StayAhead Training

the training specialist

bringing people *and* technology together

Oracle Database	Oracle Development	Oracle Fusion Middleware	Oracle Business Intelligence
Oracle e-Business	MySQL	MariaDB	MongoDB
Unit Testing Frameworks	Maven	Hibernate	Acceptance Testing Frameworks
Java Technology	Web Development HTML & CSS	JavaScript & jQuery	PHP, Python Perl & Ruby
OOAD	Spring	UNIX	Solaris, AIX & HP-UX
Red Hat Linux	SUSE Linux	LPI Linux	Business Analysis ITIL® & PRINCE2®

London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh

scheduled | closed | virtual training

www.stayahead.com +44 (0) 20 7600 6116 sales@stayahead.com

Angular Development

Update 1.0

Angular Development

3 days training

Angular Development Course Overview

The Angular Development course focuses on using modern Angular to develop single-page web applications. This course teaches the use of Angular directives and expressions in HTML5 documents, writing Angular components, filters and services to build complete Single Page Web Applications (SPA).

This course covers Angular from Version 2 upwards, currently delivered using Version 5. TypeScript is used as the primary language for development and the chosen environment for development is Microsoft Visual Studio Code together with Google Chrome.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

The delegate will learn and acquire skills as follows:

- Creating dynamic data-driven HTML5 templates, views and controllers
- Coding loosely-coupled modules, controllers and services with TypeScript
- Creating and utilising models
- Understanding and using Angular Expressions for data binding
- Managing Angular components
- Designing Angular forms
- Using Ajax to retrieve data
- Producing reusable custom Angular directives

Who will the Course Benefit?

The Angular Development course is aimed at front-end developers and engineers using Angular, HTML5 and TypeScript along with modern assistive technologies such as Node.js and Git, to develop quality software. Programmers, Designers, Testers, Quality Analysts and anyone who needs a good understanding of the use of Angular within Web development would also benefit.

Course Objectives

This course aims to provide the delegate with the knowledge to be able to construct an Angular single page web application that neatly separates presentation from business logic concerns and exploits all core elements of the framework including; components, form validation, dependency injection, property binding, event handling, service components, and routing.

Requirements

Substantial prior experience with HTML, CSS, and JavaScript. This knowledge can be obtained by attendance on the pre-requisite HTML & CSS and JavaScript 1 courses.

Pre-Requisite Courses

- HTML & CSS
- JavaScript 1

Follow-On Courses

- HTML5 & CSS3 with JavaScript
- JavaScript 2
- Bootstrap
- PHP Programming

Notes:

- Course technical content is subject to change without notice.
- Course content is structured as sessions, this does not strictly map to course timings. Concepts, content and practicals often span sessions.

Table of Contents

Chapter 1: Angular Precursors

Introduction 1- 3
Single Page Applications 1- 4
Transpilers 1- 5
Polyfills 1- 6
ES6 Features 1- 7
JavaScript Versioning 1- 7
Primitive and Reference Types 1- 9
Primitive Types 1- 9
Reference Types 1- 9
Template Literals 1-10
Destructuring 1-11
Array Destructuring 1-11
Object Destructuring 1-12
Usage 1-12
Decorators 1-13
let, const and var 1-14
Usage 1-14
Arrow Functions 1-15
Arguments 1-16
Using this 1-16
Destructuring Assignment 1-17
Default Parameters and Values 1-17
Understanding Classes 1-18
Spread and Rest Operators 1-19
Spread 1-19
Rest 1-19
Default Parameters and Values 1-20
Export and Imports 1-21
Modules 1-22

Chapter 1: Angular Precursors (continued)

Array Functions 1-23
Array.prototype.map() 1-23
Promises 1-26
Generator Functions 1-28
Sets and Maps 1-29
Functional JavaScript 1-30
First-Class Objects 1-30
Purity 1-30
Immutability 1-31
Currying 1-32
Composition 1-33
TypeScript 1-34
What is TypeScript? 1-34
Types 1-34

Chapter 2: Introducing Angular

Introduction 2- 3
Environment 2- 4
Language 2- 4
Editor 2- 4
Node 2- 4
A Simple Application 2- 5
Setup 2- 5
Add Bootstrap 2- 6
Create a Component 2- 6
Angular Module 2- 7
Hello World Component 2- 8
Directives 2- 8
Project Layout 2- 9
Folder Structure 2- 9
GitHub Files 2- 9
.EDITORCONFIG 2- 9

Chapter 2: Introducing Angular (continued)

angular-cli.json 2-10
package.json 2-10
src Folder 2-12

Chapter 3: Angular Templates

Introduction 3- 3
Interpolation 3- 4
Expressions 3- 4
Reference Other Components 3- 5
Local Variables 3- 5
Property Binding 3- 6
Lifecycle Hooks 3- 7
Event Binding 3- 8
EventEmitter 3- 8
Pipes 3- 9
Chaining Pipes 3- 9
json 3- 9
slice 3- 9
Case Modifiers 3- 9
Title Case 3- 9
number 3-10
percent 3-11
currency 3-11
date 3-11
async 3-12
Custom Pipes 3-13
Built-in Structural Directives 3-14
ngIf 3-14
ngSwitch 3-14
ngStyle 3-15
ngClass 3-15
ngFor 3-16
ngNonBindable 3-16

Chapter 4: Dependency Injection

Introduction 4- 3
What is Dependency Injection? 4- 4
Using Angular DI 4- 6
NgModule 4- 6
@Injectable 4- 7
Provider Registration 4- 7

Chapter 5: Angular Forms

Introduction 5- 3
Form Precursors 5- 4
Template Driven Forms 5- 5
#1 Create Optional Model 5- 6
#2 Create Component 5- 6
#3 Create Form Layout Template 5- 7
#4 Bind Properties with ngModel 5- 8
#5 Add name Attribute to Inputs 5- 8
#6 Add Optional CSS 5- 8
#7 Add Validation Messages 5- 9
#8 Handle Submission 5-10
#9 Disable Submission Until Valid 5-10
Reactive Forms 5-11
#2 Create Component 5-11
#3 Create Form Layout Template 5-12
#4 Bind Properties 5-12
#4 Add Validation 5-13
#5 Add Validation Messages 5-13
#5 Custom Validation 5-14

Chapter 6: Angular HTTP

Introduction 6- 3
Angular 5 Changes 6- 4
Setup 6- 5

Chapter 6: Angular HTTP (continued)

Making Requests 6- 6
HTTP Options 6- 7
Response Formats 6- 7
Headers 6- 7
URL Parameters 6- 7
Rejections and Wrapping 6- 8
Request Manipulation 6-10
Interceptors 6-11
Using Promises 6-12

Chapter 7: Angular Routing

Introduction 7- 3
Modern Client Routing 7- 3
Route Configuration 7- 4
Routes 7- 4
router-outlet 7- 4
routerLink 7- 5
RouterModule 7- 5
Setup Routing 7- 6
A Note On Navigation Paths 7- 7
Routing Errors 7- 7
Routing Redirects 7- 8
Linking Routes 7- 8
Route Parameters 7- 9
Linking to Routes With Parameters 7- 9
Reading Route Parameters 7-10
Optional Parameters 7-11
Reading Route Parameters 7-11
Auxiliary Routes 7-12
Child Routing 7-13
Route Access Control 7-14
Route Guards 7-14
Using CanActivate 7-15

Chapter 7: Angular Routing (continued)

Using CanDeactivate 7-15
Asynchronous Route Guards 7-16

Chapter 8: Observables & Reactive Programming

Introduction 8- 3
Reactive Programming 8- 4
Important Concepts 8- 4
Creating Observables 8- 5
Consuming Observables 8- 5
Differences to Promises 8- 6
Unsubscribing 8- 6
Lazy Nature 8- 7
Multiple Executions 8- 7
Operators 8- 8
Form Events 8- 8
Asynchronous Processing 8- 9
Error Handling in Streams 8- 9
Merging Results 8-10
Parallel Execution 8-10
Polling 8-11
Cold and Hot Observables 8-11
Inter-Component Communications 8-12

Chapter 9: Angular & Redux

Introduction 9- 3
Flux 9- 4
Is Flux Needed? 9- 4
Flux vs MVC 9- 5
Redux 9- 7
Inspired by Flux 9- 8
Three Principles 9- 9
Core Redux 9-10

Chapter 9: Angular & Redux (continued)

Actions 9-10
Action Creators 9-12
Reducers 9-13
The Store 9-15
Redux Data Flow 9-16
Angular & Redux 9-17
Background 9-17
Redux Concepts Recap 9-17
ngrx 9-18
Setup 9-18
Define State 9-19
Actions and ActionCreators 9-19
Create Reducers 9-19
Store Awareness 9-20
Redux DevTools 9-21
Add Redux DevTools 9-21
Redux DevTools Panel 9-22

Chapter 10: Testing Angular Applications

Introduction 10- 3
Using Jasmine 10- 4
Setup & Teardown 10- 5
Using Karma 10- 6
Angular CLI 10- 6
Angular Testing Framework 10- 7
Configure for Testing 10- 7
Angular Test Bed 10- 8
Configure ATB 10- 8
ATB Change Detection 10- 9
Testing Components 10-10
Testing HTTP Services 10-11
Testing Routes 10-12
E2E Testing 10-14

Chapter 11: Angular CLI Reference

Introduction 11-3
Setup 11-3
Create Application 11-4
Create Application in Current Folder 11-4
Compile Project 11-4
Serve Application 11-5
Add Autocomplete 11-5
Search Documentation 11-5
E2E Testing 11-5
Code Formatting 11-5
Code Generation 11-6
Get Configuration 11-7
Set Configuration 11-7
Deploying 11-8
Linting 11-8
Testing 11-8
Show Version 11-9
Add Third Party Libraries 11-9
Retrofitting CLI 11-9
Upgrading Angular CLI 11-9

CHAPTER 1

Angular Precursors

Introduction

Before diving into Angular, an update on the current status of JavaScript and modern web development is in order.

Much has changed.

This chapter cherry picks from the full JavaScript reference documentation which can be found at MDN. It would be pointless to reproduce that documentation here, so please refer to the docs for complete syntax, specification, compatibility and API references.

Single Page Applications

A concept that is often mentioned, but seldom reconciled. SPA applications are simply websites designed to operate as native applications, the code residing in a single web page. Due to SPAs typically requiring more functionality than a normal web page, the framework sits behind the page to attempt to simplify matters.

The Angular libraries typically used in SPAs are:

- Componentisation and Communication

  Separating things makes them more manageable. Once separated, communication between them must still work.

- Binding and Templating

  As the application resides in one page, it will need a templating system for rendering its pages. Likewise, since there is no posting to the server, the application needs a method for listening to the current state of the application: the binding system.

- Routing

  SPAs typically have multiple, discrete, navigable views. A router must capture state and allow for navigation through both the the url, the back and forward buttons.

SPA is applicable when the application is complex. Advantages include:

- Client-side rendering can improve performance
- More flexible UI design. Completely independent of the server-side
- A reduction in application "chattiness". Offline processing can also happen
- State tracking can be improved
- Good for data-driven applications

SPA is not so applicable when the application is fairly simple, or requires more advanced native functionality, such as graphics. Disadvantages include:

- JavaScript must be enabled
- Security means securing API endpoints too, though that should be in place
- SEO is arguable effected
- Complexity is a necessary cost

Transpilers

The only dependencies that are required are the build tools and transpilers that allow Angular to operate in multiple browsers and environments.

ES6 and ES2016 are not yet fully supported by browser vendors. Most of the large vendors have support though backward compatibility is key, and older browsers shouldn't be ignored (to a point). The solution is to use a transpiler to generate compatible code.

Transpilers, or transcompilers to give them their full name, also generate source map files, allowing ES6/7 source to be debugged within the browser.

The two major options for transpilation are:

Traceur

Google transpiler that supports ES6 as well as some experimental ES.next features.

Full details: https://github.com/google/traceur-compiler

Babel

More popular, with ever growing support.

Many frameworks support Babel; converting ES6 into ES5.

Full details: https://babeljs.io/

Online Babel: https://babeljs.io/repl/

Babel remains the better supported option, unless using TypeScript, which has it's own transpiler option.

When using TypeScript, the online tool ES6console can act as the equivalent of the above Babel tool: https://es6console.com/

Polyfills

Used everywhere, but always assumed. A polyfill is code that fills a hole in the browser which shouldn't really exist. Think the DIY crack filler; it serves the same purpose.

Polyfills are a form of regressive enhancement.

For instance, a polyfill (or a shim) can be used to add functionality, such as sessionStorage and canvas support, to browsers that don't support it.

There is a shim/polyfill for sessionStorage available here:
https://gist.github.com/remy/350433

With the polyfill referenced, a developer can use the standard API safe in teh knowledge that the capability is back-filled, and that cross-browser testing can been covered.

There's a list of polyfills available here:

https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills

With Angular, polyfills should not be utilised directly, as this builds in a tight coupling. Instead, the framework makes use of polyfills indirectly, that they may be swapped out as and when they are superseded, or no longer necessary as technology is baked into the browser.

Later sections cover this in more detail.

ES6 Features

A standard ratified in 2015, ECMAScript 2015 (referred to throughout as ES6) is a major update to the JavaScript language. The following section provides a short introduction to the more common features of the specification that will be encountered during Angular development.

JavaScript Versioning

JavaScript naming is confusing. Additions of features includes many proposals, which may or may not appear. These are often referred to as ES7, ES2016, or ES Next features.

ECMAScript:

A language standardized by ECMA International and overseen by the TC39 committee. This term is usually used to refer to the standard itself.

JavaScript

The commonly used name for implementations of the ECMAScript standard. This term isn't tied to a particular version of the ECMAScript standard, and may be used to refer to implementations that implement all or part of any particular ECMASCript edition.

ECMAScript 5 (ES5)

The 5th edition of ECMAScript, standardized in 2009. This standard has been implemented fairly completely in all modern browsers

ECMAScript 6 (ES6)/ ECMAScript 2015 (ES2015)

The 6th edition of ECMAScript, standardized in 2015. This standard has been partially implemented in most modern browsers. To see the state of implementation by different browsers and tools, see the compatibility tables:

http://kangax.github.io/compat-table/es6/

ECMAScript 2016: The 7th edition of ECMAScript, compatability tables:

http://kangax.github.io/compat-table/es2016plus/

ECMAScript Proposals: Proposed features or syntax that are being considered for future versions of the ECMAScript standard. These "next" features are listed:

http://kangax.github.io/compat-table/esnext/

Primitive and Reference Types

While JavaScript is a dynamically typed language, types are assigned. How those variables behave depends upon their type.

Primitive Types

These are copied by value. They are stored on the stack.

- String
- Number
- Boolean
- Undefined
- Null
- ES6 Symbol

To show that a copy of the variable is created on the stack:

```
var one = 1;
var two = 2;
var three = one + two;
console.log(three);          // 3
one = 10;
console.log(three);          // Still 3!
```

Reference Types

These are just references in memory. They are stored on the heap.

- Object
- Array

To prove that a reference is retained during concatenation and other operations:

```
var numbersOne = [[1]];
var numbersTwo = [2, [3]];
var newNumbers = numbersOne.concat(numbersTwo);
console.log(newNumbers);     // [[1], 2, [3]]
numbersOne[0].push(10);
console.log(newNumbers);     // [[1, 10], 2, [3]]
```

Various means of creating copies exist:

Object.assign(): which will not copy references types (shallow copy)

Array.concat(): a new array is created without changes to original arguments

These are useful, though it may be cleaner to use an optimised library to provide immutable data structures. These are covered later under Immutability.

Template Literals

String literals allowing embedded expressions, including multi-line strings and string interpolation features. Known as "template strings" in prior editions of the ES2015 specification e.g.

```
`string text`
// Multiline white space
`string text line 1

 string text line 2`
`string text ${expression} string text`
```

Template literals are enclosed by the back-tick (` `) (grave accent) character instead of double or single quotes.

Previously:

```
let name = 'Greg';
let age = 42;
let welcomeString = 'Hello ' + name + ', you are ' + age + ' and
welcome.';
console.log(welcomeString);
```

Now:

```
let name = 'Greg';
let age = 42;
let welcomeString = `Hello ${name}, you are ${age} and welcome.`;
console.log(welcomeString);
```

Template literals can contain placeholders, as 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.

If there is an expression preceding the template literal (tag here), the template string is called "tagged template literal". In that case, the tag expression (usually a function) gets called with the processed template literal, which you can then manipulate before outputting. To escape a back-tick in a template literal, use backslash before the back-tick.

`\`` === '`' // --> true

Destructuring

Destructuring assignment is the JavaScript expression to unpack variables from arrays or objects: array element or object properties may be extracted and stored in variables.

Whereas normal literal expression allow the creation of ad hoc packages of data e.g.

```
let x = [1, 2, 3, 4, 5];
```

The destructuring assignment allows the left-hand side to define those values to unpack from the sourced variable.

```
let x = [1, 2, 3, 4, 5];
let [y, z] = x;
console.log(y);      // 1
```

Array Destructuring

```
// Basic assignment
let numbers = [1, 2, 3];
let [one, two, three] = numbers;
console.log(one);                          // 1

// Assignment separate from declaration
let greeting, name;
[greeting, name] = ['Hello', 'Greg'];
console.log(greeting);                     // “Hello”

// Default values
let greeting, name;
[greeting='Howdy', name='Greg'] = ['Hello'];
console.log(greeting);                     // “Hello”

// Swap values
let greeting = 'Hello', name='Greg';
[name, greeting] = [greeting, name];
console.log(name);                         // “Hello”

// Parse array from function
function f() {
   return ['Hello', 'Greg'];
}
[greeting, name] = f();
console.log(greeting);                     // “Hello”

// Using spread syntax
let a, b, rest;
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a);                            // 10
console.log(b);                            // 20
console.log(rest);                         // [30, 40, 50]
```

Object Destructuring

```
// Basic assignment
let meaning = {answer: 42, question: 'Ultimate Question'};
let {answer, question} = meaning;
console.log(answer);                    // 42

// Assignment separate from declaration
let answer, question;
({answer, question} = {answer: 42, question: 'Ultimate Question'});
console.log(answer);                    // 42

// Assignment to new variable names
let meaning = {answer: 42, question: 'Ultimate Question'};
let {answer: new_a, question: new_q} = meaning;
console.log(new_a);                     // 42

// Default values
let {answer = 42, question = 'Ultimate Question'} = {answer: 44};
console.log(answer);                    // 44
```

Usage

Examples are not exhaustive: MDN JavaScript Reference has full documentation.

Decorators

Not strictly a ES6 feature, they are due for inclusion in ES7 having been omitted from previous specifications. However, they are extensively used by modern JS frameworks, available in TypeScript, and usable now.

Similar to annotations in Java and C#, they are used to annotate an existing class or method and complete or enhance it. They are really just functions that return functions.

