent(key) + "=" +
      encodeURIComponent(query[key]);
    separator = "&";
  }
  return url;
}
```

Creating the login form

Now we can create the login form, as shown in the following screenshot:



In lib/client/login.ts, we create the login form. We start with the imports and the template:

```
import { Component, Output, EventEmitter } from "angular2/core";
import { Http, HTTP_PROVIDERS } from "angular2/http";
import { getUrl } from "./api";
import { LoginResult } from "../shared/api";

@Component({
  selector: "login-form",
  template: `
    <h2>Login</h2>
    <form (submit)="submit($event)">
      <div>{{ message }}</div>
      <label>Username<br /><input [(ngModel)]="username">
    </label>
    <label>Password<br /><input type="password" [(ngModel)]="password" />
    <button type="submit">Log in</button>
  </form>
  `,
  viewProviders: [HTTP_PROVIDERS]
})
export class LoginForm {
  username: string;
  password: string;
  message: string;

  constructor(private http: Http) {}
}

submit(e: Event) {
  e.preventDefault();
  this.http.get(getUrl("login", { username: this.username,
  password: this.password }))
    .map(response => response.json())
    .subscribe((response: LoginResult) => {
      if (response.ok) {
        this.success.emit(undefined);
      } else {
        this.message = response.message;
      }
    });
}
```

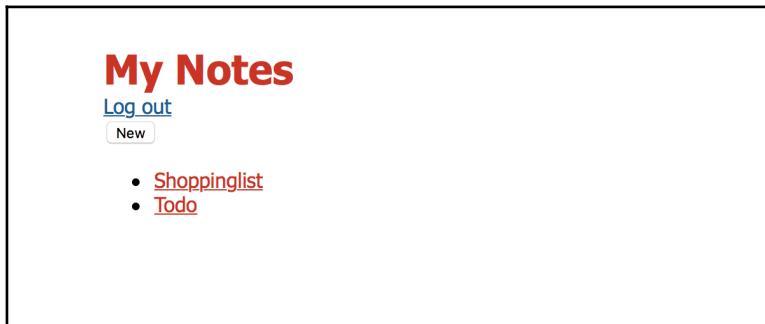
Here we will submit the username and password to the server. If the login is successful, we emit the success event. The main component can then hide the login page and show the menu:

```
submit(e: Event) {
  e.preventDefault();
  this.http.get(getUrl("login", { username: this.username,
  password: this.password }))
    .map(response => response.json())
    .subscribe((response: LoginResult) => {
      if (response.ok) {
        this.success.emit(undefined);
      } else {
        this.message = response.message;
      }
    });
}
```

```
@Output()  
success = new EventEmitter();  
}
```

Creating a menu

In `lib/client/menu.ts`, we create the menu. In the menu, the user will see his/her notes and can create a new note. The menu will look like the following:



This component can emit two different events: `create` and `open`. The second has an argument, so we have to add `string` as type argument:

```
import { Component, Input, Output, EventEmitter } from "angular2/core";  
import { MenuItem } from "../shared/api";  
  
@Component({  
  selector: "notes-menu",  
  template: `  
    <button type="button" (click)="clickCreate()">New</button>  
    <ul>  
      <li *ngFor="#item of items">  
        <a href="javascript:;" (click)="clickItem(item)">{{  
          item.title }}</a>  
      </li>  
    </ul>  
  `,  
})  
export class Menu {  
  @Input()  
  items: MenuItem[];  
  
  @Output()  
  create = new EventEmitter();
```

```

@Output()
open = new EventEmitter<string>();

clickCreate() {
  this.create.emit(undefined);
}

clickItem(item: MenuItem) {
  this.open.emit(item.id);
}

}

```

The note editor

The note editor is a simple text area. Above it, we will show the title of the note. With two-way bindings, the title is automatically updated when the content of the text area is changed.



The main component

Now we can write the main component. This component will show one of the other components, depending on the state. First we must import rxjs, Angular, and the functions and components we have already written:

```

import "rxjs";
import { Component } from "angular2/core";
import { bootstrap } from "angular2/platform/browser";
import { Http, HTTP_PROVIDERS, Response } from "angular2/http";
import { getUrl } from "./api";
import { MenuItem, MenuResult, ItemResult } from "../shared/api";
import { LoginForm } from "./login";
import { Menu } from "./menu";
import { NoteEditor } from "./note";

```

We will use an `enum` type to store the state:

```
enum State {  
    Login,  
    Menu,  
    Note,  
    Error  
}
```

The template shows the right component based on the state. These components have some event listeners attached:

```
@Component ({  
    selector: "note-application",  
    viewProviders: [HTTP_PROVIDERS],  
    directives: [LoginForm, Menu, NoteEditor],  
    template: `  
        <h1>My Notes</h1>  
        <login-form *ngIf="stateLogin" (success)="loadMenu()"></login-form>  
        <div *ngIf="!stateLogin">  
            <a href="javascript:;" (click)="logout()">Log out</a>  
        </div>  
        <notes-menu *ngIf="stateMenu" [items]="menu"  
(create)="createNote()" (open)="loadNote($event)"> </notes-menu>  
        <note-editor *ngIf="stateNote&& note" [content]="note.content"  
(save)="save($event)" (remove)="remove($event)"></note-editor>  
        <div *ngIf="stateError">  
            <h2>Something went wrong</h2>  
            Reload the page and try again  
        </div>  
    `)  
})
```

In the body of the class, we have to add some properties for the state first:

```
class Application {  
    state = State.Menu;  
  
    constructor(private http: Http) {  
        this.loadMenu();  
    }  
  
    get stateLogin() {  
        return this.state === State.Login;  
    }  
    get stateMenu() {  
        return this.state === State.Menu;
```

```

}
get stateNote() {
    return this.state === State.Note;
}
get stateError() {
    return this.state === State.Error;
}

menu: MenuItem[] = [];
note: ItemResult = undefined;

```

Error handler

Now we will write a function that will load the menu. Errors will be passed to `handleError`. If the user was not authenticated, we will find the status code 401 here and show the login form. For a successful request, we can cast the response to the interfaces we defined in `lib/shared/api.ts`:

```

handleError(error: Response) {
    if (error.status === 401) {
        // Unauthorized
        this.state = State.Login;
        this.menu = [];
        this.note = undefined;
    } else {
        this.state = State.Error;
    }
}

loadMenu() {
    this.state = State.Menu;
    this.menu = [];
    this.http.get(getUrl("note/list", {})).subscribe(response => {
        const body = <MenuResult>response.json();
        this.menu = body.items;
    }, error => this.handleError(error));
}

```

We implement the event listeners, `createNote` and `loadNote`, of the menu:

```

createNote() {
    this.note = {
        id: undefined,
        content: ""
    };
    this.state = State.Note;
}

```

```

loadNote(id: string) {
  this.note = undefined;
  this.http.get(getUrl("note/find", { id: id
})).subscribe(response => {
  this.state = State.Note;
  this.note = <ItemResult>response.json();
}, error => this.handleError(error));
}

```

In save, we have to check whether the note is new or being updated:

```

save(content: string) {
  let url: string;
  this.note.content = content;
  if (this.note.id === undefined) {
    // New note
    url = getUrl("note/insert", { content: this.note.content });
  } else {
    // Existing note
    url = getUrl("note/update", { id: this.note.id, content:
this.note.content });
  }

  this.state = State.Note;
  this.note = undefined;
  this.http.get(url).subscribe(response => {
    this.loadMenu();
  }, error => this.handleError(error));
}

remove() {
  if (this.note.id === undefined) {
    this.loadMenu();
    return;
  }
  this.http.get(getUrl("note/remove", { id: this.note.id
})).subscribe(response => {
    this.loadMenu();
  }, error => this.handleError(error));
}

logout() {
  this.http.get(getUrl("logout", {})).subscribe(response => {
    this.state = State.Login;
    this.menu = [];
    this.note = undefined;
  }, error => this.handleError(error));
}
}

```

```
bootstrap(Application).catch(err => console.error(err));
```

Running the application

To test the application, the server and the static files have to be served from the same server. To do that, you can use the `http-server` package. That server can serve the static files and pass through (proxy) the requests to the API server. If MongoDB is not running yet, open a terminal window and run `mongod --dbpath ./data`. Open a terminal window in the root of the project and run the following to start the API server on `localhost:8800`:

```
gulp && node --harmony_destructuring dist/server
```

In a new terminal window, navigate to the `static` directory. Install `http-server` using the following command:

```
npm install http-server -g
```

Now you can start the server:

```
http-server -P http://localhost:8800
```

Open `localhost:8080` in a browser and you will see the application that we have created.

Summary

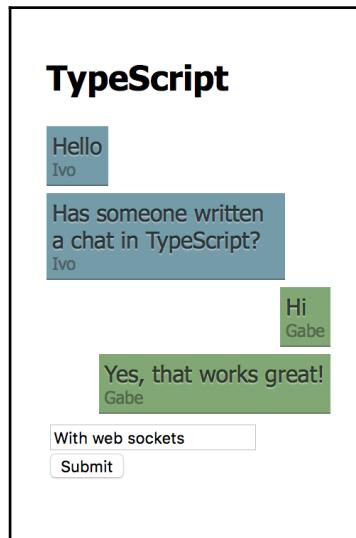
In this chapter, you created a client-server application. You used NodeJS to create a server, with MongoDB and Phaethon. You also learned more about asynchronous programming and the structural type system. We used our knowledge of Angular from the first chapter to create the client side.

In the next chapter, we will create another client-server application. That application is not a CRUD application, but a real-time chat application. We will be using React instead of Angular.

4

Real-Time Chat

After having written two applications with Angular 2, we will now create one with **React**. The server part will also be different. Instead of a connectionless server, we will now create a server with a persistent connection. In the previous chapters, the client sent requests to the server and the server responded to them. Now we will write a server that can send information at any time to the client. This is needed to send new chat messages immediately to the client, as shown in the following:



In the chat application, a user can first choose a username and join a chat room. In the room, he/she can send messages and receive messages from other users. In this chapter, we will cover the following topics:

- Setting up the project
- Getting started with React
- Writing the server
- Connecting to the server
- Creating the chat room
- Comparing React and Angular

Setting up the project

Before we can start coding, we have to set up the project. The directory structure will be the same as in Chapter 3, *Note-Taking App with Server*; static contains the static files for the webserver, lib/client contains the client-side code, lib/server contains the code for the server, lib/shared contains the code that can be used on both sides, and lib/typings contains the type definitions for React.

We can install all dependencies, for gulp, the server, and React, as follows:

```
npm init
npm install react react-dom ws --save
npm install gulp gulp-sourcemaps gulp-typescript gulp-uglify small --save-dev
```

The type definitions can be installed using npm:

```
cd lib
npm install @types/node @types/react @types/react-dom @types/ws --save
```

We create static/index.html, which will load the compiled JavaScript file:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Chat</title>
    <link href="style.css" rel="stylesheet" />
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript">
      var process = {
```

```

    env: {
      NODE_ENV: "DEBUG" // or "PRODUCTION"
    }
  };
</script>
<script src="scripts/scripts.js" type="text/javascript"></script>
</body>
</html>

```

We add styles in static/style.css:

```

body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
label, input, button {
  display: block;
}

```

Configuring gulp

We can use almost the same `gulpfile`. We do not have to load any polyfills for React, so the resulting file is even simpler:

```

var gulp = require("gulp");
var sourcemaps = require("gulp-sourcemaps");
var typescript = require("gulp-typescript");
var small = require("small").gulp;
var uglify = require("gulp-uglify");

var tsServer = typescript.createProject("lib/tsconfig.json", {
  typescript: require("typescript") });

var tsClient = typescript.createProject("lib/tsconfig.json", { typescript:
  require("typescript"), target: "es5" });

gulp.task("compile-client", function() {
  return gulp.src(["lib/client/**/*.ts", "lib/client/**/*.tsx",
    "lib/shared/**/*.ts"], { base: "lib" })
    .pipe(sourcemaps.init())
    .pipe(typescript(tsClient))
    .pipe(small("client/index.js", { outputFileName: {
      standalone: "scripts.js" }, externalResolve:
      ["node_modules"] }))
    .pipe(sourcemaps.write("."))
    .pipe(gulp.dest("static/scripts")));
});

```

```
gulp.task("compile-server", function() {
    return gulp.src(["lib/server/**/*.ts", "lib/shared/**/*.ts"], {
        base: "lib"
    })
        .pipe(sourcemaps.init())
        .pipe(typescript(tsServer))
        .pipe(sourcemaps.write("."))
        .pipe(gulp.dest("dist"));
});
gulp.task("release", ["compile-client", "compile-server"], function() {
    return gulp.src("static/scripts/**.js")
        .pipe(uglify())
        .pipe(gulp.dest("static/scripts"));
});
gulp.task("default", ["compile-client", "compile-server"]);
```

In lib/tsconfig.json, we configure TypeScript. We have to set the `jsx` option. In React, views are written in an XML-like language, **JSX**, inside JavaScript. To use this in TypeScript, you have to set the `jsx` option and use the file extension `.tsx` instead of `.jsx`.

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es6",
        "jsx": "react",
        "types": ["node"]
    }
}
```

Getting started with React

Just like Angular, React is component based. Angular is called a framework, whereas React is called a library. This means that Angular provides a lot of different functionalities and React provides one functionality, **views**. In the first two chapters, we used the HTTP service of Angular. React does not provide such a service, but you can use other libraries from npm instead.

Creating a component with JSX

A component is a class that has a `render` method. That method will render the view and is the replacement of the template in Angular. A simple component would look like the following:

```
export class Example extends React.Component<{}, {}> {
  render() {
    const name = "World";
    return (
      <div>
        Hello, { name }!
        <button onClick={() => alert("Hello")}>
          Click me
        </button>
      </div>
    );
  }
}
```

As you can see, you can embed HTML inside the `render` function. Expressions can be wrapped inside curly brackets, both in text and in properties of other components. Event handlers can be added in this way too. Instead of using built-in components, you can use custom components in these handlers. All built-in components start with a lowercase character and custom elements should start with an uppercase character. This is not just a convention, but required by React. We have to use a different syntax for type casts in `.tsx` files, as the normal syntax conflicts with the XML elements. Instead of `<Type> value`, we will now write `value as Type`. In `.ts` files, we can use both styles.

Adding props and state to a component

In the example, the component extends the `React.Component` class. That class has two type arguments, which represent the props and the state. The props contain the input that the parent component gives to this one. You can compare that to the `@Input` directive in Angular. You cannot modify the props in the containing class. The state contains the other properties of a component in Angular, which can be modified in the class. You can access the props with `this.props` and the state with `this.state`. The state cannot be modified directly, as you have to replace the state with a new object. Imagine the state contains two properties, `foo` and `bar`. If you want to modify `foo` and `bar`, it is not allowed with `this.state.foo = 42`, but you have to write `this.setState({ foo: 42, bar: true })` instead. In most cases, you do not have to change all properties of the state. In such cases, you only have to specify the properties that you want to change. For instance, `this.setState({ foo: 42, bar: true })` will change the value of `foo` and keep the old value of `bar`. The state object is then replaced by a new object. The state object will never change. Such an object is called an immutable object. We will read more on these objects in [Chapter 5, Native QR Scanner App](#).

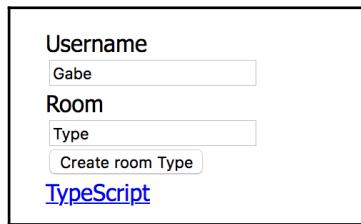
The component will be re-rendered by React after calling `setState`.

In other parts of the application, we will also need to modify a few properties of an object. For big objects, this becomes annoying. We create a helper function, which requires the old state, adds modifications to it, and returns a new state. This function does not change the old state, but returns a new one. In `lib/client/model.ts`, we create the `modify` function:

```
export function modify<U extends V, V>(old: U, changes: V) {
  const result: any = Object.create(Object.getPrototypeOf(old));
  for (const key of Object.keys(old)) {
    result[key] = old[key];
  }
  for (const key of Object.keys(changes)) {
    result[key] = changes[key];
  }
  return <U> result;
}
```

Creating the menu

We will start with the menu of our application. In the menu, the user can choose the chat room that he/she wants to join. The menu will first ask the user for a username. Afterward, the user can type the name of a chat room. The user will get completions for known rooms, but he/she can also create a new room. Let's check the following screenshot as an example of menu:



The component will delegate the completions to its parent, so we need to add the current list of completions to the props, such that the parent can set it. Also, we need to add a callback that can be called when the completions must be fetched.

The state must contain the username and the room name. React does not have two-way bindings, so we have to use event listeners to update the username and room name in the state.

We will disable the rest of the menu if the user hasn't provided the username. When the user has filled in a room, we show a list of completions and a button to create a new room with the specified name.

We write the code in `lib/client/menu.tsx`. First, we define the props and state in two different interfaces:

```
import * as React from "react";
import { modify } from "./model";

interface MenuProps {
  completions: string[];
  onRequestCompletions: (room: string) => void;
  onClick: (username: string, room: string) => void;
}

interface MenuState {
  username: string;
  roomname: string;
}
```

Second, we create the class. We set the initial state with an empty username and room name:

```
export class Menu extends React.Component<MenuProps,
MenuState> {
  state = {
    username: "",
    roomname: ""
  };
}
```

In the render function, we use JSX to show the component. We can use normal TypeScript constructs. There is no need to use something like `NgIf` or `NgFor`, as we did in Angular:

```
render() {
  const menuEnabled = this.state.username !== "";
  const menuStyle = {
    opacity: menuEnabled ? 1 : 0.5
  };
  const showCreateButton = menuEnabled
    && this.state.roomname !== ""
    && this.props.completions
      .indexOf(this.state.roomname) === -1;
  return (<div>
    <label htmlFor="username">Username</label>
    <input type="text" id="username" onChange=
      {e => this.changeUsername(
        (e.target as HTMLInputElement).value)} />
```

```

<div style={menuStyle}>
  <label htmlFor="roomname">Room</label>
  <input type="text" id="roomname"
    disabled={!menuEnabled}
    onChange={e =>
      this.changeName(
        (e.target as HTMLInputElement).value)
    } />
  { showCreateButton
    ? <button onClick={
        () => this.submit(this.state.roomname) }>
        Create room { this.state.roomname }</button>
    : "" }
  { this.props.completions.map(
    completion =>
      <a href="javascript:;""
        key={completion}
        style={{display: "block"}}
        onClick={() =>
          this.submit(completion)}>
        { completion }</a> )
  }
</div>
</div>);
}

```

Finally, we can implement the listeners:

```

private changeUsername(username: string) {
  this.setState({ username });
}

private changeName(roomname: string) {
  this.setState({ roomname });
  this.props.onRequestCompletions(roomname);
}

private submit(room: string) {
  this.props.onClick(this.state.username, room);
}

```

We can see this component in action by adding the following in lib/client/index.tsx:

```
ReactDOM.render(<Menu completions={[]} onRequestCompletions={() => {}}
  onClick={() => {}}>);
```

This will render the menu in the HTML file.

Testing the application

To view the application in a browser, you must first build it using `gulp`. You can execute `gulp` in a terminal. Afterward, you can open `static/index.html` in a browser.

Writing the server

To add interaction to the application, we must create the server first. We will use the `ws` package to easily create a websocket server. On the websocket, we can send messages in both directions. These messages are objects converted to strings with JSON, just like in the previous chapters.

Connections

In the previous chapter, we wrote a connectionless server. For every request, a new connection was set up. We could store a state using a session. Such session was identified with a cookie. If you were to copy that cookie to a different computer, you would have the same session there.

Now we will write a server that uses connections. In this way, the server can easily keep track of which user is logged in and where. The server can also send a message to the client without a direct request. This automatic updating is called pushing. The opposite, pulling, or polling, means that the client constantly asks the server whether there is new data.

With connections, the order of arrival is the same as the order of sending. With a connectionless server, a second message can use a different route and arrive earlier.

Typing the API

We will type these messages in `lib/shared/api.ts`. In the previous chapter, the URL identified the function to be called. Now, we must include that information in the message object. We type the messages from the client to the server and vice versa:

```
export enum MessageKind {  
    FindRooms,  
    OpenRoom,  
    SendMessage,  
  
    RoomCompletions,  
    ReceiveMessage,
```

```

    RoomContent
}
export interface Message {
  kind: MessageKind;
}

export type ClientMessage = OpenRoom | ChatMessage | FindRooms;
export type ServerMessage = RoomContent | ChatMessage;

export interface FindRooms extends Message {
  query: string;
}
export interface OpenRoom extends Message {
  room: string;
}
export interface RoomCompletions extends Message {
  completions: string[];
}
export interface RoomContent extends Message {
  room: string;
  messages: ChatContent[];
}
export interface SendMessage extends Message {
  text: string;
}
export interface ChatMessage extends Message {
  content: ChatContent
}

export interface ChatContent {
  room: string;
  username: string;
  content: string;
}

```

Accepting connections

In `lib/server/index.ts`, we create a server that listens for new connections. We also keep track of all open connections. When a message is sent in a chat room, it can be forwarded to all sessions that have opened that room. We use `ws` to create a websocket server:

```

import * as WebSocket from "ws";
import * as api from "../shared/api";

const server = new WebSocket.Server({ port: 8800 });

```

```
server.on("connection", receiveConnection);

interface Session {
    sendChatMessage(message: api.ChatContent): void;
}
const sessions: Session[] = [];
```

We will store the recent messages in an array. We limit the size of the array, as an attacker could otherwise fill the whole memory of a server with a (D)DOS attack: if a user sends a lot of messages (automatically), this will cost a lot of server memory. If multiple users do that, the memory can be filled entirely and the server will crash.

Storing recent messages

You can implement this with an array by removing the first message and appending the new message at the end. However, this would shift the whole array, especially large arrays that can take some time. Instead, we use a different approach. We use an array that can be seen as a circle: after the last element comes the first one. We use a variable that points to the oldest message. When a new message is added, the item at the position of the pointer is overwritten with the new message. The pointer is incremented with one and points again to the oldest message. When the messages A, B, C and D are sent with an array size of 3, this can be visualized like the following:

```
[-, -, -]; pointer = 0
[A, -, -]; pointer = 1
[A, B, -]; pointer = 2
[A, B, C]; pointer = 0
[D, B, C]; pointer = 1
```

If you are familiar with analyzing algorithms and **Big-Oh** notation, this takes $O(1)$, whereas the naive idea takes $O(n)$. We create the array in `lib/server/index.ts`:

```
const recentMessages: api.ChatContent[] = new Array(2048);
let recentMessagesPointer = 0;
```

We do not save the messages to disk. You could do that and use a cache with such array to increase the performance of the server.

Handling a session

For each connection, we have to keep track of the username and room name of the user. We can do that with variables inside the `receiveConnection` function:

```
function receiveConnection(ws: WebSocket) {
    let username: string;
    let room: string;
```

We can listen to the `message` and `close` events. The first is emitted when the client has sent a message in the websocket. The second is emitted when the websocket has been closed. When the socket is closed, we must not send any messages to it and we must remove it from the `sessions` array:

```
ws.on("message", message);
ws.on("close", close);
const session: Session = { sendChatMessage };
sessions.push(session);

function message(data) {
    try {
        const object = <api.ClientMessage> JSON.parse(data);
        if (typeof object.kind !== "number") return;
        switch (object.kind) {
            case api.MessageKind.FindRooms:
                findRooms(<api.FindRooms> object);
            case api.MessageKind.OpenRoom:
                openRoom(<api.OpenRoom> object);
                break;
            case api.MessageKind.SendMessage:
                chatMessage(<api.SendMessage> object);
                break;
        }
    } catch (e) {
        console.error(e);
    }
}

function close() {
    const index = sessions.indexOf(session);
    sessions.splice(index, 1);
}

function send(data: api.ServerMessage) {
    ws.send(JSON.stringify(data));
}
```

The server should always validate the input that it gets. The data could not be a JSON string, which would cause `JSON.parse` to throw an error. `object.kind` might not be a number, as TypeScript does not do any runtime checks. We can validate that with a `typeof` check.



If you would not have added a `try/catch`, the server would crash if the client sends a message that is not the correct JSON. To prevent this, we will catch that error. For debugging, we write the error on the console.

Implementing a chat message session

Now we can implement the functions that are called when a message comes in. We start with the function that sends a chat message to all active connections in that room and stores it in the array with recent messages:

```
function sendChatMessage(content: api.ChatContent) {
  if (content.room === room) {
    send({
      kind: api.MessageKind.ReceiveMessage,
      content
    });
  }
}

function chatMessage(message: api.SendMessage) {
  if (typeof message.content !== "string") return;

  const content: api.ChatContent = {
    room,
    username,
    content: message.content
  };

  recentMessages[recentMessagesPointer] = content;
  recentMessagesPointer++;
  if (recentMessagesPointer >= recentMessages.length) {
    recentMessagesPointer = 0;
  }

  for (const item of sessions) {
    if (session !== item) item.sendChatMessage(content);
  }
}
```

This will send a chat message to all other sessions in the same room. We insert the message at the right location in `recentMessages` and adjust the pointer.

Finally, we will write the function that gives completions for room names. We do not have an array of room names, so we have to get that information from the recent messages. The resulting array can contain duplicates, so we have to remove these. A naive approach would be to check for every element if it has occurred before in the array. However, this is a slow operation. Instead, we sort the array first. After sorting, we only have to compare each element with the element before it. If these are equal, the second is a duplicate, otherwise it is not. For those familiar with **Big-Oh**, the first approach costs $O(n^2)$ and the second one costs $O(n \log(n))$. This results in the following function:

```
function findRooms(message: api.FindRooms) {
    const query = message.query;
    if (typeof query !== "string") return;

    const rooms = recentMessages
        .map(msg => msg.room)
        .filter(room => room.toLowerCase().indexOf(query.toLowerCase()) !==
-1)
        .sort();
    const completions: string[] = [];
    let previous: string = undefined;
    for (let room of rooms) {
        if (previous !== room) {
            completions.push(room);
            previous = room;
        }
    }
    send({
        kind: api.MessageKind.RoomCompletions,
        completions
    });
}
```

We have completed the server and can focus on the client side again.

Connecting to the server

We can connect to the server with the `WebSocket` class:

```
const socket = new WebSocket("ws://localhost:8800/");
```

Since we're using React, we add the following to the state. We create a new component, `App`, that will show the menu or a chat room based on the state. In `lib/client/index.tsx`, we first define the state and props of that component:

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import * as api from "../shared/api";
import * as model from "./model";
import { Menu } from "./menu";
import { Room } from "./room";

interface Props {
  apiUrl: string;
}

interface State {
  socket: WebSocket;
  username: string;
  connected: boolean;
  completions: string[];
  room: model.Room;
}

class App extends React.Component<Props, State> {
  state = {
    socket: undefined,
    username: '',
    connected: false,
    completions: [],
    room: undefined
  };
}
```

Automatic reconnecting

Next up, we will write a function, `connect`, that connects to the server using a `WebSocket`. We call that function in `componentDidMount`, which is called by React. We must also call `connect` when the connection gets closed for some reason (for instance, network problems). We store the socket in the state and we also keep track of whether the client is connected:

```
connect() {
  if (this.state.connected) return;
```

```

const socket = new WebSocket(this.props.apiUrl);
this.setState({ socket });
socket.onopen = () => {
this.setState({ connected: true });
if (this.state.room) {
this.openRoom(this.state.username, this.state.room.name);
}
};
socket.onmessage = e => this.onMessage(e);
socket.onclose = e => {
this.setState({ connected: false });
setTimeout(() => this.connect(), 400);
};
}
onMessage(e: MessageEvent) {
const message = JSON.parse(e.data.toString()) as api.ServerMessage;
if (message.kind === api.MessageKind.RoomCompletions) {
this.setState({
completions: (message as api.RoomCompletions).completions
});
} else if (message.kind === api.MessageKind.RoomContent) {
this.setState({
room: {
name: (message as api.RoomContent).room,
messages: (message as api.RoomContent).messages.map(msg =>
this.mapMessage(msg))
}
});
} else if (message.kind === api.MessageKind.ReceiveMessage) {
this.addMessage(this.mapMessage((message as api.ReceiveMessage).content));
}
}
componentDidMount() {
this.connect();
}

```

`socket.onmessage` is called when the client receives a message from the server. Based on the kind of message, it is sent to some function that we will implement later. First, we will write the render function. After we have written the render function, we know which event handlers we have to write.

When you write **top down**, you first write the main function and afterward the helper functions that the main function requires. With the **bottom up** approach, you write the helper functions before you write the main function. In this section, we write the helper functions last, so we write top down. You can try both styles and find out what you like most.



