ANGULAR

From Theory To Practice

Asim Hussain

Learn how to build the the web applications of tomorrow, today.

CODECRAFT_ www.codecraft.tv

Angular

From Theory To Practice

Asim Hussain

Version 1.2.0, 2017-11-24

Table of Contents

Copyright . . . 1
Changelog . . . 2
- Versions . . . 3
  - 1.2.0 (2017-11-24) . . . 3
  - 1.1.0 (2017-04-27) . . . 3
  - 1.0.1 (2017-01-05) . . . 3
  - 1.0.0 (2016-11-16) . . . 3

About . . . 4
Quickstart . . . 5
- Overview . . . 6
- Plunker . . . 7
  - Learning Outcomes . . . 7
  - What is plunker and how to use it? . . . 7
  - Structure of an Angular Plunker . . . 7
  - Summary . . . 9
  - Listing . . . 10
- Intro to TypeScript . . . 11
  - Learning Outcomes . . . 11
  - TypeScript vs JavaScript . . . 11
  - Transpilation . . . 12
  - Summary . . . 12
- Writing our first app . . . 13
  - Learning Outcomes . . . 13
  - Components . . . 13
  - Imports . . . 14
  - Template . . . 14
  - Angular Modules . . . 15
  - Bootstrapping . . . 17
  - Component Tree . . . 17
  - Troubleshooting . . . 18
  - Summary . . . 18
  - Listing . . . 19
- String Interpolation . . . 21
  - Learning Outcomes . . . 21
  - Motivation . . . 21
  - Classes . . . 21
  - String Interpolation . . . 22
  - Summary . . . 23

Listing . . . 23
Looping . . . 25
Learning Outcomes . . . 25
JokeListComponent . . . 25
Configuring . . . 27
Summary . . . 28
Listing . . . 29
Property & Event Binding . . . 31
Learning Outcomes . . . 31
Hiding and showing elements . . . 31
HTML Attribute vs DOM Property . . . 31
Input Property Binding . . . 32
Output Event Binding . . . 33
Summary . . . 34
Listing . . . 35
Domain Model . . . 37
Learning Outcomes . . . 37
What are Domain Models? . . . 37
Our Domain Model . . . 37
Summary . . . 39
Listing . . . 39
Nesting Components & Inputs . . . 41
Learning Outcomes . . . 41
Create & Configure Multiple Components . . . 41
Input Property Binding on a Custom Component . . . 45
Summary . . . 47
Listing . . . 48
User Interaction & Outputs . . . 51
Learning Outcomes . . . 51
The Joke Form . . . 51
The JokeFormComponent . . . 52
Template Reference Variables . . . 55
Summary . . . 56
Listing . . . 57
Wrapping Up . . . 61
Activity . . . 62
Steps . . . 62
Solution . . . 62
ES6 JavaScript & TypeScript . . . 63
TypeScript Setup . . . 64
Transpilation . . . 64

Installing TypeScript . . . 64
Running TypeScript . . . 64
Overview . . . 66
ES6 Support . . . 66
TypeScript Support . . . 66
Let . . . 67
Block Scope . . . 67
Let . . . 69
Using let in for loops . . . 69
The for loop short-cut . . . 71
Summary . . . 71
Listing . . . 71
Const . . . 73
Declaring a const variable . . . 73
Block scoping . . . 73
Immutable variable . . . 73
Mutable Value . . . 74
Summary . . . 75
Listing . . . 75
Template Strings . . . 77
Multi-Line Strings . . . 77
Variable Substitution . . . 78
Summary . . . 78
Listing . . . 78
Fat Arrow Functions . . . 80
Syntax . . . 80
Arguments . . . 80
this . . . 81
Summary . . . 83
Listing . . . 83
Destructuring . . . 84
Object Destructuring . . . 84
Array Destructuring . . . 85
Function Parameter Destructuring . . . 85
Summary . . . 86
Listing . . . 86
For Of . . . 87
For & ForEach . . . 87
For In . . . 87
For-of loop . . . 88
Summary . . . 88

Listing . . . 89
Map & Set . . . 90
Using Object as a Map . . . 90
Map . . . 91
Set . . . 94
Summary . . . 96
Listing . . . 96
Promises . . . 98
Callbacks . . . 98
Promise API . . . 98
Creating a Promise . . . 98
Promise Notifications . . . 100
Immediate Resolution or Rejection . . . 100
Chaining . . . 101
Catch . . . 102
Summary . . . 102
Listing . . . 102
Class & Interface . . . 104
Object Orientation in JavaScript . . . 104
Class . . . 104
Inheritance . . . 106
Access Modifiers . . . 106
Constructor shortcut . . . 108
Interfaces . . . 108
Summary . . . 109
Listing . . . 109
Decorators . . . 111
Simple no-argument decorator . . . 111
Decorators with arguments . . . 112
Summary . . . 112
Listing . . . 113
Modules . . . 114
Module Loading . . . 114
ES6 Modules . . . 114
Exporting . . . 114
Importing . . . 115
Aliases . . . 115
Alternative export syntax . . . 116
Default exports . . . 117
Summary . . . 117
Listing . . . 117