In Angular, decorators are distinguished by a @ prefix, followed by properties e.g.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-guess',
  templateUrl: './guess.component.html',
  styleUrls: ['./guess.component.css']
})
export class GuessComponent implements OnInit {
...
```

Decorators are functions. They apply metadata leveraging the ReflectMetadata library. The following can be decorated:

Class

```
@Injectable()
export class MyService {
  constructor(private http:Http,
            @Inject('config') private config:string) {
  }
...
```

Property Decorators

```
class Test {
   // Decorated property is replaced by decorator value
   @Override('Keith')
   name: string = 'Greg';
}

class Test {
  @ReadOnly       // Property is read only
  name: string;
}
```

As well as **properties**, and **methods.**

Note: The method brackets after @Component indicate that there must be a Component function that returns a function matching one of the decorator signatures. This is an example of the decorator factory pattern.

If decorators still look confusing, perhaps some examples will clear things up.

let, const and var

JavaScript's concept of "hoisting" has always been odd; variables declared with the var modifier are hoisted to a global scope irrespective of where they are defined.

One of the main issues with var was that it could be quietly redefined (as opposed to reassigned) which led to potential difficulty debugging e.g.

```
var myVar = 'Greg';
var myVar = 'Larry';
console.log(myVar);
```

ES6 adds the let keyword, to allow for block-scoped binding constructs. The keyword also does not allow the pain of reassignment.

The const keyword has a similar block scope, and tends to copy the Java style for visual definition e.g.

```
const MAX_VALUE = 100;
```

const is single-assignment. Static restrictions prevent use before assignment.

```
function testFunc() {
   let myvar;
    {
         // Fine, it's block scoped.
         const myvar = "sneaky";
         // Error, just defined with const.
         myvar = "foo";
      }
      // OK, as declared with let.
      myvar = "bar";
      // Error, already declared in this block.
      let myvar = "inner";
   }
}
```

Constants may be initialised with an object, and the content of the object e.g.

```
const PERSON = {};
PERSON.colour = 'blue';          // This works.

const PERSON = [];
PERSON.push({ colour: 'red'});   // Also works.
```

So, both let and const are function-scoped, as with var, and additionally block-scoped.

Usage

Generally, don't use var in ES6. Instead, use let for variable values, and const for constant values. Start with const, and redefine if required: programming defensively.

Yes, var still works, but best to use the above.

Arrow Functions

ES6 arrow functions are a new means of creating functions. A simpler syntax. Originally then:

```
function printName(name) {
    console.log(name);
}
```

, which could also be written in ES5 as:

```
const printName = function(name) {
    console.log(name);
}
```

, becomes the following as an arrow function:

```
const printName = (name) => {
    console.log(name);
}
```

With no arguments, the empty parentheses must be present:

```
const printName = () => {
    console.log('Greg');
}
```

When having just one argument, parentheses can be omitted:

```
const printName = name => {
    console.log(name);
}
```

When just returning a value, a shorter version is possible:

```
const getName = name => name

// Equal to ...
const getName = name => {
    return name;
}
```

Arguments

Argument objects differ in ES6 arrow functions; they are no longer bound.

In ES5, the arguments object was available in functions e.g.

```
const add = function(a, b) {
  console.log(arguments);     // { 0: 10, 1: 10 }
  return a + b;
};
console.log(add(10, 10));     // 20
```

In ES6, this is no longer defined. If required, stick with an ES5 function.

Using this

The 'this' keyword is also no longer bound. With ES5, 'this' gave access to the local context e.g.

```
const user = {
  name: 'Greg',
  cities: ['Sheffield', 'London', 'Southampton'],
  printCities: function() {
    console.log(this.name);
    var that = this;
    this.cities.forEach(function(city) {
      console.log(that.name + ' lived in: ' + city);
    });
  }
}
user.printCities();        // Works
```

When adding a function to an object property, the value of 'this' is bound to that object, however with an anonymous function there is no bound 'this'. Above shows the common workaround: binding 'this' to 'that' instead.

Converting to ES6 means functions use the ' this' value of the context in which they were created, so the code becomes e.g.

```
const user = {
  name: 'Greg',
  cities: ['Sheffield', 'London', 'Southampton'],
  printCities() {
    console.log(this.name);
    this.cities.forEach((city) => {
      console.log(this.name + ' lived in: ' + city);
    });
  }
}
```

Full details at MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Destructuring Assignment

Assigning variables from objects/arrays. Up until recently:

```
var options =
{ api_base_url: 'http://app.co.uk/api/v5', test_mode: true };

var apiBaseUrl = options.api_base_url;

var testMode = options.test_mode;
```

This can now be a little more succinct:

```
let options =
{ api_base_url: 'http://app.co.uk/api/v5', test_mode: true };

let {api_base_url, test_mode} = options;
```

The same with arrays, and nested values.

Default Parameters and Values

Before ES6, default parameters and values were not viable, meaning checking:

```
function createView(title, layout) {
   title = title || 'No Title';
   layout = layout || 'landscape';
}
```

These OR-style checks would return the right value if the left-hand side of the OR is undefined. The function can then be called in a variety of ways:

```
createView('Page Title', 'portrait');

createView('Page Title');

createView(); // Equivalent to "No Title", "landscape"
```

Now, in ES6:

```
function createView(title = 'No Title', layout = 'landscape') {
   return { title: title, layout: layout };
}
```

Understanding Classes

ES6 classes are described as special functions: a syntactic sugar over the existing prototype-based inheritance. Usefully, they can make life a little more simple. Classes also enable the "super" keyword, provide the notion of "instance" and provide static methods and constructors.

Full docs at MDN:
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

To declare a class:

```
class Person {
  constructor (name) {
    this.name = name;
  }
}
const person = new Person('Greg');
console.log(person);          // { name: "Greg" }
```

Here, both a class, and a class property/attribute is defined. The constructor is not required if setting attributes from within the class:

```
class Person {
    name = 'Greg';
}
const person = new Person();
console.log(person);    // { name: "Greg" }
```

Classes may have methods, preferably using ES6 syntax and arrow functions e.g.

```
class Person {
    name = 'Curtis';
    age = 42;
    printSurname () {
        console.log(this.name);
    }
    printAge = () => {
        console.log(this.age);
    }
}
```

Classes also support inheritance, which enables the "super" keyword. See MDN documentation for examples.

Spread and Rest Operators

The spread operator is useful for cloning objects and arrays. Safely copying these references types can be difficult, and spread provides a simpler cloning approach, albeit a shallow clone.

Spread

An operator used to split object properties or array elements e.g.

```
// Arrays
const numbers = [1, 2];
const extendedNumbers = [...numbers, 3, 4];
console.log(extendedNumbers);       // [ 1, 2, 3, 4 ]

// Objects
const person = {
   name:'Greg'
};
const otherPerson = {
   ...person,
   age: 42
}
console.log(otherPerson);           // { age: 42, name: 'Greg' }
```

Rest

Rest allowing the easily handling of a variable number of function parameters. This wwas previously possible using the "arguments" property and doing a little legwork, but can now be simplified e.g.

```
function add() {
  return arguments.reduce((sum, next) => sum + next);
}
console.log(add(1, 2, 3));
// TypeError: arguments.reduce is not a function

// New syntax
function add(...numbersToAdd) {
  return numbersToAdd.reduce((sum, next) => sum + next);
}
console.log(add(1, 2, 3));       // 6
```

Default Parameters and Values

Before ES6, default parameters and values were not viable, meaning checking:

```
function createView(title, layout) {
   title = title || 'No Title';
   layout = layout || 'landscape';
}
```

These OR-style checks would return the right value if the left-hand side of the OR is undefined. The function can then be called in a variety of ways:

```
createView('Page Title', 'portrait');

createView('Page Title');

createView(); // Equivalent to "No Title", "landscape"
```

Now, in ES6:

```
function createView(title = 'No Title', layout = 'landscape') {
   return { title: title, layout: layout };
}
```


Export and Imports

Modern JavaScript and Angular splits code across multiple files, modules, discussed on the next page.

Exporting makes code available, and importing accesses statements.

Exports may be unnamed (the default) e.g.

```
export default ...;
```

or named e.g.

```
export const myInfo = ...;
```

To import a default export e.g.

```
import anyChosenName from './file.js';
```

To import a named export the correct name must be used e.g.

```
import { namedData } from './file.js';
```

A file can only contain one default export, but an unlimited number of named exports.

Importing all named exports is possible e.g.

```
import * as anyChosenName from './file.js';
```

Modules

Organising and loading files in JS has never adhered to a single standard. There is now language-level support for the popular modules: AMD and CommonJS. To review:

Functions and variables can be exported:

```
// lib/math-common.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

Asterisk (wildcard) imports everything with an alias:

```
// app.js
import * as math from "lib/math-common";
console.log("2π = " + math.sum(math.pi, math.pi));
// otherApp.js
import { sum, pi } from "lib/math-common";
console.log("2π = " + sum(pi, pi));
```

Modules themselves may also include exports e.g.

```
// lib/math-extended.js
export * from "lib/math-common";
export var e = 2.71828182846;
export default function(x) {
    return Math.exp(x);
}
// app.js
import exp, { pi, e } from "lib/math-extended";
console.log("e^π = " + exp(pi));
```

If a file exports just a class, use the default keyword. It can then be imported without using braces:

```
export default class NewClass {
}
import NewClass from './new-class';
```


Array Functions

Array Functions are not new, and are not a next-gen feature, but they do pop up everywhere and can look a little strange at first. A few are used commonly in JavaScript frameworks:

- filter()
- reduce()
- map()

It pays to dive a little deeper into one of these to ensure the concepts are clear e.g.

Array.prototype.map()

The map function is used to apply a function on every element in an array, returning a new array.

Full details from MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Syntax

```
let newArr = oldArr.map((val, index, arr) => {
  // return element to new Array
});
```

Where:

- newArr: new array that is returned
- oldArr: array to run the map function on
- val: current value being processed
- index (optional): current index of value being processed
- arr (optional): original array

Example

Simple usage:

```
const numbers = [1, 2, 3, 4, 5];
const tripleNumArray = numbers.map((num) => {
   return num * 3;
});
console.log(numbers);         // [1, 2, 3, 4, 5]
console.log(tripleNumArray);  // [3, 6, 9, 12, 15]
```

Which can be further simplified if applying an existing function:

```
const numbers = [9, 20];
const rootsArray = numbers.map(Math.sqrt);
```

```
console.log(rootsArray);        // [3, 4.47213595499958]
```

Many Angular concepts utilise immutable array manipulation, as a result, other useful functions:

find()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

findIndex()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex

filter()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

reduce()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b

concat()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b

slice()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

splice()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

Promises

Promises are the modern means of asynchronous programming, and are used in many JavaScript libraries. Prior to Promises, callbacks were the norm.

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

The Promise represents the eventual completion (or not) of an asynchronous operation and it's resulting value.

For full details:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Promises may be in one of three states:

- pending: initial state, neither fulfilled or rejected
- fulfilled: operation completed successfully
- rejected: operation failed

As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained:

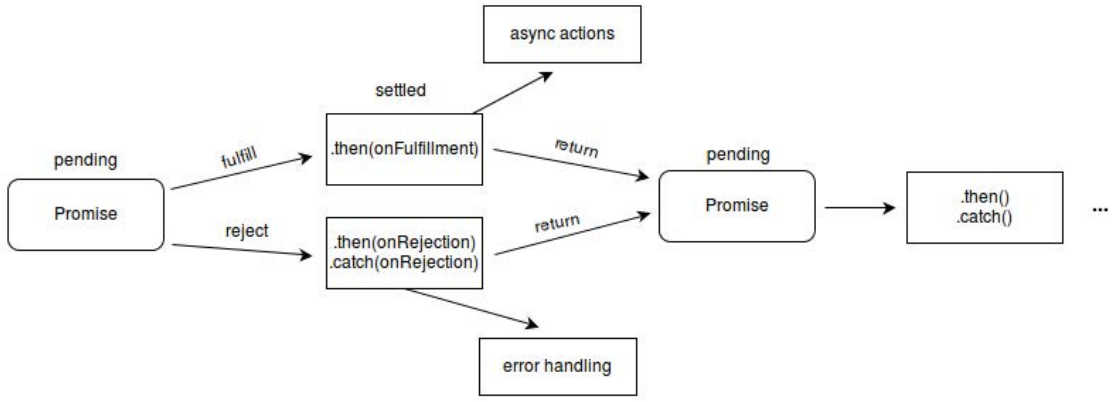


Figure 1: Promise Chaining

Promises can be fairly unwieldy until the user has aclimatised. It pays to break down the process. The following example is from MDN, and shows an image being loaded using Promises and XMLHttpRequest.

The full example, including HTML, is available here:

https://github.com/mdn/js-examples/blob/master/promises-test/index.html


```
function imgLoad(url) {
   // Create new promise with the Promise() constructor.
   // The argument is a function with resolve and reject parameters.
   return new Promise(function(resolve, reject) {
   // XHR to load an image.
   var request = new XMLHttpRequest();
   request.open('GET', url);
   request.responseType = 'blob';
   // When the request loads, check it was successful.
   request.onload = function() {
      if (request.status === 200) {
         // On success, resolve promise by returning request response.
         resolve(request.response);
      } else {
         // On failure, reject promise with error message.
         reject(Error('Image load unsuccessful; error:'
         + request.statusText));
      }
   };
   request.onerror = function() {
      // Deal with entire request failure; likely network error ...
      // ... reject the promise with a message.
      reject(Error('Network error.'));
   };
   // Send the request.
   request.send();
   });
}

// Get a reference to the body element, create a new image object.
var body = document.querySelector('body');
var myImage = new Image();
// Call the function with the URL to load, but then chain the
// promise then() method on to the end of it. This contains two callbacks
imgLoad('myLittleVader.jpg').then(function(response) {
   // The first runs when the promise resolves, with the request.response
   // specified within the resolve() method.
   var imageURL = window.URL.createObjectURL(response);
   myImage.src = imageURL;
   body.appendChild(myImage);
   // The second runs when the promise
   // is rejected, and logs the Error specified with the reject() method.
}, function(Error) {
   console.log(Error);
});
```

Generator Functions

Generators are functions which can be exited and later re-entered. Their context (variable bindings) is saved across re-entrances.

The function* keyword can be used to define a generator function inside an expression.

Parking the MDN definition for now, what is a generator?

- A generator will halt execution of a function for an indefinite period of time when "yield" is called.
- The code that invoked the generator can then control exactly when, or if, the generator resumes.
- Since a value can return every time execution halts ("yield" a value from the function), there are effectively multiple return values from a single generator.

That's the use generator (or iterator): halting function execution and yielding values.

Generators come in two parts, firstly the generator function:

```
function* quickSampleGenerator() {
  var index = 0;
  while (index < index+1)
    yield index++;
}
```

And the code that invokes it:

```
var gen = quickSampleGenerator()

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
```

Generators are a large part of the async aspect of JS, which didn't find it's way in to ES6. In async they are fully wrapped, and so their complexity effectively hidden.

In their fully unwrapped state they don't appear often, though they turn up in middleware, especially Redux Saga.

Sets and Maps

Several more efficient data structures have been added.

Sets will not allow duplicates, whereas Maps will. So, Sets should be used if unique values are to be stored e.g.

```
let s = new Set();
s.add("hi").add("bye").add("hi");
s.size === 2;
s.has("hi") === true;
let m = new Map();
m.set("hi", 42);
m.set(s, 34);
m.get(s) == 34;
```

A WeakMap is a map that doesn't stop keys from being garbage-collected. Keys need to be objects in this case. The API is the same as Map though it's not possible to iterate over a WeakMap's keys or values, or clear it. The same is true for the WeakSet e.g.

```
let wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.size === undefined
let ws = new WeakSet();
ws.add({ data: 42 });
// As the added object has no other references,
// it will not be held in the set.
```

Functional JavaScript

Functional Programming is a declarative paradigm, where side-effects are avoided and data is considered immutable: as a result code can be more maintainable. Using a Functional approach or style in JavaScript can keep things clean.

This is just a snapshot of functional programming to serve as an introduction.

First-Class Objects

Functions are first-class objects in JavaScript. They can be assigned to variables and passed as parameters to other functions.

This gives rise to the concept of **Higher Order Functions**. These are functions that take functions as parameters, other optional parameters, and in turn return a function e.g.

```
const multiply = (x, y) => x * y;
const log = func => (...args) => {
  console.log(...args);
  return func(...args);
}
const logMultiply = log(multiply);
console.log(logMultiply(2, 3));
```

The multiply() is enhanced with log() to logs all the parameters and then executes the original function.

Purity

The concept of a pure function is one that returns a value based on its arguments.

- Pure functions have no side effects. The function does not change anything external to itself
- Pure functions treat their arguments as immutable data

For example, a pure function:

```
const multiply = (x, y) => x * y;
```

An impure function:

```
let x = 0;
const multiply = y => (x = x * y);
```

Modifying global state with each execution is a side effect, hence the impurity.

Immutability

An immutable value is a value that cannot be changed.

Instead of changing a variable, a new variable should be created and returned. This way of working with data is called immutability.

For example:

```
const add5 = arr => arr.push(5)
const theArray = [1, 2]
add5(theArray) // [1, 2, 5]
add5(theArray) // [1, 2, 3, 5]
```

The add3 function doesn't follow immutability because it mutates the value of the given array. Calling the function twice obtains different results.

The function could be altered to use concat, which would return a new array, leaving the original intact e.g.

```
const add5 = arr => arr.concat(5)
const theArray = [1, 2]
const result1 = add5(theArray) // [1, 2, 5]
const result2 = add5(theArray) // [1, 2, 5]
```

After running the function twice, theArray still has it' s original value.

Why is this important?

Performance can improve

Copying sounds intensive, for both memory and computation, but immutable objects never change and can be implemented using a strategy called "structural sharing", which yields dramatically less memory overhead.

There will still be an overhead, but this is outweighed by other benefits. Overall performance may even improve, even if isolated operations become expensive.

Change tracking

Being able to track changes leads to the concept of reversing effects, and application time-travel.

There's more here. Immutability is a central concept in web developnebt, with several approaches and helper libraries available.

Further coverage can be found elsewhere in the manual.

Currying

This is a common technique of functional programming, which is effectively partial evaluation.

A multiple-arg function is converted into a fewer-args function by fixing some arguments for the next invocation.

For example:

```
var whoTalksTo = function(a) {
   return function(b) {
      var result = a + ' talks to '.concat(b);
    return result;
  };
};

// Prefilling the argument can lead to further ‘set’ functions
var gregTalksTo = whoTalksTo('Greg');
var barryTalksTo = whoTalksTo('Barry');

console.log(gregTalksTo('Andrew'));
console.log(barryTalksTo('Andrew'));
```

This is a convenient way of writing functions for reuse.

There are a number of ways this can be useful for async and other types of scenarios.

Composition

Another important concept of functional programming: the combination of functions (and components) to produce new functions with more advanced features and properties.

Composition comes with a strong positive reputation from the OO design world, an often recited mantra is *"favour object composition over class inheritance"* which originated from the GoF Design Patterns publication.

A common mistake in software development overusing class inheritance. Modeling domains using is-a relations leads to tight-coupling, and a variety of associated design issues.

Better alternatives to class inheritance include simple functions, higher order functions, and object composition.

Composition comes in many forms:

- Aggregation
- Concatenation
- Delegation

Discussion of these could be long, and is best left to a more complete source.

For now, function composition is a good start, that is, combining functions to produce a new function e.g.

```
const add = (x, y) => x + y
const square = x => x * x
```

Combining these:

```
const addAndSquare = (x, y) => square(add(x, y))
```

When it comes to Angular, the idea of composition is innate to the composition of components to form the UI.

The principles are the same.

TypeScript

What is TypeScript?