In render, we render the component based on the state—if there is no connection, we show **Connecting...**, if the user is in a room, we show that chat room, otherwise we show the menu:

```
render() {
    if (!this.state.connected) {
        return <div>Connecting...</div>;
    }
    if (this.state.room) {
        return <Room room={this.state.room} onPost={content =>
this.post(content)} />;
    }
    return <Menu
        completions={this.state.completions}
        onRequestCompletions={query =>
this.requestCompletions(query)}
        onClick={(username, room) =>
this.openRoom(username, room)}
    />;
}
```

Sending a message to the server

Before writing the event handlers, we first write a small function that sends a message to the server. It converts an object to JSON, and TypeScript will check that we are sending a correct message to the server:

```
private send(message: api.ClientMessage) {
    this.state.socket.send(JSON.stringify(message));
}
```

The `requestCompletions` and `openRoom` functions send a message to the server. In `openRoom`, we also have to store the username in the state:

```
private requestCompletions(query: string) {
    this.send({
        kind: api.MessageKind.FindRooms,
        query
    });
}
private openRoom(username, room: string) {
    this.send({
        kind: api.MessageKind.OpenRoom,
        username,
        room
    });
}
```

```
    this.setState({ username });
}
```

Writing the event handler

For iterations in React, every element should have a key that can identify it. Thus, we need to give every message such a key. We use a simple numeric key, which we will increment for every message:

```
private nextMessageId: number = 0;
private post(content: string) {
  this.send({
    kind: api.MessageKind.SendMessage,
    content
  });
  this.addMessage({
    id: this.nextMessageId++,
    user: this.state.username,
    content,
    isAuthor: true
  });
}
private addMessage(msg: model.Message) {
  const messages = [
    ...this.state.room.messages,
    msg
  ].slice(Math.max(0, this.state.room.messages.length - 10));
  const room = model.modify(this.state.room, {
    messages
  });
  this.setState({ room });
}
private mapMessage(msg: api.ChatContent) {
  return {
    id: this.nextMessageId++,
    user: msg.username,
    content: msg.content,
    isAuthor: msg.username === this.state.username
  };
}
}
```

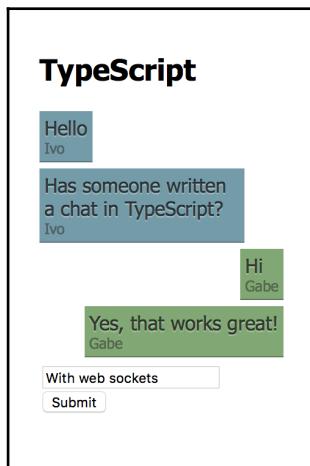
Finally, we can show the component in the HTML file:

```
ReactDOM.render(  
<App apiUrl="ws://localhost:8800/" />, document.getElementById("app")  
);
```

We have now written all event handlers and interaction with the server. We write the chat room component in the next section.

Creating the chat room

We divide the chat room into two subcomponents: a message and the input box. When the user sends a new message, it is sent to the main component. Message of the user will be shown on the right and other messages on the left, as shown in the following screenshot:



Two-way bindings

React does not have two-way bindings. Instead, we can store the value in the state and modify it when the `onChange` event is fired. For the input box, we will use this technique. The textbox should be emptied when the user has sent his/her message. With this binding, we can easily do that by modifying the value in the state to an empty string:

```
class InputBox extends React.Component<{ onSubmit(value: string): void; },  
{ value: string }> {  
  state = {  
    value: ""  
  }  
  ...  
}
```

```

};

render() {
  return (
    <form onSubmit={e => this.submit(e)}>
      <input onChange={e => this.changeValue((e.target as
HTMLInputElement).value)} value={this.state.value} />
      <button disabled={this.state.value === ""}
type="submit">Submit</button>
    </form>
  );
}

private changeValue(value: string) {
  this.setState({ value });
}

private submit(e: React.FormEvent<{}>) {
  e.preventDefault();
  if (this.state.value) {
    this.props.onSubmit(this.state.value);
    this.state.value = "";
  }
}
}

```

Stateless functional components

If a component doesn't need a state, then it does not need a class to store and manage that state. Instead of writing a class with just a render function, you can write that function without the class. These components are called stateless functional components. A message is clearly stateless, as you cannot modify a message that has already been sent:

```

function Message(props: { message: model.Message }) {
  return (
    <div>
      <div className={props.message.isAuthor ? "message"
own" : "message"}>
        { props.message.content }
        <div className="message-user">
          { props.message.user }
        </div>
      </div>
      <div style={{clear: "both"}}></div>
    </div>
  );
}

```

A stateless functional component can have a child component with a state. The input box

has a state and can be used inside Room, which is a stateless component. We have to set the key property in the array of messages. React uses this to identify components inside the array:

```
export function Room(props: { room: model.Room, onPost: (content: string) => void }) {
  return (
    <div>
      <h2>{props.room.name}</h2>
      {props.room.messages.map(message => <Message key={message.id} message={message} />)}
      <Input onSubmit={content => props.onPost(content)} />
    </div>
  );
}
```

Running the application

We can now run the whole application. First, we must compile it with gulp. Second, we can start the server by running `node dist/server` in a terminal. Finally, we can open `static/index.html` in a browser and start chatting. When you open this page multiple times, you can simulate multiple users.

Comparing React and Angular

In the previous chapters, we used Angular and in this chapter we used React. Angular and React are both focused on components, but there are differences, for instance, between the templates in Angular and the views in React. In this section, you can read more about these differences.

Templates and JSX

Angular uses a template for the view of a component. Such a template is a string that is parsed at runtime. TypeScript cannot check these templates. If you misspell a property name, you will not get a compile error.

React uses JSX, which is syntactic sugar around function calls. A JSX element is transformed, by the compiler, into a call to `React.createElement`. The first argument is the name of the element or the element class, the second argument contains the props, and the other arguments are the children of the component. The following example demonstrates the transform:

```
<div></div>;
React.createElement("div", null);

<div prop="a"></div>;
React.createElement("div", { prop: "a" });

<div>Foo</div>;
React.createElement(
  "div",
  null,
  "Foo"
);

<div><span>Foo</span></div>;
React.createElement(
  "div",
  null,
  React.createElement(
    "span",
    null,
    "Foo"
  )
);
```

Elements that start with a capital letter or contain a dot are considered to be custom components, and other elements are treated as intrinsic elements, the standard HTML elements:

```
<div></div>;
React.createElement("div", null);

<Foo></Foo>;
React.createElement(Foo, null);
```

These JSX elements are checked and transformed at compile time, so you do get an early error when you misspell a property. React is not the only framework that uses JSX, but it is the most popular one.

Libraries or frameworks

Angular is a framework and React is a library. A library provides one functionality in the case of React—the views of the application. A framework provides a lot of different functionalities. For instance, Angular renders the views of the application, but also has, for instance, dependency injection and an `Http` service. If you want such features when you are using React, you can use another library that gives that feature.

React programmers often use a Flux based architecture. Flux is an application architecture that is implemented in various libraries. In Chapter 5, *Native QR Scanner App*, we will take a look at this architecture.

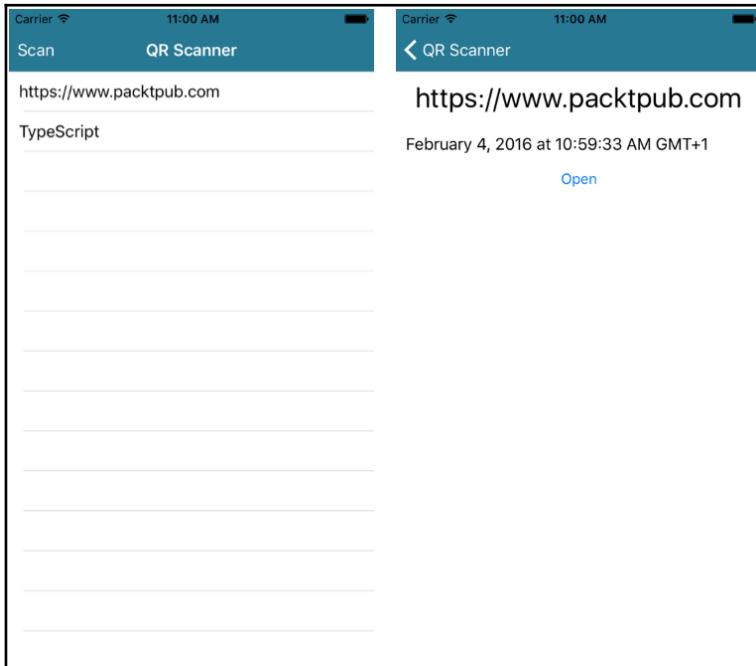
Summary

We have written an application with websockets. We have used React and JSX for the views of our application. We have seen multiple ways to create components and learned how the JSX transform works. In Chapter 5, *Native QR Scanner App*, we will use React again, but we will first take a look at mobile apps with NativeScript in the next chapter.

5

Native QR Scanner App

We have already used TypeScript to build web apps and a server. TypeScript can also be used to create mobile apps. In this chapter, we will build such an app. The app can scan QR codes. The app shows a list of all previous scans. If a QR code contains a URL, the app can open that URL in a web browser. Various frameworks exist for making mobile apps in TypeScript. We will use **NativeScript**, which provides a native user interface and runs on Android and iOS, as shown in the following:



We will create this app with the following steps:

- Creating the project structure
- Creating a Hello World page
- Creating the main view
- Adding a details view
- Scanning QR codes
- Adding persistent storage
- Styling the app
- Comparing NativeScript to alternatives

Getting started with NativeScript

Installing NativeScript requires several steps. For developing apps for Android, you have to install **Java Development Kit (JDK)** and the **Android SDK**. Android apps can be built on Windows, Linux, and Mac. Apps for iOS can only be built on a Mac. You need to install **XCode** to build these apps.

You can find more details on how to install the Android SDK at <https://docs.nativescript.org/start/quick-setup>.

After installing the Android SDK or XCode, you can install NativeScript using npm:

```
npm install nativescript -g
```

You can see whether your system is configured correctly by running the following command:

```
tns doctor
```

If you only want to develop apps for iOS, you can ignore the errors on Android and vice versa.

We can test most parts of the app in a simulator that is included in the **SDK** or **XCode**. Scanning a QR code only works on a device.



Setting up XCode for iOS development is easier than installing the Android SDK. If you can choose between iOS and Android, you want to choose iOS.

Creating the project structure

In the previous chapters, we wrote our TypeScript sources in the `lib` directory. The `static` or `dist` directory contained the compiled sources. However, in this chapter, we have to make a different structure since NativeScript has some requirements on it. NativeScript requires that the compiled sources are located in the `app` directory and it uses the `lib` directory for plugins, so we cannot use that directory for our TypeScript sources. Instead, we will use the `src` directory.

NativeScript can automatically create a basic project structure. By running the following commands, a minimal project will be created:

```
tns init  
npm install
```

The first command creates the `package.json` file and the `app` directory. NativeScript stores the icons and splash screens (which you see when the app is loading) in `app`. You can edit these files when you want to publish an app. The `npm install` command installs the dependencies that NativeScript needs. These dependencies were added to `package.json` by the first command.

We need to make some adjustments to it. We must create an `app/package.json` file. NativeScript uses this file to get the main file of the project:

```
{  
  "main": "app.js"  
}
```

Adding TypeScript

By default, NativeScript apps should be written in JavaScript. We will not use `gulp` to compile our TypeScript files since NativeScript has built-in support for transpilers like TypeScript. We can add TypeScript to it by running the following command:

```
tns install typescript
```

After running this command, NativeScript will automatically compile TypeScript to JavaScript. This command has created two files: `tsconfig.json` and `references.d.ts`. The `tsconfig` file contains the configuration for TypeScript. We will add the `outDir` option to `tsconfig.json` so that we do not have to place the source files in the same direction as the compiled files. NativeScript requires that JavaScript files are placed in the `app` folder. We will write our TypeScript sources in the `src` folder, and the compiler will write the output to the `app` folder:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "inlineSourceMap": true,
    "experimentalDecorators": true,
    "noEmitHelpers": true,
    "outDir": "app"
  },
  "exclude": [
    "node_modules",
    "platforms"
  ]
}
```

The `references.d.ts` file contains a reference to the definition files (`.d.ts` files) of the core modules of NativeScript.

Creating a Hello World page

To get started with NativeScript, we will first write a simple app. In `src/app.ts`, we must register `mainEntry` that will create the view of the app. The entry should be a function that returns a `Page`. A `Page` attribute is one of the classes that NativeScript uses for the user interface. We can create a basic page as follows:

```
import * as application from "application";
import { Page } from "ui/page";
import { Label } from "ui/label";

application.mainEntry = () => {
    const page = new Page();
    const label = new Label();
    label.text = "Hello, World";
    page.content = label;
    return page;
};

application.start();
```

This will create a single label and add it to the page. The content of the page should be a `View` class, which is the base class that all components (including `Label`) in NativeScript inherit.

You can run the app with one of the following commands for Android and iOS, respectively:

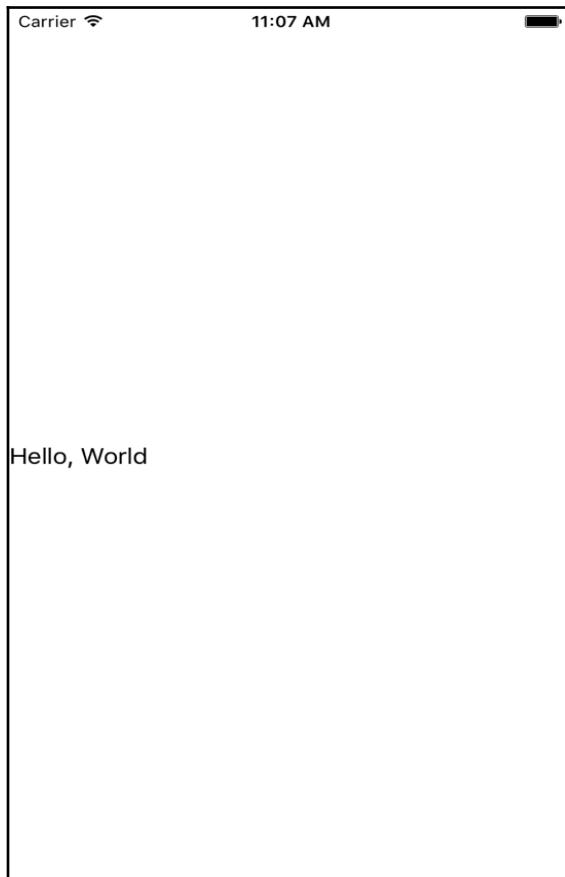
```
tns run android --emulator
tns run ios --emulator
```

You can run the app on a device by removing `--emulator`. Your device should be connected using a USB cable. You can see all connected devices by running `tns device`.

NativeScript prints a lot on the console after the output of TypeScript. Make sure you do not miss any compile errors of TypeScript.



On iOS, the app will now look like the following:



In the next sections, we will see how we can add event listeners and build bigger views.

Creating the main view

The main view will show a list of recent scans. Clicking on one of the recent scans opens a details page that shows more details on the scan. When a user clicks on the **Scan** button, the user can scan a QR code using the camera:



First, we create the model of a scan in `src/model.ts`. We need to store the content (a string) and the date of the scan:

```
export interface Scan {  
    content: string;  
    date: Date;  
}
```

In `src/view/main.ts`, we will create the view. The view should export a function that creates the page, so we can use it as the `mainEntry`. It also needs to export a function that can update the content. The view has two callbacks or events: one is called when an item is clicked and the other is called when the user clicks on the **Scan** button. This can be implemented by adding the two callbacks as arguments of the `createPage` function and returning `setItems`, which updates the content of the list, and `createView`, which creates the Page, as an object:

```
import { Page } from "ui/page";
import { ActionBar, ActionItem } from "ui/action-bar";
import { ListView } from "ui/list-view";

export function createPage(itemCallback: (index: number) => void,
scanCallback: () => void) {
    let items: string[] = [];
    let list: ListView;

    return { setItems, createView };
    function setItems(value: string[]) {
        items = value;
        if (list) {
            list.items = items;
            list.refresh();
        }
    }
}
```

An `ActionBar` is the bar at the top of the screen with the app name. We add an `ActionItem` attribute to it, which is a button in the bar. We use a `ListView` attribute to show the recent scans in a list. Elements have an `on` method, which we use to listen to events, similar to `addEventListener` in websites and `on` in NodeJS.

The `itemLoading` event is fired when an item in the list is being rendered. In that event, the view for an item of the list should be created. The `tap` event is fired when the user taps on the scan button. The `itemCallback` event will be invoked with the index of the item when that happens.

First, we create the action bar. We add it to the page and add a button to the action bar:

```
function createView() {
    const page = new Page();
    const actionBar = new ActionBar();
    actionBar.title = "QR Scanner";
    const buttonScan = new ActionItem();
    buttonScan.text = "Scan";
    buttonScan.on("tap", scanCallback);
    actionBar.actionItems.addItem(buttonScan);
```

Next, we create the list as follows:

```
list = new ListView();
list.items = items;
```

Finally, we add event listeners to the list. In itemLoading, we create Label, if it was not created yet, and set the text of it. In itemTap, we call itemCallback with the index of the tapped item:

```
list.on("itemLoading", args => {
    if (!args.view) {
        args.view = new Label();
    }
    (<Label> args.view).text = items[args.index];
});
list.on("itemTap", e => itemCallback(e.index));

page.actionBar = actionBar;
page.content = list;
return page;
}
```

In `src/app.ts`, we can call this function and show the `view` attribute:

```
import * as application from "application";
import { createPage } from "./view/main";
import * as model from "./model";

let items: model.Scan[] = [];

const page = createPage(index => showDetailsPage(items[index]), scan);
application.mainEntry = page.createView;
application.cssFile = "style.css";
application.start();
```

We will implement scanning later on. For now, we will always add a fake scan, so we can test the other parts of the application:

```
function scan() {
    addItem("Lorem");
}
```

In `addItem`, we add a new scan to the list of scans. We call `update`, which will update the list in the main view and show the details page with this scan. We limit the amount of scans in the list by 100:

```
function addItem(content: string) {
    const item: model.Scan = {
        content,
        date: new Date()
    };
    items = [item, ...items].slice(0, 100);
    update();
    showDetailsPage(item);
}
```

We will implement the details page in the next section. For now, we will only add a placeholder function so that we can test the other functions:

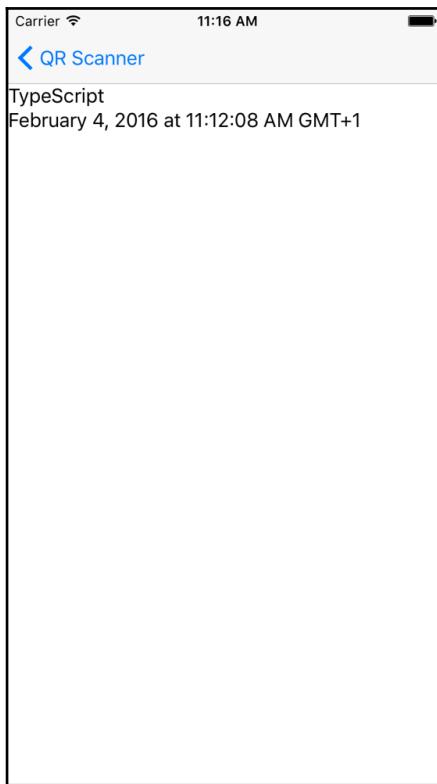
```
function showDetailsPage(scan: model.Scan) {
```

In `update`, we change the values in the list to the new items:

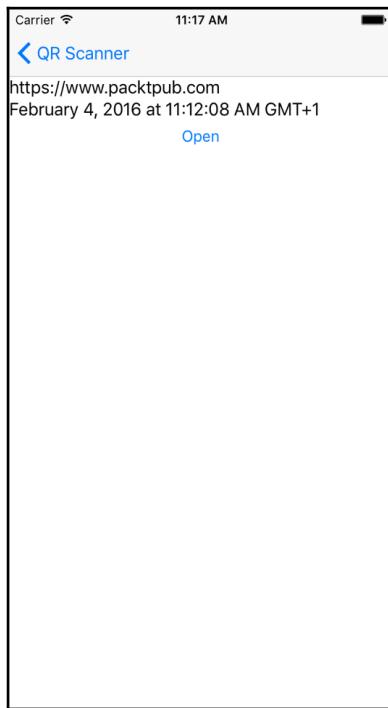
```
function update() {
    page.setItems(items.map(item => item.content));
}
```

Adding a details view

The details view is shown when the user scans a code or clicks on an item in the recent scans list. It shows the content of the scan and the date, as shown in the following screenshot:



If the content of the scan is a URL, we will show a button to open that link, as shown in the following screenshot:



At the end of this chapter, we will style this page properly.

We add a function to `src/model.ts` that will return `true` when the scan (probably) contains a URL. We consider a scan that contains no spaces and begins with `http://` or `https://` to be a URL:

```
function startsWith(input: string, start: string) {
  return input.substring(0, start.length) === start;
}
export function isUrl({ content }: Scan) {
  if (content.indexOf(" ") !== -1) {
    return false;
  }
  return startsWith(content, "http://") || startsWith(content,
"https://");
}
```

The view requires the scan itself and optionally a callback. The callback will only be provided if the scan contains a link and the button should be shown.

NativeScript has various ways to show multiple elements on a page. A page can only contain a single component, but NativeScript has components that can contain multiple components. These are called layouts. The simplest one, and probably also the most used, is the `StackLayout`. Elements will be placed below or beside each other. The `StackLayout` has a property `orientation` that indicates whether the elements should be placed below (vertical, default) or beside (horizontal) each other.

Other layouts include the following:

- **DockLayout:** Elements can be placed on the left, right, top, bottom, or center of the component.
- **GridLayout:** Elements are placed in one or multiple rows and columns in a grid. This is equal to a `<table>` tag in HTML.
- **WrapLayout:** A row is filled with elements. When it is full, the next elements are added to a new row.



You can find all components at <http://docs.nativescript.org/ui/ui-views-and-all-layout-containers> and all layout containers at <http://docs.nativescript.org/ui/layout-containers>.

In `src/view/details.ts`, we will implement this page:

```
import { EventData } from "data/observable";
import { topmost } from "ui/frame";
import { Page } from "ui/page";
import { ActionBar, ActionItem } from "ui/action-bar";
import { Button } from "ui/button";
import { Label } from "ui/label";
import { StackLayout } from "ui/layouts/stack-layout";
import * as model from "../model";

export function createDetailsPage(scan: model.Scan, callback?: () => void) {
    return { createView };
    function createView() {
        const page = new Page();
        const layout = new StackLayout();
        page.content = layout;
        layout.orientation = "vertical";
        layout.items.push(scan);
        if (callback) {
            layout.on("tap", (args) => {
                const item = args.object;
                if (item instanceof Scan) {
                    callback();
                }
            });
        }
    }
}
```

In a label, we will show the content of the scan. We can add a class name to it, just like you would do on an HTML webpage. Later on, we can style this page using CSS:

```
const label = new Label();
label.text = scan.content;
label.className = "details-content";
layout.addChild(label);
```

The date of the scan will be shown in a second label:

```
const date = new Label();
date.text = scan.date.toLocaleString("en");
layout.addChild(date);
```

If a callback is provided, we show a button that will open the link of the scan:

```
if (callback) {
    const button = new Button();
    button.text = "Open";
    button.on("tap", callback);
    layout.addChild(button);
}

return page;
}
```

In `src/app.ts`, we can now implement the `showDetailsPage` function. Using `topmost().navigate`, we can navigate to the page. Users can go back to the main page with the standard back button of Android or iOS, which is automatically shown:

```
import { topmost } from "ui/frame";
import { openUrl } from "utils/utils";
import { createDetailsPage } from "./view/details";
...
function showDetailsPage(scan: model.Scan) {
    let callback: () => void;
    if (model.isUrl(scan)) {
        callback = () => openUrl(scan.content);
    }
    topmost().navigate(createDetailsPage(scan, callback).createView());
}
```

The `openUrl` function opens a web browser with the specified URL.

Scanning QR codes

NativeScript has support for plugins. A plugin can add extra functionality, such as turning on the flash light of a phone, vibrating the phone, logging in with Facebook, or scanning QR codes. These can be installed using the command line interface of NativeScript.

We will use a NativeScript plugin to scan QR codes. The plugin is called NativeScript BarcodeScanner. It can scan QR codes and other barcode formats. The plugin can be installed using the following command:

```
tns plugin add nativescript-barcodescanner
```

Type definitions

We must add a definition file to import the plugin. The plugin does not contain type definitions, and type definitions are not available on DefinitelyTyped and TSD. It is not necessary to write definitions that are fully correct. We only have to type the parts of the library that we are using. We use the `scan` function, which can take an optional settings object and return a `Promise`. In `src/definitions.d.ts`, we write the following definition:

```
declare module "nativescript-barcodescanner" {  
    function scan(options?: any): Promise<any>;  
}
```



You do not need to specify the `export` keyword in definition files. All declarations in a module in a definition file are automatically considered to be exported.

Implementation

The `scan` function can now be implemented. We use the exported `scan` function and listen for `Promise` to resolve or reject. When `Promise` resolves, we add the item to the list and open the details page.

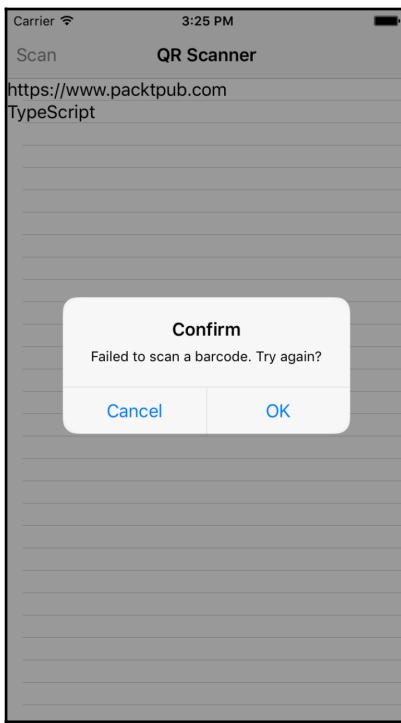
We can import the plugin in `src/app.ts`:

```
import * as barcodescanner from "nativescript-barcodescanner";
```

The `scan` function can now be rewritten as follows:

```
function scan() {  
  barcodescanner.scan().then(result => {  
    addItem(result.text);  
    return false;  
  });  
}
```

We can also show a message when the scan failed. This way, the user gets feedback when the scan failed. We will show a question asking whether the user wants to try again, as shown in the following screenshot:



This can be implemented by replacing the `scan` function with the following code:

```
import * as dialogs from "ui/dialogs";  
...  
function scan() {  
  barcodescanner.scan().then(result => {  
    addItem(result.text);  
    return false;  
  }).catch(() => {  
    dialogs.confirm("Failed to scan a barcode. Try again?",  
      "OK",  
      "Cancel");  
  });  
}
```

```
}, () => {
  return dialogs.confirm("Failed to scan a barcode. Try again?")
}).then(tryAgain => {
  if (tryAgain) {
    scan();
  }
});
```

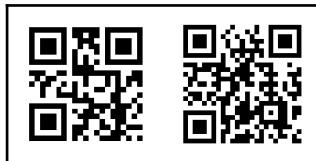
In the first callback, the scan was successful. The scan is added to the recent scan list and the details page shows. In the second callback, we show the dialog. The `dialogs.confirm` function returns a promise, which will resolve to `boolean`. In the last callback, `tryAgain` will be `false` if the scan was successful or if the user clicked on the **No** button. It will be `true` if the user clicked on the **Yes** button. In that case, we will show the barcode scanner again.



When you return a value in the second callback (or `catch` callback), the resulting `Promise` will resolve to that value. When you return `Promise`, the resulting `Promise` will be resolved or rejected with the value or error of that `Promise`. If you want to reject the resulting `Promise`, you must use `throw`.

Testing on a device

In the emulator, we cannot take a picture of a QR code; thus, we have to test the app on a device. We can do that by connecting the device using a USB cable and then running `tns run android` or `tns run ios`. You can test the app using these QR codes, which contain text (left image) and a URL (right image). You can scan the QR codes several times and notice the list build up in the main view. When you restart the app, you will see that the list is cleared. We will fix that in the next section.



Adding persistent storage

When the user closes and reopens the app, the user sees an empty list of scans. We can make the list persistent by saving it after a scan and loading it when the app starts. We can use the `application-settings` module to store the scans. The storage is based on **key-value**: a value is assigned to a specific key.

Only booleans, numbers, and strings can be stored using this module. An array cannot be stored. Instead, one could store the length under one key (for instance, `items-length`) and the items under a set of keys (`items-0, items-1, ...`). An easier approach is to convert the array to a string using **JSON**.

The list can be saved using the following function:

```
function save() {
    applicationSettings.setString("items", JSON.stringify(items));
}
```

The `Date` objects are converted to strings by `JSON.stringify`. Thus, we must convert them back to a `Date` object manually:

```
function load() {
    const data = applicationSettings.getString("items");
    if (data) {
        try {
            items = (<any[]> JSON.parse(data)).map(item => ({
                content: item.content,
                date: new Date(item.date)
            }));
        } catch (e) {}
    }
}
```

Before `application.start()`, we must call the `load` and `update` functions to show the previous scans:

```
const page = createPage(index => showDetailsPage(items[index]), scan);
application.mainEntry = page.createView;
load();
update();
application.start();
```

In `addItem`, we must call `save`:

```
function addItem(content: string) {
  const item: model.Scan = {
    content,
    date: new Date()
  };
  items = [item, ...items].slice(0, 100);
  save();
  update();
  showDetailsPage(item);
}
```

Styling the app

The app can be styled using CSS. Not all CSS properties are supported, but basic settings like fonts, colors, margin, and padding work. We can add a stylesheet in the app adding the following code before `application.start()`:

```
application.cssFile = "style.css";
```

We will change the style of the following parts of the app:

In `app/style.css`, we will first give the `ActionBar` a background color:

```
ActionBar {
  background-color: #237691;
  color: #fefefe;
}
```

 The stylesheet must be added in the `app` folder, instead of `src`. NativeScript will only load files inside `app`. TypeScript files are compiled into that folder, but the stylesheet should already be located there.

We will add some margin to the labels in the list and details page:

```
Label {
  margin: 10px;
}
```

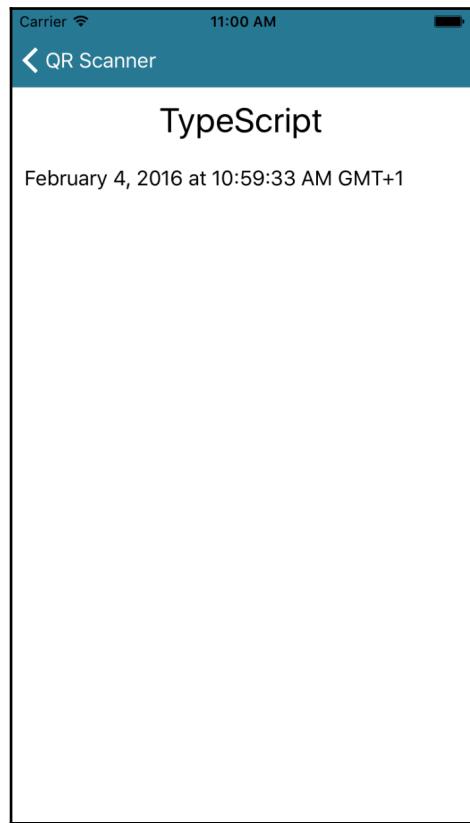
The main page is now properly styled, as shown in the following screenshot:



We can also style the label on the **detail** page, which we gave a class name. We make the text in the label bigger and center the text:

```
.details-content {  
    font-size: 28pt;  
    text-align: center;  
    margin: 10px;  
}
```

This results in the following design:



Comparing NativeScript to alternatives

Various frameworks that can build mobile apps exist. Lots of developers use **Cordova** or **Phonegap**. These tools wrap an HTML page into an app. These apps are called hybrid, as they combine HTML pages with mobile apps. The user interface is not native and can give a bad user experience.

Other tools have a native interface, which gives a good look and feel. **Titanium**, **NativeScript**, and **React Native** do this. With these tools, less code can be shared between a web app and mobile app. With React Native, apps can be written using the React framework.

In NativeScript, programmers have access to all native APIs. The disadvantage of this is that the programmer would write platform-specific code. NativeScript also includes wrappers around these classes, which work on both Android and iOS. For instance, the `Button` class, which we used in this chapter, is a wrapper around `android.widget.Button` on Android and `UIButton` on iOS.

Summary

In this chapter, we created a mobile app using NativeScript. We used a plugin to scan QR codes. The scans are saved, so the list is persisted after a restart of the app. Finally, we added custom styles to our app.

In the next chapter, we will build a spreadsheet web app using React. We will discover some principles of functional programming and learn how we can handle the state of an application. We will also see how we can build a cross-platform application.

6

Advanced Programming in TypeScript

In the previous chapters, we learned the basics of TypeScript and we worked with various frameworks. We will discover more advanced features of TypeScript in this chapter. This chapter covers the following aspects:

- Using type guards
- More accurate type guards
- Checking null and undefined
- Creating tagged union types
- Comparing performance of algorithms

Using type guards

Sometimes, you must check whether a value is of a certain type. For instance, if you have a value of a class `Base`, you might want to check if it is of a certain subclass `Derived`. In JavaScript you would write this with an `instanceof` check. Since TypeScript is an extension of JavaScript, you can also use `instanceof` in TypeScript. In other typed languages, like C#, you must then add a **type cast**, which tells the compiler that a value is of a type, different from what the compiler analyzed. You can also add type casts in two different ways. The old syntax for type casts uses `< and >`, the new syntax uses the `as` keyword. You can see them both in the next example:

```
class Base {  
    a: string;  
}  
class Derived extends Base {
```

```
b: number;  
}  
const foo: Base;  
if (foo instanceof Derived) {  
    (<Derived> foo).b;  
    (foo as Derived).b;  
}
```

When you use a type guard, you say to the compiler: trust me, this value will always be of this type. The compiler cannot check that and will assume that it is true. But, we are using a compiler to get notified about errors so we want to reduce the amount of casts that we need.

Luckily, the compiler can, in most cases, understand the usages of `instanceof`. Thus, in the previous example the compiler knows that the type of `foo` is `Derived` inside the `if`-block. Thus, we do not need type casts there:

```
const foo: Base;  
if (foo instanceof Derived) {  
    foo.b;  
}
```

An expression that checks whether a value is of a certain type is called a **type guard**. TypeScript supports three different kinds of type guards:

- The `typeof` guard checks for primitive types. It starts `typeof x ===` or `typeof x !==`, followed by `string`, `number`, `boolean`, `object`, `function`, or `symbol`.
- The `instanceof` guard checks for class types. Such a type guard starts with the variable name, followed by `instanceof` and the class name.
- **User defined type guards** a custom type guard. You can define a custom type guard as a function with a special return type:

```
function isCat(animal: Animal): animal is Cat {  
    return animal.name === "Kitty";  
}
```

You can then use it as `isCat(x)`.

You can use these type guards in the condition of `if`, `while`, `for`, and `do-while` statements and in the first operand of binary logical operators (`x && y`, `x || y`) and conditional expressions (`x ? y : z`).

Narrowing

The type of a variable will change (locally) after a type guard. This is called **narrowing**. The type will be more specific after narrowing. More specific can mean that a class type is replaced by the type of a subclass, or that a union is replaced by one of its parts. The latter is demonstrated in the following example:

```
let x: string | number;
if (typeof x === "string") {
    // x: string
} else {
    // x: number
}
```

As you can see, a type guard can also narrow a variable in the `else` block.

Narrowing any

Narrowing will give a more specific type. For instance, `string` is more specific than `any`. The following code will narrow `x` from `any` to `string`:

```
let x: any;
if (typeof x === "string") {
    // x: string
}
```

In general, a more specific type can be used on more constructs than the initial type. For instance, you can call `.substring` on a `string`, but not on a `string | number`. When narrowing from `any`, that is not the case. You may write `x.abcd` if `x` has the type `any`, but not when its type is `string`. In this case, a more specific type allows less constructs with that value. To prevent these issues, the compiler will only narrow values of type `any` to primitive types. That means that a value can be narrowed to `string`, but not to a class type, for instance. The next example demonstrates a case where the compiler would give an undesired error, if this was not implemented:

```
let x: any;
if (x instanceof Object) {
    x.abcd();
}
```

In the block after the type guard, `x` should not be narrowed to `Object`.

Combining type guards

Type guards can be combined in two ways. First, you can nest if statements and thus apply multiple type guards to a variable:

```
let x: string | number | boolean;
if (typeof x !== "string") {
    if (typeof x !== "number") {
        // x: boolean
    }
}
```

Secondly, you can also combine type guards with the logical operators (`&&`, `||`). The previous example can also be written as:

```
let x: string | number | boolean;
if (typeof x !== "string" && typeof x !== "number") {
    // x: boolean
}
```

With `||`, we can check that a value matches one of multiple type guards:

```
let x: string | number | boolean;
if (typeof x === "string" || typeof x === "number") {
    // x: string | number
} else {
    // x: boolean
}
```

More complex type guards can be created with user defined type guards.

More accurate type guards

Before TypeScript 2.0, the compiler did not use the control flow of the program for type guards. The easiest way to see what that means, is by an example:

```
function f(x: string | number) {
    if (typeof x === "string") {
        return;
    }
    x;
}
```

The type guard narrows `x` to `string` in the block after the `if` statement. If the `else` block existed, it would have narrowed `x` to `number` there. Outside of the `if` statement, no narrowing happens, because the compiler only looks at the structure or shape of the program. That means that the type of `x` on the last line would be `string | number`, even though that line can only be executed if the condition of the `if` statement is false and `x` can only be a `number` there. With some terminology, type guards were only syntax directed and were only based on the syntax, not on the control flow of the program.

As of TypeScript 2.0, the compiler can follow the control flow of the program. This gives more accurate types after type guards. The compiler understands that the last line of the function can only be reached if `x` is not a `string`. The type of `x` on the last line will now be `number`. This analysis is called **control flow based type analysis**.

Assignments

Previously, the compiler did not follow assignments of a variable. If a variable was reassigned in the block after an `if` statement, the narrowing would not be applied. Thus, in the next example, the type of `x` is `string | number`, both before and after the assignment:

```
let x: string | number = ...;
if (typeof x === "string") {
    x = 4;
}
```

With control flow based type analysis, these assignments can be checked. The type of `x` will be `string` before the assignment and `number` after it. Narrowing after an assignment works only for union types. The parts of the union type are filtered based on the assigned value. For types other than union types, the type of the variable will be reset to the initial type after an assignment.

This can be used to write a function that either accepts one value or a list of values, in one of the following ways:

```
function f(x: string | string[]) {
    if (typeof x === "string") x = [x];
    // x: string[]
}

function g(x: string | string[]) {
    if (x instanceof Array) {
        for (const item of x) g(item);
        return;
    }
    // x: string
```

}

With the same analysis, the compiler can also check for values that are possibly null or undefined. Instead of getting runtime errors saying **undefined is not an object**, you will get a compile time warning that a variable might be undefined or null.

Checking null and undefined

TypeScript 2.0 introduces two new types: `null` and `undefined`. You have to set the compiler option `strictNullChecks` to `true` to use these types. In this mode, all other types cannot contain `undefined` or `null` anymore. If you want to declare a variable that can be `undefined` or `null`, you have to annotate it with a union type. For instance, if you want a variable that should contain a `string` or `undefined`, you can declare it as `let x: string | undefined;`.

Before assignments, the type of the variable will be `undefined`. Assignments and type guards will modify the type locally.

Guard against null and undefined

TypeScript has various ways to check whether a variable could be `undefined` or `null`. The next code block demonstrates them:

```
let x: string | null | undefined = ...;
if (x !== null) {
    // x: string | undefined
}
if (x !== undefined) {
    // x: string | null
}
if (x != null) {
    // x: string
}
if (x) {
    // x: string
}
```

The last type guard can have unexpected behavior, so it is advised to use the others instead. At runtime, `x` is converted to a Boolean. `null` and `undefined` are both converted to `false`, non-empty strings to `true`, but an empty string is converted to `false`. The latter is not always desired.

To check for a string, you can also use `typeof x === "string"` as a type guard. It is not always possible to write a type guard for some types, but you can always use the type guards in the code block.

The never type

TypeScript 2.0 also introduced the `never` type, which represents an unreachable value. For instance, if you write a function that always throws an error, its return type will be `never`.

```
function alwaysThrows() {
    throw new Error();
}
```

In a union type, `never` will disappear. Formally, `T | never` and `never | T` are equal to `T`. You can use this to create an assertion that a certain position in your code is unreachable:

```
function unreachable() {
    throw new Error("Should be unreachable");
}

function f() {
    switch (...) {
        case ...:
            return true;
        case ...:
            return false;
        default:
            return unreachable();
    }
}
```

The compiler takes the union of the types of all expressions in return statements. That gives `boolean | never` in this example, which is reduced to `boolean`.

We will use `strictNullChecks` in the next chapters.

Creating tagged union types

With TypeScript 2.0, you can add a tag to union types and use these as type guards. That feature is called: **discriminated union types**. This sounds very difficult, but in practice it is very easy. The following example demonstrates it:

```
interface Circle {
    type: "circle";
    radius: number;
```

```
}

interface Square {
    type: "square";
    size: number;
}

type Shape = Circle | Square;
function area(shape: Shape) {
    if (shape.type === "circle") {
        return shape.radius * shape.radius * Math.PI;
    } else {
        return shape.size * shape.size;
    }
}
```

The condition in the `if` statements works as a type guard. It narrows the type of `shape` to `circle` in the `true` branch and `square` in the `false` branch.

To use this feature, you must create a union type of which all elements have a property with a string value. You can then compare that property with a string literal and use that as a type guard. You can also do that check in a `switch` statement, like the next example.

```
function area(shape: Shape) {
    switch (shape.type) {
        case "circle":
            return shape.radius * shape.radius * Math.PI;
        case "square":
            return shape.size * shape.size;
    }
}
```

We have now seen the new major features of the type system of TypeScript 2.0. We will see most of them in action in the next chapters. We will also write some simple algorithms in these chapters. We will learn some background information on writing and analyzing algorithms in the next section.

Comparing performance of algorithms

We will also write some small algorithms in the next chapters. This section shows how the performance of an algorithm can be estimated. During such analysis, it is often assumed that only a large input gives performance problems. The analysis will show how the running time scales when the input scales.

The next section requires some knowledge of basic mathematics. However, this section is not foreknowledge for the next chapters. If you do not understand a piece of this section, you can still follow the rest of the book.

For instance, if you want to find the index of an element in a list, you can use a for loop:

```
function indexOf(list: number[], item: number) {  
    for (let i = 0; i < list.length; i++) {  
        if (list[i] === item) return i;  
    }  
    return -1;  
}
```

This function loops over all elements of the array. If the array has size n , then the body of the loop will be evaluated n times. We do not know how long the body of the loop runs. It could be hundreds or tens of a second, but that depends on the computer. When you run the program twice, the time will probably not be exactly the same.

Luckily, we do not need these numbers for the analysis. It is important to see that the running time of the body does not depend on the size of the array.

The function first sets i to 0, and then executes the code in the loop at most n times. In the worst case, the body is executed n times and the final `return -1` runs. The running time will then be something + n * something + something, where all instances of something do not depend on n . When the input is big enough, we can neglect the time of the initialization of i and the final `return -1`. So, for a large n , the running time is approximately n * something.

Big-Oh notation

Mathematicians created a notation to write this more simply, called Big-Oh notation. When you say that the running time is $\mathcal{O}(n)$, you mean that the running time is at most n * something, for a big enough n . In general, $\mathcal{O}(f(n))$, where $f(n)$ is a formula, means that the running time is at most a multiple of $f(n)$. More formally, if you say that the running time is $\mathcal{O}(f(n))$, you mean that for some numbers N and c the following holds: if $n > N$ then the running time is at most $c * f(n)$. The condition $n > N$ is a formal way of saying: if n is big enough, the value of c is the replacement of something.

For the original problem, this would result in $O(n)$. When you analyze some other algorithms, you can count how often it can be executed for each piece of code. From these terms, you must choose the highest one. We will analyze the next example:

```
function hasDuplicate(items: number[]) {
    for (let i = 0; i < items.length; i++) {
        for (let j = 0; j < items.length; j++) {
            if (items[i] === items[j] && i !== j) return true;
        }
    }
    return false;
}
```

The first line, where `i` is declared, is only evaluated once. The line where `j` is declared is executed at most n times, because it is in the first `for` loop. The `if` statement runs at most $n * n$, or n^2 times. The last line is evaluated at most once. The highest term of these is n^2 . Thus, this algorithm runs in $O(n^2)$.

Optimizing algorithms

For a large array, this function might be too slow. If we would want to optimize this algorithm, we can make the second `for` loop shorter.

```
function hasDuplicate(items: number[]) {
    for (let i = 0; i < items.length; i++) {
        for (let j = 0; j < i; j++) {
            if (items[i] === items[j]) return true;
        }
    }
    return false;
}
```

With the old version, we would compare every two items twice, but now we compare them only once. We also can remove the check `i !== j`. It requires some more work to analyze this algorithm. The body of the second `for` loop is now evaluated $0 + 1 + 2 + \dots + (n - 1)$ times. This is a sum of n terms and the average of the terms is $(n - 1) / 2$. This results in $n * (n-1) / 2$, or $n^2 / 2 - n / 2$. With the Big-Oh notation, you can write this as $O(n^2)$. This is the highest term, so the whole algorithm still runs in $O(n^2)$. As you can see, there is no difference in the Big-Oh between the original and optimized versions. The algorithm will be about twice as fast, but if the original algorithm was way too slow, this one is probably too slow. Real optimization is a bit harder to find.

Binary search

We will first take a look at the `indexOf` example, which runs in $O(n)$. What if we knew that the input is always a sorted list? In such a case, we can find the index much faster. If the element at the center of the array is higher than the value that we search, we do not have to take a look at all elements on the right side of the array. If the value at the center is lower, then we can forget all elements on the left side. This is called **binary search**. We can implement this with two variables, `left` and `right`, which represent the section of the array in which we are searching: `left` is the first element of that section, and `right` is the first element after the section. So `right - 1` is the last element of the section. The code works as follows: it chooses the center of the section. If that element is the element that we search, we can stop. Otherwise, we check whether we should search on the left or right side. When `left` equals `right`, the section is empty. We will then return `-1`, since we did not find the element.

```
function binarySearch(items: number[], item: number) {  
    let left = 0;  
    let right = items.length;  
    while (left < right) {  
        const mid = Math.floor((left + right) / 2);  
        if (item === items[mid]) {  
            return mid;  
        } else if (item < items[mid]) {  
            right = mid;  
        } else {  
            left = mid + 1;  
        }  
    }  
    return -1;  
}
```

What is the running time of this algorithm? To find that, we must know how often the body of the loop is evaluated. Every time that the body is executed, the function returns, or the length of the section is approximately divided by two. In the worst case, the length of the section is constantly divided by two, until the section contains one element. That element is the searched element and the function returns or the length becomes zero and the function while loop stops. So, we can ask the question: how often can you divide n by two, until it becomes less than one? $\log n$ gives us that number. This algorithm runs in $O(\log(n))$, which is a lot faster than $O(n)$. However, it only works if the array is sorted.



In Big-Oh notation, $O(\log(n))$ and $O(^2\log(n))$ are the same. They only differ by some constant number, which disappears in Big-Oh notation.

Built-in functions

When you use other functions in your algorithm, you must be aware of their running time. We could for instance implement `indexOf` like this:

```
function fastIndexOf(items: number[], item: number) {  
    items.sort();  
    return binarySearch(items, item);  
}
```

Both lines of the function are only executed once, but $O(1)$ is not the running time of this algorithm! This function calls `binarySearch`, and we know that the body of the while loop in that function runs, at most, approximately $^2\log n$ times. We do not need to know how the function is implemented, we only need to know that it takes $O(^2\log(n))$. We also call `.sort()` on the array. We have not written that function ourselves and we cannot analyze the code for it. For these functions, you must know (or look up) the running time. For sorting, that is $O(n ^ 2\log(n))$. So our `fastIndexOf` is not faster than the original version, as it runs in $O(n ^ 2\log(n))$.

We can however use sorting to improve the `hasDuplicate` function.

```
function hasDuplicate(items: number[]) {  
    items.sort();  
    for (let i = 1; i < items.length; i++) {  
        if (items[i] === items[i - 1]) return true;  
    }  
    return false;  
}
```

The loop costs $O(n)$ and the sorting costs $O(n ^ 2\log(n))$, so this algorithm runs in $O(n ^ 2\log(n))$. This is faster than our initial implementation, that took $O(n^2)$.

With this basic knowledge, you can analyze simple algorithms and compare their speeds for large inputs. In the next chapters, we will analyze some of the algorithms that we will write.

Summary

In this chapter, we have seen various new features of TypeScript 2.0. In this release, lots of new features for more accurate type analysis were added. We have seen control flow based type analysis, null and undefined checking, and tagged union types. Finally, we have also taken a look at analyzing algorithms. We will use most of these topics in the next three chapters. In [Chapter 7, *Spreadsheet Application with Functional Programming*](#), we will build a spreadsheet application. We will also discover functional programming there.

7

Spreadsheet Applications with Functional Programming

In this chapter, we will explore a different style of programming: **functional programming**. With this style, functions should only return something and not have other side effects, such as assigning a global variable. We will explore this by building a spreadsheet application.

Users can write calculations in this application. The spreadsheet contains a grid and every field of the grid can contain an expression that will be calculated. Such expressions can contain constants (numbers), operations (such as addition, multiplying), and they can reference other fields of the spreadsheet. We will write a parser, that can convert the string representation of such expressions into a data structure. Afterwards, we can calculate the results of the expressions with that data structure. If necessary, we will show errors such as division by zero to the user.

Tax calculator		
0	1	2
0 Price without VAT	42	€
1 VAT percentage	21	%
2 VAT	8.82	€
3 Price with VAT	50.82	€

Add column

Add row

We will build this application using the following steps:

- Setting up the project
- Functional programming
- Using data types for expressions
- Writing unit tests
- Parsing an expression
- Defining the sheet
- Using the Flux architecture
- Creating actions
- Writing the view
- Advantages of Flux

Setting up the project

We start by installing the dependencies that we need in this chapter using NPM:

```
npm init -y
npm install react react-dom -save
npm install gulp gulp-typescript small --save-dev
```

We set up TypeScript with lib/tsconfig.json:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "noImplicitAny": true,
    "jsx": "react"
  }
}
```

We configure gulp in gulpfile.js:

```
var gulp = require("gulp");
var ts = require("gulp-typescript");
var small = require("small").gulp;

var tsProject = ts.createProject("lib/tsconfig.json");

gulp.task("compile", function() {
  return gulp.src(["lib/**/*.ts", "lib/**/*.tsx"])
    .pipe(ts(tsProject))
```

```
.pipe(gulp.dest("dist"))
.pipe(small("client/index.js", { externalResolve:
["node_modules"], outputFileName: { standalone: "client.js" } }))})
.pipe(gulp.dest("static/scripts/"));
});
```

We install type definitions for React:

```
npm install @types/react @types/react-dom --save
```

In static/index.html, we create the HTML structure of our application:

```
<!DOCTYPE HTML>

<html>
  <head>
    <title>Chapter 5</title>
    <link href="style.css" rel="stylesheet" />
  </head>
  <body>
    <div id="wrapper"></div>
    <script type="text/javascript">
      var process = {
        env: {
          NODE_ENV: "DEBUG" // or "PRODUCTION"
        }
      };
    </script>
    <script type="text/javascript" src="scripts/client.js"></script>
  </body>
</html>
```

We add some basic styles in static/style.css. We will add more styles later on:

```
body {
  font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande',
'Lucida Sans', Arial, sans-serif;
}

a:link, a:visited {
  color: #5a8bb8;
  text-decoration: none;
}
a:hover, a:active {
  color: #406486;
}
```

Functional programming

When you ask a developer what the definition of a function is, he would probably answer something like “something that does something with some arguments”. Mathematicians have a formal definition for a function:

A function is a relation where an input has exactly one output.

This means that a function should always return the same output for the same input.

Functional programming (FP) uses this mathematical definition. The following code would violate this definition:

```
let x = 1;
function f(y: number) {
    return x + y;
}

f(1);
x = 2;
f(1);
```

The first call to `f` would return `2`, but the second would return `3`. This is caused by the assignment to `x`, which is called a **side effect**. A reassignment to a variable or a property is called a side effect, since function calls can give different results after it.

It would be even worse if a function modified a variable that was defined outside of the function:

```
let x = 1;
function g(y: number) {
    x = y;
}
```

Code like this is hard to read or test. These mutations are called side effects. When a piece of code does not have side effects, it is called **pure**. With functional programming, all or most functions should be pure.

Calculating a factorial

We will take a look at the factorial function to see how we can surpass the limitations of functional programming. The factorial function, written as $n!$ is defined as $1 * 2 * 3 * \dots * n$. This can be programmed with a simple `for` loop:

```
export function factorial(x: number) {
    let result = 1;
    for (let i = 1; i <= x; i++) {
        result *= i;
    }
    return result;
}
```

However, the value of `i` is increased in the loop, which is a reassignment and thus a side effect. With functional programming, recursion should be used instead of a loop. The factorial of `x` can be calculated using the factorial of `x - 1` and multiplying it with `x`, since $x! = x * (x-1)!$ for $x > 1$. The following function is pure and smaller than the iterative function. Calling a function from the same function is called recursion.

```
export function factorial(x: number): number {
    if (x <= 1) return 1;
    return x * factorial(x - 1);
}
```



When you define a function with recursion, TypeScript cannot infer the return type. You have to specify the return type yourself in the function header.

We will use this function later on, so save this as `lib/model/utils.ts`.

Using data types for expressions

Fields of the spreadsheet can contain expressions, that can be calculated. To calculate these values, the input of the user must be converted to a data structure, which can then be used to calculate the result of that field.

Tax calculator			
	0	1	2
0	Price without VAT	42	€
1	VAT percentage	21	%
2	VAT	8.82	€
3	Price with VAT	50.82	€

Add column

A =1:0+1:2

Save Cancel

These expressions can contain constants, operations, references to other fields or a parenthesized expression:

- Constants: 0, 42, 10.2, 4e6, 7.5e8
- Unary expression: -expression, expression!
- Binary expression: expression + expression, expression / expression
- References: 3:1 (third column, first row)
- Parenthesized expression: (expression)

We will create these types in `lib/model/expression.ts`. First we import `factorial`, since we will need it later on.

```
import { factorial } from "./utils";
```

Creating data types

We can declare data types for these expression kinds. We define them using a class. We can distinguish these kinds easily using `instanceof`. We can declare `Constant` as follows:

```
export class Constant {  
    constructor(  
        public value: number  
    ) {}
```

}



Adding public or private before a constructor argument is syntactic sugar for declaring the property and assigning to it in the constructor:

```
export class Constant {
    value: number;
    constructor(value: number) {
        this.value = value;
    }
}
```

A `UnaryExpression` has a kind (minus or factorial) and the operand on which it is working. We define the kind using an `enum`. For the expression, we reference the `Expression` type that we will define later on:

```
export class UnaryOperation {
    constructor(
        public expression: Expression,
        public kind: UnaryOperationKind
    ) {}
}
export enum UnaryOperationKind {
    Minus,
    Factorial
}
```

A `binary expression` also has a kind (`Add`, `Subtract`, `Multiply`, or `Divide`) and two operands.

```
export class BinaryOperation {
    constructor(
        public left: Expression,
        public right: Expression,
        public kind: BinaryOperationKind
    ) {}
}
export enum BinaryOperationKind {
    Add,
    Subtract,
    Multiply,
    Divide
}
```

We will call the reference to another field, a **Variable**. It contains the column and the row of the referenced field:

```
export class Variable {  
    constructor(  
        public column: number,  
        public row: number  
    ) {}  
}
```

A parenthesized expression simply contains an expression:

```
export class Parenthesis {  
    constructor(  
        public expression: Expression  
    ) {}  
}
```

We can now define **Expression** as the union type of these classes:

```
export type Expression = Constant | UnaryOperation | BinaryOperation |  
Variable | Parenthesis;
```

The preceding definition means that an **Expression** is a **Constant**, **UnaryExpression**, **BinaryExpression**, **Variable** or **Parenthesis**.

Traversing data types

We can distinguish these classes using `instanceof`. We will demonstrate that by writing a function that converts an expression to a string. TypeScript will change the type of a variable after an `instanceof` check. These checks are called type guards. In the code below, `formula instanceof Constant` narrows the type of `formula` to `Constant` in the block after the `if`. In the `else` block, `Constant` is removed from the type of `formula`, resulting in `UnaryOperation | BinaryOperation | Variable | Parenthesis`.

Using a sequence of `if` statements, we can distinguish all cases. For a constant, we can simply convert the value to a string:

```
export function expressionToString(formula: Expression): string {  
    if (formula instanceof Constant) {  
        return formula.value.toFixed();  
    }
```

For a UnaryOperation, we show the operator before or after the rest of the expression. We convert the rest to a string using recursion and we call `expressionToString` on the expression. Because of that, we had to specify the return type manually:

```
    } else if (formula instanceof UnaryOperation) {
        const { expression, kind } = formula;
        switch (kind) {
            case UnaryOperationKind.Factorial:
                return expressionToString(expression) + "!";
            case UnaryOperationKind.Minus:
                return "-" + expressionToString(expression);
        }
    }
```

We convert a BinaryOperation to a string by inserting the operator between the converted operands:

```
    } else if (formula instanceof BinaryOperation) {
        const { left, right, kind } = formula;
        const leftString = expressionToString(left);
        const rightString = expressionToString(right);
        switch (kind) {
            case BinaryOperationKind.Add:
                return leftString + "+" + rightString;
            case BinaryOperationKind.Subtract:
                return leftString + "-" + rightString;
            case BinaryOperationKind.Multiply:
                return leftString + "*" + rightString;
            case BinaryOperationKind.Divide:
                return leftString + "/" + rightString;
        }
    }
```

A variable is shown as the column, a colon and the row:

```
    } else if (formula instanceof Variable) {
        const { column, row } = formula;
        return column + ":" + row;
```

A parenthesized expression is shown as the containing expression wrapped in parentheses:

```
    } else if (formula instanceof Parenthesis) {
        const { expression } = formula;
        return "(" + expressionToString(expression) + ")";
    }
}
```

This function is a good example of walking through (traversing) a data structure with recursion. Such a function can be written in the following steps:

- Distinguish different cases (for instance using `instanceof` or `typeof`)
- Handle the containing nodes recursively (for instance, `left` and `right` of a `BinaryOperation`)
- Combine the results

In the next session, we will write another function that traverses an expression to validate it.