Types . . . 118
Transpile-time type checking . . . 118
Supported Types . . . 119
Generics . . . 121
Optional Types . . . 123
Type Inference. . . . 123
Types for external libraries . . . 123
Summary . . . 124
Listing . . . 124
Wrapping Up . . . 127
Further Reading . . . 127
Angular CLI . . . 128
Angular CLI . . . 129
Features . . . 129
Installing the Angular CLI . . . 130
Start an application with `ng new` . . . 130
Serve an application with `ng serve` . . . 137
Generate code with `ng generate` . . . 137
Create a build with `ng build` . . . 139
Testing Angular . . . 142
Summary . . . 143
Activity . . . 144
Steps . . . 144
Solution . . . 144
Components . . . 145
Overview . . . 146
Architecting with Components . . . 147
Learning Objectives . . . 147
Process . . . 147
Data Flow . . . 148
Summary . . . 149
Templates, Styles & View Encapsulation . . . 150
Learning Objectives . . . 150
templateURL . . . 150
styles . . . 150
View Encapsulation . . . 152
styleURLs . . . 158
Depreciated Properties . . . 159
Summary . . . 159
Listing . . . 159
Content Projection . . . 164

Goals . . . 164
Learning Objectives . . . 164
Motivation . . . 164
Content projection . . . 164
Multi-content projection . . . 165
Summary . . . 167
Listing . . . 167
Lifecycle Hooks . . . 171
Learning Objectives . . . 171
Phases . . . 171
Adding hooks . . . 172
Detecting what has changed . . . 177
Interfaces . . . 178
Summary . . . 179
Listing . . . 179
ViewChildren & ContentChildren . . . 184
Learning Objectives . . . 184
Example application . . . 184
ViewChild . . . 186
ViewChildren . . . 188
ViewChild referencing a template local variable . . . 189
ContentChild & ContentChildren . . . 191
Summary . . . 192
Listing . . . 192
Wrapping Up . . . 196
Activity . . . 197
Steps . . . 197
Solution . . . 197
Built-in Directives . . . 198
Overview . . . 199
NgFor . . . 200
Learning Objectives . . . 200
Basics . . . 200
Index . . . 202
Grouping . . . 203
Summary . . . 205
Listing . . . 205
NgIf & NgSwitch . . . 209
Learning Objectives . . . 209
NgIf . . . 209
NgSwitch . . . 211

Summary . . . 214
Listing . . . 214
NgStyle & NgClass . . . 218
Learning Objectives . . . 218
NgStyle . . . 218
NgClass . . . 222
Summary . . . 225
Listing . . . 225
NgNonBindable . . . 229
Learning Objectives . . . 229
Description . . . 229
Summary . . . 230
Listing . . . 230
Structural Directives . . . 232
Learning Objectives . . . 232
Long form structural directives . . . 232
Syntax sugar and * . . . 233
Summary . . . 233
Listing . . . 234
Wrapping Up . . . 238
Activity . . . 239
Steps . . . 240
Solution . . . 240
Custom Directives . . . 241
Overview . . . 242
Creating a custom directive . . . 243
Learning Objectives . . . 243
Directive decorator . . . 243
Attribute selector . . . 243
Directive constructor . . . 244
Summary . . . 246
Listing . . . 247
HostListener & HostBinding . . . 250
Learning Objectives . . . 250
HostListener . . . 250
HostBinding . . . 252
Summary . . . 254
Listing . . . 254
Inputs & Configuration . . . 258
Learning Objectives . . . 258
Configuration . . . 258

Summary . . . 260
Listing . . . 260
Wrapping Up . . . 264
Activity . . . 265
Steps . . . 265
Solution . . . 265
Reactive Programming with RxJS . . . 266
Overview . . . 267
Streams & Reactive Programming . . . 268
Learning Objectives . . . 268
What are Streams? . . . 268
What is Reactive Programming? . . . 269
Summary . . . 272
Observables & RxJS . . . 273
Learning Objectives . . . 273
Observables . . . 273
RxJS . . . 273
Other operators . . . 276
Summary . . . 278
Listing . . . 278
RxJS & Angular . . . 280
Learning Objectives . . . 280
Angular observables . . . 280
Reactive form example . . . 280
Not using observables . . . 286
Summary . . . 287
Listing . . . 287
Wrapping Up . . . 291
Activity . . . 292
Steps . . . 292
Solution . . . 292
Pipes . . . 293
Overview . . . 294
Built-in Pipes . . . 295
Learning Objectives . . . 295
Pipes provided by Angular . . . 295
Summary . . . 305
Listing . . . 306
Async Pipe . . . 310
Learning Objectives . . . 310
Overview . . . 310