TypeScript is a superset of ES6, available since 2012, which is backwards compatible with ES5 and adds features to that version. Features include:

- Class-based Object Oriented Programming
- Static Type System
- Generics
- Lambdas
- Iterators
- For/Of loops
- Python-style generators
- Reflection

TypeScript looks very much like JavaScript, though files have a .ts extension and is compiled to JavaScript, often at build time, with the use of a TypeScript compiler.

The full TypeScript documentation: https://www.typescriptlang.org/docs goes into detail, but the following covers just a little of the basics.

Types

Many of the same types as JavaScript are available, and in most cases the type are only optional as the TS compiler can infer them from the associated values using type inference e.g.

```
let isComplete: boolean = false;    // Boolean.
```

Numbers are all floating point values e.g.

```
let decimal: number = 10;        // Number.
```

Strings may be single or double quoted, and may also be created using template strings, which can be multiline with embedded expressions; these are created inside backticks, (`).

```
let colour: string = “red”;         // String.

// Template String with expression.
let sentence: string =
`Hi, I’m Greg.
My favourite colour is ${ colour }.`
```


Arrays may be created either similarly to JS, or using a generic array type e.g.

```
let numberList: number[]= [10, 20, 30];
let numberList: Array<number> = [10, 20, 30];
```

Tuples are arrays with known types e.g.

```
let collection: [string, number];      // Declaration.
collection = ["a string in here", 100];   // Initialisation.
```

Enums allow more friendly and understandable names for numeric types. By default the enum members begin numbering at 0, though these can be overridden and set manually e.g.

```
enum Colour { Red, Orange, Yellow, Green };
let c: Colour = Colour.Orange;

// Override.
enum Colour { Red = 1, Orange = 2, Yellow = 6 };
let c: Colour = Colour.Orange;
```

While catching problems are compile time is a bonus, there is also sometimes the need for a variable to have multiple types, for this, the keyword any comes into play e.g.

```
let thisWillChange: any = 100;
thisWillChange = "one hundred";
```

The strangeness of null and undefined continues from JS, in that they are subtypes of all other types, and hance are able to be assigned to the other types.

```
let undef: undefined = undefined;
let nu: null = null;
```

Return types can now be set for functions e.g.

```
function getPerson(): Person {
   return person;
}
```

When returning nothing, void is used e.g.

```
function setPerson(person: Person): void {
   this.person = person;
}
```

In all of the above snippets, the let keyword has been used instead of the traditional JS var keyword. The let keyword should be used instead of var in all cases.

For full detail and documentation: http://www.typescriptlang.org/

CHAPTER 2

Introducing Angular

Introduction

Angular is the rewrite of the popular AngularJS framework.

There have been many changes since AngularJS was launched; JavaScript moved to ECMASCRIPT 6 in 2015, commonly known as ES6. New frameworks have introduced new ideas and ways of working; frameworks such as Vue.js, ReactJS, Aurelia and Polymer.

Common themes among the new approaches is that of the mentioned, ES6, Web Components, and support for mobile applications.

Angular 2 released, and has quickly advanced to Angular 5, though this is marketed as an "invisible makeover" and remains largely backward compatible with Angular 2.x.x applications.

But all this does not answer the question of what Angular actually is. Well, Wikipedia describe Angular as:

"a TypeScript-based open-source front-end web application platform
led by the Angular Team at Google and by a community of individuals
and corporations to address all of the parts of the developer's workflow
while building complex web applications."

Version 5 is absolutely not the end of the story. Google are planning a major release every six months; hopefully keeping the framework fresh and incorporating advances in the ever changing web application landscape.

In this chapter we look at the underpinnings of the framework. While much of the precursor information may be already known, it would be remiss to gloss over the core concepts and technologies that make Angular what it is.

Environment

Language

Angular is built with HTML, CSS, and JavaScript or usually TypeScript.

Editor

Any editor can be used, though supporting TypeScript is a must.

Sublime is a good option, but tooling and linting are available far more readily in other tools, and so it comes to one or two that are a more natural fit.

WebStorm

Provided by JetBrains, this is a licensed product, but feature complete and well supported.

Microsoft Visual Studio Code

This remains free, and is the preferred IDE for Angular development.

As VS Code has evolved quickly alongside Angular, it is feature complete, has good TypeScript support, plus:

- Debugger
- Git integration
- Plugin store
- Angular support
- Supports intellisense

Node

Angular dependencies are best managed by npm, which is of course powered by Node.js.

Download and install Node.js from: https://nodejs.org/en/

Node comes with npm, though npm may be updated itself e.g.

```
npm install npm@latest -g
```

A Simple Application

Setup

Install Node.js as per the website instructions, and check version, which should be 8.1.2 or higher:

```
node -v
```

Check npm version, which should be 5.0.0 or higher:

```
npm -v
```

Install TypeScript:

```
npm install -g typescript
```

Use Chrome as the browser, mostly for the debugger.

The core team of Angular developed a CLI tool for Angular, which bootstrap applications and offers accelerated build and deploy tooling.

Install Angular CLI, should be 1.7.0 or higher, the latest being 1.7.4 at time of writing:

```
npm install -g @angular/cli
npm install --save-dev @angular/cli@latest
ng --version
```

Create a simple project using Angular CLI:

```
ng new hello-world
```

Run the application from the newly created project directory:

```
cd hello-world
ng serve
```

Add Bootstrap

As the project will make use of Bootstrap, at project level, install bootstrap from npm:

```
npm install --save bootstrap
```

, and update angular-cli.json with details of the required CSS file:

```
"styles": [
   "../node_modules/bootstrap/dist/css/bootstrap.min.css",
   "styles.css"
],
```

That's it, Bootstrap is now available for the project.

An alternative is not to serve Bootstrap with the application, but to use it from an external repository instead. Add the following to index.html:

```
<!-- Bootstrap CSS -->
<link
href="//netdna.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
rel="stylesheet" />
```

Create a Component

Using Angular CLI:

```
ng generate component hello-world
```

This will create a new component with the given name in a new directory:

```
installing component
  create src\app\hello-world\hello-world.component.css
  create src\app\hello-world\hello-world.component.html
  create src\app\hello-world\hello-world.spec.ts
  create src\app\hello-world\hello-world.component.ts
  update src\app\app.module.ts
```

Using Angular CLI for this creation also updates the module with details of the new component.

Angular Module

The module is the application's injector and compiler that helps organise components, and their dependencies.

Applications must have at least one module, the root module.

New modules may optionally be introduced for parts of the application.

The class for the module resides in a file called app.module.ts, and is decorated with @NgModule which takes a configuration object.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { HelloWorldComponent } from './hello-world/hello-

@NgModule({
  declarations: [
    AppComponent,
    HelloWorldComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Some points on the class:

- For browser applications, the root module should import BrowserModule.
- Application components are declared in the declarations section.
- The imports sections makes their exported components, directives and properties of other modules available here.
- The providers section is for services that are to be made available to the entire application as singletons. These can be provided at component level for more limited scope, and lazy loading.
- The bootstrap section informs Angular of the root component.

This list is not exhaustive, but there is enough detail for this simple application.

Hello World Component

Components are the building blocks of Angular. Revisiting the component built earlier:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent implements OnInit {

  name: String;

  constructor() {
    this.name = 'Greg';
  }

  ngOnInit() {
  }
}
```

The @Component decorator is passed configuration object, to reference templates for both HTML and CSS, if required.

Adding detail here, such as the name variable, initialised in the constructor, makes the values directly available in the template.

Adding a reference to name in the template, will display the name e.g.

```
<div class="alert alert-success">
    <strong>Success!</strong>
    Hello: {{ name }}
</div>
```

Directives

Making the component variable an array, and building in the constructor e.g.

```
  names: string[];
  constructor() {
    this.names = ['Greg', 'Julia', 'Keith', 'Ted']
  }
```

, lends itself towards the use of directives in the template that have changed somewhat from the AngularJS experience e.g.

```
<ul>
   <li *ngFor="let name of names">
      {{ name }}
   </li>
</ul>
```


Project Layout

Folder Structure

Using Angular CLI creates a fairly complex initial folder structure.

src: The main folder for the application's source code.

node_modules: The folder where libraries are stored.

e2e: The end-to-end testing folder.

GitHub Files

.gitignore

The standard file instructing GitHub and Git which files in a folder should be excluded (or ignored) from Git version control. Specify files or folders not to check into the repository, such as:

- /dist: Ignore files in the distribution folder.
- .map: Don't commit any map files.
- /node_modules: Don't commit the various Angular or other libraries.

Lines beginning with a # are comment lines.

README.md

GitHub projects contain a README.md file that is displayed as part of the GitHub listing. The Angular CLI will include a readme file when creating a new project.

.EDITORCONFIG

Provide configuration settings for the editor, such as indent_style (tab or space) and indent_size (number). Visual Studio Code doesn't natively support .editorconfig, but a plugin can be installed that supports it by pressing Ctrl+Shift+P and entering the following command.

```
ext install .EditorConfig
```

Tip: Trying to create a file beginning with a period, Windows will view it as a file extension and ask for a file name. However, with an additional period at the end as well, Windows will create the file.

angular-cli.json

Configure the behaviour of Angular CLI. There are various sections, such as project, apps, and defaults. The apps group contains most of the options for customisation.

Some settings that may be used:

apps/root: Folder where the root application is, using src.

apps/outDir: Folder to hold the distribution, typically dist.

apps/assets: Files and folders that should be included with distribution.

apps/styles: Collection of style sheets to bundle into the application.

apps/scripts: Collection of JavaScript files to bundle.

package.json

Tells NPM which dependencies are needed. There is no issue in adding more packages than used. Client applications will only get the packages actually used in the code, not the entire list from this file. Since this is a JSON file, the elements described here will be between { } in the actual file.

Package information

Contains information about the package, such as its name and version.

```
"name": "helloworld",
"version": "1.0.0",
"description": "Angular Hello Application",
"author":"Bob Bobson",
```

The name attribute is required, and must be lowercase. The version, description, and author attributes are optional, but should be provided in case another developer wants to find out information about the application.

Scripts

The scripts section is used to provide commands to the NPM command-line application.

```
"scripts": {
"start": "ng serve",
"lint": "tslint \"src/**/*.ts\"",
"test": "ng test",
"pree2   e": "webdriver-manager update ",
"e2e": "protractor"
},
```

Using NPM start to launch the application, actually runs the serve command, which trans-compiles the TypeScript files and opens the server on the local host. Add additional scripts here; compress JavaScript files for instance.

License

The purpose of the package.json file is to provide information about the application, and if the package is open source, specify the license on the package, such as MIT. The license may also be set to UNLICENSED and add the private attribute (set to true). Once complete, packages can be published to the Node repository for others.

```
"license": "UNLICENSED",
"private":true,
```

Dependencies

The dependencies section lists all the modules relied upon, and will be automatically added when someone installs a package e.g.

```
"dependencies": {
  "@angular/animations": "^5.2.0",
  "@angular/common": "^5.2.0",
  "@angular/compiler": "^5.2.0",
  "@angular/core": "^5.2.0",
  "@angular/forms": "^5.2.0",
  "@angular/http": "^5.2.0",
  "@angular/platform-browser": "^5.2.0",
  "@angular/platform-browser-dynamic": "^5.2.0",
  "@angular/router": "^5.2.0",
  "core-js": "^2.4.1",
  "rxjs": "^5.5.6",
  "zone.js": "^0.8.19"
},
```

The install command adds these dependencies to the application folder from the Node repository.

devDependencies

Lists dependencies that are only needed for development and testing. When doing a production release, these dependencies will not be added.

The package.json file interacts with npm to install dependencies.

tsLint.json

Set up the options for the codelyzer lint checker. A lint checker is a nice additional tool (installed as part of Angular CLI), but sometimes can cause unnecessary warnings. These can clutter lint checking. Customise to adapt the lint checker to a preferred coding style.

For example, the any type in TypeScript is allowed, but could be seen as pointless. Control whether to report usage of any with the following entry:

```
"no-any": true
```

Tip: Lint checkers are useful for reducing the likelihood of errors that will compile, but may cause problems at run-time.

src Folder

The src folder contains the configuration file for the TypeScript compiler and the type specifications.

tsConfig.json

The tsConfig.json file contains the compiler options for TypeScript. It allows us to specify how we want the .ts files to be transpiled into JavaScript files. The options we use for our default are

```
{
"compilerOptions": {
"target": "es5",
"module": "commonjs",
"moduleResolution": "node",
"sourceMap": true,
"emitDecoratorMetadata": true,
"experimentalDecorators": true,
"removeComments": false,
"noImplicitAny": false
}
}
```

This file is not required, but included to specify exact behaviour, including ES5 or ES6.

typings.json

Some JavaScript libraries (jQuery) add features and syntax to the JavaScript environment that the TypeScript compiler doesn't know how to handle. When the compiler doesn't know how to handle the syntax, it throws an error.

Library developers might create their own type declaration files (- d.ts extension). These would provide the compiler with information about features in the libraries. The typings.json file tells the compiler where to find the definition files for libraries in use e.g.

```
{
"globalDependencies": {
"core-js": "registry:dt/core-js#0.0.0+20160914114559",
"jasmine": "registry:dt/jasmine#2.5.0+20161003201800",
"node": "registry:dt/node#6.0.0+20161014191813"
}
```

These indicate that particular libraries will be used, so npm should find and install them as part of the build.

To fix the warning message, update the library using the typings install feature. The syntax is as follows.

```
npm run typings -- install dt~jasmine --save --global
```

This command runs the typings application and passes everything after the double dash to the typings program, so it becomes:

```
typings install dt~jsamine --save –global
```

Notice that the typings d.ts file has been updated (see typings\globals\jasmine):

```
"jasmine": "registry:dt/jasmine#2.5.0+20161003201800"
```

This command.

```
npm run typings list38
```

, shows a list of all the typings files (d.ts) installed for the application.

Note: typings.json should not need to be edited manually, rather use the typings tool to update dependencies and version numbers.

CHAPTER 3

Angular Templates

Introduction

Angular is all about displaying data by binding controls in a template to properties of the component.

In order to facilitate this, there are a number of aspects to be aware of:

- Template syntax
- Lifecycle hooks
- Component interactions
- Directives
- Expressions
- Bindings
- Pipes
- Event Binding

, and more.

Interpolation

Interpolation is the simplest form of data binding. Using the double-brace inserts a property value, which updates with changes to the property as a result of 'change detection' e.g.

```
{{ theName }}
```

Expressions

Expressions may also be present within the interpolation braces. Angular evaluates the expression and converts to a string for display.

JavaScript expressions that have or promote side effects are prohibited, including:

- assignments (=, +=, -=, ...)
- new
- chaining expressions with ; or ,
- increment and decrement operators (++ and --)

Other notable differences from JavaScript syntax include:

- no support for the bitwise operators | and &
- new template expression operators, such as |, ?. and !.

Note:

- Expressions must return a value.
- Expressions that update or create variables are not supported.
- Expressions may not reference global variables or objects.
- Expressions should have no side effects (idempotent/nullipotent).
- Expressions should be fast to execute, for a positive user experience.

Reference Other Components

Other components may be reference by their selector e.g.

```
import { Component } from '@angular/core';
import { LoginComponent } from './login/login.component';
import { LogoutComponent } from './logout/logout.component';

@Component({
  selector: 'app-root',
  template: ` <h1>{{title}}</h1>
         <login></login>
                   <logout></logout>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Local Variables

Local variables are variables dynamically declared in the template using the # syntax.

For example, to focus on an element, the focus() method is part of the DOM API, and can be used:

```
<input type="text" #name>
<button (click)="name.focus()">Focus</button>
```

Property Binding

Interpolation is a simple means of using the Angular templating system, which is build on property binding.

Using interpolation:

```
<p>{{ person.name }}</p>
```

, is a simpler way than the underlying:

```
<p [textContent]="person.name"></p>
```

Where textContent is the DOM property.

This becomes more useful for boolean values:

```
<option [selected]="isPersonSelected" value="Greg">Greg</option>
```

, where the property updates with changes to isPersonSelected.

This extends to hidden for hiding an element:

```
<div [hidden]="isHidden">Something here</div>
```

And to nested properties such as:

```
<p [style.color]="foreground">Something here</div>
```

The formula is property="{{ expression }}" e.g. the following are the same:

```
<person name="{{ person.name }}"></person>
<person [name]="person.name"></person>
```

To include a string in the output:

```
<person name="Name {{ person.name }}"></person>
<person [name]="Name ' + person.name"></person>
```

If the value is static, use property="value":

```
<person name="Greg"></person>
```

Expression can also contain function calls:

```
<person name="{{ person.getFullName() }}"></person>
<person [name]="person.getFullName()"></person>
```

Lifecycle Hooks

Angular creates a component, renders it, creates and renders child components, checks for property changes, and destroys it.

Lifecycle hooks allow the developer to act when these events occur.

The lifecycle sequence occurs **after a component or directive constructor** is called:

Lifeycle Hook	Detail
ngOnChanges()	Respond when resetting data-bound input properties.
ngOnInit()	Init component after first displaying data-bound properties.
ngDoCheck()	Detect and act upon changes that Angular does not detect alone.
ngAfterContentInit()	Respond after projecting external content into component view.
ngAfterContentChecked()	Respond after checking content projected into component.
ngAfterViewInit()	Respond after init of component views and child views.
ngAfterViewChecked()	Respond after checking views & child views.
ngOnDestroy()	Cleanup before destroy.

This is not exhaustive, the full detail is available in the docs:
https://angular.io/guide/lifecycle-hooks

Event Binding

Event binding syntax consists of a target event name within parentheses on the left of an equal sign, and a quoted template statement on the right.

The following event binding listens for the button's click events, calling the component's save() method on click:

```
<button (click)="save()">Save</button>
```

The canonical form is also viable:

```
<button on-click="save()">Save</button>
```

The resultant event is available in the method, but including it in the call e.g.

```
<div (click)="onClick($event)">
<button>Go</button>
</div>

onClick(event) {
console.log(event);
// Could alter event lifecycle.
event.preventDefault();
event.stopPropagation();
}
```

If the target event is a native DOM element event, then $event is a DOM event object, with properties such as target and target.value.

<input [value]="person.name"

(input)="person.name=$event.target.value" >

When the user makes changes, the input event is raised, and the binding executes the statement within a context that includes the DOM event object, $event.

EventEmitter

Directives typically raise custom events with an Angular EventEmitter. The directive creates an EventEmitter and exposes it as a property. The directive calls EventEmitter.emit(payload) to fire an event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the $event object.

Pipes

Pipes are a means of transforming, filtering and limiting view output. In AngularJS these were known simply as "filters".