Validating an expression

When you are writing a function with recursion, you should always be sure that you are not creating infinite recursion, similar to an infinite loop. For instance, when you forget the base cases of the factorial function (`x <= 1`), you would get infinite recursion.

We would also get recursion when a field of the spreadsheet references itself (directly or indirectly). To prevent these issues, we will validate an expression before calculating it. We create the restriction that a reference should not point to itself and it may not reference a higher column or row index.

Later on, we will also show errors when a number is divided by zero, when the factorial of a negative or non-integer is calculated, when a referenced field contains an error, and when a referenced field contains text instead of a number. We define a class `Failure` to represent such an error:

```
export class Failure {
    constructor(
        public kind: FailureKind,
        public location: Expression
    ) {}
}

export enum FailureKind {
    ForwardReference,
    SelfReference,
    TextNotANumber,
    DivideByZero,
    FactorialNegative,
    FactorialNonInteger,
    FailedDependentRow
}
```

Next, we define a function which gives a string description of the error:

```
export function failureText({ kind }: Failure) {
  switch (kind) {
    case FailureKind.ForwardReference:
      return "This expression contains a forward reference to
another variable";
    case FailureKind.SelfReference:
      return "This expression references itself";
    case FailureKind.TextNotANumber:
      return "This expression references a field that does
contain a number";
    case FailureKind.DivideByZero:
      return "Cannot divide by zero";
    case FailureKind.FactorialNegative:
      return "Cannot compute the factorial of a negative number";
    case FailureKind.FactorialNonInteger:
      return "The factorial can only be computed of an integer";
    case FailureKind.FailedDependentRow:
      return "This expression references a field that has
more errors";
  }
}
```

Now we can define a validate function, which will generate an array of errors. The function has two base cases: **constants** and **variables**.

A **constant** can never have errors. A **variable** is an error if it is a self or forward reference. For a unary, binary, or parenthesized expression we must validate the children recursively:

```
export function validate(column: number, row: number, formula: Expression): Failure[] {
  if (formula instanceof UnaryOperation || formula instanceof Parenthesis)
  {
    return validate(column, row, formula.expression);
  } else if (formula instanceof BinaryOperation) {
    return [
      ...validate(column, row, formula.left),
      ...validate(column, row, formula.right)
    ];
  } else if (formula instanceof Variable) {
    if (formula.column === column && formula.row === row) {
      return [new Failure(FailureKind.SelfReference, formula)];
    }
    if (formula.column > column || formula.row > row) {
      return [new Failure(FailureKind.ForwardReference, formula)];
    }
  }
  return [];
}
```

```
    } else {
        return [];
    }
}
```

In the first `if` statement, the type of `formula` is `UnaryOperation | Parenthesis`. Since both types have the property `expression`, we can access it.

Calculating expressions

The last traversal is calculating the expression. This function will return a number if the calculation succeeded. Otherwise, it will return a list of errors. The arguments of the function are the expression and a function that gives the value of a referenced field:

```
export function calculateExpression(formula: Expression, resolve:
(variable: Variable) => number | Failure[]): number | Failure[] {
```

For a constant, we can simply return its value:

```
if (formula instanceof Constant) {
    return formula.value;
```

To calculate the value of a `UnaryOperation`, we first calculate its operand. If that contains an error, we propagate it. Otherwise, we calculate the factorial or the negative value of it. For a factorial we also show an error if it is not a non-negative integer. Because of the type guard, TypeScript narrows the type of `value` to a number in the `else` block:

```
} else if (formula instanceof UnaryOperation) {
    const { expression, kind } = formula;
    const value = calculateExpression(expression, resolve);
    if (value instanceof Array) {
        return value;
    } else {
        switch (kind) {
            case UnaryOperationKind.Factorial:
                if (value < 0) {
                    return [new Failure(FailureKind.FactorialNegative,
formula)];
                }
                if (Math.round(value) !== value) {
                    return [new Failure(FailureKind.FactorialNonInteger, formula)];
                }
                return factorial(Math.round(value));
            case UnaryOperationKind.Minus:
                return -value;
        }
    }
}
```

```
}
```

For a binary operation, we calculate the left and right side. If one of these contains errors, we return those. Otherwise we apply the operator to both values:

```
} else if (formula instanceof BinaryOperation) {
    const { left, right, kind } = formula;
    const leftValue = calculateExpression(left, resolve);
    const rightValue = calculateExpression(right, resolve);
    if (leftValue instanceof Array) {
        if (rightValue instanceof Array) {
            return [...leftValue, ...rightValue];
        }
        return leftValue;
    } else if (rightValue instanceof Array) {
        return rightValue;
    } else {
        switch (kind) {
            case BinaryOperationKind.Add:
                return leftValue + rightValue;
            case BinaryOperationKind.Subtract:
                return leftValue - rightValue;
            case BinaryOperationKind.Multiply:
                return leftValue * rightValue;
            case BinaryOperationKind.Divide:
                if (rightValue === 0) {
                    return [new Failure(FailureKind.DivideByZero,
                        formula)];
                }
                return leftValue / rightValue;
        }
    }
}
```

For a variable, we delegate the calculation to the `resolve` function:

```
} else if (formula instanceof Variable) {
    return resolve(formula);
} else if (formula instanceof Parenthesis) {
    return calculateExpression(formula.expression, resolve);
}
}
```

Finally, we calculate the value of a parenthesized expression with the expression it contains.

Parsing an expression

A parser can convert a string to some data type. The first guess of the type of a parser would be:

```
type Parser<T> = (source: string) => T;
```

Since we will also use a parser to parse a part of the source. For instance, when parsing a factorial, we first parse the operand (which hopefully has one character remaining, the exclamation mark) and then parse the exclamation mark. Thus, a parser should return the resulting data and the remaining source:

```
type Parser<T> = (source: string) => [T, string];
```

A constant (such as 5.2) and a variable (5:2) both start with a number. Because of that, a parser should return an array with all options:

```
type Parser<T> = (source: string) => [T, string][];
```

To demonstrate how this works, imagine that there are two parsers: one that parses A, one that parses AA and one that parses AB. The string AAA could be parsed with a sequence of these parsers in three different ways: A-A-A, A-AA, and AA-A. Now imagine that the parsers can first parse A or AA, and then only AB. We will parse AAB. The first part would result in the following result:

```
[  
  ["A", "AB"],  
  ["AA", "B"]  
]
```

The remaining string of the first element (AB), can then be parsed by the second parser (AB). This would have an empty string as the remaining part. The remaining string of the second item (B) cannot be parsed. Thus, these parses can parse AAB as A-AB.

Creating core parsers

We will first create two core parsers in `lib/model/parser.ts`. The function `parse` runs a parser and returns the result if successful, `epsilon` will always succeed and `token` will try to parse a specific string. The value can be specified as the last argument for both functions:

```
type ParseResult<T> = [T, string][];  
type Parser<T> = (source: string) => ParseResult<T>;  
  
export function parse<U>(parser: Parser<U>, source: string): U | undefined
```

```

    const result = parser(source)
        .filter(([result, rest]) => rest.length === 0)[0];
    if (!result) return undefined;
    return result[0];
}

const epsilon = <U>(value: U): Parser<U> => source =>
  [[value, source]];

const token = <U>(term: string, value: U): Parser<U> => source => {
  if (source.substring(0, term.length) === term) {
    return [[value, source.substring(term.length)]];
  } else {
    return [];
  }
};

```

We will combine these core parsers into more complex and useful parsers. First, we will create a function that tries different parsers:

```
const or = <U>(...parsers: Parser<U>[]): Parser<U> => source =>
  (<[U, string>[]>[]).concat(...parsers.map(parser => parser(source)));
```

We can use this to parse a digit. We combine the parsers that parse the number 0 to 9:

```

const parseDigit = or(
  token("0", 0), token("1", 1),
  token("2", 2), token("3", 3),
  token("4", 4), token("5", 5),
  token("6", 6), token("7", 7),
  token("8", 8), token("9", 9)
);

```



Functions that have functions as an argument or return type are called **high order functions**. These functions can easily be reused. With functional programming, you often create such functions.

Running parsers in a sequence

Another way to combine parsers is running them in a sequence. Before we can write these functions, we must define two helper functions in `lib/model/utils.ts`. `flatten` will convert an array of arrays into an array. `flatMap` will first call `map` on the array and secondly `flatten`:

```
export function flatten<U>(source: U[][][]) {
  return (<U[]>[]).concat(...source);
}
export function flatMap<U, V>(source: U[], callback: (value: U) => V[]): V[] {
  return flatten(source.map(callback));
}
```

Back in `lib/model/parser.ts`, we define a `map` function, which can convert a `Parser<U>` to a `Parser<V>`:

```
const map = <U, V>(parser: Parser<U>, callback: (value: U) => V): Parser<V>
=> source =>
  parser(source).map<[V, string]>(([item, rest]) => [callback(item), rest]);
```

We also define a `bind` function, which will run a parser after another parser:

```
const bind = <U, V>(parser: Parser<U>, callback: (value: U) => Parser<V>): Parser<V> => source =>
  flatMap(parser(source), ([result, rest]) => callback(result)(rest));
```

With functional programming, the type of a function can sometimes already describe the implementation. When the implementation gives no type errors, the implementation is in most cases correct.

Next up, we create two functions that can run two or three parsers in a sequence and can combine the results of these parsers into a specific type:

```
const sequence2 = <U, V, W>(
  left: Parser<U>,
  right: Parser<V>,
  combine: (x: U, y: V) => W) =>
  bind(left, x => map(right, y => combine(x, y)));

const sequence3 = <U, V, W, T>(
  first: Parser<U>,
  second: Parser<V>,
  third: Parser<W>,
  combine: (x: U, y: V, z: W) => T) =>
  bind(first, x => sequence2(second, third, (y, z) => combine(x, y, z)));
```

With these functions, we can write a function that can match a sequence of any length, or a list. A list is either one element or one element followed by a list. As you can see, this requires recursion. We need the resulting parser inside the definition of the parser, which is not possible. Instead, we can create a function that will evaluate the parser (`source => parser(source)`):

```
function list<U>(parseItem: Parser<U>) {
  const parser: Parser<U[]> = or(
    map(parseItem, item => [item]),
    sequence2(
      parseItem,
      source => parser(source),
      (item, items) => [item, ...items]
    )
  );
  return parser;
}
```

We can also create a separated list parser, which will either parse only one element, or parse the first element and a list of separators and items. We create an interface to store the result of the function:

```
interface SeparatedList<U, V> {
  first: U;
  items: [V, U][];
}
const separatedList = <U, V>(parseItem: Parser<U>, parseSeparator: Parser<V>) =>
  or(
    map(parseItem, first => ({ first, items: [] })),
    sequence2(
      parseItem,
      list(sequence2(parseSeparator, parseItem, (sep, item) => <[V, U]>[sep, item])),
      (first, items) => ({ first, items })
    )
  );

```

We can now parse a list of digits:

```
const parseDigits = list(parseDigit);
```

This can parse a list of digits. We can convert that to a number with the `map` function that we have defined. Since an integer can be written as $1337 = 1 * 10^3 + 3 * 10^2 + 3 * 10^1 + 7 * 10^0$. We can use the `reduce` function of arrays for this. `reduce` works as follows: `[1, 2, 3, 4].reduce(f, 0) === f(f(f(f(0, 1), 2), 3), 4)`

We can now define the conversion function:

```
const toInteger = (digits: number[]) => digits.reduce(  
  (previous, current, index) =>  
    previous + current * Math.pow(10, digits.length - index - 1),  
    0  
)
```

With map, we can define parseInteger:

```
const parseInteger = map(parseDigits, toInteger);
```

A variable can be parsed as a sequence of an integer (the `column`), a colon, and another integer (the `row`):

```
const parseVariable = sequence3(parseInteger, token(":"), undefined),  
  parseInteger,  
  (column, separator, row) => new Variable(column, row));
```

Parsing a number

A number or constant can be written in the following ways:

- 8 (integer)
- 8.5 (with decimal part)
- 8e4 = 80000 (with exponent)
- 8.5e4 = 85000 (with decimal part and exponent)

We create two parsers, that will parse the decimal part and exponent of a number. They fallback to a default value (0 and 1) in case the number does not have a decimal part or exponent:

```
const parseDecimal = or(  
  epsilon(0),  
  sequence2(  
    token(".", undefined),  
    parseDigits,  
    (dot, digits) => toInteger(digits) / Math.pow(10, digits.length)  
)  
)  
;  
const parseExponent = or(  
  epsilon(1),  
  sequence2(  
    token("e", undefined),  
    parseDigits,
```

```
(e, digits) => Math.pow(10, toInteger(digits))  
)  
);
```

With these functions, we can easily define the `parseConstant` function:

```
const parseConstant = sequence3(  
  parseInteger,  
  parseDecimal,  
  parseExponent,  
  (int, decimal, exp) => new Constant((int + decimal) * exp)  
);
```

We can now define a parser called `parseConstantVariableOrParenthesis`, which will parse a constant, variable, or parenthesized expression (as the name suggests). `parseParenthesis` will be implemented later on:

```
const parseConstantVariableOrParenthesis = or(parseConstant, parseVariable,  
parseParenthesis);
```

Order of operations

When evaluating an expression, the order of execution is important. For instance, $(3 * 4) + 2$ equals 14, while $3 * (4 + 2)$ equals 18. The correct evaluation of $3 * 4 + 2$ is the first one. An expression should be evaluated in this order:

1. Parenthesis
2. Multiplication and division
3. Addition and subtraction
4. Unary expressions

Multiple instances of the same group should be evaluated from left to right, so $10 - 2 + 3 = (10 - 2) + 3$.

Two ways exist to implement this: parsing the source in the right order, or parsing it left to right and correcting it during calculation. Since we already wrote the calculation part, we will parse the source in the right order. That is also the easiest option.

Based on these rules, the left or right side of a multiplication or division can never be an addition or subtraction. The operand of a unary expression can only be a constant, variable, or parenthesized expression. With these restrictions, one can create the following abstract representation:

```
Expression ← Term | Expression ('+' | '-') Term
Term ← Factor | Term ('*' | '/') Factor
Factor ← ConstantVariableOrParenthesis | '-' ConstantVariableOrParenthesis
| ConstantVariableOrParenthesis '!'
Parenthesis ← '(' Expression ')'
ConstantVariableOrParenthesis ← Constant | Variable | Parenthesis
```

This means that an expression is either a single term, or an addition and subtraction of multiple terms. A term is a factor or a multiplication and division of factors. A factor can be a constant, variable or parenthesized expression, optionally with a minus or an exclamation mark. With these rules, an expression will always be parsed in the right order.

We can easily convert this abstract representation to parsers. We start with `parseFactor`, which can be built with `or` and `sequence2`.

```
const parseFactor = or(
    parseConstantVariableOrParenthesis,
    sequence2(
        token("-", undefined),
        parseConstantVariableOrParenthesis,
        (t, value) => new UnaryOperation(value, UnaryOperationKind.Minus)
    ),
    sequence2(
        parseConstantVariableOrParenthesis,
        token("!", undefined),
        (value) => new UnaryOperation(value, UnaryOperationKind.Factorial)
    )
);
```

We can implement `parseTerm` and `parseExpression` using the function `seperatedList`. We will use `reduce` to transform the array into a `BinaryOperation`, just like we used it to convert an array of numbers into a single number in `toInteger`. First, we create the function that transforms the array into a `BinaryOperation`.

```
function foldBinaryOperations({ first, items }: SeparatedList<Expression, BinaryOperationKind>) {
    return items.reduce(fold, first);

    function fold(previous: Expression, [kind, next]: [BinaryOperationKind, Expression]) {
        return new BinaryOperation(previous, next, kind);
    }
}
```

```
    }  
}
```

We use that function in `parseTerm` and `parseExpression`.

```
const parseTerm = map(  
  separatedList(  
    parseFactor,  
    or(  
      token("*", BinaryOperationKind.Multiply),  
      token("/", BinaryOperationKind.Divide)  
    )  
)  
,  
  foldBinaryOperations  
)  
;  
export const parseExpression = map(  
  separatedList(  
    parseTerm,  
    or(  
      token("+", BinaryOperationKind.Add),  
      token("-", BinaryOperationKind.Subtract)  
    )  
)  
,  
  foldBinaryOperations  
)  
;
```

We have not defined `parseParenthesis` yet. Because it depends on `parseExpression`, we must place it below its definition. However, if we would define it here with `const`, it cannot be referenced in `parseConstantVariableOrParenthesis`. Instead we will define it as a function.

```
function parseParenthesis(source: string): ParseResult<Expression> {  
  return sequence3(  
    token("(", undefined),  
    parseExpression,  
    token(")", undefined),  
    (left, expression, right) => new Parenthesis(expression)  
)  
(source);  
}
```

Functions can be used before their definition. We add the source as an argument, as defined in the `Parser` type.

Defining the sheet

A spreadsheet will be a grid of fields. Every field can contain a string or an expression, as demonstrated in the following screenshot:

The screenshot shows a "Tax calculator" application window. At the top is a title bar with the text "Tax calculator". Below it is a table with 4 rows and 3 columns. The columns are labeled 0, 1, and 2 at the top. The rows are numbered 0 through 3 on the left. Row 0 contains "Price without VAT" with value "42 €". Row 1 contains "VAT percentage" with value "21 %". Row 2 contains "VAT" with value "8.82 €". Row 3 contains "Price with VAT" with value "50.82 €". To the right of the table, there are two buttons: "Add column" and "Add row".

	0	1	2	
0	Price without VAT	42	€	
1	VAT percentage	21	%	
2	VAT	8.82	€	
3	Price with VAT	50.82	€	

[Add row](#)

[Add column](#)

In `lib/model/sheet.ts`, we will define the sheet and create functions to parse, show and calculate all expressions in the field.

First, we will import types and functions that we will use in this file.

```
import { Expression, Variable, calculateExpression, Constant, Failure,
FailureKind, validate, expressionToString } from "./expression";
import { parse, parseConstant, parseExpression } from "./parser";
```

We can define a field as an expression or a string, and a sheet as a grid of fields:

```
export type Field = Expression | string;
export class Sheet {
    constructor(
        public title: string,
        public grid: Field[][][]
    ) {}
}
```

Now we will write functions that give the amount of columns and rows of the sheet.

```
export function columns(sheet: Sheet) {
    return sheet.grid.length;
}
export function rows(sheet: Sheet) {
    const firstColumn = sheet.grid[0];
    if (firstColumn) return firstColumn.length;
    return 0;
}
```

The user can write text or an expression in the fields of the spreadsheet. When the content of a field starts with an equals token, it is considered an expression. We write a function `parseField` that parses the content to an expression if it starts with the equals token. Otherwise, it will return the string as-is.

```
export function parseField(content: string): Field {
    if (content.charAt(0) === "=") {
        return parse(parseExpression, content.substring(1));
    } else {
        return content;
    }
}
```

We also create a function that changes a field to a string.

```
export function fieldToString(field: Field) {
    if (typeof field === "string") {
        return field;
    } else {
        return "=" + expressionToString(field);
    }
}
```

In case of an expression, it converts it to a string and adds an equals token before it. Otherwise, it just returns the string.

Calculating all fields

We will write a function that calculates all expressions in the spreadsheet. A field that contains an expression is converted to a number, if the calculation succeeded, or an array of errors otherwise. A field that contains text does not need calculation, so the content is immediately returned.

This yields this type for the result of the calculation:

```
export type Result = ResultField[][];
export type ResultField = string | number | Failure[];
```

We will use two nested loops to loop over each field. This is not pure, but it makes it easier to resolve variables in expressions. When a valid expression is to be calculated, the referenced fields would already be evaluated.

```
export function calculateSheet({ grid }: Sheet) {
    const result: ResultField[][] = [];
```

```

for (let column = 0; column < grid.length; column++) {
    const columnContent = grid[column];
    result[column] = [];
    for (let row = 0; row < columnContent.length; row++) {
        result[column][row] = calculateField(column, row);
    }
}

return result;

```

For each field, we first check whether it is a string. If so, we can immediately return it. Otherwise, we validate the expression. If the expression is invalid, we return the errors and otherwise we run `calculateExpression` on it.

```

function calculateField(column: number, row: number): ResultField {
    const field = grid[column][row];
    if (typeof field === "string") {
        return field;
    } else {
        const errors = validate(column, row, field);
        if (errors.length !== 0) return errors;
        return calculateExpression(field, resolveVariable);
    }
}

```

When a variable reference needs to be resolved, we can access the calculated value from the array `result`. If it contains a string, we try to convert it to a number.

```

function resolveVariable(location: Variable): number | Failure[] {
    const { column, row } = location;
    const value = result[column][row];
    if (typeof value === "string") {
        const num = parse(parseConstant, value);
        if (num === undefined) {
            return [new Failure(FailureKind.TextNotANumber,
location)];
        }
        return num.value;
    } else if (value instanceof eArray) {
        return [new Failure(FailureKind.FailedDependentRow,
location)];
    } else {
        return value;
    }
}

```

We have already written a parser that can parse a constant, so we can reuse it here. If it

contains an array of errors, we return a new error, which says that a referenced field contains an error. Otherwise, the field contains a number and we can simply return that.

Using the Flux architecture

In React, every class component can have a state. Maintaining a state is a side effect and not pure, so we will not use that in this application. Instead, we will use Stateless Functional Components, which are pure. We still need to maintain the state of the application. We will use the Flux architecture to do that. With Flux, you need to write a small piece of non-pure, but the other parts of the application can be written pure. The architecture can be divided into these parts:

- **Store:** Contains the state of the application
- **View:** React components that render the state to HTML
- **Action:** A function that can modify the state (example: rename the spreadsheet)
- **Dispatcher:** A hub modifies the state by executing an action

Several implementations of Flux exist. We will build our own, so that we can understand the ideas better and we can create an implementation that can be properly typed using TypeScript.

We will implement these parts in the following sections.

Defining the state

In `lib/model/state.ts`, we can define an interface that contains the state of the application. The state should contain this information:

- Active spreadsheet
- Calculated results of all expressions
- Selected column and row if a popup is opened
- Content of the textbox of the popup
- Whether or not the textbox of the popup contains a syntax error

This yields the following declaration:

```
import { Sheet, Result } from "./sheet";

export interface State {
    sheet: Sheet;
```

```
    result: Result,  
  
    selectedColumn: number;  
    selectedRow: number;  
    popupInput: string;  
    popupSyntaxError: boolean;  
}
```

If the `popup` is not shown, we will set `selectedColumn` and `selectedRow` to `undefined`. Otherwise, these properties will contain the column and row of the selected field.

```
const emptyRow = ["", ""];  
const emptyGrid = [  
  emptyRow,  
  emptyRow  
]  
export const emptySheet = new Sheet("Untitled", emptyGrid)  
  
export const empty: State = {  
  sheet: emptySheet,  
  result: emptyGrid,  
  
  selectedColumn: undefined,  
  selectedRow: undefined,  
  popupInput: "",  
  popupSyntaxError: false  
}
```

We should also construct the state of the application when it starts. It should contain an empty sheet and the `popup` should not be open.

Creating the store and dispatcher

We will create the store and dispatcher in `lib/model/store.ts`. The dispatcher should take an action and execute it. We first define an action as a function that modifies a state. Since we cannot assign to the state, as that is not pure, an action should not adjust the old state, but create a new state object with a certain modification.

```
export type Action<T> = (state: T) => T;
```

The dispatcher should accept such action. We define the dispatcher as a function with an action as an argument.

```
export type Dispatch<T> = (action: Action<T>) => void;
```

We can now create the store. The store should fire a callback when the state changes.

```
export function createStore<U>(state: U, onChange: (newState: U) => void) {
  const dispatch: Dispatch<U> = action => {
    state = action(state);
    onChange(state);
  }
  return dispatch;
}
```

The store also needs an initial state. We add these two as arguments to the `createStore` function. The function will return the dispatcher.

Creating actions

An action should modify the state. To do that, we will first create three helper functions. One to modify a part of an object, one to modify a part of an array, and one to easily create a new array.

We use the same update function as we did in [Chapter 3, Note-Taking App with a Server](#). We add this function to `lib/model/utils.ts`.

```
export function update<U extends V, V>(old: U, changes: V): U {
  const result = Object.create(Object.getPrototypeOf(old));
  for (const key of Object.keys(old)) {
    result[key] = (<any> old)[key];
  }
  for (const key of Object.keys(changes)) {
    result[key] = (<any> changes)[key];
  }
  return result;
}
```

We also create a function that changes the element at a certain `index` of an array. The other elements will remain at the same location. We will use this function to change the content of a field of the spreadsheet later on.

```
export function updateArray<U>(array: U[], index: number, item: U) {
  return [...array.slice(0, index), item, ...array.slice(index + 1)];
}
```

We define a function `rangeMap`, which creates an array. The callback argument is used to create each element of the array.

```
export function rangeMap<U>(start: number, end: number, callback: (index: number) => U): U[] {
  const result: U[] = [];
  for (let i = start; i < end; i++) {
    result[i] = callback(i);
  }
  return result;
}
```



The functions `update` and `rangeMap` are not pure, since the functions contain several assignments. Sometimes it is not possible or very hard to write a function pure. However, these functions keep the side effects local and other functions will not perceive that the function is pure.

Adding a column or a row

In `lib/model/action.ts`, we will create the actions for our application. First we must import the types and function that we have written before.

```
import { State } from "./state";
import { calculateSheet, Field, rows, fieldToString, parseField } from
"./sheet";
import { update, updateArray, rangeMap } from "./utils";
```

Now we can create an action that calculates all expressions. We will not export this action, but we will use it in other actions.

```
const modifyResult = (state: State) =>
  update(state, {
    result: calculateSheet(state.sheet)
 });
```

With this definition, `modifyResult` is a function that takes a state and returns an updated state with a modified `result` property. This conforms to the Action type that we defined earlier.

We can use this function to create the actions that add a row or column. If a new row needs to be added, every column should get an extra field at the end. This field should be empty; it should contain the empty string. Afterwards, we need to update the `result` property of the state. We will use the `modifyResult` function for that.

```
export const addRow = (state: State) =>
```

```
modifyResult(update(state, {
  sheet: update(state.sheet, {
    grid: state.sheet.grid.map(column => [...column, ""])
  })
}));
```

To add a new column, we must add a new array with empty strings. We will use `rangeMap` to create such an array. We can use `rows` to get the amount of rows, and thus the length of the new array.

```
export const addColumn = (state: State) =>
  modifyResult(update(state, {
    sheet: update(state.sheet, {
      grid: [
        ...state.sheet.grid,
        rangeMap(0, rows(state.sheet), () => "")
      ]
    })
  }));
});
```

Later on, these actions can be triggered by `dispatch(addRow)` or `dispatchaddColumn)`. We will see that in action when we create the view.

Changing the title

Adding a row or a column is an action that does not have arguments. Changing the title does require an argument, namely the new title. Since the definition of an action does not allow extra arguments, we cannot write it as a function that requires the new title and the current state. Instead, we can create a function that takes the new title, and then returns a function that requires the current state. That will give this definition:

```
export const setTitle = (title: string) => (state: State) =>
  update(state, {
    sheet: update(state.sheet, { title })
  });
});
```

This action can be fired by running `dispatch(setTitle("Untitled"))`. If you forget the argument, or specify a wrong argument, TypeScript will give an error. Other implementations of Flux make it hard to type such actions.

Showing the input popup

We need to create several actions for the popup:

- Open the popup
- Close it
- Toggle it (close if it is already open, close it otherwise)
- Change the input of the textbox
- Save the new value

To open the popup, we need the column and the row of the field and save them in the state. We set the input of the popup to the content of that field, either a string or the expression converted to a string. When the popup is opened, it cannot have any syntax errors so we set that property to false.

```
export const popupOpen = (selectedColumn: number, selectedRow: number) =>
(state: State) =>
  update(state, {
    selectedColumn,
    selectedRow,
    popupInput:
      fieldToString(state.sheet.grid[selectedColumn][selectedRow]),
    popupSyntaxError: false
  });

```

The popup can be closed by setting the column and row to undefined.

```
export const popupClose = (state: State) =>
  update(state, {
    selectedColumn: undefined,
    selectedRow: undefined,
    popupInput: ""
  });

```

To toggle the popup, we check whether it opened in the specified location, and close it or open it afterwards.

```
export const popupToggle = (column: number, row: number) => (state: State) =>
  (column === state.selectedColumn && row === state.selectedRow)
    ? popupClose(state) : popupOpen(column, row)(state);

```

We can update the content of the input box:

```
export const popupChangeInput = (popupInput: string) => (state: State) =>
  update(state, {
    popupInput
 });
```

Finally, we can create an action that saves the input in the popup and closes it. However, when the popup contains a syntax error, we will not close the popup, but we will tell the user that the input contains an error. In such a case, `parseField` will return undefined. Otherwise, we change the field that is selected and recalculate the whole spreadsheet.

```
export const popupSave = (state: State) => {
  const input = state.popupInput;
  const value = parseField(input);
  if (value === undefined) {
    return update(state, {
      popupSyntaxError: true
    });
  }
  return modifyResult(update(state, {
    sheet: update(state.sheet, {
      grid: updateArray(state.sheet.grid, state.selectedColumn,
        updateArray(state.sheet.grid[state.selectedColumn],
        state.selectedRow, value)
      )
    }),
    selectedColumn: undefined,
    selectedRow: undefined,
    popupInput: ""
  )));
};
```

These are all actions of our applications.