There are a number of build-it pipes; the full list is available in the API docs:

https://angular.io/api?type=pipe

Chaining Pipes

Pipes may be chained e.g.

```
Today is {{ theDay | date | uppercase}}
```

json

Applies json.stringify(). Useful when debugging for checking inside an array for example.

```
<p>{{ data }}</p>
<p>{{ data | json }}</p>
```

slice

As per JavaScript slice, takes two arguments: a start index and optional end index:

```
<p>{{ data | slice:0:3 | json}}</p>
```

Case Modifiers

Transform to upper or lower case:

```
<p>{{ 'The test string' | uppercase }}</p>
```

Title Case

Capitalise the first letter of all words:

```
<p>{{ 'The test string' | titlecase }}</p>
```

number

Formats a number. Takes one string parameter, formatted as:

{integerDigits}.{minFractionDigits}-{maxFractionDigits}, all optional. Each part indicates:

- numbers required in the integer section
- least numbers required in the decimal section
- most numbers required in the decimal section

For example:

```
<p>{{ 54321 }}</p>
<!-- '54321' -->
```

Using the number pipe will group the integer part, even with no digits required:

```
<p>{{ 54321 | number }}</p>
<!-- '54,321' -->
```

The integerDigits parameter will left-pad the integer part with zeros if required:

```
<p>{{ 54321 | number:'6.' }}</p>
<!-- '054,321' -->
```

The minFractionDigits is the min size of the decimal part, so will pad zeros on the right until reached:

```
<p>{{ 54321 | number:'.2' }}</p>
<!-- '54,321.00' -->
```

The maxFractionDigits is the maximum size of the decimal part. You have to specify a minFractionDigits, even at 0, if you want to use it. If the number has more decimals than that, then it is rounded:

```
<p>{{ 12345.13 | number:'.1-1' }}</p>
<!-- '12,345.1' -->
<p>{{ 12345.16 | number:'.1-1' }}</p>
<!-- '12,345.2' -->
```

Note: This pipe relies on the browser Internationalization API, and is not available in every browser. This means using a polyfill in some scenarios.

percent

Allows display of percentages:

```
<p>{{ 0.1 | percent }}</p>
<!-- ‘10%’ -->
<p>{{ 0.1 | percent: ‘.4’ }}</p>
<!--’10.0000%’ -->
```

currency

Requires at least one parameter: the ISO string for the currency, plus an optional boolean to choose to include the symbol or ISO string; defaulting to false. The string may also be formatted, as per number.

```
<p>{{ 12.5 | currency:'EUR' }}</p>
<!-- 'EUR12.50' -->
<p>{{ 12.5 | currency:'USD':true }}</p>
<!-- '$12.50' -->
<p>{{ 12.50 | currency:'USD':true:'.3' }}</p>
<!-- '$12.500' -->
```

date

Formats a date value to a string in the chosen format, using either a Date object or a number of milliseconds. Format may be a pattern 'dd/MM/yyyy', 'MM-yy' or one of a predefined symbolic name e.g. 'short', 'longDate':

```
<p>{{ theDay | date:'dd/MM/yyyy' }}</p><!-- '12/05/1999' -->
<p>{{ theDay | date:'longDate' }}</p><!-- 'July 18, 1926' -->
<p>{{ theDay | date:'HH:mm' }}</p><!-- '12:30' -->
<p>{{ theDay | date:'shortTime' }}</p><!-- '13:30 PM' -->
```

Other date formats:

- medium (Aug 25, 2016, 12:59:08 PM)
- short (8/25/2016, 12:59 PM)
- fullDate (Thursday, August 25, 2016)
- longDate (August 25, 2016)
- mediumDate (Aug 25, 2016)
- shortDate (8/25/2016)
- mediumTime (12:59:08 PM)
- shortTime (12:59 PM)

async

This defines an asynchronous function, returning an AsyncFunction object, allowing the display of data obtained asynchronously. PromisePipe or ObservablePipe may be used depending data comes from a Promise or an Observable.

An empty string is returned until data is available. Once resolved, the resolved value is returned, and change detection triggered.

Usage with Promise

```
<import { Component } from '@angular/core';

@Component({
   selector: 'ns-greeting',
   template: `<div>{{ asyncGreeting | async }}</div>`
})
export class GreetingComponent {

   asyncGreeting = new Promise(resolve => {
      // Resolve after 5s
      window.setTimeout(() => resolve('hello'), 5000);
   });
}
```

Usage with Observable

Here, the pipe unsubscribes automatically when the component is destroyed. Avoid multiple subscriptions multiple promise calls by storing the result e.g.

```
@Component({
   selector: 'ns-user',
   template: `<div *ngIf="asyncUser | async as user">
   {{ user.name }}</div>`
})
export class UserComponent {

   asyncUser = new Promise(resolve => {
      window.setTimeout(() => resolve({ name: ‘Greg’ }), 5000);
   });
}
```


Custom Pipes

Where AngularJS enabled custom filters, Angular allows custom pipes.

Create a class that implements the PipeTransform interface, overriding the transform() method.

Add the @Pipe decorator to register the pipe with a name. To use the pipe in a template, add it to the declarations of @NgModule.

```
import { PipeTransform, Pipe } from '@angular/core';

@Pipe({ name: 'fromNow' })
export class FromNowPipe implements PipeTransform {

   transform(value, args) {
      // do something here
   }
}
```

The Moment.js fromNow function displays elapsed time from a date. Install Moment.js using NPM:

```
npm install moment
```

And add it to SystemJS configuration:

```
map: {
'@angular': 'node_modules/@angular',
'rxjs': 'node_modules/rxjs',
'moment': 'node_modules/moment/moment'
}
```

The end result:

```
import { PipeTransform, Pipe } from '@angular/core';
import * as moment from 'moment';

@Pipe({ name: 'fromNow' })
export class FromNowPipe implements PipeTransform {

   transform(value, args) {
      return moment(value).fromNow();
   }
}
```

To make use of the pipe:

```
<p>Elapsed time: {{ theDay | fromNow }}</p>
```

Built-in Structural Directives

Angular provides a number of built-in directives to provide dynamic behaviour.

ngIf

Display or hide an element based on a condition. The condition is determined by the result of the expression that you pass into the directive.

If the result of the expression returns a false value, the element will be removed from the DOM.

```
<div *ngIf="false"></div> <!-- never displayed -->
<div *ngIf="a > b"></div> <!-- display if a is more than b -->
<div *ngIf="str == 'yes'"></div> <!-- display if str is "yes" -->
<div *ngIf="myFunc()"></div> <!-- display if myFunc returns truthy -->
```

To change the CSS visibility of an element, use either ngStyle or the class directives.

ngSwitch

To render different elements depending on a given condition. Allows a single evaluation of an expression, and then display nested elements based on the value that resulted from that evaluation.

```
<div class="container" [ngSwitch]="myVar">
<div *ngSwitchCase="'A'">Var is A</div>
<div *ngSwitchCase="'B'">Var is B</div>
<div *ngSwitchDefault>Var is something else</div>
</div>
```

ngSwitchDefault is optional.

ngStyle

Set a given DOM element CSS properties from Angular expressions. The simplest way to use this directive is by doing [style.<cssproperty>]="value".

```
<div [style.background-color]="'yellow'">
Uses fixed yellow background
</div>
```

Alternatively, set fixed values using the NgStyle attribute and using key value pairs for each property

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
Uses fixed white text on blue background
</div>
```

For dynamic values e.g. setting font size based on the input value:

```
<div>
   <span [ngStyle]="{color: 'red'}" [style.font-size.px]="fontSize">
      red text
   </span>
</div>
```

ngClass

This allows the setting and change the CSS classes for a given DOM element.

```
.bordered {
   border: 1px dashed black;
   background-color: #eee;
}

<div [ngClass]="{bordered: false}">Not bordered</div>
<div [ngClass]="{bordered: true}">Bordered</div>
```

To make it dynamic we add a variable as the value for the object value

```
<div [ngClass]="{bordered: isBordered}">
   Using object literal. Border {{ isBordered ? "ON" : "OFF" }}
</div>
```

A classesObj object may be defined and used directly:

```
<div [ngClass]="classesObj">
   Using object var. Border {{ classesObj.bordered ? "ON" : "OFF" }}
</div>
```

ngFor

To repeat a given DOM element (or a collection of) and pass an element of the array on each iteration. The syntax is *ngFor="let item of items".

The let item syntax specifies a (template) variable that's receiving each element of the items array

The items is the collection of items from your controller.

```
this.cities = ['Soton', 'Brum', 'London'];

<div *ngFor="let city of cities">
   <div class="item">{{ city }}</div>
</div>
```

ngNonBindable

Used to tell Angular not to compile or bind a particular section of the page.

```
<div>
   <span >{{ content }}</span>
   <span ngNonBindable>
      This is what {{ content }} rendered.
   </span>
</div>
```


CHAPTER 4

Dependency Injection

Introduction

DI was core to AngularJS, and has been built upon for Angular.

Angular improves the prior DI model by unifying the injection systems, as well as fixing tooling, and some other associated bugs.

The saving grace of DI is that it ... can ... aid object lifecycle management. Applications often need to create, manage, and delete associated objects: when there are many objects, lifecycle-management becomes tedious, error-prone, and far easier to push to a DI system.

On the server-side there are several technologies that have made DI ubiquitous: Spring, Google Guice, Dagger and the core JEE/Jakarta EE CDI.

On the client-side, DI has been woven in to modern frameworks, and is often provided via annotation.

What is Dependency Injection?

Dependency Injection is a core feature of Angular. Dependency Injection, or DI, makes dependency management easier, with less coupling between component and simplified unit testing.

DI has been simplified in comparison to AngularJS.

Prior to DI, code has the new keyword littered everywhere, with concerns about the complexity of resultant object maps, as well as instance lifecycle management.

Simple dependencies can be resolved using a standard import:

```
export class FirstClass ... {
}

import { FirstClass } from './first-class'; // Dependency

export class SecondClass ... {
   FirstClass.work();
}
```

Fairly simple, but more complex cases are common. The following class has two dependencies; what happens when once of those classes constructors is updated: the whole codebase needs to be revisited e.g.

```
class Person {
   private address: Address;
   private mobile: Mobile;

   constructor(postcode: string, mobileBrand: string) {
      this.address = new Address(postcode);
      this.mobile = new Mobile(mobileBrand);
   }
}
```

Poor. Also, what about:

- Using a MockFirstClass in a unit test?
- Share an instance of FirstClass across the application?
- Create a new instance each time FirstClass is utilised?

A simpler version would as follows, which means the Person needs to know nothing about how Address and Mobile are constructed:

```
class Person {
   private address: Address;
   private mobile: Mobile;

   constructor(address: Address, mobile: Mobile) {
      this.address = address;
      this.mobile = mobile;
   }
}
```

TypeScript has a very useful shorthand notation for this common setup:

```
class Person {

   constructor(private address: Address,
               private mobile: Mobile) { }
}
```

At this point, a DI framework can make life easier with the concept of an Injector. An Injector has a factory for a collection of objects. Using an Injector would simply the creation of a Person, for example in a unit test, to:

```
const injector = new Injector([Hamburger, Bun, Patty, Toppings]);
const burger = injector.get(Hamburger);

const injector = new Injector([Person, Address, Mobile]);
const person = injector.get(Person);
```

Using Angular DI

Thankfully, in Angular, there is no need to be too worried about the details of injection. The DI system starts with NgModule specification.

NgModule

The Angular DI implementation is mostly covered by @NgModule providers and declarations.

- Providers: declare services
- Providers: declare pipes, components and directives

Here, app.module.ts is declaring a UserService service, as well as LogingComponent and LogoutComponent components e.g.

```
@NgModule({
   declarations: [
      AppComponent,
      LoginComponent,
      LogoutComponent
   ],
imports: [
   BrowserModule,
   FormsModule,
   HttpModule
],
providers: [UserService, HttpService],
bootstrap: [AppComponent]
})
export class AppModule { }
```

The UserService added through providers can then be referenced in a component, typically via the constructor e.g.

```
constructor(private httpService: HttpService) { }
```

, where the methods of the service then becomes available e.g.

```
list() {
   return this.httpService.get('/staffList');
}
```


@Injectable

It could be that the service to be injected and used within the application is a facade for an API service, thus hiding the innards of the HTTP implementation.

In this case, the more app-specific service would inform Angular that it has some of its own dependencies, via the @Injector class decorator e.g.

```
import { Injectable } from '@angular/core';
import { HttpService } from './http.service';

@Injectable()
export class StaffService {

   constructor(private apiService: ApiService) { }

   list() {
      return this.httpService.get('/races');
   }
}
```

Provider Registration

So, service registration is possible via:

- Module
- Component

The choice depends upon the chosen service scope and lifetime.

Module providers are registered with the root injector, and are thus injectable into any class. Once created, a service instance lives for the life of the app and Angular injects this one service instance in every class that needs it.

Services, such as core HttpService or ApiService are likely to be used throughout the application, and are best provided in this application scope.

Alternatively, a component may register a service e.g.

```
@Component({
  selector: 'my-component',
  providers: [ApiService ],
  template: `my-component.html`
})
```

A component-provided service may have a limited lifetime. Each new instance of the component gets its own instance of the service which is destroyed along with the component.

CHAPTER 5

Angular Forms

Introduction

Forms are ubiquitous on the web. There have been several attempts to tidy them up, and replace them with newer constructs, yet they persist in a very similar form (!) to as they were created.

Capturing data from the user can be complex, and making forms easy to use, while being dynamic, can be an art.

Angular adds quite a lot of functionality to form handling, ngModel can be replaced with a completely custom approach where desired.

The two approaches available now are:

- **Template-Driven;** which deals with the majority of form handling logic in the template of the form itself.
- **Reactive Forms;** which deal with the logic in the component class, and deals with user interactions using Observables.

Form Precursors

To use forms, bootstrap the application with the FormsModule and/or ReactiveFormsModule e.g.

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
```

, and add these to the imports section e.g.

```
imports: [
   BrowserModule,
   FormsModule,
   ReactiveFormsModule
],
```

It's also worth noting that Angular is happy with the HTML5 form labeling approach e.g.

```
<label for="username">Username</label>
<input type="text" name="username" id="username">
```

, or without the id attribute e.g.

```
<label>Name<input type="text" name="username"></label>
```


Template Driven Forms

The most straightforward approach to forms in Angular is to make use of the directives that Angular provides for form building, and use the common template syntax (interpolation, expressions and binding) alongside specific form directives.

The steps to create a new form to submit user data is as follows:

1. Optionally create a model class.
2. Create the component that controls the form.
3. Create a template with the initial form layout.
4. Bind data properties to form controls with ngModel two-way data-binding syntax.
5. Add a name attribute to each form-input control.
6. Add custom CSS where required.
7. Show and hide validation-error messages.
8. Handle submission with ngSubmit.
9. Disable submit button until the form is valid.

The following pages will follow this forms recipe and create a useful form.

#1 Create Optional Model

An optional model can be created, perhaps to provide part of the template for the form's initial state e.g.

```
export class User {

    constructor(
        public id: number,
        public username: string,
        public email: string,
    ) {}
}
```

Instantiate the POJO (Plain Ordinary JavaScript Object) in the component e.g.

```
user = new User(111, 'gcurtis', 'greg@greg.com');
```

#2 Create Component

The component holds the POJO template instance, submission logic, and anything else required for the form e.g.

```
export class UserFormComponent implements OnInit {

  user = new User(111, 'gcurtis', 'greg@greg.com');
  submitted = false;
  questions = ['Favourite football team?', 'Name of first pet?',
'City where you were born?'];
  defaultQuestion = 'Favourite football team?';
  answer = '';

  constructor() { }

  ngOnInit() {}

  onSubmit() {
    this.submitted = true;
  }

  get diagnostic() {
    return JSON.stringify(this.user);
  }
}
```

#3 Create Form Layout Template

The template has the bulk of the form logic e.g.

```
<div class="form-group">
  <label for="username">Username</label>
  <input type="text"
         class="form-control"
         id="username"
         required
         [(ngModel)]="user.username" name="username"
         #spy>
         <br> {{spy.className}}
</div>
```

Where:

required – standard HTML5 validation control

[(ngModel)] – an optional two-way binding to populate the form with initial data from the optional POJO

#spy – adds a template reference variable to provide live visual output of the Angular-provided classes

form-control – a cosmetic Bootstrap addition.

Other form template additions can inspect the component for details, and use template syntax to bulk-create and populate the form e.g.

```
<div class="form-group">
  <label for="secret">Questions</label>
  <select
    id="questions"
    class="form-control"
    [ngModel]="defaultQuestion"
    name="questions">
    <option *ngFor="let q of questions" [value]="q">{{ q }}</option>
  </select>
</div>
```

Additionally, the form requires the declaration of the template variable e.g.

```
<form #f="ngForm">
```

The variable f is now a reference to the NgForm directive that governs the form. This is added automatically by Angular.

The NgForm directive supplements the form element with additional features. It monitors properties, including validity. It also has its own valid property which is true only if every contained control is valid.

#4 Bind Properties with ngModel

As can be seen above, some ngModel bindings can be one-way, and others two-way, depending on the requirement.

Add two-way binding to populate the template from component state, and update that state dynamically e.g.

```
[(ngModel)]="user.username" name="username"
```

And use one-way binding to simply add detail to the form from the component only e.g.

```
[ngModel]="defaultQuestion"
```

#5 Add name Attribute to Inputs

A name attribute is added to the <input> tag and set to "email". Any unique value is viable, but being descriptive is helpful e.g.

```
<div class="form-group">
  <label for="email">Email</label>
  <input type="text"
         class="form-control"
         id="email"
         [(ngModel)]="user.email" name="email">
</div>
```

Defining a name attribute is required when using [(ngModel)] in a form.

Internally, Angular creates FormControl instances and registers them with an NgForm directive that is attached to the <form> tag. Each FormControl is registered under the name assigned to the name attribute.

#6 Add Optional CSS

Custom CSS can be added to the relevant template css file. This can provide visual clues as to the validity of that input e.g.

```
.ng-valid[required], .ng-valid.required  {
    border-left: 5px solid #42A948;
}

.ng-invalid:not(form)  {
    border-left: 5px solid #a94442;
}
```


#7 Add Validation Messages

As #f is available via f.value, and controls via the .controls property, erros are available on properties by inspecting the property e.g.

```
<form novalidate #f="ngForm">
  {{ f.controls.username?.errors | json }}
</form>
```

By using ?.errors, called the Safe navigation operator, this ensures that the property does not fail is it does not yet exist, is null, or undefined.

As username is required, this will display a JSON string e.g.

```
{ "required": true }
```

This can be displayed more usefully e.g.

```
<div *ngIf="f.controls.username?.required" class="error">
   Userame is required
</div>
```

As this may not be needed at fist load, touched can be added e.g.

```
<div *ngIf="f.controls.username?.errors &&
      f.controls.username?.touched" class="error">
   Username is required
</div>
```

Using ngModel also provides information via CSS classes; these update dynamically to reflect state.

State	Class (if true)	Class (if false)
The control has been visited	ng-touched	ng-untouched
The control's value has been changed	ng-dirty	ng-pristine
The control's value is valid	ng-valid	ng-invalid

A template reference variable named spy can be added to the chosen <input> to display the input's CSS classes e.g.

```
<input type="text"
     class="form-control"
     id="username"
     required
     [(ngModel)]="user.username" name="username"
     #spy>
     <br> {{spy.className}}
```

#8 Handle Submission

The standard submit button triggers a form submit due to the type property.

The form's ngSubmit event property can be bound to the form component's onSubmit() method:

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

As the onSubmit function is in the component, it has access to current form values, and so can do anything with them at this stage e.g.

```
onSubmit() {
   this.submitted = true;
   console.log(this.submitted);
   console.log(this.user.username);
   // Call service ...
}
```

#9 Disable Submission Until Valid

To disable the submit, bind to the disabled property of the submit button, and setting it to true dynamically when f.invalid is true.

When the form is valid, the submit allows submission.

```
<form novalidate (ngSubmit)="onSubmit()" #f="ngForm">
  ...
  <button type="submit" class="btn btn-success"
       [disabled]="f.invalid">Submit</button></form>
```


Reactive Forms

These are the newer functional Reactive means of building forms.

To recap setup, include the ReactiveFormsModule:

```
import {ReactiveFormsModule} from "@angular/forms";

@Component({...})
export class App { }

@NgModule({
    declarations: [App],
    imports: [BrowserModule, ReactiveFormsModule],
    bootstrap: [App]
})
```

#2 Create Component

The component holds the POJO template instance, submission logic, and anything else required for the form e.g.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, FormBuilder }
    from '@angular/forms';

@Component({
    selector: 'app-user-form',
    templateUrl: './user-form.component.html',
    styleUrls: ['./user-form.component.css'],
})
export class UserForm {
    form: FormGroup;

    username = new FormControl("", Validators.required);
    password = new FormControl("", Validators.minLength(8));

    constructor(fb: FormBuilder) {
        this.form = fb.group({
            "username": this.username,
            "password": this.password
        });

    }
    onSubmit() {
        console.log(this.form);
    }
}
```

Some fundamental form classes are used here, namely:

Class	Description
FormGroup	FormGroup tracks the value and validity state of a group of AbstractControl instances. The group's properties include its child controls. The top-level form in your component is a FormGroup.
FormControl	FormControl tracks the value and validity status of an individual form control. It corresponds to an HTML form control such as an <input> or <select>.
FormBuilder	Helper class to reduce repetition and clutter by handling details of control creation.
Validators	Provides simple, composable functions to aid in validation.

#3 Create Form Layout Template

The template is somewhat simplified e.g.

```
<form [formGroup]="form" (ngSubmit)="submit()">
    <label for="username">username</label>
    <input type="text" name="username" id="username"
      [formControl]="username">
    <br>

    <label for="password">password</label>
    <input type="password" name="password" id="password"
       [formControl]="password">
    <br>
    <button type="submit">Submit</button>
</form>
```

#4 Bind Properties

Above, the inputs to be associated to the FormControl in the class are linked, and formGroup is added. This takes an existing FormGroup instance and associates it with an HTML element.

Elements of the form are accessible via the FormControl with dot notation e.g.

```
console.log(this.password.valid);   // Boolean output
```


#4 Add Validation

As can be seen previously, the Validators are added to the FormControls e.g.

```
username = new FormControl("", Validators.required);
password = new FormControl("", Validators.minLength(8));
```

, and are also available in the template e.g.

```
{{ username.value | json }
<label for="username">username</label>
<input type="text" name="username" id="username"
   [formControl]="username">
<br>

{{ password.valid }}
<label for="password">password</label>
<input type="password" name="password" id="password"
   formControl]="password">
<br>
```

Multiple validation is added as an array e.g.

```
password = new FormControl("", [
    Validators.required,
    Validators.minLength(8)
]);
```

#5 Add Validation Messages

The same properties are available as with Template forms, so tailoring the is very similar e.g.

```
<div [hidden]="username.valid || username.untouched">
   <div [hidden]="!username.hasError('required')">
      Username is required.
   </div>
</div>

<div [hidden]="password.valid || password.untouched">
    <div>
        Problems with password:
    </div>
    <div [hidden]="!password.hasError('required')">
        The password is required.
    </div>
    <div [hidden]="!password.hasError('minlength')">
        Password must be longer than 8 characters.
    </div>
</div>
```

#5 Custom Validation

Custom validation can be added with very little effort.

Assuming that the password cannot contain special characters e.g.

```
hasSpecialChars = (input: FormControl) => {
    const hasExclam = input.value.includes('!');
    const hasHash = input.value.includes('#');
    if (hasExclam || hasHash) {
        return { validPass: true }
    } else {
        return null;
    }
}

password = new FormControl("", [
    Validators.required,
    Validators.minLength(8),
    this.hasSpecialChars
]);
```

The view can then check for the presence and value of the resulting JSON e.g.

```
<div [hidden]="!password.hasError('validPass')">
   Your password must not have special characters.
</div>
```


CHAPTER 6

Angular HTTP

Introduction

Modern JavaScript applications support both the classic XMLHttpRequest interface, and the more modern fetch() API.

Angular has a provided modules for HTTP; @angular/http and @angular/common/http, that build on XMLHttpRequest. As calls using these modules are asynchronous, a number of approaches are possible; including using:

- Callbacks
- Promises
- Observables

Observables are the preferred approach here due to their better better error handling; these are covered from the ground up in the appropriate manual section.

Angular 5 Changes

The @angular/http module was officially deprecated and replaced by @angular/common/http, introduced in version 4.3. It seems likely that @angular/http will be removed in a future release. The new version is smaller, simpler, and recommended for new applications.

HttpClient has been improved; object literals may be directly used as headers or parameters, previously it was the classes HttpHeaders and HttpParams.

The previous header construction that was necessary:

```
const headers = new HttpHeaders().set('Authorization', 'secret');
const params = new HttpParams().set('page', '1');
return this.http.get('/api/users', { headers, params });
```

, has been simplified:

```
const headers = { 'Authorization': 'secret' };
const params = { 'page': '1' };
return this.http.get('/api/users', { headers, params });
```

To upgrade:

- replace HttpModule with HttpClientModule from @angular/common/http
- inject the HttpClient service
- remove all now superfluous map(res => res.json()) calls

Http worked with Observable<Response> by default and had to convert response bodies using:

```
http.get('/api').map(res => res.json());
```

HttpClient does response body conversion and, by default, assumes a JSON response returning an Observable<Object>.

Essentially, previous calls that mapped results to JSON can be removed.

Some classes are renamed:

- HttpResponse<T>
- HttpRequest<T>
- HttpHeaders
- HttpParams

When sending a request, declare the expected response type (arraybuffer, blob, text, json) and the type of the response body will be either an ArrayBuffer or Blob or string.

For JSON responses, either use a generic Object or pass an interface describing the structure of the JSON document. An HttpResponse<MyInterface> or Observable<MyInterface> is then worked with.

Setup

Import the chosen module, which supports both XHR and JSONP requests through the HttpModule and JsonpModule.

Import in app.module.ts:

```
import { HttpModule } from '@angular/http';
```

or,

```
import { HttpClientModule } from '@angular/common/http';
```

Add to the import list:

```
imports: [
  BrowserModule,
  FormsModule,
  HttpModule,
  HttpClientModule
],
```

Inject service into chosen component:

```
class MyComponent {

   constructor(public httpClient: HttpClient) {
   }

   doGet(): void {
      // Use service.
   }
}
```

Making Requests

Use the get() method of HttpClient to make requests:

```
@Component(...)
export class MyComponent implements OnInit {

  results: string[];

  // Inject HttpClient into your component or service.
  constructor(private http: HttpClient) {}

  ngOnInit(): void {
    // Make the HTTP request:
    this.http.get('/api/items').subscribe(data => {
      // Read the result field from the JSON response.
      this.results = data['results'];
    });
  }
}
```

Often GET requests are exposed as observables. The service provides a string response, which must then be consumed and converted as appropriate.

Here, the code is using the Http service indirectly, via the discogs service, and subscribing to the :

```
search(): void {
  if (!this.query) {
    return;
  }
  this.discogs
    .searchTrack(this.query)
    .subscribe((res: any) => this.renderResults(res));
}
```

HTTP Options

A range of options, parameters, headers and formats are available.

Response Formats

As of 4.3, the default response format is JSON, with no requirement to parse the response manually.

Although JSON is the most common response format, there is obviously need for other types. Define the 'responseType' property of the options object to prevent the JSON default:

```
{ responseType: 'text' }
```

Headers

To add headers to the request, use the headers property of the options object. A HttpHeaders object contains header definitions (the Headers object when using the older Http client).

```
const headers = new HttpHeaders({ 'Content-Type': 'application/json'
});
const options = { headers: headers };
```

URL Parameters

Define url parameters inside the options object with a HttpParams object:

```
const params = new HttpParams().set('id', '1');
const options = { params: params };
```

Rejections and Wrapping

Using the subscriber's error and complete callbacks, it is simple to deal with potential issues:

```
onSubmit(username: string, email: string) {
  this.httpService.sendData({ username: username, email: email })
    .subscribe(
      data => console.log(data),
      (err) => console.log(err),
      () => console.log('Submission Complete')
    );
}
```

Using the optional catch operator allows for more control. Catch errors, possibly do something, and pass the exception on perhaps.

Here, the service is designed to return a 500 error, which is then wrapped and dealt with in a controlled fashion in the component:

```
// Service.
getData() {
  return
    this.http.get('http://www.mocky.io/v2/5a26d0e43000007d2a0e895c')
    .map((response: Response) => response.json())
    .catch((e) => {
      return Observable.throw(
        new Error(`${ e.status } ${ e.statusText }`)
      );
    });
}
```

```
// Component.
ngOnInit() {
  this.httpService.getData()
    .subscribe(
      (data: any) => console.log('GET: ' + data),
      err => {
        this.errorMessage = err.message;
        console.log(this.errorMessage);
      },
      () => { console.log('Completed'); }
    );
}
```

Note: mocky.io can be used to generate URLs with desired responses, including error codes.

An alternative could have been to provide a pre-built response, perhaps from a cache:

```
getData() {
  return
   this.http.get('http://www.mocky.io/v2/5a26d0e43000007d2a0e895c')
    .map((response: Response) => response.json())
    .catch(e => {
      if (e.status >=  500) {
        return this.cachedVersion();
      } else {
        return Observable.throw(
          new Error(`${ e.status } ${ e.statusText }`)
        );
      }
    });
}
```

Request Manipulation

Cancellation may be required after a timeout, or if a priority request renders another pointless. To cancel, use the unsubscribe function of the subscription. Here, the request is cancelled if it takes longer than 2 seconds to complete:

```
ngOnInit() {
  const request = this.httpService.getData()
    .subscribe(
      // 3 Callbacks.
      (data: any) => console.log('GET: ' + data),
      err => {
        this.errorMessage = err.message;
        console.log(this.errorMessage);
      },
      () => {
        console.log('Completed');
      }
    );

    setTimeout(() => {
      request.unsubscribe();
      this.errorMessage = 'Request cancelled';
    }, 2000)
}
```

As well as cancellation, RxJS has a retry operator which can either retry continually, or with a retryCount:

```
getData() {
  return this.http.get('https://angular-
bdbaa.firebaseio.com/title.json')
    .map((response: Response) => response.json())
    .retry(3);
}
```


Interceptors

Interceptors are able to modify the request before it hits the server, or modify the response. This approach may be used for authentication; modifying the authorization header.

As with a service, interceptors are injectable, and as there may be several of them, they are chained; the first called by Angular, and each then invoked by the previous:

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>,
next: HttpHandler): Observable<HttpEvent<any>> {
req.headers.append('Authorization', '<EXAMPLE-TOKEN>')
         return next.handle(req);
   }
}
```

Interceptors are provided, also much like services:

```
...
providers: [
DISCOGS_PROVIDERS,
{provide: APP_BASE_HREF, useValue: '/'},
{provide: LocationStrategy, useClass: HashLocationStrategy},
{provide: HTTP_INTERCEPTORS, useClass: Interceptor, multi: true }
]
 ...
```

Using Promises

Promises are still a viable option for Http, however they do not allow for request cancellation or the chaining of RxJS operators, so are deemed less useful than Observables.

Here, the RxJS toPromise operator converts the observable to a promise:

```
getArtist() {
  return this.http.get('https://api.discogs.com/artists/41')
    .map((response) => response.json())
    .toPromise();
}
```

, to be consumed in the component:

```
ngOnInit() {
  this.musicService.getArtist()
    .then(result => {
      console.log(result.name);
    })
    .catch((error) => console.error(error));
}
```

CHAPTER 7

Angular Routing

Introduction

Routing is the approach of separating application locations within a SPA. Even in a SPA, separation of concerns is needed: some parts of the application may be only applicable in certain situations, or be accessed only by certain users.

So, the main reasons for separation:

- Logical separation of concerns:

  Dealing with users, should be separate to dealing with vehicles.

- State can be maintained throughout the application:

  If a user has logged in, components can be made aware of their logged in condition.

- Protection of location:

  Some components may be inaccessible, or even unloaded, dependent upon user status or some other logic.

Locations are also useful for helping bookmarking to work, for saving/sharing locations and for allowing the browser navigation buttons to operate smoothly.

Modern Client Routing

Client-side routing was upgraded with HTML5, which did away with the older hashState approach (though this is still available).

Browsers can now create new browser history objects without making new requests; effectively baking AJAX into navigation at a low level.

The history.pushState method exposes navigational history to JavaScript, allowing frameworks to manipulate history without reloading. This was available in AngularJS, but is the default in Angular.

Hash-based routing, where the # character appears in the URL, is still available as a fall back.

Route Configuration

The main routing components provided by Angular are:

Routes

An array of routes to define routing for the application. Sets up expected paths, components to be used how they are to be interpreted.

Common route attributes:

path – browser URL when application is at the route.

component - component to be rendered at this route.

redirectTo - redirect route; routes can have either component or redirect attribute defined.

pathMatch – optional. Defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set to 'full', otherwise it will match all paths.

children - array of route definitions for child routes.

Create routes as an array of route configurations e.g.

```
const myRoutes: Routes = [
  { path: 'page-one', component: PageOneComponent },
  { path: 'page-two', component: PageTwoComponent }
];
```

router-outlet

This acts as a placeholder for Angular to fill dynamically based on the current route state e.g.

```
<router-outlet></router-outlet>
<router-outlet name='left-col'></router-outlet>
<router-outlet name='right-col'></router-outlet>
```


routerLink

This allows specific parts of the application to be linked e.g.

When using the following route configuration:

```
[{ path: 'user/:id, component: UserComponent }]
```

When linking to this user/:id route, a hardcoded use would be:

```
<a routerLink='/user/10>link to user with id = 10</a>
```

RouterModule

Using RouterModule.forRoot takes the Routes array as an argument and returns a configured router module.

It is possible to split the router from the module definition e.g.

```
import { RouterModule, Routes } from '@angular/router';

const myRoutes: Routes = [
  { path: 'page-one', component: PageOneComponent },
  { path: 'page-two', component: PageTwoComponent }
];

export const routing = RouterModule.forRoot(routes);
```

And import to the module e.g.

```
import { routing } from './app.routes';

@NgModule({
  imports: [
    BrowserModule,
    routing
  ],
  …
```

Setup Routing

The precursor to routing in Angular is to have a base path set in the head tag of index.html e.g.

```
<base href="/">
```

The steps required to get routing working are as follows:

1. Import routing components in the module definition:

```
import { Routes, RouterModule } from '@angular/router';
```

2. Define application routes:

```
const myRoutes: Routes = [
   { path: '', component: HomeComponent },
   { path: 'sales', component: SalesComponent },
   { path: 'hr', component: HRComponent },
];
```

Where:

- Path indicates the URL this route will handle
- Component ties a route to a component
- An optional redirectTo redirects a given path to an existing route

3. Provide the routes to the application:

```
imports: [
   RouterModule.forRoot(myRoutes)
]
```

4. Output the router in a chosen template:

```
<router-outlet>
```

5. Create template links to allow user-controlled navigation to routes:

```
<a routerLink="/sales">Sales</a>
<a [routerLink]="['/sales']">Sales</a> // Allows more detail.
```

The routerLink directive is used to link to routes with no page reload.

The first link is simply a string, so can be used without property binding syntax, though the second uses a more specific syntax; this can build links from separate parts, to allow for more dynamic link building e.g.

```
<a [routerLink]="['/users', user.id]">{{ user.id }}</a>
```

Pushing to routes is also achievable programmatically, covered later in this section.

A Note On Navigation Paths

Absolute paths are used unless specified:

```
/users (/ is always appended to root domain)
```

Relative paths may be used:

```
No slash or ./ (add to currently loaded path)
../ or ../../ etc to go back from currently loaded path
```

Routing Errors

Cater for unknown routes by adding a component that is presented upon error. This component can then be declared in the router, and passed a data object which contains detail interpretable by that component e.g.

```
{ path: 'page-not-found', component: ErrorComponent,
data: {message: '404. Page not found.'} },
```

The ErrorComponent can then access the data object using the ActivatedRoute e.g.

```
import { ActivatedRoute, Data } from '@angular/router';

export class ErrorComponent implements OnInit {
  errorMessage: string;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    // Either access via route snapshot (one time) …
    this.errorMessage = this.route.snapshot.data['message'];
    // Or use the observable for continuous monitoring.
    this.route.data.subscribe(
      (data: Data) => {
        this.errorMessage = data['message'];
      }
   …
```

Routing Redirects

At startup, the application navigates to the empty route. This can be altered to a named route instead e.g.

```
export const myRoutes: Routes = [
  { path: '', redirectTo: 'page-one', pathMatch: 'full' },
  { path: 'page-one', component: PageOneComponent },
  { path: 'page-two', component: PageTwoComponent }
];
```

The pathMatch property is required for redirects. Here, using pathMatch: full means the router will redirect if the entire URL matches the empty path.

Linking Routes

Routes may be linked both declaratively and programmatically.

The RouterLink has already been mentioned. Add links to routes using the RouterLink directive e.g.

```
<a routerLink="/page-one">Page One</a>
```

In order to navigate programmatically, use the navigate function provided by the router e.g.

```
constructor(private route: ActivatedRoute,
   private router: Router) {}

navigate() {
   this.router.navigate(['page-two'],
      {relativeTo: this.route, queryParamsHandling: 'preserve'});
}
```

Route Parameters

In many situations, the user will drive which component they wish to view.

If the user is given a list of people, they want to click on a person and display a page with details on that person; this requires the use of route parameters e.g.

```
export const myRoutes: Routes = [
  { path: '', redirectTo: 'users', pathMatch: 'full' },
  { path: 'users', component: UsersComponent },
  { path: 'user/:id', component: UserComponent }
];
```

The id places the parameter in the path.

Linking to Routes With Parameters

The UsersComponent might display a list of users, each with a link to the user route with the appropriate id e.g.

```
<a *ngFor="let user of users"
  [routerLink]="['/user', user.id]">
  {{ user.name }}
</a>
```

routerLink uses an array to specify the path and the route parameter. This can also be achieved programmatically:

```
goToUser(id) {
  this.router.navigate(['/user', id]);
}
```

Reading Route Parameters

The UserComponent will read the parameter, loading the user based on the provided id. The ActivatedRoute service provides a params Observable for access to route parameters e.g.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Subscription } from 'rxjs/Subscription';

@Component({
  selector: 'user',
  template: `
    <div>Details for user: {{id}}</div>
  `,
})
export class UserComponent implements OnInit, OnDestroy {
  id: number;
  mySubscription: Subscription;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this. mySubscription = this.route.params.subscribe(params => {
       this.id = +params['id'];
    });
  }

  ngOnDestroy() {
    this. mySubscription.unsubscribe();
  }
}
```

Using the observable rather than the snapshot is a better choice, as the router will likely not recreate the component when navigating from itself.

Optional Parameters

Query parameters can optionally be used for optional parameters to a route.

Use the [queryParams] directive along with [routerLink] to pass query parameters e.g.

```
<a [routerLink]="['products']" [queryParams]="{ order: 'common' }">
```

Go to Common Products

The same is viable programmatically e.g.

```
goToCommonProducts(ordering) {
  this.router.navigate(['/products'],
      { queryParams: { order: ordering } });
}
```

Reading Route Parameters

Reading query parameters is possible either via the snapshot of ActivatedRoute, or using an Observable e.g.