Testing actions

Since actions are pure functions, we can easily test them. They can be tested without the store, dispatcher, and view. To demonstrate this, we will write tests for `addColumn`, `addRow` and `setTitle`. We start with importing AVA, these functions and some helper functions.

```
import * as test from "ava";
import { empty } from "../model/state";
import { addColumn, addRow, setTitle } from "../model/action";
import { columns, rows } from "../model/sheet";
```

We will write a test for `addColumn`. We validate that the amount of columns is increased by one and that the amount of rows has not been changed.

```
test("addColumn", t => {
  const state = addColumn(empty);
  t.is(columns(state.sheet), columns(empty.sheet) + 1);
  t.is(rows(state.sheet), rows(empty.sheet));
});
```

We write a test for `addRow` too. This time, we validate that the amount of columns stayed the same but the amount of rows increased.

```
test("addRow", t => {
  const state = addRow(empty);
  t.is(columns(state.sheet), columns(empty.sheet));
  t.is(rows(state.sheet), rows(empty.sheet) + 1);
});
```

For `setTitle`, we check that the title has indeed been changed and that the grid has not changed.

```
test("setTitle", t => {
  const state = setTitle("foo")(empty);
  t.is(state.sheet.title, "foo");
  t.is(state.sheet.grid, empty.sheet.grid);
});
```

When you get a bug report, try to create a unit test that demonstrates that error. When you have fixed the bug, you can easily validate it by running the tests and you prevent the bug from returning in the feature.



Writing the view

The application will show an input box at the top of the screen, which is used to type the title of the spreadsheet. Below the title, a table is shown which contains all fields of the spreadsheet. When the user clicks on a field, a popup is created which allows the user to change the content of that field. If the field contains errors, these errors are shown in the popup:

A screenshot of a "Tax calculator" application. At the top, there's a title bar with the text "Tax calculator". Below it is a table with three rows and four columns. Row 0 contains the header "0" and the value "Price without VAT" with the value "42 €". Row 1 contains the header "1" and the value "Price without VAT" with an empty input field. Row 2 contains the header "2" and the buttons "Save" and "Cancel". A modal dialog is open over the table, covering the first two rows. The dialog has a title "1 Price without VAT" and a message "2 Save Cancel". At the bottom of the dialog is a button "Add row".

We will use React to create the view of our application. With Stateless Functional Components, we can write pure functions that render the state.

Rendering the grid

In `lib/client/sheet.tsx`, we will import React and functions and types that we created before:

```
import * as React from "react";
import { Dispatch } from "../model/store";
import { Expression, expressionToString, failureText } from
"../model/expression"
import { State } from "../model/state";
import { Sheet, Field, Result, ResultField, columns, rows, parseField,
fieldToString } from "../model/sheet";
import { update, rangeMap } from "../model/utils";
import * as action from "../model/action";
```

We will render the spreadsheet in `RenderSheet`. That function requires the `state` and the `dispatcher`.

```
export function RenderSheet({ state, dispatch }: { state: State, dispatch:  
Dispatch<State> }) {  
  const { sheet, result } = state;  
  const columnCount = columns(sheet);  
  const rowCount = rows(sheet);
```

At the top of the screen, we show the input box. When the user changes the title, we adjust the state to it with the `setTitle` action.

```
return (  
  <div className="sheet">  
    <input className="sheet-title" value={sheet.title}  
      onChange={e => dispatch(action.setTitle((e.target as  
HTMLInputElement).value))} />
```

We show the table below the title. In this table, we show the calculated values of all fields. We also show two buttons to add a new row or column. These buttons dispatch the actions that we defined earlier.

```
<table>  
  <tbody>  
    <tr>  
      <th></th>  
      { rangeMap(0, columnCount, index => <th key={index}>{  
index }</th>) }  
      <th rowspan={rowCount + 1} className="sheet-add-  
column">  
        <a href="javascript:;"  
          onClick={() => dispatch(action.addColumn)}>Add  
column</a>  
        </th>  
    </tr>  
    { rangeMap(0, rowCount, renderRow) }  
    <tr><th colSpan={columnCount + 2}>  
      <a href="javascript:;"  
        onClick={() => dispatch(action.addRow)}>Add row</a>  
      </th></tr>  
  </tbody>  
</table>  
</div>  
);
```

We render a row in the `renderRow` function. We use `rangeMap` to call this function, and to call `renderColumn`. React requires that we use the `key` property in a loop. We assign the `row` and `column` index to it, since these will be unique.

```
function renderRow(row: number) {
  return (
    <tr key={row}>
      <th>{ row }</th>
      { rangeMap(0, columnCount, renderColumn) }
    </tr>
  );
}

function renderColumn(column: number) {
  return (
    <RenderField key={column} column={column} row={row}
state={state} dispatch={dispatch} />
  );
}

}
```

React components should start with a capital letter. Normal functions should be named with a lower letter as a convention, but for components we have to break that rule.

Rendering a field

To render a field, we will first query the content of the field and check whether the popup is open on this field.

```
function RenderField({ column, row, state, dispatch }: {column: number,
row: number, state: State, dispatch: Dispatch<State> }) {
  const field = state.sheet.grid[column][row];
  const result = state.result[column][row];
  const open = state.selectedColumn === column
    && state.selectedRow === row;
```

Now we check whether the field contains text or an expression. In case of an expression, it can either be a successful calculation or a failed one. If it failed, we will show the amount of errors. In the popup, the user can read all errors. We generate a class name based on this, and on whether the popup is opened in this field.

```
let text: string;
let className: string;

if (typeof result === "string") {
    text = result;
    className = "field-string";
} else if (typeof result === "number") {
    text = result.toString();
    className = "field-value";
} else {
    text = result.length === 1 ? "1 error" : result.length + " errors";
    className = "field-error";
}
className += " field";
if (open) {
    className += " field-open";
}
```

With these variables, we can render the field. If we must show the popup, we will do that with `RenderPopup`.

```
return (
    <td className={className}>
        <span onClick={() => dispatch(action.popupToggle(column, row))}>
            { text }
        </span>
        { open ?
            <RenderPopup
                field={field}
                content={result}
                syntaxError={state.popupSyntaxError}
                input={state.popupInput}
                dispatch={dispatch} />
            : undefined
        }
    </td>
);
}
```

We define that function in the next section. We attach an event listener to the field which will open or close the field when the user clicks on it.

Showing the popup

We will show the popup in `RenderPopup`. The popup contains an input box, a save, and cancel button:

Tax calculator

	0	1	2	
0	Price without VAT	42	€	Add column
1	VAT percentage	21	%	
2	VAT	8.82	€	
3	Price with VAT	<code>=1:0*1:1/100</code>		

If the field contains an error, we show it below the two buttons:

Tax calculator

	0	1	2	
0	Price without VAT	42	€	Add column
1	VAT percentage	21	%	
2	VAT	8.82	€	
3	Price with VAT	<code>=1:0*1:1/100 lorem</code>		

Could not parse this expression.

For errors other than syntax errors, we show the location where it happened:

Tax calculator			
0	1	2	
0 Price without VAT	42	€	
1 VAT percentage		21	%
2 VAT	1 error	€	
3 Price with VAT			

=1:0*1:1/0

[Save](#) [Cancel](#)

Cannot divide by zero 1:0*1:1/0

We will first store all errors in a variable. In case of a syntax error, we cannot give details. For other errors, we show a description and the location of the error.

```
function RenderPopup({ field, content, syntaxError, input, dispatch }: {  
    field: Field, content: ResultField, syntaxError: boolean, input: string,  
    dispatch: Dispatch<State> }) {  
    let errors: JSX.Element | JSX.Element[] = [];  
    if (syntaxError) {  
        errors = <div className="failure">  
            Could not parse this expression.  
        </div>;  
    } else if (content instanceof Array) {  
        errors = content.map((failure, index) => <div className="failure"  
key={index.toString()}>  
            <span className="failure-text">{ failureText(failure) }</span>  
            <span className="failure-source">{  
                expressionToString(failure.location)  
            }</span>  
        </div>);  
    }  
}
```

Now we can build the full view. We attach event listeners to the input box, save, and close button. We also wrap the input box in a form, such that the user can press *Enter* (instead of clicking **Save**) to accept the changes.

```
return (
  <div className="field-popup">
    <form onSubmit={(e) => {e.preventDefault();
dispatch(action.popupSave);}}>
      <input value={input} autoFocus
        onChange={e => dispatch(action.popupChangeInput((e.target as
HTMLInputElement).value))} />
    </form>
    <a href="javascript:;" onClick={() =>
dispatch(action.popupSave)}>Save</a>
    <a href="javascript:;" onClick={() =>
dispatch(action.popupClose)}>Cancel</a>
    <br />
    { errors }
  </div>
);
}
```

Adding styles

In `static/style.css`, we will add some more styles. We will make the text of the input box for the title bigger.

```
.sheet-title {
  font-size: 24pt;
  margin: 0 0 10px;
  border: 1px solid #ccc;
  width: 200px;
}
```

We will add a border to the table and style the button to add a column.

```
.sheet > table, .sheet tr, .sheet th, .sheet td {
  border: 1px solid #ccc;
  border-collapse: collapse;
}
.sheet-add-column > a {
  width: 70px;
  display: block;
}
```

We will style the fields so they show their value properly and can contain a popup.

```
.field {  
    position: relative;  
}  
.field > span {  
    display: block;  
    min-width: 42px;  
    font-size: 10pt;  
    height: 18px;  
    padding: 3px;  
}  
.field-value > span {  
    font-family: Cambria, Cochin, Georgia, Times, Times New Roman, serif;  
    text-align: right;  
}  
.field-error > span {  
    color: #aa2222;  
}  
.field-open {  
    background-color: #eee;  
}
```

We add some styles to the popup:

```
.field-popup {  
    position: absolute;  
    left: 0px;  
    top: 20px;  
    z-index: 10;  
    background-color: #eee;  
    border-bottom: 4px solid #5a8bb8;  
    border-right: 1px solid #ddd;  
    padding: 8px;  
    width: 300px;  
}  
.field-popup > input {  
    margin-right: 10px;  
}  
.field-popup > a {  
    margin-left: 10px;  
}
```

Finally, we change the looks of error messages in the popup.

```
.failure-text {  
    font-style: italic;  
}  
.failure-source {  
    margin-left: 10px;  
    color: #555;  
    font-family: Cambria, Cochin, Georgia, Times, Times New Roman, serif;  
}
```

Gluing everything together

In `lib/client/index.tsx`, we will combine all parts of our application. We will create a component that contains the state and renders the view. When the state is updated in the store, we will propagate that to this component and render the view again.

```
import * as React from "react";  
import { render } from "react-dom";  
import { createStore, Dispatch } from "../model/store";  
import { State, empty } from "../model/state";  
import { RenderSheet } from "./sheet";  
  
class App extends React.Component<{}, State> {  
    dispatch: Dispatch<State>;  
    state = empty;  
  
    constructor(props: {}) {  
        super(props);  
        this.dispatch = createStore(this.state, state =>  
            this.setState(state));  
    }  
    render() {  
        return (  
            <div className="sheet">  
                <RenderSheet  
                    state={this.state}  
                    dispatch={this.dispatch} />  
            </div>  
        );  
    }  
}
```

Finally, we can render this component in the HTML file.

```
render(<App />, document.getElementById("wrapper"));
```

We can view the result by running `gulp compile` and opening `static/index.html` in your browser.

Advantages of Flux

In this section you can find some of the advantages of using Flux, the architecture that we used in this chapter.

Flux is based on the unidirectional flow of data. Angular supports two way bindings, which allow data to flow in two directions. With this data flow, a lot of properties might get changed after a single change is made. This can lead to unpredictable behavior in big applications. Flow and React do not have such bindings, but instead there is a clean flow of data (`store | view | action | dispatch | store`).

The parts of Flux are not strictly bound to each other. This makes it easy to test specific parts of the application with unit tests. We already saw that the actions do not depend on the view.

Going cross-platform

Since the parts of Flux are not bound, we can, relatively, replace the HTML views of the application with views of a different platform. The user interface does not store the state of the application, but it is managed in the store. The other parts need no modification when the HTML views are replaced. This way we can port the application to a different platform and go cross-platform.

Summary

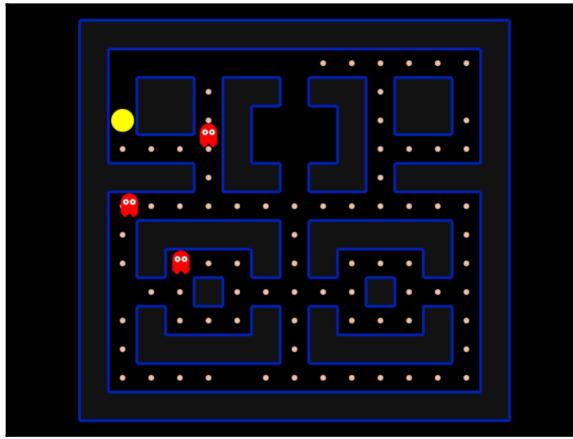
We have built a spreadsheet application with functional programming, React, and Flux in this chapter. We have discovered the limitations of functional programming and learned how we can take advantage of it. We have written automated unit tests for parts of the code that we have written. We also saw how we can traverse data structures and write a parser with functional programming. With the Flux architecture, we learnt how we can write the biggest part of the application with pure functions.

In the next chapter, we will see more of functional programming. We will rebuild Pac-Man with the HTML5 canvas.

8

Pac Man in HTML5

In this chapter, we will recreate Pac Man with the HTML5 canvas. Just like the previous chapter, we will be using functional programming. With the HTML5 canvas and JavaScript, you can play games in the browser.



Pac Man is a classic game where the player (Pac Man, the yellow circle) must eat all of the dots. The ghosts are the enemies of Pac Man: when you get caught by a ghost, you lose. If you eat all of the dots without being caught by a ghost, you win the game.

Drawing on a canvas is, just like modifying the HTML elements of a page, a side effect and thus not pure. Since we will be using functional programming, we will create some abstraction around it, similar to what React does. We will build a small non-pure framework so we can use that to build the rest of the game with pure functions. We will also use `strictNullChecks` in this chapter. The compiler will check which values can be `undefined` or `null`.

We will build the game in these steps:

- Setting up the project
- Using the HTML5 canvas
- Designing the framework
- Drawing on the canvas
- Adding utility functions
- Creating the models
- Drawing the view
- Handling events
- Creating the time handler
- Running the game
- Adding a menu

Setting up the project

The project structure will be similar to the previous projects. In `lib`, we will place our sources. We separate the files for the framework and the game in `lib/framework` and `lib/game`. In `lib/tsconfig.json`, we configure TypeScript:

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "strictNullChecks": true  
  }  
}
```

In the root directory, we set up `gulp` in `gulpfile.js`:

```
var gulp = require("gulp");  
var ts = require("gulp-typescript");  
var small = require("small").gulp;  
  
var tsProject = ts.createProject("lib/tsconfig.json");  
  
gulp.task("compile", function() {  
  return gulp.src("lib/**/*.{ts,html}")  
    .pipe(ts(tsProject))  
    .pipe(small("game/index.js", { outputFileName: { standalone:  
      "scripts.js" }}))  
    .pipe(gulp.dest("static/scripts/"));  
});
```

```
});  
gulp.task("default", ["compile"]);
```

We can install our dependencies with NPM.

```
npm init -y  
npm install gulp gulp-typescript small --save-dev
```

Finally, we create a simple HTML file in static/index.html.

```
<!DOCTYPE HTML>  
<html>  
  <head>  
    <title>Pac Man</title>  
  </head>  
  <body style="background-color: black;">  
    <canvas id="game" width="800" height="600"></canvas>  
    <script src="scripts/scripts.js"></script>  
  </body>  
</html>
```

Before we start writing the framework, we will have a quick look at how the HTML5 canvas works.

Using the HTML5 canvas

The **HTML5 canvas** is an HTML element, just like `<div>`. However, the canvas does not contain other HTML elements, but it can contain a drawing generated by JavaScript code. In `lib/game/index.ts` we will quickly experiment with it.

We can get a reference to the canvas using `document.getElementById` the same way we got a reference to a `<div>` element:

```
const canvas = <HTMLCanvasElement> document.getElementById("game");
```

We cannot directly draw on the canvas; we have to get a rendering context first. Currently, two kinds of rendering contexts exist: a two dimensional context and a `webgl` context, used for 3D rendering. The `webgl` context is a lot harder to use. Luckily, Pac Man is 2D, so we can use the 2D context:

```
const context = canvas.getContext("2d");
```

In an editor with completions, you can check which functions exist on the context. For instance, you can use `context.fillRect(10, 10, 100, 100)` to draw a filled rectangle from 10,10 to 110,110. The x-axis starts at the left side and goes to the right, and the y-axis starts at the top of the canvas and goes down.

Before you can draw anything on the canvas, you must set the drawing color. The canvas distinguishes two different color settings: the fill color and the stroke color. The fill color is used to paint a filled shape. The stroke color is used to draw a shape that only consists of an outline.

We can set these colors using `context.fillStyle` and `context.strokeStyle`:

```
context.fillStyle = "#ff0000";
context.strokeStyle = "#0000ff";
```

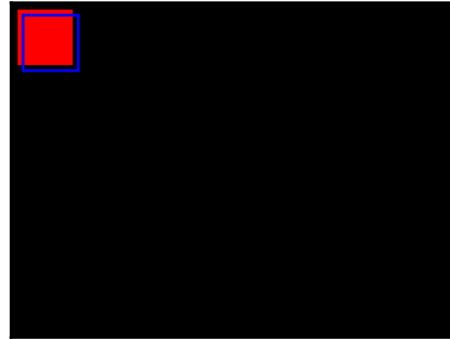
We can also set the weight of a line with a similar property.

```
context.lineWidth = 5;
```

We can draw rectangles with these styles.

```
context.fillRect(10, 10, 100, 100);
context.strokeRect(20, 20, 100, 100);
```

This results in the following image:



Saving and restoring the state

The context also has the functions `save()` and `restore()`. With these functions, you can restore the current draw styles, such as `fillStyle`, and `lineWidth`. `restore()` resets the state to the last time that `save()` was called, based on a LIFO stack (Last In, First Out).

In the following example, the restore on position 3 resets the state to the state saved on position 2, and restore on position 4 resets it to position 1:

```
context.save(); // 1
context.fillStyle = "#ff0000";
context.save(); // 2
context.strokeStyle = "#0000ff";
context.restore(); // 3
context.restore(); // 4
```

We will use these functions in the framework as they can easily be used with recursion.

Designing the framework

We will design the framework based on functional programming. The framework will do all non-pure work, so that the rest of the application can be built with pure functions (except for `Math.random`).

Strictly speaking, `Math.random` is not a pure function. Given that `Math.random()` is not always equal to `Math.random()`, that function will update some internal state.



In pure functional languages, such a function can still exist. That function takes a state and returns a random number and a new state. Since every call to random will get a different state, it can return different random values.

A game consists of an event loop. The amount of iterations that this loop does per second is called FPS or frames per second. Every step of the loop, the game state needs to be updated. For instance, enemies and the player can move, and the player can eat dots in Pac Man. At the end of each step, the game state must be redrawn.

The game must also handle user input. When the user presses the left button, the player should start moving to the left.

We will split the event loop into the following components:

- The view, which will draw the game every step
- A time handler, which will be called once in every step
- An event handler, which will be called for every event that occurs

With functional programming it can often be useful to think about the types of functions before you write them. We will take a quick look at the types of these three components. Imagine the state is stored in some interface `State`. The view will transform this state into a picture. The view might need the width and height of the canvas, so we add these as arguments. We will create the definition of a `Picture` later on:

```
function draw(state: State, width: number, height: number): Picture
```

The time handler should transform the state into a new state. It should not have any other arguments:

```
function timeHandler(state: State): State
```

The event handler also transforms the `state`, but it can use an extra argument, which contains the event that has occurred:

```
function eventHandler(state: State, event: Event): State
```

In the next sections, we will create a framework that manages these three components.

Creating pictures

We will start by creating data types for pictures. Some examples of a picture are a circle, a line, text, or a combination of those. Such pictures can also be scaled, repositioned (translated), or rotated. An empty picture is also a picture.

We define a picture as the union of these different kinds:

```
export type Picture
= Empty
| Rectangle
| RectangleOutline
| Circle
| CircleOutline
| Line
| Text
| Color
| Translate
| Rotate
| Scale
| Pictures;
```

We start by creating some basic types. The `Empty` picture can be defined as follows:

```
export class Empty {  
    __emptyBrand: void;  
}
```

This class does not need any properties. However, if you do not add any properties to a class, values of every type will be assignable to it. This is because TypeScript has a structural type system, and for instance a string has, at the least, all properties of an empty class (that is, no properties). For instance, a string or a number is assignable to that class. To prevent that, we add a **brand** to the class. A brand is a property that does not exist at runtime, but is used to prevent issues with structural typing.

For rectangles and circles, we create different types. One is filled, one has only the outline. For such outlines, we can set the thickness:

```
export class Rectangle {  
    __rectangleBrand: void;  
  
    constructor(  
        public x = 0,  
        public y = 0,  
        public width = 1,  
        public height = width  
    ) {}  
}  
export class RectangleOutline {  
    __rectangleOutlineBrand: void;  
  
    constructor(  
        public x = 0,  
        public y = 0,  
        public width = 1,  
        public height = width,  
        public thickness = 1  
    ) {}  
}
```

If we do not add a brand to these definitions, a Rectangle will be assignable to a RectangleOutline. These brands are also necessary to differentiate a rectangle and a circle:

```
export class Circle {  
    __circleBrand: void;  
  
    constructor(  
        public x = 0,  
        public y = 0,  
        public width = 1,  
        public height = width  
    ) {}  
}  
  
export class CircleOutline {  
    __circleOutlineBrand: void;  
  
    constructor(  
        public x = 0,  
        public y = 0,  
        public width = 1,  
        public height = width,  
        public thickness = 1  
    ) {}  
}
```

We define a line as a list of points and a thickness:

```
export type Point = [number, number];  
export type Path = Point[];  
export class Line {  
    __lineBrand: void;  
  
    constructor(  
        public path: Path,  
        public thickness: number  
    ) {}  
}
```

Next, we define the type for text:

```
export class Text {  
    __textBrand: void;  
  
    constructor(  
        public text: string,  
        public font: string  
    ) {}  
}
```

Wrapping other pictures

We can wrap other pictures and create new ones. For instance, we will change the color of a picture with `Color`. With this definition we can write `new Color("#ff0000", new Circle(0, 0, 2, 2))` to get a red circle:

```
export class Color {  
    __colorBrand: void;  
  
    constructor(  
        public color: string,  
        public picture: Picture  
    ) {}  
}
```

We could also reposition a picture. This is usually called translating. `new Translate(100, 100, new Circle(0, 0, 2, 2))` draws a circle around (100, 100) instead of (0, 0):

```
export class Translate {  
    __translateBrand: void;  
  
    constructor(  
        public x: number,  
        public y: number,  
        public picture: Picture  
    ) {}  
}
```

As the name suggests, Rotate rotates some other picture:

```
export class Rotate {
    __rotateBrand: void;

    constructor(
        public angle: number,
        public picture: Picture
    ) {}  
}
```

We can resize a picture with Scale. new Scale(5, 5, new Circle(0, 0, 2, 2)) would draw a circle of 10×10 instead of 2×2:

```
export class Scale {
    __scaleBrand: void;

    constructor(
        public x: number,
        public y: number,
        public picture: Picture
    ) {}  
}
```

The last class provides a way to show multiple pictures as one picture:

```
export class Pictures {
    __picturesBrand: void;

    constructor(
        public pictures: Picture[]
    ) {}  
}
```

In lib/framework/draw.ts, we will draw these pictures on a canvas. We will implement that function later; we will now only define its header:

```
import { Picture, Rectangle, RectangleOutline, Circle, CircleOutline, Line,
Text, Color, Translate, Rotate, Scale, Pictures, Path } from "./picture";

export function drawPicture(context: CanvasRenderingContext2D, item:
Picture) { }
```

We have now defined all the data types needed to draw a picture. We will create events before we implement the drawPicture function.

Creating events

The application can accept keyboard events. We will distinguish between two kinds of event: a key press and a key release. We will not add mouse events, but you can add these yourself. We define these events in `lib/framework/event.ts`.

Every key has a certain key code, a number that identifies a key. For instance, the left arrow key has code 37. We will add the key code to the `event` class:

```
export const enum KeyEventKind {
    Press,
    Release
}
export class KeyEvent {
    constructor(
        public kind: KeyEventKind,
        public keyCode: number
    ) {}
}
```

We define the event source as a function that will be invoked every step. It will return a list of events that occurred in that step.

```
export function createEventSource(element: HTMLElement) {
    let queue: KeyEvent[] = [];

    const handleKeyEvent = (kind: KeyEventKind) => (e: KeyboardEvent) => {
        e.preventDefault();
        queue.push(new KeyEvent(
            kind,
            e.keyCode
        ));
    };
    const keypress = handleKeyEvent(KeyEventKind.Press);
    const keyup = handleKeyEvent(KeyEventKind.Release);
    element.addEventListener("keydown", keypress);
    element.addEventListener("keyup", keyup);
    function events() {
        const result = queue;
        queue = [];
        return result;
    }
    return events;
}
```

We will call this function in every step to check for new events. In the next section, we will pass these events to the event handler, which will update the game state.

Binding everything together

In `lib/framework/game.ts`, we will bind these components together. We will create a function that starts the event loop and updates the state every step. The function has these arguments:

- The canvas element on which the game will be drawn.
- The event element. Events on this element will be sent to the event handler. This does not have to be the same element as the canvas. An element needs focus to get keyboard events. Since the canvas does not always have focus, it can be better to listen for events on the body element, if there is only one game on the web page.
- The amount of frames per second.
- The initial state of the game.
- A function that draws the state.
- The time handler.
- The event handler.

We register the type of the state as a generic or type argument. Users of this function can provide their own type. TypeScript will automatically infer this type based on the value of the `state` argument:

```
import { Picture } from "./picture";
import { drawPicture } from "./draw";
import { createEventSource, KeyEvent } from "./event";

export function game<UState>(
  canvas: HTMLCanvasElement,
  eventElement: HTMLElement,
  fps: number,
  state: UState,
  drawState: (state: UState, width: number, height: number) => Picture,
  timeHandler: (state: UState) => UState = x => x,
  eventHandler: (state: UState, event: KeyEvent) => UState) {
```

With `createEventSource`, which we have written before, we can get an event source for the specified element:

```
const eventSource = createEventSource(eventElement);
```

To set up drawing, we must acquire the rendering context:

```
const context = canvas.getContext("2d")!;
```

The function `getContext` may return `null` when the context type is not supported. The type `2d` is supported in all browsers that support a canvas, so we can safely cast it with an exclamation mark. This cast will remove the null ability from the type. We create an interval, such that the step function will be called multiple times per second, based on the `fps` parameter:

```
setInterval(step, 1000 / fps);
```

We will use the function `requestAnimationFrame` to render the view. This function takes a callback that will be called when the browser wants to redraw the page. If the browser does not need to redraw, or it has no time for it, it will not try to redraw it. If the draw function is pure, this does not affect the game:

```
let drawAnimationFrame = -1;
draw();

function step() {
    let previous = state;
    for (const event of eventSource()) {
        state = eventHandler(state, event);
    }
    state = timeHandler(state);
    if (previous !== state && drawAnimationFrame === -1) {
        drawAnimationFrame = requestAnimationFrame(draw);
    }
}
```

Finally, we create the draw function. This function renders the picture in the center of the screen. A canvas has an *x*-axis that goes to the left and a *y*-axis that goes down. In mathematics, however, the *y*-axis goes to the top. We will choose the latter and flip the whole picture. `context.restore()` will restore the state to the state at `context.save()`. The transformations do not influence any drawings after the draw function, for instance in the next step:

```
function draw() {
    drawAnimationFrame = -1;
    const { width, height } = canvas;

    context.clearRect(0, 0, width, height);

    context.save();
    context.translate(Math.round(width / 2), Math.round(height / 2));
    context.scale(1, -1);

    drawPicture(context, drawState(state, width, height));
```

```
    context.restore();
}
}
```

We will use the save and restore function in the next section too. We will then draw all kinds of picture on the canvas.

Drawing on the canvas

In `lib/framework/draw.ts`, we will implement the `drawPicture` function that we created before. Using `instanceof` we can check which kind of picture we must draw.

We will interpret the location of an object as the center of it. Thus, `new Rectangle(10, 10, 100, 100)` will draw a rectangle around 10,10. We can draw the outline of a rectangle or the whole rectangle with `strokeRect` and `fillRect`:

```
import { Picture, Rectangle, RectangleOutline, Circle, CircleOutline, Line,
Text, Color, Translate, Rotate, Scale, Pictures, Path } from "./picture";

export function drawPicture(context: CanvasRenderingContext2D, item:
Picture) {
    context.save();
    if (item instanceof RectangleOutline) {
        const { x, y, width, height, thickness } = item;
        context.strokeRect(x - width / 2, y - height / 2, width, height);
    } else if (item instanceof Rectangle) {
        const { x, y, width, height } = item;
        context.fillRect(x - width / 2, y - height / 2, width, height);
    }
}
```

To draw a circle, we use the `arc` function. That function does not draw the circle itself, but only registers its path. We can draw the line or fill it using `stroke` or `fill`. We must wrap `arc` with `beginPath` and `closePath` to do that:

```
} else if (item instanceof CircleOutline || item instanceof Circle) {
    const { x, y, width, height } = item;
    if (width !== height) {
        context.scale(1, height / width);
    }
    context.beginPath();
    context.arc(x, y, width / 2, 0, Math.PI * 2);
    context.closePath();
    if (item instanceof CircleOutline) {
        context.lineWidth = item.thickness;
        context.stroke();
    } else {
```

```
    context.fill();
}
```

For a line, we must do something similar. With `lineTo`, we can draw one section of the line. A line does not have to be closed; it does not have to end at the location it started. Thus, we do not call `closePath`:

```
} else if (item instanceof Line) {
  const { path, thickness } = item;
  context.lineWidth = thickness;
  context.beginPath();
  if (path.length === 0) return;
  const [head, ...tail] = path;
  const [headX, headY] = head;
  context.moveTo(headX, headY);
  for (const [x, y] of tail) {
    context.lineTo(x, y);
  }
  context.stroke();
```

With `fillText`, we can draw text on the canvas. We will center the text. We must also scale the text, since we have flipped the whole canvas in `game.ts`. If you forget this, the text would be upside down:

```
} else if (item instanceof Text) {
  const { text, font } = item;
  context.scale(1, -1);
  context.font = font;
  context.textAlign = "center";
  context.textBaseline = "middle";
  context.fillText(text, 0, 0);
```

We will draw pictures that contain other pictures, such as `Color` or `Pictures`, with recursion. For `Color`, we can simply set the color on the context:

```
} else if (item instanceof Color) {
  const { color, picture } = item;
  context.fillStyle = color;
  context.strokeStyle = color;
  drawPicture(context, picture);
```

For `Translate`, `Rotate`, and `Scale`, we can use the `translate`, `rotate`, and `scale` functions that exist on the rendering context:

```
} else if (item instanceof Translate) {
  const { x, y, picture } = item;
  context.translate(x, y);
```

```

    drawPicture(context, picture);
} else if (item instanceof Rotate) {
    const { angle, picture } = item;
    context.rotate(angle);
    drawPicture(context, picture);
} else if (item instanceof Scale) {
    const { x, y, picture } = item;
    context.scale(x, y);
    drawPicture(context, picture);
}

```

For Pictures, we can use a loop to render all pictures:

```

} else if (item instanceof Pictures) {
    const { pictures } = item;
    for (const picture of pictures) {
        drawPicture(context, picture);
    }
}

```

Finally, we restore the state of the context:

```

    context.restore();
}

```

We have now finished the work on the framework. We will develop the game in the next section.

Adding utility functions

We will write several utility functions in `lib/game/utils.ts`. With `flatten`, we will transform an array of arrays into one array.

```

export function flatten<U>(source: U[][]): U[] {
    return (<U[]>[]).concat(...source);
}

```

With `update`, we can modify some properties of an object. This is the same function as in previous chapters.

```

export function update<U extends V, V>(old: U, changes: V): U {
    const result = Object.create(Object.getPrototypeOf(old));
    for (const key of Object.keys(old)) {
        result[key] = (<any> old)[key];
    }
    for (const key of Object.keys(changes)) {
        result[key] = (<any> changes)[key];
    }
}

```

```
    }
    return result;
}
```

Next, we will create a function for working with `Math.random`. `randomInt` will return a random integer in a certain range and `chance` has a chance to return true:

```
export function randomInt(min: number, max: number) {
    return min + Math.floor(Math.round(
        Math.random() * (max - min + 1)
    ));
}
export function chance(x: number) {
    return Math.random() < x;
}
```

We can calculate the difference between two points with the Pythagorean theorem:

```
export function square(x: number) {
    return x * x;
}
export function distance(x1: number, y1: number, x2: number, y2: number) {
    return Math.sqrt(square(x1 - x2) + square(y1 - y2));
}
```

Finally, we write a function that checks whether a number is an integer:

```
export function isInt(x: number) {
    return Math.abs(Math.round(x) - x) < 0.001;
}
```

Due to rounding errors, we must check that the value is near an integer.

Creating the models

In `lib/game/model.ts`, we will create the models for the game. These models will contain the state of the game, such as the location of the enemies, walls, and dots. The state must also contain the current movement of the player and the difficulty level, as the game will have multiple difficulties.

Using enums

We start with several enums. We can store the difficulty with such an enum:

```
export enum Difficulty {
    Easy,
    Hard,
    Extreme
}
```

The values of an `enum` are converted to numbers during compilation. TypeScript gives the first element zero as the value, the next item one, and so on. In this example, `Easy` is 0, `Hard` is 1, and `Extreme` is 2. However, you can also provide other values. For some applications, this can be useful. We will use custom values to define `Movement`. This enum contains the four directions in which the player can move. In case the user does not move, we use `None`. We give the members a value:

```
export enum Movement {
    None = 0,
    Left = 1,
    Right = -1,
    Top = 2,
    Bottom = -2
}
```

With these values, we can easily create a function that checks whether two movements are in the opposite direction: their sum should equal zero:

```
export function isOppositeMovement(a: Movement, b: Movement) {
    return a + b === 0;
}
```

Other useful patterns that are often used are bitwise values. A number is stored in a computer as multiple bits. For instance, 00000011 (binary) equals 3 (decimal). You can calculate the decimal value of a binary number as follows. The first position from the right has value 1. The next has value 2, then 4, 8, and so on. Summing the values of the positions with a one results in the decimal value.

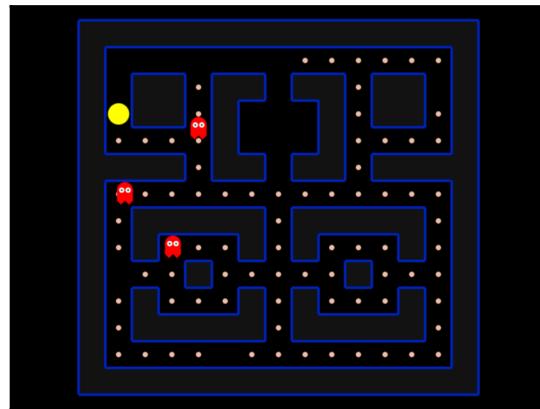
You can use this binary representation to store multiple Booleans in a number. 00000011 would then mean that the first two values are true, and the other values are 0. We use an `enum` to define the names of these properties. `<<` is the bitwise shift operator. `1 << x` means that 00000001 is shifted `x` bits to the left. For instance, `1 << 4` results in 00010000:

```
export enum Side {
    Left = 1 << 0,
    Right = 1 << 1,
```

```
Top = 1 << 2,  
Bottom = 1 << 3,  
LeftTop = 1 << 4,  
RightTop = 1 << 5,  
LeftBottom = 1 << 6,  
RightBottom = 1 << 7  
}
```

We can combine multiple values using the bitwise or operator, `|`. A bit of the output is 1 if at least one of the input bits on that position is 1. Thus, `Side.Left | Side.Right` equals `00000011`. We can check whether some bit is true with the bitwise and operator, `&`. A bit of the output of this operator is 1 if both input bits on that position are 1. For instance, `00000011` and `Side.Right` results in `00000010`. This is not zero, so the Boolean value of `Side.Right` in that number is true.

We will use this later on to draw the edges of the walls. As you can see in the following screenshot, the edges of all walls are drawn.



Storing the level

Now, we will define a model that can store the state of a level. A level contains several objects that are placed in a certain location:

```
export interface Object {  
    x: number;  
    y: number;  
}
```

An enemy should also contain the location at which it is targeted. An enemy might not always know where the player is, so it cannot always run toward the player:

```
export interface Enemy extends Object {  
    toX: number;  
    toY: number;  
}
```

For a wall, we store the sides on which walls exist. These neighbors are used to draw the walls:

```
export interface Wall extends Object {  
    neighbours: Side;  
}
```

We can store these objects in the level. We also store the size of the grid, the current movement and the movement based on the keyboard input in the level:

```
export interface Level {  
    walls: Wall[];  
    dots: Object[];  
    enemies: Enemy[];  
    player: Object;  
    width: number;  
    height: number;  
    inputMovement: Movement;  
    currentMovement: Movement;  
    difficulty: Difficulty;  
}
```

Creating the default level

We will write a function that can parse a level, based on a string. This allows us to create the level as follows:

```
const defaultLevel = parseLevel([
    "WWWWWWWWWWWWWWWW",
    "W.....E.....W",
    "W.WWWW.WWWW.W",
    "W.W...W.W...W.W",
    "WE..W....W...W",
    "W.W...W.W...W.W",
    "W.WWWW.WWWW.W",
    "W.....W",
    "WWWW.WW WW.WWWW",
    "W....W W....W",
```

```
"W.WW.W P W.WW.W",
"W.WW.WW WW.WWEW",
"W.....W",
"WWWWWWWWWWWWWWWWWW"
```

]);

A `W` means that there should be a wall in that location, `E` stands for an enemy, `P` for the player, and a dot for a dot that Pac Man can eat.

To parse a level, we will first split these strings into an array of arrays, our grid:

```
function parseLevel(data: string[]): Level {
    const grid = data.map(row => row.split(""));
```

We will create a function `mapBoard`, which will transform this grid into an array of objects. `toObject` creates an object if the grid contains the specified character in that location:

```
return {
    walls: mapBoard(toWall),
    dots: mapBoard(toObject(".")),
    enemies: mapBoard(toEnemy),
    player: mapBoard(toObject("P"))[0],
    width: grid[0].length,
    height: grid.length,
    inputMovement: Movement.None,
    currentMovement: Movement.None,
    difficulty: Difficulty.Easy
};
```

In `mapBoard`, we first apply the callback to each element of the grid. We then flatten the grid to a one dimensional array. We filter elements that are `undefined` out of this array, as the callback should return `undefined` when the element in the grid at that location is not the expected kind:

```
function mapBoard<U>(callback: (field: string, x: number, y: number) => U | undefined): U[] {
    const mapped = grid.map((row, y) => row.map((field, x) =>
        callback(field, x, y)));
    return flatten(mapped).filter(item => item !== undefined)
        as U[];
}
```

We have to cast the value in the return-statement. The TypeScript compiler cannot follow that the call to `filter` only passes through values that are not `undefined`.

In `toObject`, we create a function that will create an object if the grid contains the specified character at that position. A function that returns a function can be used to curry. Currying means that you first provide some arguments, and later on the other arguments. In this case, we provide the first argument, the kind within the `return` statement, some preceding lines. The other arguments are provided by the `mapBoard` function:

```
function toObject(kind: string) {
    return (value: string, x: number, y: number) => {
        if (value !== kind) return undefined;
        return { x, y };
    } We will return the content of a field of the grid in get. If the
index is out of bounds, we
}

We will return the content of a field of the grid in get. If the index is out of bounds, we return undefined:
```

```
function get(x: number, y: number) {
    const row = grid[y];
    if (!row) return undefined;
    return row[x];
}
```

We use this function to check for the neighbors of a wall. Using the bitwise defined `enum`, we can register all sides on which the wall has a neighbor:

```
function toWall(kind: string, x: number, y: number): Wall | undefined {
    if (kind !== "W") return undefined;
    let neighbours: Side = 0;
    if (get(x - 1, y) === "W") neighbours |= Side.Left;
    if (get(x + 1, y) === "W") neighbours |= Side.Right;
    if (get(x, y - 1) === "W") neighbours |= Side.Bottom;
    if (get(x, y + 1) === "W") neighbours |= Side.Top;
    if (get(x - 1, y - 1) === "W") neighbours |= Side.LeftBottom;
    if (get(x - 1, y + 1) === "W") neighbours |= Side.LeftTop;
    if (get(x + 1, y - 1) === "W") neighbours |= Side.RightBottom;
    if (get(x + 1, y + 1) === "W") neighbours |= Side.RightTop;
    return {
        x,
        y,
        neighbours
    }
}
```

In `toEnemy`, we set the initial location of the enemy and the target location to the same values:

```
function toEnemy(kind: string, x: number, y: number) {
  if (kind !== "E") return undefined;
  return {
    x,
    y,
    toX: x,
    toY: y
  };
}
```

Finally, we create a small function that checks whether an object is aligned on the grid:

```
export function onGrid({ x, y }: Object) {
  return isInt(x) && isInt(y);
}
```

We have now created the default level. You can easily add another level later on, when the main menu has been added.

Creating the state

We store the state in a new interface. We will also define the default state for our game:

```
export interface State {
  level: Level;
}
export const defaultState: State = {
  level: defaultLevel
};
```

This interface is, at the moment, not very useful as you might be better off using the `Level` as the game state. However, later on in this chapter, we will also add a menu that should exist in the state.

Drawing the view

In `lib/game/view.ts`, we will render the game. We start with importing types that we defined earlier:

```
import { State, Level, Object, Wall, Side, Menu } from "./model";
import { Picture, Pictures, Translate, Scale, Rotate, Rectangle, Line,
Circle, Color, Text, Empty } from "../framework/picture";
```

We will store the font name in a variable, so we can easily change it later:

```
const font = "Arial";
```

In `draw`, we will render the game. For now, that means only drawing the level. Later on, we will add a menu to the game:

```
export function draw(state: State, width: number, height: number) {
    drawLevel(state.level, width, height),
}
```

We render the level in `drawLevel`. We calculate the size of all objects with the size of the grid and the canvas:

```
function drawLevel(level: Level, width: number, height: number) {
    const scale = Math.min(width / (level.width + 1), height / (level.height + 1));
```

We scale and center the whole level with this calculated scale:

```
    return new Scale(scale, scale,
        new Translate(-level.width / 2 + 0.5, -level.height / 2 + 0.5, new
Pictures([
```

Next, we draw all objects on the canvas. We use several functions that we create as follows:

```
        drawObjects(level.walls, drawWall),
        drawObjects(level.walls, drawWallLines),
        drawObjects(level.dots, drawDot),
        drawObjects(level.enemies, drawEnemy),
        drawObject(drawPlayer)(level.player)
    )));
};
```

In `drawObject`, we draw an object using the specified callback. We translate the picture of the object to the right location:

```
function drawObject<U extends Object>(callback: (item: U) => Picture) {
    return (item: U) =>
        new Translate(item.x, item.y, callback(item));
}
```

With `drawObjects`, we can draw a list of objects:

```
function drawObjects<U extends Object>(items: U[], callback: (item: U) => Picture) {
    return new Pictures(items.map(drawObject(callback)));
}
```

In `drawWall`, we render the background of a wall:

```
function drawWall() {
    return new Color("#111", new Rectangle(0, 0, 1, 1));
}
```

We render the edges of a wall in `drawWallLines`. We check the neighbors of a wall with the bitwise enum that we defined earlier. First, we list all possible sides in an array:

```
const leftTop: [number, number] = [-0.5, 0.5];
const leftBottom: [number, number] = [-0.5, -0.5];
const rightTop: [number, number] = [0.5, 0.5];
const rightBottom: [number, number] = [0.5, -0.5];
const wallLines: [Side, Line][] = [
    [Side.Left, new Line([leftTop, leftBottom], 0.1)],
    [Side.Right, new Line([rightTop, rightBottom], 0.1)],
    [Side.Top, new Line([leftTop, rightTop], 0.1)],
    [Side.Bottom, new Line([leftBottom, rightBottom], 0.1)]
];
```

We filter this array with the bitwise enum, and color the remaining pieces:

```
function drawWallLines({ neighbours }: Wall) {
    const lines = wallLines
        .filter(([side]) => (side & neighbours) === 0)
        .map(([side, line]) => line);
    return new Color("#0021b3", new Pictures(lines));
}
```

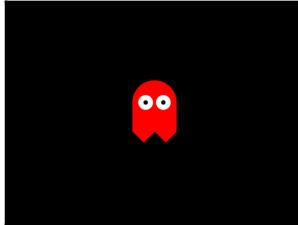
In `drawDot`, we will show a small circle for a dot:

```
function drawDot() {  
    return new Color("#f0c0a8", new Circle(0, 0, 0.2, 0.2));  
}
```

We render the player as a circle. You can try to create the famous, eating Pac Man yourself later on:

```
function drawPlayer() {  
    return new Color("#ffff00", new Circle(0, 0, 0.8, 0.8));  
}
```

We do some more work to draw an enemy. The enemy will look as follows:



The background of the enemy consists of a circle for its head, a rectangle for the body, and two rotated rectangles for the feet.

```
function drawEnemy() {  
    const shape = new Color("#ff0000", new Pictures([  
        new Circle(0, 0.15, 0.6),  
        new Rectangle(0, -0.05, 0.6, 0.4),  
        new Translate(-0.15, -0.25,  
            new Rotate(Math.PI / 4, new Rectangle(0, 0, 0.2, Math.SQRT2 *  
0.15))),  
        new Translate(0.15, -0.25,  
            new Rotate(Math.PI / 4, new Rectangle(0, 0, 0.2, Math.SQRT2 * 0.15))))  
    ]);
```

The eyes consist of two white circles with two smaller black circles as pupils.

```
const eyes = new Color("#ffff", new Pictures([
    new Circle(-0.12, 0.15, 0.2),
    new Circle(0.12, 0.15, 0.2)
]));
const pupils = new Color("#0000", new Pictures([
    new Circle(-0.12, 0.15, 0.06),
    new Circle(0.12, 0.15, 0.06)
]));
return new Pictures([shape, eyes, pupils]);
}
```

Handling events

We will create an event handler in `lib/game/event.ts`. The event handler must set the correct movement direction in the state. The time handler will then use this to update the direction of the player. The step can only do that when the player is aligned to the grid. If the player is between two fields on the grid, we will not change the direction of the player, since he will then probably head into a wall.

Working with key codes

An event provides the key code of the pressed or released key. We can get this code of a certain character with `"x".charCodeAt(0)` (where `x` is the character). The key codes of left, top, right, and bottom are 37, 38, 39, and 40.

First, we must create a helper function that transforms a key code to the `Movement` enum that we defined earlier. We store the different keys that we use in a new enum:

```
import { KeyEvent, KeyEventKind } from "../framework/event";
import { State, Movement } from "./model";
import { update } from "./utils";

enum Keys {
    Top = 38,
    Left = 37,
    Bottom = 40,
    Right = 39,
    Space = " ".charCodeAt(0)
}
```

Now we can transform a key code to a Movement:

```
function getMovement(key: number) {
    switch (key) {
        case Keys.Top:
            return Movement.Top;
        case Keys.Left:
            return Movement.Left;
        case Keys.Bottom:
            return Movement.Bottom;
        case Keys.Right:
            return Movement.Right;
    }
    return undefined;
}
```

The event handler will invoke `eventHandlerPlaying`, which we will define later on in this section. When we add a menu to the application, we will adjust this handler:

```
export function eventHandler(state: State, event: KeyEvent) {
    return eventHandlerPlaying(state, event);
}
```

In `eventHandlerPlaying`, we update the movement in the state. When the user presses a key, we set the movement to that corresponding direction. When the user releases the key that maps to the current movement, we set the movement to None:

```
function eventHandlerPlaying(state: State, event: KeyEvent) {
    if (event instanceof KeyEvent) {
        const inputMovement = getMovement(event.keyCode);
        if (event.kind === KeyEventKind.Press) {
            if (inputMovement) {
                return update(state, {
                    level: update(state.level, { inputMovement })
                });
            }
        } else {
            if (inputMovement === state.level.inputMovement) {
                return update(state, {
                    level: update(state.level, { inputMovement: Movement.None })
                });
            }
        }
    }
    return state;
}
```

We have now finished the event handler for the game. When the user presses or releases a key, this is updated in the state. However, the real work is being done in the time handler, which we create in the next section.

Creating the time handler

The time handler requires some more work. First, we import other types and functions.

```
import { State, Level, Object, Enemy, Wall, Movement, isOppositeMovement, onGrid, Difficulty } from "./model";
import { update, randomInt, chance, distance, isInt } from "./utils";
```

We define a step function so that we can add the menu later on.

```
export function step(state: State) {
    return stepLevel(state);
}
```

In stepLevel, we can update the objects in the level. First, we update the location of the enemies. We use stepEnemy, which we define later on.

```
function stepLevel(state: State): State {
    const level = state.level;
    const enemies = level.enemies.map(enemy => stepEnemy(enemy, level.player, level.walls, level.difficulty));
```

We update the location of the player based on the current movement:

```
const player = stepPlayer(level.player, level.currentMovement, level.walls);
```

Dots that are near the player, are eaten by the player and removed from the level:

```
const dots = stepDots(level.dots, player);
```

We change the current movement if the player is aligned on the grid or when they wants to move in the opposite direction:

```
const currentMovement = onGrid(player) ||
isOppositeMovement(level.inputMovement, level.currentMovement) ?
level.inputMovement : level.currentMovement;
```

We use these values to update the level:

```
const newLevel = update(level, { enemies, dots, player, currentMovement
});
return update(state, { level: newLevel });
}
```

Now, we create a function that checks whether an object collides with a wall:

```
function collidesWall(x: number, y: number, walls: Wall[]) {
  for (const wall of walls) {
    if (Math.abs(wall.x - x) < 1 && Math.abs(wall.y - y) < 1) {
      return true;
    }
  }
  return false;
}
```

Next, we create a function that updates the position of an enemy. The enemy can walk 0.0125 points if the difficulty is easy, otherwise, they can move 0.025 point. These values are chosen so that after a certain amount of steps, the enemy has walked exactly 1 point on the grid. Thus, the enemy will always be aligned to the grid again:

```
function stepEnemy(enemy: Enemy, player: Object, walls: Wall[], difficulty:
Difficulty): Enemy {
  const enemyStepSize = difficulty === Difficulty.Easy ? 0.0125 : 0.025;

  let { x, y, toX, toY } = enemy;
```

With a certain chance, the enemy will target on the player again. An enemy cannot always see where the player is, and the chance simulates that. Also, the enemy will get a small deviation:

```
if (chance(1 / (difficulty === Difficulty.Extreme ? 30 : 10))) {
  toX = Math.round(player.x) + randomInt(-2, 2);
  toY = Math.round(player.y) + randomInt(-2, 2);
}
```

If the enemy is aligned on the grid, it can move in all directions. Otherwise, it can only walk ahead or back:

```
if (!isInt(x)) {
  x += toX > x ? enemyStepSize : -enemyStepSize;
} else if (!isInt(y)) {
  y += toY > y ? enemyStepSize : -enemyStepSize;
} else {
```

The player is aligned on the grid, but the location might have a small rounding error. Thus, we round the values here.

```
x = Math.round(x);  
y = Math.round(y);
```

To walk around, we first create an array of all options. Then, we filter these options and sort them. With a chance of 0.2, the enemy will choose the second-best option. Otherwise, it will choose the best option. The best option is the option that brings the enemy as close to the enemy:

```
const options: [number, number][] = [  
    [x + enemyStepSize, y],  
    [x - enemyStepSize, y],  
    [x, y + enemyStepSize],  
    [x, y - enemyStepSize]  
];  
const possible = options  
    .filter(([x, y]) => !collidesWall(x, y, walls))  
    .sort(compareDistance);  
if (possible.length !== 0) {  
    if (possible.length > 1 && chance(0.2)) {  
        [x, y] = possible[1];  
    }  
    [x, y] = possible[0];  
}  
return {  
    x, y, toX, toY  
};
```

At the end of this function, we define the `compare` function that we used to sort the array. Such `compare` functions should return a negative value if the first argument comes after the second argument, and a positive value if the first argument should come before the other:

```
function compareDistance([x1, y1]: [number, number], [x2, y2]: [number, number]) {  
    return distance(toX, toY, x1, y1) - distance(toX, toY, x2, y2);  
}
```

We update the location of the player in `stepPlayer`:

```
const playerStepSize = 0.04;
function stepPlayer(player: Object, movement: Movement, walls: Wall[]): Object {
    let { x, y } = player;
```

When the player is aligned on the grid, we round the location to eliminate rounding errors:

```
if (onGrid(player)) {
    x = Math.round(x);
    y = Math.round(y);
}
```

If the user has no movement, we do not modify the player and we can return it directly:

```
switch (movement) {
    case Movement.None:
        return player;
```

Otherwise, we update the x or y coordinate of the player:

```
case Movement.Left:
    x -= playerStepSize;
    break;
case Movement.Right:
    x += playerStepSize;
    break;
case Movement.Top:
    y += playerStepSize;
    break;
case Movement.Bottom:
    y -= playerStepSize;
    break;
}
```

If the user was not aligned on the grid, we do not have to check whether the player collides with a wall. Otherwise, we must validate it. If the user then does collide with a wall, we return the old player with the old location:

```
if (onGrid(player) && collidesWall(x, y, walls)) {
    return player;
}
return { x, y };
}
```

We can filter the dots by calculating the distance to Pac Man. When they are close to the player, they are eaten by Pac Man and filtered out:

```
function stepDots(dots: Object[], player: Object) {
    return dots.filter(dot => distance(dot.x, dot.y, player.x, player.y) >=
0.55)
}
```

The time handler can now update the state: the player moves, the enemies try to move toward the player, and the player can eat dots.

Running the game

To start the game, we must call the `game` function with the default state, draw function, time handler, and event handler. In `lib/game/index.ts`, we write the following code to start the game:

```
import { game } from "../framework/game";
import { defaultState } from "./model";
import { draw } from "./view";
import { step } from "./step";
import { eventHandler } from "./event";

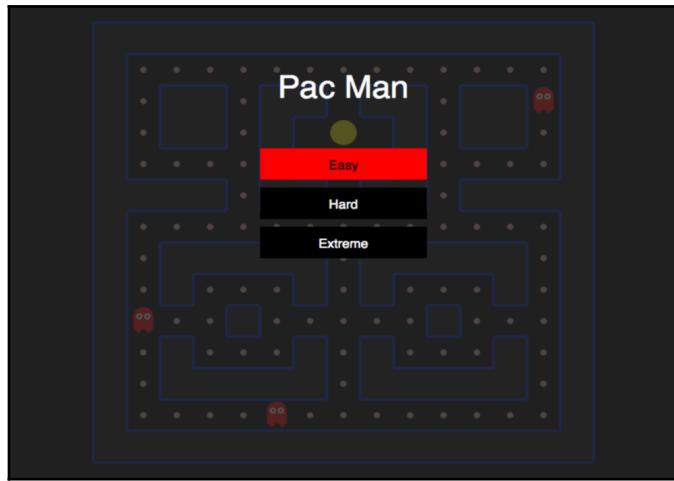
const canvas = <HTMLCanvasElement> document.getElementById("game");
game(canvas, document.body, 60, defaultState, draw, step, eventHandler);
```

We can compile the game by executing `gulp`. You can play the game by opening `static/index.html`.

As you will see, nothing happens when you have eaten all of the dots, or when you get hit by an enemy. In the next section, we will implement a menu. When the player wins or loses, we will show this menu.

Adding a menu

To finish off the game, we will add some menus to it. In the main menu, the user can choose a difficulty. The user can select an option using the arrow keys and confirm using the spacebar. The menu will look like this:



To implement the menu, we must add it to the state. Then we can render the menu and update the menu state in the event handler. We start by updating the state.

Changing the model

In `lib/game/model.ts`, we will add the menus to the state. First, we will create a new type for the menu. The menu contains a title, a list of options, and the index of the selected button. Each option has a string and a function that applies the action by transforming the state:

```
export interface Menu {  
    title: string;  
    options: [string, (state: State) => State][];  
    selected: number;  
}
```

We add the menu to the State:

```
export interface State {  
    menu: Menu | undefined;  
    level: Level;  
}
```

The main menu will contain three buttons; to start an easy, hard, or extreme game. We will define a function that can start the game with a specified difficulty:

```
const startGame = (difficulty: Difficulty) => (state: State) => ({  
    menu: undefined,  
    level: update(defaultLevel, { difficulty })  
});
```

Now we can define the main menu:

```
export const menuMain: Menu = {  
    title: "Pac Man",  
    options: [  
        ["Easy", startGame(Difficulty.Easy)],  
        ["Hard", startGame(Difficulty.Hard)],  
        ["Extreme", startGame(Difficulty.Extreme)]  
    ],  
    selected: 0  
}
```

We can define two more menus, which are shown when the user wins or dies:

```
export const menuWon: Menu = {  
    title: "You won!",  
    options: [  
        ["Back", state => ({ menu: menuMain, level: state.level })]  
    ],  
    selected: 0  
}  
export const menuLost: Menu = {  
    title: "Game over!",  
    options: [  
        ["Back", state => ({ menu: menuMain, level: state.level })]  
    ],  
    selected: 0  
}
```

We can use this menu in the starting state of the application:

```
export const defaultState: State = {
    menu: menuMain,
    level: defaultLevel
};
```

Since the menu is a part of the default state, the game will start with the menu. In the next sections, we will render the menu and handle its events.

Rendering the menu

We must update `lib/game/view.ts` to draw the menu on the canvas. We change the draw function:

```
export function draw(state: State, width: number, height: number) {
    return new Pictures([
        drawLevel(state.level, width, height),
        drawMenu(state.menu, width, height)
    ]);
}
```

Next, we create `drawMenu`, that will render the level. It will show the title and the buttons. The selected button gets a different color:

```
function drawMenu(menu: Menu | undefined, width: number, height: number): Picture {
    if (menu === undefined) return new Empty();
    const selected = menu.selected;
    const background = new Color("rgba(40,40,40,0.8)", new Rectangle(0, 0, width, height));
    const title = new Translate(0, 200, new Scale(4, 4,
        new Color("#ffff", new Text(menu.title, font)))
    );
    const options = new Pictures(menu.options.map(showOption));

    return new Pictures([background, title, options]);

    function showOption(item: [string, (state: State) => State],
    index: number) {
        const isSelected = index === selected;
        return new Translate(0, 100 - index * 50, new Pictures([
            new Color(isSelected ? "#ff0000" : "#000000",
                new Rectangle(0, 0, 200, 40)),
            new Color(isSelected ? "#000000" : "#ffffff",
                new Scale(1.6, 1.6, new Text(item[0], font)))
        ]));
    }
}
```

```
    ]));
}
}
```

This function will now draw the menu when it is active. We must still handle the events of the menu. We will do that in the next section.

Handling events

In `lib/game/event.ts`, we will handle the events for the menu. We must update the index of the selected item when the user presses the up or down key. When the user presses space, we execute the action of the selected button. First, we must adjust `eventHandler` to call `eventHandlerMenu` when the menu is visible.