```
paramsSubscription: Subscription;

constructor(private route: ActivatedRoute) { }

ngOnInit() {
  this.user = {
    id: this.route.snapshot.params['id'],
    name: this.route.snapshot.params['name']
  };
  this.paramsSubscription = this.route.params
    .subscribe(
      (params: Params) => {
        this.user.id = params['id'];
        this.user.name = params['name'];
      }
    );
```

Note the subscription should also be explicitly unsubscribed to prevent memory leaks.

Auxiliary Routes

These allow multiple independent routes in a single application. Component have one primary route and zero or more auxiliary ones. Auxiliary routes must have unique name within a component e.g.

```
<nav>
  <a [routerLink]="['/page-one']">Page One</a>
  <a [routerLink]="['/page-two']">Page Two</a>
  <a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">
   Component Aux</a>
</nav>
<div >Outlet:</div>
<div>
  <router-outlet></router-outlet>
</div>

<div>Sidebar</div>
<div>
  <router-outlet name="sidebar"></router-outlet>
</div>
```


Child Routing

It is often appropriate to view routes from within other routes, without navigating away. The standard Master-Child view would require this.

If user address details were to be visible in the same view as user:

```
export const myRoutes: Routes = [
  { path: '', redirectTo: 'users', pathMatch: 'full' },
  { path: 'users', component: UsersComponent },
  { path: 'user/:id', component: UserComponent,
    children: [
      { path: '', redirectTo: 'details', pathMatch: 'full' },
      { path: 'address', component: AddressComponent },
      { path: 'vehicle', component: VehicleComponent }
    ]
  }
];
```

These can then be displayed with another router outlet inside the UsersComponent e.g.

```
<a
[routerLink]="['/users', user.id]"
href="#"
class="list-group-item"
*ngFor="let user of users">
{{ user.id }}
</a>
…
<router-outlet></router-outlet>
```

Route Access Control

An element of security can be added to route navigation, allowing access or not to chosen endpoints.

Routes may only be accessible only with a token, or with a login, and exiting a route can also be controlled, in order to not lose unsaved information for example.

Route Guards

A guard must be registered with chosen routes.

To guard with a login, as well as for potential unsaved info, a route could be configured as follows:

```
import { Routes, RouterModule } from '@angular/router';
import { UserComponent } from './user-component;
import { LoginRouteGuard } from './login-route-guard';
import { SaveContentGuard } from './save-content-guard';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  {
    path: 'user',
    component: UserComponent,
    canActivate: [LoginRouteGuard],
    canDeactivate: [SaveContentGuard]
  }
];
```

LoginRouteGuard is checked on activation of the user route, and SaveContentGuard upon leaving the route.

Using CanActivate

Implements the CanActivate interface and implement the canActivate function e.g.

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { LoginService } from './login-service';

@Injectable()
export class LoginRouteGuard implements CanActivate {

  constructor(private loginService: LoginService) {}

  canActivate() {
    return this.loginService.isLoggedIn();
  }
}
```

If canActivate returns true, the user can activate the route, otherwise the route remain inaccessible.

An unsuccessful attempt at accessing the route could result in a notification or a redirect.

Using CanDeactivate

Very similar to canActivate, firstly create an interface for the function implementation e.g.

```
export interface CanDeactivate<T> {
  canDeactivate(component: T,
route: ActivatedRouteSnapshot,
state: RouterStateSnapshot):
      Observable<boolean>|Promise<boolean>|boolean;
}
```

Then, implement the interface in the chosen component e.g.

```
import { CanDeactivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { UserComponent } from './user-component';

@Injectable()
export class SaveContentGuard
implements CanDeactivate<UserComponent> {

  canDeactivate(component: UserComponent) {
    return component.areEditFormsSaved();
  }
}
```

Asynchronous Route Guards

Notice in the previous code that both functions can return values of **Observable<boolean>**, **Promise<boolean>** or simply **boolean**.

This is because the decision making may take place over HTTP with an external service, or have to carry out several functions that may take some time.

If an asynchronous request is required, use the Observable<boolean> over the Promise, and the router will wait for resolution to determine access.

When navigating away from a component, a dialog for confirmation could be used, and again, this returns an observable which resolves to true for 'OK' and false for 'cancel' e.g.

```
canDeactivate() {
   return dialogService.confirm('Discard unsaved changes?');
}
```

Generally, favour Observables for more fine-grained control in asynchronous interactions.

CHAPTER 8

Observables & Reactive Programming

Introduction

A new feature with Angular is the Observable. This is a proposed standard for managing asynchronous data coming in ES7.

Observables open up a continuous channel of communication in which multiple values of data may be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular uses observables extensively – not just for HTTP but also in events.

Observables are quite similar to promises, though with improvement. The major differences being:

Observables	**Promises**
Handle multiple values over time	Called once and return a single value
Are cancellable	Are not cancellable

As working with real-time date is becoming more usual, observables offer an improvement over promises, and are generally preferred to promises for HTTP.

As Observables are a future JS feature, in order to make use of them a library is needed, usually RxJS, which provides Observable operators to handle emitted data, such as:

- Map, Filter, Take, Skip, Debounce, Subscribe, Buffer ...

Reactive Programming

In order to use Observables in Angular, some knowledge of RxJS and reactive programming is required: but what is reactive programming?

Put simply:

"Reactive programming is programming with asynchronous data streams".

Create and listen for streams (data emissions) on click, hover, user inputs, property changes, data streams etc. Observe these streams and react to changes.

Usefully, RxJS provides a toolbox of functions that can combine, decorate, filter, merge and manipulate those streams.

Important Concepts

1. Promises emit single values, while streams emit multiple values.

Moving from callbacks, to promises, and then on to streams effectively offers the ability to continuously respond to data changes.

Imperative code "pulls" data whereas reactive streams "push" data.

2. Reactive Programming involves subscribers that are notified of changes, with streams pushing data to subscribers

3. RxJS is functional. Functional operators like map, reduce, and filter are commonplace.

4. Streams are composable.

Streams are pipelines of operations. Subscribe to streams and even combine them to produce new streams if necessary.

Creating Observables

After importing from rxjs/Observable, declare an object of the type, which itself contains data of a chosen type.

```
private data: Observable<number>;
```

Create an instance, and update the observer with data over time:

```
this.data = new Observable(observer => {
    setTimeout(() => {
        observer.next(23);
    }, 2000);

    setTimeout(() => {
        observer.next(89);
    }, 4000);

    setTimeout(() => {
        observer.complete();
    }, 6000);

    this.finished = false;
});
```

Observables can produce several notifications with these methods:

- next() emits an event. May be called multiple times
- error() throws an error. Break the stream when called. The error callback will take place
- complete() marks the observable as complete

Consuming Observables

Subscribe to the Observable to listen for changes. The subscribe method accepts three callbacks as parameters:

- onNext called when an event is triggered
- onError called when an error is thrown
- onCompleted called when the observable completes

These are used like so:

```
let subscription = this.data.subscribe(
   value => this.values.push(value),
   error => this.errors = true,
   () => this.finished = true
);
```

Differences to Promises

Unsubscribing

Observables may be cancelled, by returning a function during initialisation of the Observable.create function:

```
var observable = Observable.create((observer) => {
  var id setTimeout(() => {
    observer.next('test event');
  }, 500);

  return () => {
    clearTimeout(id);
  };
});
```

If the unsubscribe method is called before 500 milliseconds, the event will never be triggered e.g.

```
var subscription = observable.subscribe();

setTimeout(() => {
  subscription.unsubscribe();
}, 300);
```

Interestingly, the browsers XHR API supports call cancelling, allowing HTTP calls to be cancelled if they timeout, and even before they return:

```
var observable = this.http.get('http://test');

var subscription = observable.subscribe(res => {
  console.log('Response received.');
});

setTimeout(() => {
  subscription.unsubscribe();
}, 100);
```

Check the network tools tab in Chrome to see the request cancelled unless it completes within the allotted time.

Lazy Nature

As opposed to promises, observables are only enabled when a first observer subscribes. When creating promises, the initialization processing is immediately called. Observables are lazy, a callback must be subscribed to let them execute their initialization callback.

With no subscribe method called on the observable, setTimeout will never be called and the event never triggered e.g.

```
var observable = Observable.create((observer) => {
  setTimeout(() => {
    observer.next('test event');
  }, 500);
});
```

With HTTP observables, a request will only be executed on subscription.

Multiple Executions

Observables may be calls multiple times, where promises cannot be reused after resolution or rejection.

Here, an observable sends events every 1s, completing after 10s e.g

```
var observable = Observable.create((observer) => {
   var id = setInterval(() => {
      observer.next('test event');
   }, 1000);

   setTimeout(() => {
      clearInterval(id);
      observer.complete();
   }, 10000);
});

observable.subscribe(
   (data) => {
      console.log('data received');
   },
   (error) => { },
   () => {
      console.log('completed');
   }
);
```

Operators

Reactive Programming leverages asynchronous data streams, with multiple parts of the application connected on streams. Observables are adorned with operators to transform and adapt streams.

Common operators:

map – Projects each element of an observable sequence into a new form by incorporating the element's index.

flatMap – Projects each element of an observable sequence to another observable sequence.

flatMapLatest – Same as flatMap but cancels previous observables if they are in progress.

filter – Filters the elements of an observable sequence based on a predicate.

Form Events

While extremely useful is HTTP, Observables are able to be used elsewhere fairly extensively. Angular form fields are treated as observables, and can be subscribed to in order to listen for changes e.g.

```
<h2>Observable Forms</h2>
<form [formGroup]="testForm">
  <input formControlName="textInput">
</form>
<div>
  <b>In reverse:</b> {{data}}
</div>
```

Form control field have a valueChanges property which returns an Observable:

```
this.textInput.valueChanges
   .map(n=>n.split('').reverse().join(''))
   .subscribe(value => this.data = value);
```

Asynchronous Processing

Add the map operator to extract the JSON payload from the response.

```
addBookWithObservable(book:Book): Observable<Book> {
   let headers = new Headers({ 'Content-Type': 'application/json' });
   let options = new RequestOptions({ headers: headers });
   return this.http.post(this.url, book, options)
      .map(this.extractData)
      .catch(this.handleErrorObservable);
}
```

As observables are lazy, call the subscribe method when executing the method.

```
addBook(): void {
   this.bookService.addBookWithObservable(this.book)
   .subscribe( book => {
      this.fetchBooks();
      this.reset();
      this.bookName = book.name;
   },
   error => this.errorMessage = <any>error);
}
```

These can be linked with user events, perhaps to display when the user enters values into a search field, or clicks on a particular part of the screen.

Error Handling in Streams

The catch operator can be used within services to extract the actual error from an HTTP response payload or call a method of the service to handle errors globally.

To propagate a new error in the stream, use the Observable.throw method e.g.

```
addBookWithObservable(book:Book): Observable<Book> {
   let headers = new Headers({ 'Content-Type': 'application/json' });
   let options = new RequestOptions({ headers: headers });
      return this.http.post(this.url, book, options)
      .map(this.extractData)
      .catch(this.handleErrorObservable);
}

private handleErrorObservable (error: Response | any) {
   console.error(error.message || error);
   return Observable.throw(error.message || error);
}
```

Merging Results

For chaining calls, the mergeMap operator can be used. Here the result of the second call is merged with the first:

```
loadContractByCustomer(): void {
   this.http.get('assets/customer.json')
      .map((res: Response) => {
         this.customer = res.json();
         return this.customer;
   })
   .mergeMap((customer) =>
      this.http.get(customer.contractUrl))
   .map((res: Response) => res.json())
   .subscribe(res => this.contract = res);
}
```

Parallel Execution

Use Observable.forkJoin method e.g.

```
loadPeopleAndCustomers(): void {
   this.combined = { people: [], customer: {} };
   Observable.forkJoin(
      this.http.get('assets/people.json')
         .map((res: Response) => res.json()),
      this.http.get('assets/customer.json')
         .map((res: Response) => res.json())
      )
      .subscribe(res => this.combined = {
         people: res[0].people, customer: res[1]
      }
   );
}
```

Polling

Polling a service can be useful in lieu of WebSockets. Observables have an interval operator for just this e.g.

```
initializePolling() {
   return Observable
      .interval(5000)
      .flatMap(() => this.getMessages());
}
```

Polling may be cancelled using unsubscription, or until a completion flag is switched using the takeUntil operator:

```
initializePolling(stopPolling) {
   return Observable
      .interval(5000)
      .flatMap(() => this.getMessages())
      .takeUntil(stopPolling);
}
```

Cold and Hot Observables

Cold observables start running upon subscription i.e. values are pushed to observers by the observable sequence only on subscription e.g. multiple subscribers will receive all data.

Hot observables are producing values before subscription. Values are being published whether listened to or no.

An example in raw RxJS:

```
const observable = new Rx.Observable(observer => {
   setTimeout(() => { observer.next(1); }, 1000);
   setTimeout(() => { observer.next(2); }, 2000);
   setTimeout(() => { observer.next(3); }, 3000);
   setTimeout(() => { observer.next(3); }, 5000);
   setTimeout(() => { observer.next(4); }, 5000);
}).publish();

observable.connect();
```

Subscribers will effectively be listening to the same stream, and will only receive data from the time they subscribe; prior data will have been missed.

Hot observables can use refCount instead of connect. Broadcasting beings as soon as there is more than one subscriber.

Inter-Component Communications

There are several means of communicating between components. The Angular-provided EventEmitter is one, and is actually built upon the Subject type provided by RxJS.

A Subject is a subtype of Observable that can both create and listen to data streams. The documentation for Subject can be found here:

http://reactivex.io/rxjs/manual/overview.html#subject

To use:

1. Create a Subject and call next([value]) to emit a chosen value at a chosen time.
2. Elsewhere, subscribe to the Subject using subscribe().
3. Emit values on demand, and subscribe in a separate component to observe changes.

For example, a MessageService could be injected where required to both send and receive messages:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Subject } from 'rxjs/Subject';

@Injectable()
export class MessageService {
    private subject = new Subject<any>();

    sendMessage(message: string) {
        this.subject.next({ text: message });
    }

    clearMessage() {
        this.subject.next();
    }

    getMessage(): Observable<any> {
        return this.subject.asObservable();
    }
}
```

CHAPTER 9

Angular & Redux

Introduction

Large Angular applications undertake a lot of asynchronous activity, manage a lot of state, and share that state across components. State management is hard.

Among others, applications typically deal with state in the form of:

Server state

Stored on the server and usually available via REST endpoints.

Persistent state

A subset of the server state stored on the client, in memory. Naively, we can treat the persistent state as a cache of the server state.

URL and router state

Client state

Being that state not stored on the server. Alerts, toggles, forms, filters, searches etc. This is often reflected in the URL.

Transient client state

Where state is stored on the client, but not represented in the URL. Last recently viewed video, position in a file etc.

Local UI state

Individual components in charge of their own behaviour. Button colours on change, whether to expand a view area etc.

Keeping tabs on where state is changed and the knock on effects of those changes can become laboured.

There are several approaches to dealing with state in Angular applications; here we discuss Angular alongside Redux.

Flux

Flux is an alternative way of managing communication between components, and the general flow of data in applications. It's popular in React, but appears as a technique elsewhere.

Flux aims to help manage application data flow by applying an architectural style onto proceedings. It should be noted that Flux is not a framework or library, but an approach/idea, or set of guiding principles.

"Flux is a system architecture that encourages single-directional data flow through your application" - Tom Occhino, Facebook

Flux treats application data as the primary concern, and concerns itself with managing the flow of data using four parts:

- Store
- Dispatcher
- Views (React components, Angular views)
- Action

To summarise: Flux is essentially another name for the Observer pattern.

Is Flux Needed?

If the application deals with dynamic data then Flux (or an implementation, or an extension) is needed.

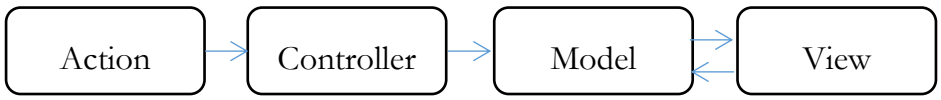
If the application has static views that don't share state, and data is not saved or updated then Flux will be of little benefit.

There is some complexity here, but the addition of clarity is invaluable.

Flux vs MVC

The notion of MVC has been in favour since 1976; and is still used for many projects. The issues with MVC is scalability, and at Facebook it proved easier to use Flux at scale.

To visualise MVC:

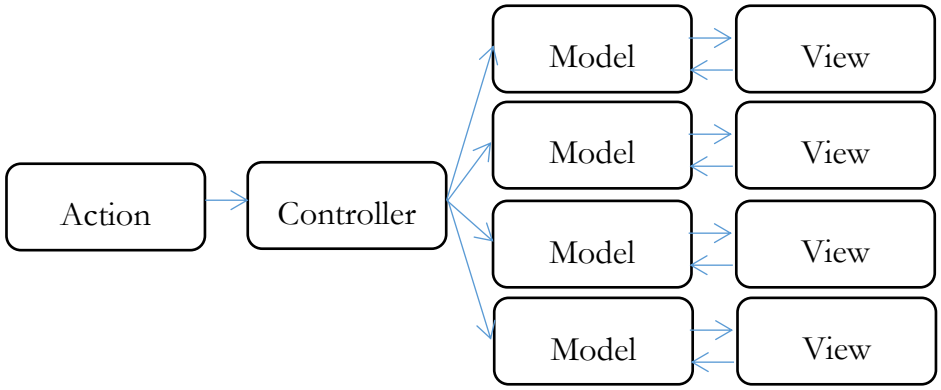


Here, the standard flow and responsibility is:

- **Model** maintains domain data and behaviour
- **View** is the display aspect
- **Controller** takes user input, manipulate the model, update the view

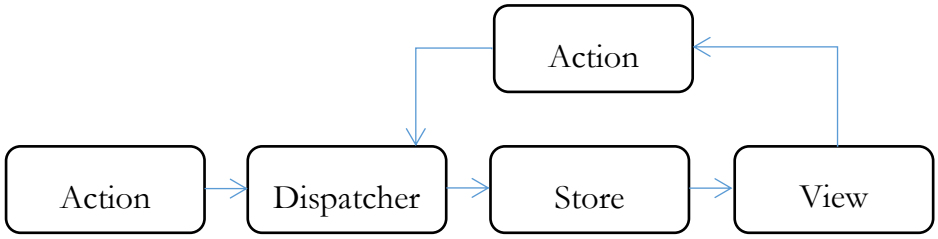
With bidirectional data flow, the Controller eventually starts to feel overly responsible for both state maintenance and data, and can be overwhelmed with throughput.

As MVC scales, we a faced with the following:



It becomes much harder to understand and debug. Anyone that has worked with MVC will recognise this difficulty, and while server-side frameworks such as Struts and Spring MVC can take remove some pain through making controllers declarative, they add pain in other areas. Anyway, server-side MVC is losing advocates.

To visualise Flux:



Importantly, and differentiating Flux from MVC, this is a **Unidirectional Data Flow**, which promotes predictability: the process governing the presentation of application state to the user should be deterministic:

1. **Views** send **Actions** to the **Dispatcher**
2. **Dispatcher** send **Actions** to **Stores**
3. **Views** get data from **Stores**

Flux is strict about the flow of application. Unidirectional flow means all the changes go to the Dispatcher through Actions: the Store cannot change by itself. The Store is also able to maintain any data related to the application.

Being able to assume that an Action simply flows through the system makes understanding the system far easier.

The Dispatcher is very much the traffic controller. It enforces that until the Store layer has completed, the Views are unable to put another Action through the system. This constraint is a guarantee of understanding the downstream effects of an Action.

Moving to Flux:

- Improves data consistency
- Makes it easier to pinpoint bugs
- Unit tests (if used) can be more meaningful

At Facebook, implementing Flux led to a dramatic decrease in the numbers of tickets (bugs) relating to the corresponding system.

Redux

Redux is a continuation of the work done with Flux; it's essentially an evolution of Flux plus some new additions, such as hot reloading and time-travel.

Redux is described as:

"a predictable state container for JavaScript applications".

Redux helps applications behave consistently, run in different environments (client, server, and native), and enables easier testing.

Development helpers such as time traveling debugging via Redux DevTools improve working conditions.

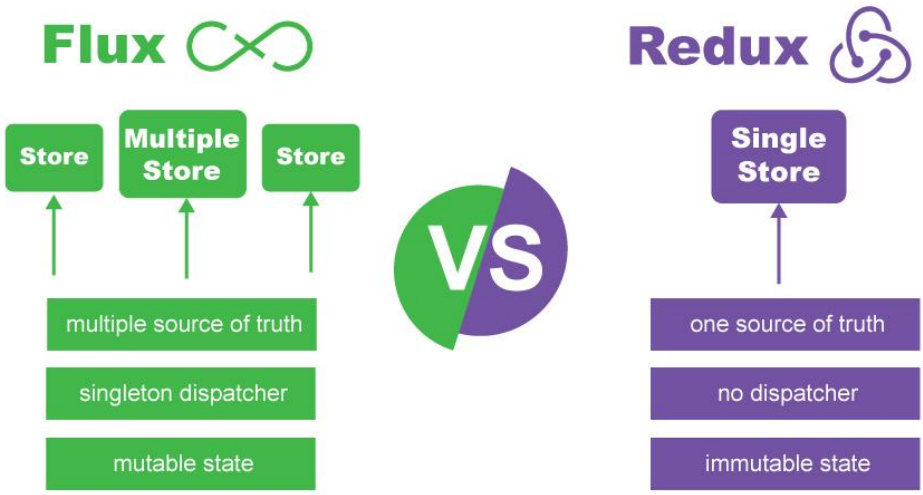
Redux started out with React, but can be used independently of React: there are implementations in Angular, Vue and others.

Surprisingly, Redux is 2k in size, with dependencies.

Inspired by Flux

The big differences that set Redux apart from Flux are:

Flux	Redux
A pattern.	A library.
Has actions: simple JS objects.	Has actions, though these may also be functions or promises.
Has multiple singleton stores, with logic.	Has a single store, holding no logic. Uses a reducer function called for each dispatched action.
The store is smart.	The reducer is smart.
Has a single dispatcher, with all actions passing through it.	Has no dispatcher, the store handles the dispatch process.
Data can be mutated as desired.	Store state is immutable. Reducers copy state and return modified versions of the state's copy.



Three Principles

It's extremely easy to over-complicate Redux. A simple gist of the framework can be gleaned from the readme of the package, plus a little more additional information, in order to outline the fundamentals.

1. The whole state of your app is stored in an object tree inside a single store.

   This is the first principle of Redux: the concept of the Single Immutable State Tree, or **Single Source of Truth**. The entire state of the application is represented by one JavaScript object.

   In the example TODO application supplied by the Redux library, the following object is the state:

```
"current_state:"
[object Object] {
   todos: [[object Object] {
      completed: true,
      id: 0,
      text: "hey",
   }, [object Object] {
      completed: false,
      id: 1,
      text: "ho",
   }],
   visibilityFilter: "SHOW_ACTIVE"
}
```

2. The only way to change the state tree is to emit an action, an object describing what happened.

   This second principle of Redux is that **the state tree is read only**. State may only be changed by dispatching actions.

3. To specify how the actions transform the state tree, you write pure reducers.

   So **changes are made with pure functions**. Reducers are pure functions that take the previous state and an action, and return the next state. New state objects are returned, instead of mutating the previous state.

   - Pure functions return values dependant only upon argument values.
   - Pure functions don't have side effects like network or database calls.
   - Pure functions also do not override the values of anything.

Core Redux

Over the following pages, the core components of Redux are introduced, independently of a framework.

Actions

Actions are payloads of information; sending data from application to store.

Actions are the only source of information for the store.

Actions are sent to the store using store.dispatch().

Actions are plain JavaScript objects.

Actions must have a type property to indicate the type of action being performed. Types should be defined as string constants.

Actions may have any structure (adding to the type property), though often pass as little data as possible. There is a standard type for Actions, described by the flux-standard-type: https://github.com/redux-utilities/flux-standard-action

The flux-standard-action dictates that Actions **may have:**

4. an error property
5. a payload property
6. a meta property

For example, a basic Flux Standard Action:

```
{
   type: 'ADD_TODO',
   payload: {
      text: 'Do something.'
   }
}
```

, which could be described by a TypeScript interface:

```
interface Action {
   type: string;
   payload?: any;
}
```


To provide further guidance on the properties, from the FSA specification:

type

The type of an action identifies to the consumer the nature of the action that has occurred. type is a string constant. If two types are the same, they MUST be strictly equivalent (using ===).

payload

Represents the payload of the action, and MAY be any type of value.

Any information about the action that is not the type or status of the action should be part of the payload field.

By convention, if error is true, the payload SHOULD be an error object. This is akin to rejecting a promise with an error object.

error

This MAY be set to true if the action represents an error.

An action whose error is true is analogous to a rejected Promise. By convention, the payload SHOULD be an error object.

If error has any other value besides true, including undefined and null, the action MUST NOT be interpreted as an error.

meta

The optional meta property MAY be any type of value. It is intended for any extra information that is not part of the payload.

It should be noted that the FSA is just guidance, and many Actions will be seen with additional properties. The key is to keep the data minimal.

For example the following are valid Actions:

```
{
   type: 'PLUS',
   payload: 7
}

{
  type: 'PHRASE_SET',
  payload: ERROR_OBJECT,
  error: true,
}
```

Note

Actions describe the fact that something happened, they do not describe how the state changes.

Action Creators

Functions that create actions.

Action creators just return an action:

```
// Actions
const COUNTER_INCREMENT = 'COUNTER_INCREMENT';
const COUNTER_DECREMENT = 'COUNTER_DECREMENT';

// Action Creators
export const incrementCounter = () => ({
  type: COUNTER_INCREMENT,
});

export const decrementCounter = () => ({
  type: COUNTER_DECREMENT,
});
```

To dispatch an action, pass the result of the action creator to the dispatch() function:

```
dispatch(incrementCounter())
```

Alternatively, create a bound action creator:

```
const boundIncrementCounter = () => dispatch(incrementCounter())
```

The dispatch() function can be accessed from the store (using store.dispatch()).

Also, bindActionCreators() can automatically bind multiple action creators to a dispatch() function.

As Action Creators are not reducers, they may have side-effects. This becomes important for Redux Middleware, and for async data flow, covered in later sections.

Reducers

Reducers specify how the state changes in response to actions sent to the store.

Reducers are pure functions that takes the previous state and an action, and return the next state:

```
(previousState, action) => newState
```

Reducers are pure functions. They **do not:**

- Mutate its arguments
- Perform side effects like API calls and routing transitions
- Call non-pure functions, e.g. Date.now() or Math.random()

For example:

```
function reducer(state = initialState, action) {
  switch(action.type) {
    case 'INCREMENT':
      // Works, but not strictly correct.
      // return {
      //   count: state.count + 1
      // };
      return Object.assign({}, state, {
        count: state.count + 1
      });
    case 'DECREMENT':
      return {
        count: state.count - 1
      };
    default:
      return state;
  }
}
```

State is copied using Object.assign(), with an empty object as the first parameter.

The object spread operator can be used to copy other properties { ...state, ...newState } if they are present.

The default case returns the previous state, or what would have been the initial state with no action present.

State may be split between reducers, using a process called **reducer composition**, so that it may be dealt with in an appropriate place, before being recombined.

For example, from the provided TODO application:

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

In large application, reducers may be in separate files, before being recombined using combineReducers(), for example:

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

, or, in ES6 syntax:

```
import { combineReducers } from 'redux'
import * as reducers from './reducers'

const todoApp = combineReducers(reducers)
```


The Store

The Store is provided by Redux, and:

- Holds application state
- Allows access to state via getState()
- Allows state to be updated via dispatch(action)
- Registers listeners via subscribe(listener)
- Handles unregistering of listeners via the function returned by subscribe(listener)

There is only a single store in a Redux application.

Rather than multiple stores, complexity bring multiple reducers via reducer composition.

Stores are created with reducers:

```
import { createStore } from 'redux'
import exampleApp from './reducers'

const store = createStore(exampleApp)
```

Initial state may be optionally provided as the second argument. In most cases this is used to hydrate client state as a match to server state:

```
const store = createStore(todoApp, window.STATE_FROM_SERVER)
```

With store access, actions can be dispatched; a UI is not required to test that the setup is wired together:

```
store.dispatch({ type: 'INCREMENT' });
store.dispatch(addTodo('Build Redux application.'));
```

Redux Data Flow

Redux adheres to **strict unidirectional data flow**. This means predictable, easier to understand processes, and encourages only a single copy of the data, rather than multiple independent copies with their own lifecycle.

The Redux data lifecycle follows four steps:

1. Dispatch Action

A plain object, the Action, is dispatched and describes the event.

For example:

```
{ type: 'ARTICLE_HIGHLIGHT', articleId: 88 }
{ type: 'ADD_TODO', text: 'Create Redux simple example code.' }
```

An action is a news snippet. The store.dispatch(action) can be called from anywhere.

2. Store calls the Reducer function

The store passes two arguments to the reducer: the current state tree and the action.

The reducer computes the next state; predictably. Calling it with the same inputs many times should produce the same outputs. There are no side effects. Side effects, if required, should happen before an action is dispatched.

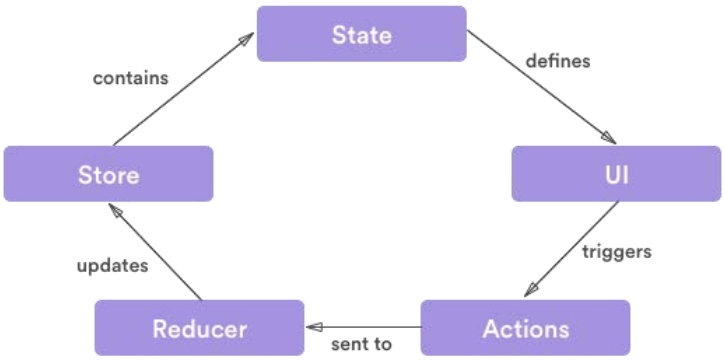
3. Multiple Reducers may be combined

Redux has combineReducers(), or one may be written.

4. Store saves the state tree

The new tree is now the next state. Listener registered with store.subscribe(listener) will be invoked and may call store.getState() to obtain current state.

The UI, if present, can be updated to reflect the new state.



Angular & Redux

Background

Redux is not Angular specific. It is a pure JS predictable state container, an evolution of Facebook's Flux, and influenced by the Elm functional language.

Redux is a convention for data consumption in client applications. The idea is to store every piece of data in a centralised container and change it only using set of predefined actions. Data flows in one direction only, reducing complexity.

The emphasis is on:

- A single, immutable data store
- One-way data flow
- Change based on pure functions and action streams

Confusingly, in the Angular space there are two main implementations of Redux:

1. **ngrx**
2. **angular-redux**

The libraries are similar, with ngrx favouring RxJS, and angular-redux trying to maintain compatibility with Redux libraries. Both expose the store as an observable stream.

It seems that some of the Google team are involved with ngrx, while Rangle.io have a hand in angular-redux, as the main developer works with that company.

Redux Concepts Recap

Application data is in a single data structure called the state, held in the store

- Application reads state from the store
- The store is never mutated directly
- User interaction fires actions which describe state changes
- A new state is created by combining the old state and the action by a reducer function

ngrx

Most of the ngrx implementation is through the ngrx/store module. Other modules are available for better integration and development:

ngrx/store-devtools - ngrx implementation of Redux DevTools

ngrx/effects - a model for performing side-effects similar to redux-saga

ngrx/router and ngrx/router-store - a router for Angular that can be connected to the ngrx store

Setup

Add support for ngrx using npm:

```
npm install @ngrx/core @ngrx/store @ngrx/effects –-save
```

Add store support to the code module e.g.

```
...
import { StoreModule } from '@ngrx/store';
import { employees } from './reducers/employee.reducer';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot(employees)
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```


Define State

Using Redux, it pays to spend time on deciding what state the store will manage. Generally, it tends to be:

- Data from server-side APIs
- Data from user input, including UI state (visible/invisible etc)
- Data regarding user preferences/themes
- Data for internationalization

An interface called AppState, or AppStore is used to define state, with readonly to ensure compile-time immutability e.g. in app.store.js:

```
import { Employee } from './models/employee';

export interface AppStore {
    readonly employees: Employee[];
}
```

Here, the state is based up an Employee model.

Actions and ActionCreators

Many Redux patterns dispatch actions directly from components. This can be viewed as incorrect by some, and instead an action creator is used that dispatches actions. This can be defined in a separate file.

Action types could just be simple string e.g.

```
export const INCREMENT: String = "INCREMENT";
```

Create Reducers

Actions are reduced, and new application state is returned.

Reducers return a new copy of the state, and are pure, side-effect free functions.

```
import { Action } from '@ngrx/store';

export function employees (state: any = [], action: Action) {
    switch (action.type) {
        case 'ADD_EMPLOYEES':
            return [...state, payload];
        default:
            return state;
    }
}
```

As the state is immutable, the ES6 feature Object.assign() or the spread operator, used above, for arrays can help maintain that immutability.

Store Awareness

Here, an EmployeeService makes use of an HttpService to dispatch actions, which engage the reducer to produce an updated store object. Awareness of change is via the observable employees.

```
@Injectable()
export class EmployeeService {

    employees: Observable<Array<Employee>>;

    url = '';

    constructor(private http: HttpService,
        private store: Store<AppStore>) {
        this.employees = store.select(store => store.employees);
    }

    loadEmployees() {
        return this.http.get(this.url)
            .map((res: Response) => {
                console.log('Response: ', res);
                return res || {};
            })
            .map((payload: Employee[]) => {
                console.log('Payload: ', payload);
                return { type: 'ADD_EMPLOYEES', payload };
            })
            .subscribe((action) => {
                console.log('Action: ', action);
                this.store.dispatch(action);
            });
    }
}
```

Elsewhere, a view component can engage the EmployeeService when a button is clicked:

```
<h2>List of employees</h2>
<button class="btn btn-primary" (click)="loadNew()">Load New</button>
<ol>
    <li *ngFor="let employee of (employees | async)">
        {{ employee.surname }}
    </li>
</ol>
```

, where loadNew calls the injected service:

```
loadNew() {
   this.employeeService.loadEmployees();
}
```


Redux DevTools

An extremely useful development time package to provide tooling to the browser. Connecting the browser to a Redux application and add tools, real-time monitoring, and time-travel to an additional browser panel.

This guide uses the Chrome browser extension; the same as used by React applications.

Add Redux DevTools

The first step is to add the browser extension via either Chrome extensions or Firefox Add-ons.

The simplest way to add the tooling follows.

1. Install the tooling:

```
npm install --save @ngrx/store-devtools
```

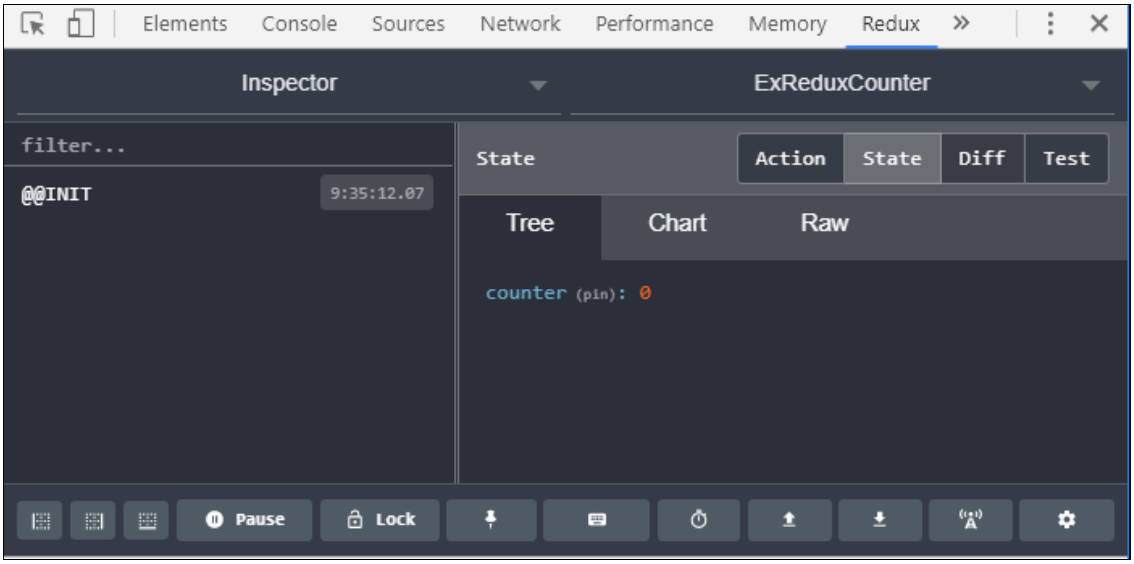
2. Update the app module:

```
import { StoreDevtoolsModule } from '@ngrx/store-devtools';

@NgModule({
  imports: [
    StoreModule.forRoot({ post: postReducer }),
    StoreDevtoolsModule.instrument({
      maxAge: 10 // number of states to retain
    })
  ]
})

export class AppModule { }
```

This is not an exhaustive setup; additional steps can be found in the documentation.

Redux DevTools Panel

Once installed and running, an additional Redux panel is available in the browser tooling. If installed correctly, there should be a visualisation of the store contents available under 'State' e.g.

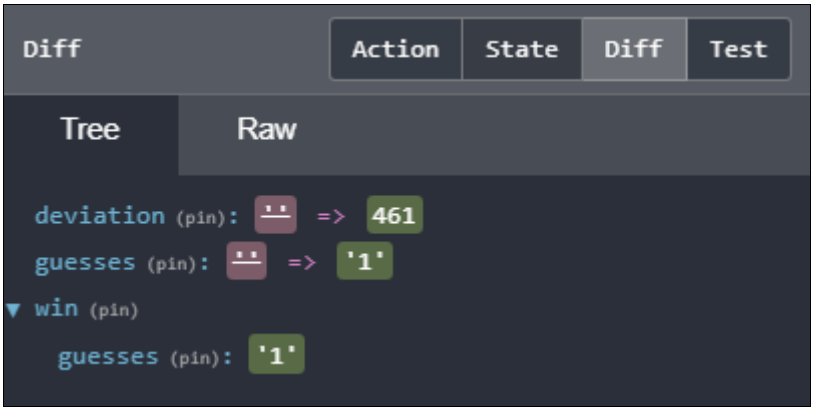


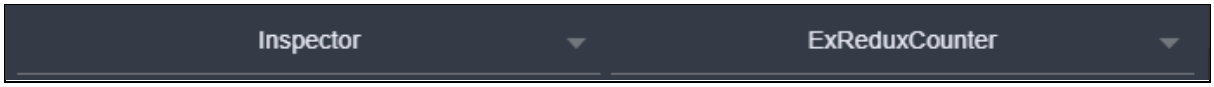
The left side shows actions and two options (Jump and Skip); enabling time traveling and changing the application view:

- **Jump** takes the application to the state of the app at the time this action fired.
- **Skip** shows the application without that action.