```
export function eventHandler(state: State, event: KeyEvent) {
    if (state.menu) {
        return eventHandlerMenu(state, event);
    } else {
        return eventHandlerPlaying(state, event);
    }
}
```

Next, we create `eventHandlerMenu`.

```
function eventHandlerMenu(state: State, event: KeyEvent) {

    if (event instanceof KeyEvent && event.kind === KeyEventKind.Press) {
        const menu = state.menu!;
        let selected = menu.selected;
        switch (event.keyCode) {
            case Keys.Top:
                selected--;
                if (selected < 0) {
                    selected = menu.options.length - 1;
                }
                return {
                    menu: update(menu, {
                        selected
                    }),
                    level: state.level
                };
            case Keys.Bottom:
                selected++;
                if (selected >= menu.options.length) {
                    selected = 0;
                }
        }
    }
}
```

```

        return {
            menu: update(menu, {
                selected
            }),
            level: state.level
        };
    case Keys.Space:
        const option = menu.options[menu.selected];
        return option[1](state);
    default:
        return state;
    }
}
return state;
}

```

You can navigate through the menu using the arrow keys and the space bar. However, in the background, the game is still running. In the next section, we will not update the `state` of the level when the menu is `active`. Also, we will show a menu when the user has won or lost.

Modifying the time handler

In `lib/game/step.ts`, we must show the menu when the user won or lost. We must change the import-statement to import `menuLost` and `menuWon` from `model`:

```
import { State, Level, Object, Enemy, Wall, Movement, isOppositeMovement,
menuLost, menuWon, onGrid, Difficulty } from "./model";
```

In `newMenu`, we check whether such a menu should be shown.

```

function newMenu(player: Object, dots: Object[], enemies: Enemy[]) {
    for (const enemy of enemies) {
        if (distance(enemy.x, enemy.y, player.x, player.y) <= 1) {
            return menuLost;
        }
    }
    if (dots.length === 0) return menuWon;
    return undefined;
}

```

In `stepLevel`, we must call this function.

```

function stepLevel(state: State): State {
    const level = state.level;
    const enemies = level.enemies.map(enemy => stepEnemy(enemy, level.player,

```

```
level.walls, level.difficulty));
const player = stepPlayer(level.player, level.currentMovement,
level.walls);
const dots = stepDots(level.dots, player);
const currentMovement = onGrid(player) ||
isOppositeMovement(level.inputMovement, level.currentMovement) ?
level.inputMovement : level.currentMovement;
const menu = newMenu(player, dots, enemies);
const newLevel = update(level, { enemies, dots, player, currentMovement
});
return update(state, { level: newLevel, menu });
}
```

Finally, we must not call `stepLevel` in `step` if the menu is active.

```
export function step(state: State) {
if (state.menu === undefined) {
    return stepLevel(state);
} else {
    return state;
}
}
```

We can now compile the game again with `gulp` and run it by opening `static/index.html`.

Summary

In this chapter, we have explored the HTML canvas. We have seen how we can design a framework to use functional programming. The framework provides abstraction around drawing on the canvas, which is not pure.

We have built the game Pac Man. The structure of this application was similar to a Flux architecture, like we saw in the previous chapter.

The enemies in this game are not very smart. They easily get stuck behind a wall. In the next chapter, we will take a look at another game, but we will only focus on the artificial intelligence (AI). We will create an application that can play Tic-Tac-Toe without losing. We will see how a Minimax strategy works and how we can implement it in TypeScript.

9

Playing Tic-Tac-Toe against an AI

We built the game Pac Man in the previous chapter. The enemies were not very smart; you can easily fool them. In this chapter, we will build a game in which the computer will play well. The game is called **Tic-Tac-Toe**. The game is played by two players on a grid, usually three by three. The players try to place their symbols three in a row (horizontal, vertical or diagonal). The first player can place crosses, the second player places circles. If the board is full, and no one has three symbols in a row, it is a draw.

The game is usually played on a three-by-three grid and the target is to have three symbols in a row. To make the application more interesting, we will make the dimension and the row length variable.

We will not create a graphical interface for this application, since we have already done that in *Chapter 6, Advanced Programming in TypeScript*. We will only build the game mechanics and the **artificial intelligence (AI)**. An AI is a player controlled by the computer. If implemented correctly, the computer should never lose on a standard three by three grid. When the computer plays against the computer, it will result in a draft. We will also write various unit tests for the application.

We will build the game as a command-line application. That means you can play the game in a terminal. You can interact with the game only with text input:

```
It's player one's turn!
Choose one out of these options:
1   X|X|
     +-+-
     |O|
     +-+-
     | |
```

```
2  x| |x
   +-+-
   |o|
   +-+-
   | |
3  x| |
   +-+-
   x|o|
   +-+-
   | |
4  x| |
   +-+-
   |o|x
   +-+-
   | |
5  x| |
   +-+-
   |o|
   +-+-
   x| |
6  x| |
   +-+-
   |o|
   +-+-
   |x|
7  x| |
   +-+-
   |o|
   +-+-
   | |x
```

We will build this application in the following steps:

- Creating the project structure
- Adding utility functions
- Creating the models
- Implementing the AI using Minimax
- Creating the interface
- Testing the AI
- Summary

Creating the project structure

We will locate the source files in `lib` and the tests in `lib/test`. We use `gulp` to compile the project and `AVA` to run tests. We can install the dependencies of our project with NPM:

```
npm init -y
npm install ava gulp gulp-typescript --save-dev
```

In `gulpfile.js`, we configure `gulp` to compile our TypeScript files:

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

var tsProject = ts.createProject("./lib/tsconfig.json");

gulp.task("default", function() {
  return tsProject.src()
    .pipe(ts(tsProject))
    .pipe(gulp.dest("dist"));
});
```

Configure TypeScript

We can download type definitions for NodeJS with NPM:

```
npm install @types/node --save-dev
```

We must exclude browser files in TypeScript. In `lib/tsconfig.json`, we add the configuration for TypeScript:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs"
  }
}
```

For applications that run in the browser, you will probably want to target ES5, since ES6 is not supported in all browsers. However, this application will only be executed in NodeJS, so we do not have such limitations. You have to use NodeJS 6 or later for ES6 support.

Adding utility functions

Since we will work a lot with arrays, we can use some utility functions. First, we create a function that flattens a two dimensional array into a one dimensional array:

```
export function flatten<U>(array: U[][]) {  
    return (<U[]>[]).concat(...array);  
}
```

Next, we create a function that replaces a single element of an array with a specified value. We will use functional programming in this chapter again, so we must use immutable data structures. We can use map for this, since this function provides both the element and the index to the callback. With this `index`, we can determine whether that element should be replaced:

```
export function arrayModify<U>(array: U[], index: number, newValue: U) {  
    return array.map((oldValue, currentIndex) =>  
        currentIndex === index ? newValue : oldValue);  
}
```

We also create a function that returns a random integer under a certain upper bound:

```
export function randomInt(max: number) {  
    return Math.floor(Math.random() * max);  
}
```

We will use these functions in the next sessions.

Creating the models

In `lib/model.ts`, we will create the model for our game. The model should contain the game state.

We start with the player. The game is played by two players. Each field of the grid contains the symbol of a player or no symbol. We will model the grid as a two dimensional array, where each field can contain a player:

```
export type Grid = Player[][];
```

A player is either `Player1`, `Player2`, or no player:

```
export enum Player {  
    Player1 = 1,  
    Player2 = -1,
```

```
    None = 0
}
```

We have given these members values so we can easily get the opponent of a player:

```
export function getOpponent(player: Player): Player {
    return -player;
}
```

We create a type to represent an index of the grid. Since the grid is two dimensional, such an index requires two values:

```
export type Index = [number, number];
```

We can use this type to create two functions that get or update one field of the grid. We use functional programming in this chapter, so we will not modify the grid. Instead, we return a new grid with one field changed:

```
export function get(grid: Grid, [rowIndex, columnIndex]: Index) {
    const row = grid[rowIndex];
    if (!row) return undefined;
    return row[columnIndex];
}
export function set(grid: Grid, [row, column]: Index, value: Player) {
    return arrayModify(grid, row,
        arrayModify(grid[row], column, value)
    );
}
```

Showing the grid

To show the game to the user, we must convert a grid to a string. First, we will create a function that converts a player to a string, then a function that uses the previous function to show a row, finally a function that uses these functions to show the complete grid.

The string representation of a grid should have lines between the fields. We create these lines with standard characters (+, -, and |). This gives the following result:

```
x|x|o
-+-
|o|
-+-
x| |
```

To convert a player to the string, we must get their symbol. For Player1, that is a cross and for Player2, a circle. If a field of the grid contains no symbol, we return a space to keep the grid aligned:

```
function showPlayer(player: Player) {  
    switch (player) {  
        case Player.Player1:  
            return "X";  
        case Player.Player2:  
            return "O";  
        default:  
            return " ";  
    }  
}
```

We can use this function to the tokens of all fields in a row. We add a separator between these fields:

```
function showRow(row: Player[]) {  
    return row.map(showPlayer).reduce((previous, current) => previous + "|" +  
    current);  
}
```

Since we must do the same later on, but with a different separator, we create a small helper function that does this concatenation based on a separator:

```
const concat = (separator: string) => (left: string, right: string) =>  
    left + separator + right;
```

This function requires the separator and returns a function that can be passed to reduce. We can now use this function in showRow:

```
function showRow(row: Player[]) {  
    return row.map(showPlayer).reduce(concat("|"));  
}
```

We can also use this helper function to show the entire grid. First we must compose the separator, which is almost the same as showing a single row. Next, we can show the grid with this separator:

```
export function showGrid(grid: Grid) {  
    const separator = "\n" + grid[0].map(() => "-").reduce(concat("+")) +  
    "\n";  
    return grid.map(showRow).reduce(concat(separator));  
}
```

Creating operations on the grid

We will now create some functions that do operations on the grid. These functions check whether the board is full, whether someone has won, and what options a player has.

We can check whether the board is full by looking at all fields. If no field exists that has no symbol on it, the board is full, as every field has a symbol:

```
export function isFull(grid: Grid) {
    for (const row of grid) {
        for (const field of row) {
            if (field === Player.None) return false;
        }
    }
    return true;
}
```

To check whether a user has won, we must get a list of all horizontal, vertical and diagonal rows. For each row, we can check whether a row consists of a certain amount of the same symbols on a row. We store the grid as an array of the horizontal rows, so we can easily get those rows. We can also get the vertical rows relatively easily:

```
function allRows(grid: Grid) {
    return [
        ...grid,
        ...grid[0].map((field, index) => getVertical(index)),
        ...
    ];
    function getVertical(index: number) {
        return grid.map(row => row[index]);
    }
}
```

Getting a diagonal row requires some more work. We create a helper function that will walk on the `grid` from a start point, in a certain direction. We distinguish two different kinds of diagonals: a diagonal that goes to the lower-right and a diagonal that goes to the lower-left.

For a standard three by three game, only two diagonals exist. However, a larger grid may have more diagonals. If the grid is 5 by 5, and the users should get three in a row, ten diagonals with a length of at least three exist:

1. 0, 0 to 4, 4
2. 0, 1 to 3, 4
3. 0, 2 to 2, 4

4. 1, 0 to 4, 3
5. 2, 0 to 4, 2
6. 4, 0 to 0, 4
7. 3, 0 to 0, 3
8. 2, 0 to 0, 2
9. 4, 1 to 1, 4
10. 4, 2 to 2, 4

The diagonals that go toward the lower-right, start at the first column or at the first horizontal row. Other diagonals start at the last column or at the first horizontal row. In this function, we will just return all diagonals, even if they only have one element, since that is easy to implement.

We implement this with a function that walks the grid to find the diagonal. That function requires a start position and a `step` function. The `step` function increments the position for a specific direction:

```
function allRows(grid: Grid) {
    return [
        ...grid,
        ...grid[0].map((field, index) => getVertical(index)),
        ...grid.map((row, index) => getDiagonal([index, 0], stepDownRight)),
        ...grid[0].slice(1).map((field, index) => getDiagonal([0, index + 1], stepDownRight)),
        ...grid.map((row, index) => getDiagonal([index, grid[0].length - 1], stepDownLeft)),
        ...grid[0].slice(1).map((field, index) => getDiagonal([0, index], stepDownLeft))
    ];
    function getVertical(index: number) {
        return grid.map(row => row[index]);
    }
    function getDiagonal(start: Index, step: (index: Index) => Index) {
        const row: Player[] = [];
        let index: Index | undefined = start;
        let value = get(grid, index);
        while (value !== undefined) {
            row.push(value);
            index = step(index);
            value = get(grid, index);
        }
        return row;
    }
    function stepDownRight([i, j]: Index): Index {
```

```

        return [i + 1, j + 1];
    }
    function stepDownLeft([i, j]: Index): Index {
        return [i + 1, j - 1];
    }
    function stepUpRight([i, j]: Index): Index {
        return [i - 1, j + 1];
    }
}

```

To check whether a row has a certain amount of the same elements on a row, we will create a function with some nice looking functional programming. The function requires the array, the player, and the index at which the checking starts. That index will usually be zero, but during recursion we can set it to a different value. `originalLength` contains the original length that a sequence should have. The last parameter, `length`, will have the same value in most cases, but in recursion we will change the value. We start with some base cases. Every row contains a sequence of zero symbols, so we can always return `true` in such a case:

```

function isWinningRow(row: Player[], player: Player, index: number,
originalLength: number, length: number): boolean {
    if (length === 0) {
        return true;
    }
}

```

If the row does not contain enough elements to form a sequence, the row will not have such a sequence and we can return `false`:

```

if (index + length > row.length) {
    return false;
}

```

For other cases, we use recursion. If the current element contains a symbol of the provided player, this row forms a sequence if the next `length-1` fields contain the same symbol:

```

if (row[index] === player) {
    return isWinningRow(row, player, index + 1, originalLength, length -
1);
}

```

Otherwise, the row should contain a sequence of the original length in some other position:

```

return isWinningRow(row, player, index + 1, originalLength,
originalLength);
}

```

If the grid is large enough, a row could contain a long enough sequence after a sequence that was too short. For instance, XXXXXX contains a sequence of length three. This function handles these rows correctly with the parameters `originalLength` and `length`.

Finally, we must create a function that returns all possible sets that a player can do. To implement this function, we must first find all indices. We filter these indices to indices that reference an empty field. For each of these indices, we change the value of the grid into the specified player. This results in a list of options for the player:

```
export function getOptions(grid: Grid, player: Player) {
  const rowIndices = grid.map((row, index) => index);
  const columnIndices = grid[0].map((column, index) => index);
  const allFields = flatten(rowIndices.map(
    row => columnIndices.map(column => <Index> [row, column]))
  ));
  return allFields
    .filter(index => get(grid, index) === Player.None)
    .map(index => set(grid, index, player));
}
```

The AI will use this to choose the best option and a human player will get a menu with these options.

Creating the grid

Before the game can be started, we must create an empty grid. We will write a function that creates an empty grid with the specified size:

```
export function createGrid(width: number, height: number) {
  const grid: Grid = [];
  for (let i = 0; i < height; i++) {
    grid[i] = [];
    for (let j = 0; j < width; j++) {
      grid[i][j] = Player.None;
    }
  }
  return grid;
}
```

In the next section, we will add some tests for the functions that we have written. These functions work on the grid, so it will be useful to have a function that can parse a grid based on a string.

We will separate the rows of a grid with a semicolon. Each row contains tokens for each field. For instance, "XXO; O ;X" results in this grid:

```
x|x|o  
+-+-  
|o|  
+-+-  
x| |
```

We can implement this by splitting the string into an array of lines. For each line, we split the line into an array of characters. We map these characters to a Player value:

```
export function parseGrid(input: string) {  
    const lines = input.split(";");
    return lines.map(parseLine);
    function parseLine(line: string) {
        return line.split("").map(parsePlayer);
    }
    function parsePlayer(character: string) {
        switch (character) {
            case "X":
                return Player.Player1;
            case "O":
                return Player.Player2;
            default:
                return Player.None;
        }
    }
}
```

In the next section we will use this function to write some tests.

Adding tests

Just like in [Chapter 5, Native QR Scanner App](#), we will use AVA to write tests for our application. Since the functions do not have side effects, we can easily test them.

In `lib/test/winner.ts`, we test the `findWinner` function. First, we check whether the function recognizes the winner in some simple cases:

```
import test from "ava";
import { Player, parseGrid, findWinner } from "../model";

test("player winner", t => {
    t.is(findWinner(parseGrid("    ;XXX;    ")), Player.Player1);
    t.is(findWinner(parseGrid("    ;OOO;    ")), Player.Player2);
    t.is(findWinner(parseGrid("    ;    ")), Player.None);
});
```

We can also test all possible three-in-a-row positions in the three by three grid. With this test, we can find out whether horizontal, vertical, and diagonal rows are checked correctly:

```
test("3x3 winner", t => {
  const grids = [
    "XXX;   ;   ",
    "   ;XXX;   ",
    "   ;   ;XXX",
    "X  ;X  ;X  ",
    " X ; X ; X ",
    " X;  X;  X",
    "X  ; X ; X",
    "  X; X ;X  "
  ];
  for (const grid of grids) {
    t.is(findWinner(parseGrid(grid), 3), Player.Player1);
  }
});
```

We must also test that the function does not claim that someone won too often. In the next test, we validate that the function does not return a winner for grids that do not have a winner:

```
test("3x3 no winner", t => {
  const grids = [
    "XXO;OXX;XOO",
    "   ;   ;   ",
    "XXO;   ;OOX",
    "X  ;X  ; X "
  ];
  for (const grid of grids) {
    t.is(findWinner(parseGrid(grid), 3), Player.None);
  }
});
```

Since the game also supports other dimensions, we should check these too. We check that all diagonals of a four by three grid are checked correctly, where the length of a sequence should be two:

```
test("4x3 winner", t => {
  const grids = [
    "X  ; X  ;   ",
    " X ;  X ;   ",
    "   X ;   X;   ",
    "   ;X  ; X  ",
    "   X ;   X;   ",
    " X ;   X;   ",
    "X  ; X  ;   "
  ];
});
```

```
"      ;  X;  X "
];
for (const grid of grids) {
    t.is(findWinner(parseGrid(grid)), 2), Player.Player1);
}
});
```

You can of course add more test grids yourself.



Add tests before you fix a bug. These tests should show the wrong behavior related to the bug. When you have fixed the bug, these tests should pass. This prevents the bug returning in a future version.

Random testing

Instead of running the test on some predefined set of test cases, you can also write tests that run on random data. You cannot compare the output of a function directly with an expected value, but you can check some properties of it. For instance, `getOptions` should return an empty list if and only if the board is full. We can use this property to test `getOptions` and `isFull`.

First, we create a function that randomly chooses a player. To higher the chance of a full grid, we add some extra weight on the players compared to an empty field:

```
import test from "ava";
import { createGrid, Player, isFull, getOptions } from "../model";
import { randomInt } from "../utils";

function randomPlayer() {
    switch (randomInt(4)) {
        case 0:
        case 1:
            return Player.Player1;
        case 2:
        case 3:
            return Player.Player2;
    default:
        return Player.None;
    }
}
```

We create 10000 random grids with this function. The dimensions and the fields are chosen randomly:

```
test("get-options", t => {
  for (let i = 0; i < 10000; i++) {
    const grid = createGrid(randomInt(10) + 1, randomInt(10) + 1)
      .map(row => row.map(randomPlayer));
```

Next, we check whether the property that we describe holds for this grid:

```
const options = getOptions(grid, Player.Player1)
t.is(isFull(grid), options.length === 0);
```

We also check that the function does not give the same option twice:

```
for (let i = 1; i < options.length; i++) {
  for (let j = 0; j < i; j++) {
    t.notSame(options[i], options[j]);
  }
}
});
```

Depending on how critical a function is, you can add more tests. In this case, you could check that only one field is modified in an option or that only an empty field can be modified in an option:

Now you can run the tests using `gulp && ava dist/test`. You can add this to your `package.json` file. In the `scripts` section, you can add commands that you want to run. With `npm run xxx`, you can run task `xxx.npm test` that was added as shorthand for `npm run test`, since the `test` command is often used:

```
{
  "name": "chapter-7",
  "version": "1.0.0",
  "scripts": {
    "test": "gulp && ava dist/test"
  },
  ...
}
```

Implementing the AI using Minimax

We create an AI based on **Minimax**. The computer cannot know what his opponent will do in the next steps, but he can check what he can do in the worst-case. The minimum outcome of these worst cases is maximized by this algorithm. This behavior has given Minimax its name.

To learn how Minimax works, we will take a look at the value or score of a grid. If the game is finished, we can easily define its value: if you won, the value is 1; if you lost, -1 and if it is a draw, 0. Thus, for player 1 the next grid has value 1 and for player 2 the value is -1:

X X X
-++-
O O
-++-
X O

We will also define the value of a grid for a game that has not been finished. We take a look at the following grid:

X X
-++-
O O
-++-
O X

It is player 1's turn. He can place his stone on the top row, and he would win, resulting in a value of 1. He can also choose to lay his stone on the second row. Then the game will result in a draw, if player 2 is not dumb, with score 0. If he chooses to place the stone on the last row, player 2 can win resulting in -1. We assume that player 1 is smart and that he will go for the first option. Thus, we could say that the value of this unfinished game is 1.

We will now formalize this. In the previous paragraph, we have summed up all options for the player. For each option, we have calculated the minimum value that the player could get if he would choose that option. From these options, we have chosen the maximum value.

Minimax chooses the option with the highest value of all options.

Implementing Minimax in TypeScript

As you can see, the definition of Minimax looks like you can implement it with recursion. We create a function that returns both the best option and the value of the game. A function can only return a single value, but multiple values can be combined into a single value in a tuple, which is an array with these values.

First, we handle the base cases. If the game is finished, the player has no options and the value can be calculated directly:

```
import { Player, Grid, findWinner, isFull, getOpponent, getOptions } from
"./model";

export function minimax(grid: Grid, rowLength: number, player: Player): [Grid, number] {
    const winner = findWinner(grid, rowLength);
    if (winner === player) {
        return [undefined, 1];
    } else if (winner !== Player.None) {
        return [undefined, -1];
    } else if (isFull(grid)) {
        return [undefined, 0];
    }
}
```

Otherwise, we list all options. For all options, we calculate the value. The value of an option is the same as the opposite of the value of the option for the opponent. Finally, we choose the option with the best value:

```
} else {
    let options = getOptions(grid, player);
    const opponent = getOpponent(player);
    return options.map<[Grid, number]>(
        option => [option, -(minimax(option, rowLength, opponent)[1])])
        .reduce(
            (previous, current) => previous[1] < current[1] ? current : previous
        )!;
}
}
```

When you use tuple types, you should explicitly add a type definition for it. Since tuples are arrays too, an array type is automatically inferred. When you add the tuple as return type, expressions after the return keyword will be inferred as these tuples. For options.map, you can mention the union type as a type argument or by specifying it in the callback function (options.map((option): [Grid, number] => ...);).

You can easily see that such an AI can also be used for other kinds of games. Actually, the minimax function has no direct reference to Tic-Tac-Toe, only `findWinner`, `isFull` and `getOptions` are related to Tic-Tac-Toe.

Optimizing the algorithm

The Minimax algorithm can be slow. Choosing the first set, especially, takes a long time since the algorithm tries all ways of playing the game. We will use two techniques to speed up the algorithm.

First, we can use the symmetry of the game. When the board is empty it does not matter whether you place a stone in the upper-left corner or the lower-right corner. Rotating the grid around the center 180 degrees gives an equivalent board. Thus, we only need to take a look at half the options when the board is empty.

Secondly, we can stop searching for options if we found an option with value 1. Such an option is already the best thing to do.

Implementing these techniques gives the following function:

```
import { Player, Grid, findWinner, isFull, getOpponent, getOptions } from
"./model";

export function minimax(grid: Grid, rowLength: number, player: Player):
[Grid, number] {
    const winner = findWinner(grid, rowLength);
    if (winner === player) {
        return [undefined, 1];
    } else if (winner !== Player.None) {
        return [undefined, -1];
    } else if (isFull(grid)) {
        return [undefined, 0];
    } else {
        let options = getOptions(grid, player);
        const gridSize = grid.length * grid[0].length;
        if (options.length === gridSize) {
            options = options.slice(0, Math.ceil(gridSize / 2));
        }
        const opponent = getOpponent(player);
        let best: [Grid, number] | undefined = undefined;
        for (const option of options) {
            const current: [Grid, number] = [option, -(minimax(option, rowLength,
opponent)[1])];
            if (current[1] === 1) {
                return current;
            }
        }
    }
}
```

```

    } else if (best === undefined || current[1] > best[1]) {
      best = current;
    }
  return best!;
}
}

```

This will speed up the AI. In the next sections we will implement the interface for the game and we will write some tests for the AI.

Creating the interface

NodeJS can be used to create servers, as we did in chapters 2 and 3. You can also create tools with a **command line interface (CLI)**. For instance, gulp, NPM and typings are command line interfaces built with NodeJS. We will use NodeJS to create the interface for our game.

Handling interaction

The interaction from the user can only happen by text input in the terminal. When the game starts, it will ask the user some questions about the configuration: width, height, row length for a sequence, and the player(s) that are played by the computer. The highlighted lines are the input of the user:

```

Tic-Tac-Toe

Width
3
Height
3
Row length
2
Who controls player 1?
1 You

2 Computer

1
Who controls player 2?
1 You

2 Computer

```

During the game, the game asks the user which of the possible options he wants to do. All possible steps are shown on the screen, with an index. The user can type the index of the option he wants:

```

X| |
-+-
O|O|
-+-
|X|
It's player one's turn!
Choose one out of these options:
1 X|X|
-+-
O|O|
-+-
|X|
2 X| |X
-+-
O|O|
-+-
|X|
3 X| |
-+-
O|O|X
-+-
|X|
4 X| |
-+-
O|O|
-+-
X|X|
5 X| |
-+-
O|O|
-+-
|X|X

```

A NodeJS application has three standard streams to interact with the user. Standard input, **stdin**, is used to receive input from the user. Standard output, **stdout**, is used to show text to the user. Standard error, **stderr**, is used to show error messages to the user. You can access these streams with `process.stdin`, `process.stdout` and `process.stderr`.

You have probably already used `console.log` to write text to the console. This function writes the text to `stdout`. We will use `console.log` to write text to `stdout` and we will not use `stderr`.

We will create a helper function that reads a line from `stdin`. This is an asynchronous task, the function starts listening and resolves when the user hits enter. In `lib/cli.ts`, we start by importing the types and function that we have written.