The four tabs on the right side are:

- **Action** shows the selected action type and data being transfered to the reducers.
- **State** shows the entire state tree at the time of the selected action.
- **Diff** shows only what the selected individual action changed in the state tree.
- **Test** create a test format in some pre-provided testing frameworks. Takes the root state and provides a written test on what the end state should be.





The top of the console shows the (initially) Inspector and App tabs:

- **App** allows choosing a running application, with different store.

The left set of tabs has different modes:

- **Inspector** (default) shows the tools mentioned above.
- **Log Monitor** shows complete state and actions, while Inspector shows diffs.
- **Chart** shows the state in a tree-like structure.

Finally, the bottom of the console provides more functionality for inspecting actions and state. It features time travel, dispatching actions, importing/exporting state, and remote control.

From the left, the first three options are for placing the console view as required.

- **Pause** recording stops the recording of actions within the console.
- **Lock changes** freezes the running applications future actions.
- **Persist** keeps the current state even when the page reloads.
- **Dispatcher** either shows or hides the dispatching module.
- **Slider** allows scrolling through actions.
- **Import and Export** do just that, and allow JSON to be used for custom state.
- **Remote** creates a separate console.

Note. This is a development-time enhancement, and should be removed prior to release.

CHAPTER 10

Testing Angular Applications

Introduction

Whether it's test-first, tes-ish-first, or retrofitting tests, there is little argument that writing testable code is a powerful addition to the building of client applications. Testing comes in the form of end-to-end tests and unit tests.

End-to-End Testing tests the flow of the application from start to finish, across both components and systems.

Unit Testing isolates aspects of the application and applies tests to prove that output meets the expectation of input.

Here, the focus is on Unit Testing. Unit testing and End-to-End testing Angular applications means using Jasmine and Karma.

Unit tests:

- Should have a clear objective
- Should be isolated; that is, have minimal dependencies, independence from other tests
- Should be deterministic. Either the test passes all of the time, or fails all of the time
- Should be named descriptively, verbosity is fine here if it meets the need
- Should happily break the DRY rule if this adds to an understanding of the system under test (SUT)

Using Jasmine

Jasmine is a BDD framework for unit testing JavaScript. Similar to Mockito, expectations can be set and mocks also used to build fairly intuitive testing code. A function may be tested:

```
function helloWorld() {
  return "Hello world!";
}
```

, with the tests being written as follows:

- describe() declares a test suite, or group of tests
- it() declares a single test
- expect() declares an assertion for the test outcome
- matcher(expected) expression is a matcher; a boolean comparison which if false, fails the spec

For example:

```
describe("Hello world", function() {
  it("says hello", function() {
    expect(helloWorld()).toEqual("Hello world!");
  });
});

describe("Hello world", function() {
  it("says hello", function() {
    expect(helloWorld()).toContain("world");
  });
});
```

Jasmine supplies multiple matchers:

```
expect(array).toContain(member);
expect(fn).toThrow(string);
expect(fn).toThrowError(string);
expect(instance).toBe(instance);
expect(mixed).toBeDefined();
expect(mixed).toBeFalsy();
expect(mixed).toBeNull();
expect(mixed).toBeTruthy();
expect(mixed).toBeUndefined();
expect(mixed).toEqual(mixed);
expect(mixed).toMatch(pattern);
expect(number).toBeCloseTo(number, decimalPlaces);
expect(number).toBeGreaterThan(number);
expect(number).toBeLessThan(number);
expect(number).toBeNaN();
expect(spy).toHaveBeenCalled();
expect(spy).toHaveBeenCalledTimes(number);
expect(spy).toHaveBeenCalledWith(...arguments);
```

Your instructor can show Jasmine being run standalone, where output is in the expected red/green style.

Setup & Teardown

Tests often involve some setup, and some cleanup (teardown). Jasmine provides functions to work with:

- beforeAll – called once, before all the specs in the suite are run
- afterAll – called once after all the specs in a test suite are complete
- beforeEach - called before each it function is run
- afterEach - called after each test is run

Using Karma

Karma is used to run the Jasmine-powered unit tests in a browser. Chrome or Firefox may be used, or a headless browser (no UI) like PhantomJS.

Jasmine tests are run within the browser engines from the command line, and displays the results, while optionally watching for changes and re-running the tests automatically.

Angular CLI handles Karma configuration, and tests can be run quite simply from project root using:

```
ng test
```

Angular CLI

Using Angular CLI to create components, pipes, services etc defaults to creating simple jasmine spec files alongside the generated code:

```
ng generate component employee
```

Creates:

```
employee.component.ts
employee.component.spec.ts
```

The spec is bootstrapped according to the type of generator used.

Angular Testing Framework

Angular provides classes that work around Jasmine; these are found in **@angular/core/testing**, as well as **@angular/compiler/testing** and **@angular/platform/browser/testing**.

Configure for Testing

Newer version of Angular have moved the core libraries. Check that karma.conf.js is referencing the newer @angular/cli instead of the older angular-cli:

```
frameworks: ['jasmine', '@angular/cli'],
...
plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-remap-istanbul'),
      require('@angular/cli/plugins/karma')
    ],
...
preprocessors: {
      './src/test.ts': ['@angular/cli']
    },
...
```

angular-cli.json may also need to be updated.

From:

```
        "environments": {
          "source": "environments/environment.ts",
          "dev": "environments/environment.ts",
          "prod": "environments/environment.prod.ts"
        }
```

To:

```
        "environmentSource": "environments/environment.ts",
        "environments": {
          "dev": "environments/environment.ts",
          "prod": "environments/environment.prod.ts"
        }
```

Angular Test Bed

The Angular Test Bed (ATB) is a higher level testing framework for Angular.

Test are still built with Jasmine and run with Karma, though it is easier to create components, handle injection, test asynchronous behaviour, and provide interaction.

ATB enables:

- testing interaction of a directive or component with it's template
- testing of change detection, and user interactions
- testing using Angular DI
- testing using NgModule configuration

Configure ATB

```
describe('Component: MyComponent', () => {

  let component: MyComponent;
  let fixture: ComponentFixture<MyComponent>;
  let myService: MyService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [MyComponent],
      providers: [MyService]
    });

    // Create component and fixture.
    fixture = TestBed.createComponent(MyComponent);

    // Obtain component from fixture.
    component = fixture.componentInstance;

    // Obtain service.
    myService = TestBed.get(MyService);

  });
});
```

beforeEach configures a testing module using the TestBed, creating a test Angular Module able to instantiate components, perform DI etc, configured as a normal NgModule.

The **fixture** is a wrapper for a component and it's associated template.

Create an instance of a component fixture through the TestBed, injecting MyService into the component constructor.

The service and component can then be used in tests as usual.

ATB Change Detection

The DebugElement and By classes allow testing component interactions:

```
import {DebugElement} from "@angular/core";
import {By} from "@angular/platform-browser";
```

The beforeEach can then configure a wrapper to the low level DOM element that represents the components view, via the debugElement property e.g.

```
let el: DebugElement;

// get the "a" element by CSS selector (e.g., by class name)
el = fixture.debugElement.query(By.css('a'));
```

Child nodes may be queried using the debugElement with a By class.

Find the text content of tags by calling el.nativeElement.textContent.trim(), and trigger change detection using fixture.detectChanges() e.g.

```
it('button to be hidden when user clicks', () => {
  expect(el.nativeElement.textContent.trim()).toBe('');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Click Me');
});
```

Additionally, a spy could be used to set expectations, and re-test e.g.

```
spyOn(myService, 'isClicked').and.returnValue(true);
expect(el.nativeElement.textContent.trim()).toBe('');
fixture.detectChanges();
expect(el.nativeElement.textContent.trim()).toBe('Click Me');
```

Testing Components

Testing component properties and methods components is quite easy.

To test a component's set-type methods then e.g.

```
import { MyComponent } from './my-component.component';

describe('Testing data state in my-component.component', () => {
  let app: MyComponent;

  beforeEach(() => {
    app = new MyComponent();
  });

  it('should set new data', () => {
    app.setData('Test data');
    expect(app.data).toBe('Testing data');
  });

  it('should clear message', () => {
    app.clearData();
    expect(app.data).toBe('');
  });
});
```

The test first imports, then initialises the component class using the beforeEach before each test.

As per standard Jasmine tests, an expectation is set, and the test rusn to see if the expectation is met.

Testing HTTP Services

With the new Http API, boilerplate for test setup has reduced. Previously, custom providers were required, but now just HttpClientModule and HttpClientTestingModule are needed:

```
import { TestBed, async, inject } from '@angular/core/testing';
import { HttpClientModule, HttpClient } from '@angular/common/http';
import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';

describe(`FakeHttpClientResponses`, () => {

  beforeEach(() => {
    // 1. Set up test environment.
    TestBed.configureTestingModule({
      imports: [
        // Much simplified.
        HttpClientModule,
        HttpClientTestingModule
      ]
    });
  });

  it(`should issue a request`,
    // 2. Declare as async test as HttpClient works with Observables
    async(
      // 3. Inject HttpClient and HttpTestingController into test
      inject([HttpClient, HttpTestingController],
         (http: HttpClient, backend: HttpTestingController) => {
        // 4. Send request.
        http.get('/foo/bar').subscribe();

        // 5. HttpTestingController replaces old `MockBackend`
        backend.expectOne({
          url: '/foo/bar',
          method: 'GET'
        });
      })
    )
  );

});
```

The API for matching requests is build around:

- expectOne(expr): expect exactly one request that matches
- expectNone(expr): expect that no requests matches
- match(expr): match the request but do not verify/assert

Testing Routes

With three routes, Home, Sales and Consulting, a stock test with TestBed is set up with a spy implementation; a spy is able to programmatically navigate the routes e.g.

```
import {Location} from "@angular/common";
import {TestBed, fakeAsync, tick} from '@angular/core/testing';
import {RouterTestingModule} from "@angular/router/testing";
import {Router} from "@angular/router";

import {
    HomeComponent,
    SalesComponent,
    ConsultingComponent,
    AppComponent,
    routes
} from "./router"

describe('Router: App', () => {

   let location: Location;
   let router: Router;
   let fixture;

   beforeEach(() => {
      TestBed.configureTestingModule({
         // Route tester, with routes to test.
         imports: [RouterTestingModule.withRoutes(routes)],
         declarations: [
            HomeComponent,
            SalesComponent,
            ConsultingComponent,
            AppComponent
         ]
      });
   });

   // Reference injected router and location.
   router = TestBed.get(Router);
   location = TestBed.get(Location);

   // A top level component to insert components,
   // and initial navigation.
   fixture = TestBed.createComponent(AppComponent);
   router.initialNavigation();
});
```

From here, tests fake asynchronous visits to endpoints, and check location paths e.g.

```
it('navigate to "" redirects you to /home', fakeAsync(() => {
   router.navigate(['']);
   tick(50);
   expect(location.path()).toBe('/home');
}));

it('navigate to "search" takes you to /search', fakeAsync(() => {
   router.navigate(['/search']);
   tick(50);
   expect(location.path()).toBe('/search');
}));
```

As routing is async by nature, the fakeAsync method can be used to test.

- Trigger router to navigate to chosen path.
- Wait for promises to resolves with tick()
- Inspect path with location.path()

E2E Testing

Up to now, testing has been for single units; be they components, services or routes.

E2E is End to end Testing, that is:

"Testing complete processes, in real environments with the whole application."

At this stage, it's important to know that full application features work in a live scenario. The focus here is no longer on specific unit, but on overall functionality.

Using Angular CLI, a home for E2E tests is created e.g

```
/e2e
   /app.e2e-spec.ts
   /app.po.ts
```

The spec is where the tests will be written, and the po file is a default page object to test a page of the application.

The page object is used to create functions that return page elements needed for the test e.g.

```
import { browser, by, element } from 'protractor';

export class MyComponentPage {
  navigateTo() {
    return browser.get('/');
  }

  getTitle() {
    return element(by.css('h1')).getText();
  }

  getPoints() {
    return element(by.cssContainingText('div',
      'Score')).$('span').getText();
  }

  getIncrementButton() {
    return element(by.cssContainingText('button', '+1'));
  }

  getResetButton() {
    return element(by.cssContainingText('button', 'Reset'));
  }
}
```


The spec can then be used to import the page object, and perform actions on the page, with expectations, in a very similar fashion to unit tests e.g.

```
import { MyComponentPage } from './app.po';

describe('ex-e2e App', () => {
  let page: MyComponentPage;

  beforeEach(() => {
    page = new MyComponentPage();
  });

  it('Should display E2E Testing title', () => {
    page.navigateTo();
    expect(page.getTitle()).toEqual('E2E Testing');
  });

  it('Should start with score of 0', () => {
    page.navigateTo();
    expect(page.getPoints()).toEqual('0');
  });

  it('Should increase score by clicking increment button', () => {
    page.navigateTo();
    expect(page.getPoints()).toEqual('0');
    page.getIncrementButton().click();
    expect(page.getPoints()).toEqual('1');
    page.getIncrementButton().click();
    page.getIncrementButton().click();
    expect(page.getPoints()).toEqual('3');
  });
});
```

Run the tests using Angular CLI:

```
ng e2e
```

This runs Protractor: a wrapper on top of WebDriverJS, which itself is a language binding of Selenium Web Driver for JavaScript.

CHAPTER 11

Angular CLI Reference

Introduction

Angular CLI makes it easy to create and deploy Angular applications.

What follows are the core functions, followed by the core options; not an exhaustive list.

The application is a moving target; up to date implementation details, as wel ass full content and options, can be found at the Github page:

https://github.com/angular/angular-cli

Setup

Ensure the latest versions of node and NPM are installed. NPM must be version 3 or higher and Node version 4 or higher. Check with:

```
npm --version
node -v
```

Update NPM using:

```
npm install npm@latest -g
```

Though it's simpler to download the latest version of Node.js from the website.

Install Angular CLI:

```
npm install -g @angular-cli
```

Check using:

```
ng help
```

Create Application

```
ng new <project-name> [options]
```

By default, the project is created under the current directory.

Options:

--dry-run only output the files created and operations performed, do not actually create the project. Alias: 'd'.

--verbose output more information. Alias: 'v'.

--skip-npm do not run any npm command once the project is created.

--skip-git do not create a git repository for the project.

--directory parent directory to create the new project into.

Create Application in Current Folder

```
ng init <project-name> [options]
```

Options:

--dry-run only output the files created and operations performed, do not actually create the project. Alias: 'd'.

--verbose output more information. Alias: 'v'.

--skip-npm do not run any npm command once the project is created.

--name name of project to create.

Compile Project

Compiles application to an output directory.

```
ng build
```

The build artifacts will be stored in the dist/ directory. All commands that build or serve your project, ng build/serve/e2e, will delete the output directory (dist/ by default). This can be disabled via the --no-delete-output-path (or --delete-output-path=false) flag.

For up to date implementation details, see the Angular CLI docs:

https://github.com/angular/angular-cli/wiki/build

Serve Application

Serve the application with live reload. This compiles and copies application-specific files to the dist folder prior to serving. By default, port 4200 is used; changed using: ng serve --port=8080.

```
ng serve
```

Add Autocomplete

Adds autocomplete functionality to your shell for ng commands.

```
ng completion
```

Search Documentation

Open a browser with the keyword as search in Angular documentation.

```
ng doc <keyword>
```

E2E Testing

Runs all end-to-end tests defined in your application, using protractor.

```
ng e2e
```

Code Formatting

Formats the code of this project using clang-format.

```
ng format
```

Code Generation

Generate new project code. Alias: 'g'.

```
ng generate <type> [options]
```

Valid types:

component <path/to/component-name>	Generates a component.
directive <path/to/directive-name>	Generates a directive.
pipe <path/to/pipe-name>	Generates a pipe.
service <path/to/service-name>	Generates a service.
class <path/to/class-name>	Generates a class.
guard	Generates a guard.
interface	Generates an interface.
enum	Generates an enum.
module	Generates a module.

The generated component has its own directory, unless the --flat options is specified.

Options:

--flat	do not create the code in its own directory.
--**default**	specify that the route should be a default route.
--lazy	specify that the route is lazy. Default to true.

To create a new component run:

```
ng generate component [component-name]
```

This creates a folder, [component-name], in the app path by default, and creates the following:

[component-name].component.ts	component class file
[component-name].component.css	component styles
[component-name].component.html	component html
[component-name].component.spec.ts	component tests
index.ts	exports the component

Get Configuration

Get a value from the Angular CLI configuration.

```
ng get <path1, path2, ...pathN> [options]
```

The pathN arguments is a valid JavaScript path like "users[1].userName". If the value isn't set, "undefined" will be shown. This command by default only works inside a project directory.

Options:

--global returns the global configuration value instead of the local one (if both are set). Also makes the command work outside of a project directory.

Set Configuration

Set a value in the Angular CLI configuration.

```
ng get <path1=value1, path2=value2, ...pathN=valueN> [options]
```

By default, sets the value in the project's configuration if ran inside a project, or fails if not inside a project. The pathN arguments is a valid JavaScript path like "users[1].userName". The value will be coerced to the proper type or will throw an error if the type cannot be coerced.

Options:

--global sets the global configuration value instead of a local one. Also makes `ng set` works outside a project.

Deploying

Build application for production, setup GitHub repository, then publish.

```
ng github-pages:deploy
```

Options:

--message=<message>	Commit message to include with build.
	Defaults to "new gh-pages version".
--environment=<env>	Angular environment to build.
	Defaults to "production".
--branch=<branch-name>	The git branch to push the pages to.
	Defaults to "gh-branch".
--skip-build	Skip building the project before publishing.
--gh-token=<token>	API token to use to deploy. Required.
--gh-username=<username>	Github username to use. Required.

Linting

Run the codelyzer linter on your project. Very useful for low-level feedback on naming, code-style, declarations etc.

```
ng lint
```

A useful TSLint reference: https://palantir.github.io/tslint/rules/

Testing

Run unit tests, using karma.

```
ng test [options]
```

Options:

--watch	Keep running the tests. Default to true.
--browsers, --colors	Arguments passed directly to karma.
--reporters, --port	
--log-level	

Show Version

Outputs the version of angular-cli, node and the operating system.

```
ng version
```

Add Third Party Libraries

Angular CLI generates automation code, and can integrate external libraries via npm. The dev environment checks entries in package.json and bundles them with the application.

Retrofitting CLI

Angular CLI may be retrofitted to applications created with a skeleton, for example. Use the previously noted ng init for this.

Folder structures may differ, so some config is possible:

--source-dir	Relative path to source files (default = src)
--prefix	Path within source dir that Angular application files reside (default = app)
--style	Path where additional style files are located (default = css)

Upgrading Angular CLI

To update the angular-cli package globally, run:

```
npm uninstall -g angular-cli
npm cache clean
npm install -g @angular/cli@latest
```

Also update the local project version, as it will be selected with higher priority than the global one:

```
npm uninstall --save-dev angular-cli
npm install --save-dev @angular/cli@latest
npm install
```


StayAhead Training Limited
6 Long Lane
London
EC1A 9HF

020 7600 6116

www.stayahead.com