```
import { Grid, Player, getOptions, getOpponent, showGrid, findWinner, isFull, createGrid } from "./model";
import { minimax } from "./ai";
```

We can listen to input from `stdin` using the `data` event. The process sends either the string or a buffer, an efficient way to store binary data in memory. With `once`, the callback will only be fired once. If you want to listen to all occurrences of the event, you can use `on`:

```
function readLine() {
    return new Promise<string>(resolve => {
        process.stdin.once("data", (data: string | Buffer) =>
            resolve(data.toString()));
    });
}
```

We can easily use `readLine` in `async` functions. For instance, we can now create a function that reads, parses and validates a line. We can use this to read the input of the user, parse it to a number, and finally check that the number is within a certain range. This function will return the value if it passes the validator. Otherwise it shows a message and retries.

```
async function readAndValidate<U>(message: string, parse: (data: string) => U, validate: (value: U) => boolean): Promise<U> {
    const data = await readLine();
    const value = parse(data);
    if (validate(value)) {
        return value;
    } else {
        console.log(message);
        return readAndValidate(message, parse, validate);
    }
}
```

We can use this function to show a question where the user has various options. The user should type the index of his answer. This function validates that the index is within bounds. We will show indices starting at 1 to the user, so we must carefully handle these.

```
async function choose(question: string, options: string[]) {
    console.log(question);
    for (let i = 0; i < options.length; i++) {
        console.log((i + 1) + "\t" + options[i].replace(/\n/g, "\n\t"));
        console.log();
    }
    return await readAndValidate(
```

```
    `Enter a number between 1 and ${ options.length }`,
    parseInt,
    index => index >= 1 && index <= options.length
  ) - 1;
}
```

Creating players

A player could either be a human or the computer. We create a type that can contain both kinds of players.

```
type PlayerController = (grid: Grid) => Grid | Promise<Grid>;
```

Next we create a function that creates such a player. For a user, we must first know whether he is the first or the second player. Then we return an async function that asks the player which step he wants to do.

```
const getUserPlayer = (player: Player) => async (grid: Grid) => {
  const options = getOptions(grid, player);
  const index = await choose("Choose one out of these options:",
    options.map(showGrid));
  return options[index];
};
```

For the AI player, we must know the player index and the length of a sequence. We use these variables and the grid of the game to run the Minimax algorithm.

```
const getAIPlayer = (player: Player, rowLength: number) => (grid: Grid) =>
  minimax(grid, rowLength, player)[0]!;
```

Now we can create a function that asks the player whether a player should be played by the user or the computer.

```
async function getPlayer(index: number, player: Player, rowLength: number): Promise<PlayerController> {
  switch (await choose(`Who controls player ${ index }?`, ["You",
    "Computer"])) {
    case 0:
      return getUserPlayer(player);
    default:
      return getAIPlayer(player, rowLength);
  }
}
```

We combine these functions in a function that handles the whole game. First, we must ask the user to provide the width, height and length of a sequence.

```
export async function game() {
    console.log("Tic-Tac-Toe");
    console.log();
    console.log("Width");
    const width = await readAndValidate("Enter an integer", parseInt,
isFinite);
    console.log("Height");
    const height = await readAndValidate("Enter an integer", parseInt,
isFinite);
    console.log("Row length");
    const rowLength = await readAndValidate("Enter an integer", parseInt,
isFinite);
```

We ask the user which players should be controlled by the computer.

```
const player1 = await getPlayer(1, Player.Player1, rowLength);
const player2 = await getPlayer(2, Player.Player2, rowLength);
```

The user can now play the game. We do not use a loop, but we use recursion to give the player their turns.

```
return play(createGrid(width, height), Player.Player1);
async function play(grid: Grid, player: Player): Promise<[Grid, Player]>
{
```

In every step, we show the grid. If the game is finished, we show which player has won.

```
    console.log();
    console.log(showGrid(grid));
    console.log();
    const winner = findWinner(grid, rowLength);
    if (winner === Player.Player1) {
        console.log("Player 1 has won!");
        return <[Grid, Player]> [grid, winner];
    } else if (winner === Player.Player2) {
        console.log("Player 2 has won!");
        return <[Grid, Player]> [grid, winner];
    } else if (isFull(grid)) {
        console.log("It's a draw!");
        return <[Grid, Player]> [grid, Player.None];
    }
}
```

If the game is not finished, we ask the current player or the computer which set he wants to do.

```
    console.log(`It's player ${ player === Player.Player1 ? "one's" : "two's" } turn!`);
    const current = player === Player.Player1 ? player1 : player2;
    return play(await current(grid), getOpponent(player));
}
}
```

In `lib/index.ts`, we can start the game. When the game is finished, we must manually exit the process.

```
import { game } from "./cli";

game().then(() => process.exit());
```

We can compile and run this in a terminal:

```
gulp && node --harmony_destructuring dist
```

At the time of writing, NodeJS requires the `--harmony_destructuring` flag to allow destructuring, like `[x, y] = z`. In future versions of NodeJS, this flag will be removed and you can run it without it.

Testing the AI

We will add some tests to check that the AI works properly. For a standard three by three game, the AI should never lose. That means when an AI plays against an AI, it should result in a draw. We can add a test for this. In `lib/test/ai.ts`, we import AVA and our own definitions.

```
import test from "ava";
import { createGrid, Grid, findWinner, isFull, getOptions, Player } from
"../model";
import { minimax } from "../ai";
import { randomInt } from "../utils";
```

We create a function that simulates the whole gameplay.

```
type PlayerController = (grid: Grid) => Grid;
function run(grid: Grid, a: PlayerController, b: PlayerController): Player
{
    const winner = findWinner(grid, 3);
    if (winner !== Player.None) return winner;
```

```
    if (isFull(grid)) return Player.None;
    return run(a(grid), b, a);
}
```

We write a function that executes a step for the AI.

```
const aiPlayer = (player: Player) => (grid: Grid) =>
minimax(grid, 3, player)[0]!;
```

Now we create the test that validates that a game where the AI plays against the AI results in a draw.

```
test("AI vs AI", t => {
  const result = run(createGrid(3, 3), aiPlayer(Player.Player1),
aiPlayer(Player.Player2))
  t.is(result, Player.None);
});
```

Testing with a random player

We can also test what happens when the AI plays against a random player or when a player plays against the AI. The AI should win or it should result in a draw. We run these multiple times; what you should always do when you use randomization in your test.

We create a function that creates the random player.

```
const randomPlayer = (player: Player) => (grid: Grid) => {
  const options = getOptions(grid, player);
  return options[randomInt(options.length)];
};
```

We write the two tests that both run 20 games with a random player and an AI.

```
test("random vs AI", t => {
  for (let i = 0; i < 20; i++) {
    const result = run(createGrid(3, 3), randomPlayer(Player.Player1),
aiPlayer(Player.Player2))
    t.not(result, Player.Player1);
  }
});

test("AI vs random", t => {
  for (let i = 0; i < 20; i++) {
    const result = run(createGrid(3, 3), aiPlayer(Player.Player1),
randomPlayer(Player.Player2))
    t.not(result, Player.Player2);
```

```
});
```

We have written different kinds of tests:

- Tests that check the exact results of single function
- Tests that check a certain property of results of a function
- Tests that check a big component

Always start writing tests for small components. If the AI tests should fail, that could be caused by a mistake in `hasWinner`, `isFull` or `getOptions`, so it is hard to find the location of the error. Only testing small components is not enough; bigger tests, such as the AI tests, are closer to what the user will do. Bigger tests are harder to create, especially when you want to test the user interface. You must also not forget that tests cannot guarantee that your code runs correctly, it just guarantees that your test cases work correctly.

Summary

In this chapter, we have written an AI for Tic-Tac-Toe. With the command line interface, you can play this game against the AI or another human. You can also see how the AI plays against the AI. We have written various tests for the application.

You have learned how Minimax works for turn-based games. You can apply this to other turn-based games as well. If you want to know more on strategies for such games, you can take a look at game theory, the mathematical study of these games.

Reading this means that you have finished the Functional Programming part of this book. One chapter remains, which will explain how you can migrate a legacy JavaScript code base to TypeScript.

10

Migrate JavaScript to TypeScript

In the previous chapters, we have built new applications in TypeScript. However, you might also have old code bases in JavaScript which you want to migrate to TypeScript. We will see how these projects can be converted to TypeScript.

You could rewrite the whole project from scratch, but that would require a lot of work for big projects. Since TypeScript is based on JavaScript, this transition can be done more efficiently.

Since the migration process is dependent on the project, this chapter can only give guidance. For various common topics, this chapter contains a recipe to migrate code. Migrating a project requires knowledge of the frameworks and the structure of the project.

The following steps are related to migrating a code base:

- Gradually migrating to TypeScript
- Adding TypeScript
- Migrating each file
- Refactoring the project

Gradually migrating to TypeScript

As of TypeScript 1.8, it is possible to combine JavaScript and TypeScript in the same project. Thus, you can migrate a project file by file.

During the migration of the files, the project should be working at every step. That way, you can check that the work is going well, and you can still work on the project. If an urgent bug is reported, you do not have to fix it in the old and migrated version; you only have to fix it in the current version.

You can convert the project in the following steps:

- Add the TypeScript compiler to the project
- Migrate each file
- Refactor and enable stricter checks of TypeScript

In the next sections, we will see how these steps can be done.

Adding TypeScript

Before you can convert JavaScript files to TypeScript, you must add the TypeScript compiler to a project. If the project already uses a build step, the TypeScript compiler must be integrated into the build step. Otherwise, a new build step must be created. In the following sections, we will set up TypeScript and the build system.

Configuring TypeScript

In all cases, you should start with configuring TypeScript. This configuration will be used by code editors and the build tool. The most important setting is `allowJs`. This setting allows JavaScript files in the TypeScript project. Other important options are `target` and `module`. For `target`, you can choose between `es3`, `es5`, and `es2015`. The latest version of JavaScript, `es2015`, is not supported in all browsers at the time of writing. You can target `es2015` when you write an application for NodeJS. You can target `es5` for browsers. For very old environments, you must target `es3`.

If the project uses external modules, you should also set the `module` setting. If your project uses `CommonJS`, mostly used in combination with NodeJS, browserify or webpack, you should use `"module": "commonjs"`. An import in CommonJS can be recognized by a call to `require` and an export by an assignment to `exports.[...]` or `module.exports`, and files are not wrapped in a `define` function:

```
var path = require("path");

exports.lorem = ...;
module.exports = ...;
```

Another module format is **AMD**, Asynchronous Module Definition. This format is used a lot for web applications. You can configure TypeScript for AMD with "module": "amd". The most popular implementation of AMD is `require.js`.

An AMD file starts with a `define` call.

```
define(["require", "exports", "path"], function (require, exports, path) {  
    exports.lorem = ...;  
});
```

Recent projects might use es2015 modules, with a build step. You can recognize such files by the `import` and `export` keywords.

```
import * as path from "path";  
export var lorem = ...;
```

If you use es2015 modules, you can set "module": "es2015". However, since these modules are often used with a certain build step to compile these modules to CommonJS, AMD or SystemJS, you can also do that directly with TypeScript. The TypeScript compiler will also do this transformation for the JavaScript files of the project, so you can remove the other build step that does this (for instance, Babel). If you want to do this, you must use "commonjs", "amd", or "systemjs".

If your project did not use a build step, you might want to change the directory structure of your project. You must not store the source files (TypeScript/JavaScript) in the same directory as the compiled files (JavaScript). In the previous chapters, we used `lib` for the sources and `dist` for the compiled files. We can configure that by setting "outDir": "dest". If you use a build tool such as gulp where temporary files can stay in memory and are not written to the disk, you do not need to set this option since the compiled files are not directly written to the disk.

This should result in a `tsconfig.json` file similar to this one:

```
{  
    "compilerOptions": {  
        "target": "es5",  
        "module": "commonjs",  
        "outDir": "dist"  
    }  
}
```

You should place this file in the directory that contains the source files. If your project did not use a build tool, you can now compile the project with `tsc -p ./lib` (where `./lib` references the directory that contains the source files and `tsconfig.json` file), provided that you have TypeScript installed (`npm install typescript -g`). If your project already used a build system, you have to integrate the TypeScript compiler into it, which we will do in the next section.

Configuring the build tool

Configuring the build depends on the build tool you use. For a few commonly used tools, you can find the steps here. Most build tools require you to install a TypeScript plugin. For browserify, you must install `tsify` using NPM; for Grunt, `grunt-ts`; for gulp, `gulp-typescript`; and for webpack, `ts-loader`. If your project uses JSPM, you do not have to install a plugin.

You can find various configurations with these tools at: <http://www.typescriptlang.org/docs/handbook/integrating-with-build-tools.html>. If you use a different build tool, you should look in the documentation of the tool and search for a TypeScript plugin.

Since TypeScript accepts (and needs) the JavaScript files in your project, you must provide all source files to the TypeScript compiler. For gulp, that would mean that you add TypeScript before other compilation steps. Imagine a task in your `gulp` file looks like this:

```
gulp.src("lib/**/*.js")
  .pipe(plugin())
  .pipe(gulp.dist("dest"));
```

You can add TypeScript to this `gulp` file:

```
var ts = require("gulp-typescript");
var tsProject = ts.createProject("lib/tsconfig.json");
...
gulp.src("lib/**/*.js")
  .pipe(ts(tsProject))
  .pipe(plugin())
  .pipe(gulp.dist("dest"));
```

For a build tool that cannot store temporary files in memory, such as Grunt, you should compile TypeScript to a temporary directory. Other steps from the build should read the sources from this directory.

For more information on how to configure a specific build tool, you can look at the documentation of the tool and the plugin.

Acquiring type definitions

For dependencies that you use, such as jQuery, you must acquire type definitions. You can install them using `npm`. You can find these type definitions on <https://aka.ms/types>.

Testing the project

Before going to the next step, make sure that the build is working. TypeScript should only show syntax errors in JavaScript files. Also, the application should be working at runtime.



If you are now using TypeScript to transpile ES modules to CommonJS, you might run into problems. Babel and TypeScript handle default imports differently. Babel looks for the `default` property, and if that does not exist, it behaves the same as a full module import. TypeScript will only look for the `default` property. If you get runtime errors after the migration, you might need to replace a default import (`import name from "package"`) with a full module import (`import * as name from "package"`).

Migrating each file

When the build system is set up, you can start with migrating files. It is easiest to start with files that do not depend on other files, as these do not depend on types of other files. To migrate a file, you must rename the file extension to `.ts`, convert the module format to ES modules, correct types that are inferred incorrectly, and add types to untyped entities. In the next sections, we will take a look at these tasks.

Converting to ES modules

In TypeScript files you cannot use CommonJS or AMD directly. Instead you must use ES modules, like we did in the previous chapters. For an import, you must choose from these:

- `import * as name from "package"`, imports the whole package, similar to `var name = require("package")` in CommonJS.
- `import name from "package"`, imports the default export, similar to `var name = require("package").default`.
- `import { name } from "package"`, imports a named export, similar to `var name = require("package").name`.

ES modules offer various constructs to export values from modules:

- `export function name() {}`, `export class Name {}`, `export var name`, exports a named variable. Compiles to:

```
function name() {}  
exports.name = name;
```

- `export default function name() {}`, `export default class Name {}`, `export default var name`, exports a variable as the default export. Compiles to:

```
function name() {}  
exports.default = name;
```

- `export default x`, where `x` is an expression, exports an expression as the default export.
- `export { x, y }`, exports variables `x` and `y` as named exports. This compiles to:

```
exports.x = x;  
exports.y = y;
```

With CommonJS and AMD you can also export an expression as the full module, with `module.exports = x` in CommonJS or `return x` in AMD. This is not possible with ES modules. If this pattern is used in a file that you must migrate, you can either switch to an ES export or patch all files that import this file, or use a legacy export statement in TypeScript. With `export = x`, you can export an expression as the full module. However, since this is not standard, it is not advised to do this but it can help during the migration. This compiles to `module.exports = x` or `return x`.

The file should give no syntax errors when you compile it. It might show type errors, which the next session will discuss.

Correcting types

Since the file was a JavaScript file, it does not have any type annotations. At some locations, TypeScript will infer types for variables. When you declare a variable and directly assign to it, TypeScript will infer the type based on the type of the assignment. Thus, when you write `let x = "",` TypeScript will type `x` as a `string`. However, in some cases the inferred type is not correct. You can see that in the next examples.

```
let x = {};
x.a = true;
```

The type of `x` is inferred as `{}`, an empty object. Thus, the assignment to `x.a` is not allowed, since the property `a` does not exist. You can fix this by adding a type to the definition: `let x: { a?: boolean } = {}.`

```
class Base {
  a: boolean;
}
class Derived extends Base {
  b: string;
}
let x = new Derived();
x = new Base();
```

In this case, the type of `x` is `Derived`. The assignment on the last line is not allowed, since `Base` is not assignable to `Derived`. You can again fix this by adding a type: `let x: Base = new Derived();`.

If the type of a variable is unknown or very complex, you can use `any` as the type for the variable, which disables type checking for that variable.

Other possible sources of errors are classes. When you use the `class` keyword to create classes, you can get errors that a property does not exist in the class.

```
class A {
  constructor() {
    this.x = true;
  }
}
```

In this example, you would get an error that the property `x` does not exist in `A`. In TypeScript, you must declare all properties of a class.

```
class A {  
  x: boolean;  
  constructor() {  
    this.x = true;  
  }  
}
```

Most errors of TypeScript should now be fixed. Some cases however can still generate type errors, for example when a variable has different types, which is discussed in the next session.

Adding type guards and casts

A common pattern in JavaScript is that a function accepts either a single value of a certain type, or an array of multiple types. You can express such a type with a union type:

```
function foo(input: string | string[]) {  
  ...  
}
```

In the body of such a function, you would check if the argument is an array or a single string. In most cases, TypeScript can correctly follow this pattern. For instance, TypeScript can change the type of `input` in the next example.

```
function foo(input: string | string[]) {  
  if (typeof input === "string") {  
  } else {  
  }  
}
```

The type of `input` is `string` in the block after the `if` and `string[]` in the `else`-block. The changing of a type is called narrowing and the checks for a type are called **type guards**. TypeScript has built-in support for `typeof` and `instanceof` type guards, but you can also define your own. A user defined type guard function is a function that takes the value as one of its arguments and returns `true` when the value is of a certain type. A type guard can be written like this:

```
function isBar(value: Foo): value is Bar {  
  ...  
}
```

As you can see, the return type of `isBar` is `value is Bar`, a special boolean type. If you have a function that checks that a value is of a certain type, you should add a return type to make it a type guard.

If the TypeScript compiler still cannot correctly narrow the type of a variable on a certain location, you must add a cast. A cast is a compiler instruction in which the programmer guarantees that a value is of a certain type. A type guard can be written in two different ways.

```
<Bar> value  
value as Bar
```

The first syntax is most used, but not supported in **JSX** or **TSX** files. In a **TSX** file, you must use the second syntax.

When you have fixed all these errors, the project should compile without errors again.

Using modern syntax

The `class` keyword was introduced in **ES2015**, a recent version of JavaScript. Older projects created classes with a function and prototypes should migrate to the new class syntax. In TypeScript, these classes can be typed better. Following are two code fragments, which show the same class written with the prototype and with the class syntax.

```
var A = (function () {  
    function A() {  
        this.x = true;  
    }  
    A.prototype.a = function () {  
    };  
    return A;  
}());  
  
class A {  
    x: boolean;  
    constructor() {  
        this.x = true;  
    }  
    a() {  
    }  
}
```

You can also use the new, block scoped variable declarations. Instead of `var x` you should write `const x` for a variable that is not reassigned and `let x` for a variable that will be reassigned.

Finally, you can also use arrow functions (or lambda expressions). Using normal function definitions, the value of `this` is lost in callbacks. Thus, you had to store that in a variable (`self` or `_this` was commonly used). You can replace that with an arrow function.

```
var _this = this;
function myCallback() {
    _this ...
}
```

This code fragment can be rewritten to:

```
const myCallback = () => {
    this ...
};
```

Adding types

The file compiles now, but lots of variables and parameters might be typed as `any`. For complex types, you can first create an interface, for object types, or a type alias, for function types or union types.

TypeScript does not infer types in the following cases:

- Variable declaration without an initializer (like `var x;`)
- Parameters of a function definition without a default value
- Return type of a function that uses recursion

In an editor like VS Code, you can check the type of a variable, parameter or function by hovering over it. On these locations you should add a type annotation yourself.

Refactoring the project

When you have ported the project to TypeScript, you can refactor the program more easily. You should remove patterns that do not fit well with TypeScript. For instance, magic string values should be replaced by enums. When you have a project that uses a framework, you can also do some framework related refactoring. For a React project, you might want to upgrade from the old class creation with `React.createClass` to the new class syntax.

A proper editor can help during refactoring. VS Code can rename an identifier in the whole project or find all references of an identifier. It can also format your code if it's messy or jump to the definition of an identifier. You can access these options with a right-click on the identifier in the code.

Which steps you must do for refactoring depends on your project. You should look for parts of the code that are not typed or incorrectly typed, because of a bad structure or some dynamic behavior.

Enable strict checks

You can enable stricter checks in TypeScript. These checks can improve the quality of your program. Here are a few options that can be useful.

- `noImplicitAny`: Checks that no variables are typed as any unless you explicitly annotated them with any.
- `noImplicitReturns`: Checks that all execution paths of a function return a value.
- `strictNullChecks`: Enables strict checks for variables that might be undefined or null.

Summary

In this chapter, we have looked at various steps involved in migrating a project from JavaScript to TypeScript. We saw how a project can be migrated gradually. We looked at various ways to update an old project so that it can use new JavaScript features and how it can use the type system of TypeScript. You can use your knowledge from the previous chapters to make the project even better.

Index

A

- about-page component
 - using, in other components 26
- actions
 - column, adding 161
 - creating 160
 - input popup, showing 163
 - rows, adding 161
 - testing 165
 - title, changing 162
- algorithms
 - Big-Oh notation 129
 - binary search 131
 - built-in functions 132
 - optimizing 130
 - performances, comparing 128, 129
- Angular 2 45
- Angular
 - as framework 98
 - comparing, with React 96
 - templates, using 96
- API
 - testing 62
- artificial intelligence (AI)
 - about 216
 - implementing, with Minimax 230
 - testing 238
 - testing, with random player 239
- asynchronous code, NodeJS
 - about 50
 - callback approach 50
 - callback approach, disadvantages 51
- Asynchronous Module Definition (AMD) 243
- authentication
 - adding 58
 - users, adding to database 61

users, implementing in database 60

B

- binary search 131
- build tool
 - reference link 244

C

- canvas
 - drawing on 190, 192
- chat room
 - application, running 96
 - creating 94
 - stateless functional components 95
 - two-way bindings 94
- client side
 - login form, creating 68, 69
 - main component 71
 - main component, error handler 73
 - menu, creating 70
 - note editor 71
 - writing 66
- command line interface (CLI)
 - creating 233
 - interaction, handling 233
 - players, creating 236, 238
- CommonJS 242
- component
 - creating 20
 - event listeners 24
 - interactions, adding 22
 - one-way variable binding 23
 - template 21
 - testing 21
- components
 - reference link 111
- conditions, adding to templates

about template, modifying 26
directives 25
template tag 25
constants 144
control flow base type analysis 125
Cordova 119
Create, Read, Update, and Delete (CRUD) 45
CRUD operations
 adding 63
 handlers, implementing 64
 request handling 66

D

decorators 21
discriminated union types 127

E

ES2015 249
event handler
 creating 203
 key code, working with 203
 key codes, working with 205
expressions
 calculating 145
 core parsers, creating 147
 data types, creating 139, 141
 data types, traversing 141, 142
 data types, using 139
 number, parsing 151
 order of operations 152, 154
 parsers, running in sequence 148, 150
 parsing 147
 validating 143

F

file migration
 about 245
 casts, adding 248
 converting, to ES modules 245
 correcting types 247
 modern syntax, using 249
 type guards, adding 248
 types, adding 250

Flux architecture
 action 158

advantages 175
cross-platform feature 175
dispatcher 158
dispatcher, creating 159
state, defining 158
store 158
store, creating 159
using 158
view 158
forecast component
 @Output, adding 36
 creating 29
 data, downloading 32, 35
 templates, using 30
forecast
 API, typing 28
 API, using 27
 displaying 27
framework
 components, binding 188
 designing 181
 events, creating 187
 other pictures, wrapping 185
 pictures, creating 182, 184
functional programming (FP)
 about 134, 137
 factorial, creating 138

H

hash 60
HTML5 canvas
 state, restoring 181
 state, saving 180
 using 179, 180

J

JavaScript
 array spread 8
 arrow functions 6
 classes, creating 6
 const 5
 destructuring 8
 ES2015 (ES6) version 4
 ES2016 4
 ES3 version 4

ES5 version 4
function arguments 7
let 5
new classes 9
template strings 8
JSON 116
JSON strings 59
JSX files 249

K

key-value 116

L

layout container
reference link 111
layouts, NativeScript
 DockLayout 111
 GridLayout 111
 WrapLayout 111

M

main component
 about 40
 event, listening to 41
 geolocation API, using 41
 other components, using 40
 sources 41
 two-way bindings 40
menu, PacMan
 adding 210
 events, handling 213
 model, changing 210
 rendering 212
 time handler, modifying 214

Minimax algorithm
 optimizing 232
Minimax
 implementing, in TypeScript 231
 used, for implementing AI 230
model, Tic-Tac-Toe game
 creating 219
 grid, creating 225
 grid, displaying 220
 operations, creating on grid 222
 random testing 228, 229

tests, adding 226
models, PacMan
 creating 193
 default level, creating 196, 199
 enums, using 194, 195
 level, storing 195
 state, creating 199
modules
 using 14
MongoDB 45
MongoDB database, NodeJS
 connecting to 53
 functions, wrapping in promises 52
 querying 54
 reference link 52

N

NativeScript
 about 99
 comparing 119
 detail view, creating 112
 details view, creating 109, 110, 111
 Hello World page, creating 103
 main view, creating 105, 107, 108
 persistent storage, adding 116
 structure, creating 101
 styling 117, 118
 TypeScript, adding 102
 working with 100

never type

 about 127

NodeJS

 about 45, 233
 asynchronous code 50
 MongoDB database 52
 starting with 49

note-taking app project

 build tool, configuring 46, 47
 directories 46
 running 75
 setting up 46
 type definitions 48

null type

 checking 126

O

open weather map
reference link 27

P

Pac Man
about 177
menu, adding 210
running 209
setting up 178
Phonegap 119
polyfill
reference link 9
pure code 137

Q

QR codes
scan function, implementing 113, 114
scanning 113
testing, on device 115
type definitions 113

R

React Native 119
React
about 76
as libraries 98
comparing, with Angular 96
components, creating with JSX 79
JSX, using 97
menu, creating 81
props, adding to component 80
starting with 79
state, adding to component 80

Real-Time Chat project
gulp, configuring 78
setting up 77
testing 84
rest argument 7

S

side effect 137
spreadsheet application project
setting up 135

summarizing 174
spreadsheet
all fields calculating 156
defining 155
structural type system
about 55
API, typing 58
generics 55

T

tagged union types
creating 127
template
conditions, adding 25
Tic-Tac-Toe game
about 216
structure, creating 218
TypeScript, configuring 218
time handler
creating 205, 209
Titanium 119
TSX file 249
type cast 121
type guard
about 248
accuracy, checking 124
assignments, checking 125
combining 124
instanceof guard 122
narrowing any 123
narrowing process 123
typeof guard 122
user defined 122
using 122

types
annotations 11
any 10
boolean 9
checking 9
defining 10
enum type 10
function type 11
intersection type 10
literal types 10
never 10

null 11
number 10
Object type 10
string 9
tuple type 11
undefined 11
union type 10
void 10

TypeScript 2.0 1

TypeScript compiler
checker 1
transpiler 1

TypeScript
about 1
adding 242
build tool, configuring 244
configuring 242
example 2, 3
migrating to 241
project, testing 245
refactoring 250
strict checks, enabling 251
transpiling 3, 4
typechecking 4

U

undefined type
 checking 126
utility functions, PacMan
 adding 192
utility functions, Tic-Tac-Toe game
 adding 219

V

variable
 about 144
 null type, checking 126
 undefined type, checking 126
view
 drawing 200, 203
 field, rendering 168
 grid, rendering 166
 popup, displaying 170
 rendering 168
 styles, adding 172
 writing 166

W

weather forecast project
 directory structure 15
 first component, creating 20
 HTML file 18
 setting up 15
 system, building 16, 18
 system, building 17
 TypeScript, configuring 16
websocket server
 API, typing 84
 automatically reconnecting 90
 chat message session, implementing 88
 connecting to 90
 connections 84
 connections, accepting 85
 event handler, writing 93
 event handlers, writing 94
 message, sending 92
 recent messages, storing 86
 session, handling 87
 writing 84

Bibliography

This learning path has been prepared for you to build stunning applications by leveraging the features of TypeScript. It comprises of the following Packt products:

- *Learning TypeScript, Remo H. Jansen*
- *TypeScript Design Patterns, Vilic Vane*
- *TypeScript Blueprints, Ivo Gabe de Wolff*