Summary . . . 314
Listing . . . 315
Custom Pipes . . . 317
Learning Objectives . . . 317
Pipe decorator . . . 317
Transform function . . . 317
Multiple parameters . . . 319
Summary . . . 320
Listing . . . 320
Wrapping Up . . . 322
Activity . . . 323
Steps . . . 323
Solution . . . 323
Forms . . . 324
Overview . . . 325
Model Driven Forms . . . 326
Learning Objectives . . . 326
Form setup . . . 326
Form model . . . 329
Linking the form model to the form template . . . 330
Summary . . . 334
Listing . . . 334
Model Driven Form Validation . . . 338
Learning Objectives . . . 338
Validators . . . 338
Form control state . . . 339
Validation styling . . . 340
Validation messages . . . 344
Summary . . . 346
Listing . . . 346
Submitting & Resetting . . . 351
Learning Objectives . . . 351
Submitting . . . 351
Resetting . . . 352
Summary . . . 352
Listing . . . 352
Reactive Model Form . . . 358
Learning Objectives . . . 358
Setting up a reactive form . . . 358
React to changes in our form . . . 359
Summary . . . 362

Listing . . . 362
Template Driven Forms . . . 365
Learning Objectives . . . 365
Overview . . . 365
Form setup . . . 365
Directives . . . 366
Two way data binding . . . 369
Domain model . . . 370
Validation . . . 370
Submitting the form . . . 374
Resetting the form . . . 374
Summary . . . 376
Listing . . . 376
Wrapping Up . . . 381
Activity . . . 382
Steps . . . 382
Solution . . . 382
Dependency Injection & Providers . . . 383
Overview . . . 384
Components . . . 387
Injectors . . . 388
Learning Objectives . . . 388
Creating Injectors . . . 388
Dependency caching . . . 389
Child Injectors . . . 389
Summary . . . 390
Listing . . . 391
Provider . . . 393
Learning Objectives . . . 393
Providers . . . 393
Summary . . . 397
Listing . . . 397
Tokens . . . 401
Learning Objectives . . . 401
Summary . . . 404
Listing . . . 404
Configuring Dependency Injection in Angular . . . 406
Learning Objectives . . . 406
The Injector Tree . . . 406
Using Dependency Injection in Angular . . . 408
Summary . . . 411

Listing . . . 411
NgModule.providers vs Component.providers vs Component.viewProviders . . . 414
Learning Objectives . . . 414
Setup . . . 414
NgModule.providers . . . 417
Component.providers . . . 419
Component.viewProviders . . . 420
Summary . . . 422
Listing . . . 423
Wrapping Up . . . 426
Activity . . . 427
Steps . . . 427
Solution . . . 427
HTTP . . . 428
Overview . . . 429
Core HTTP API . . . 430
Learning Objectives . . . 430
Providing the Http dependencies . . . 430
Demo API & sample app . . . 431
HTTP Verbs . . . 433
Promises . . . 437
Handling errors . . . 437
Headers . . . 439
Summary . . . 440
Listing . . . 441
HTTP Example with Promises . . . 445
Learning Objectives . . . 445
Cross Origin Resource Sharing . . . 445
Creating the app component . . . 448
Creating a search service . . . 448
Using promises . . . 449
Consuming our search service . . . 451
Adding a loading indicator . . . 454
Using a domain model . . . 455
Summary . . . 458
Listing . . . 458
HTTP Example with Observables . . . 461
Learning Objectives . . . 461
Returning an Observable from the service . . . 461
Using the `async` pipe . . . 463
Observables all the way down . . . 464

Summary . . . 470
Listing . . . 471
JSONP Example with Observables . . . 474
Learning Objectives . . . 474
What is JSONP? . . . 474
Refactor to JSONP . . . 475
Summary . . . 477
Listing . . . 477
Wrapping Up . . . 480
Activity . . . 481
Steps . . . 481
Solution . . . 481
Routing . . . 482
Overview . . . 483
Route Configuration . . . 486
Learning Objectives . . . 486
Components . . . 486
Routes & RouterModule . . . 488
RouterOutlet Directive . . . 489
Redirects . . . 490
Catch all route . . . 490
Summary . . . 491
Listing . . . 491
Navigation . . . 495
Learning Objectives . . . 495
Navigating by hardcoded URLS . . . 495
Navigating programmatically via the router . . . 495
Navigating via a routerLink directive . . . 497
Summary . . . 499
Listing . . . 499
Parameterised Routes . . . 504
Learning Objectives . . . 504
Configuration . . . 504
Activated route . . . 505
Example . . . 505
Summary . . . 509
Listing . . . 509
Nested Routes . . . 514
Learning Objectives . . . 514
Goal . . . 514
Setup . . . 517

Route Configuration . . . 518
Relative routes . . . 518
Child routes . . . 519
Parent route params . . . 522
Summary . . . 523
Listing . . . 523
Router Guards . . . 531
Learning Objectives . . . 531
Guard Types . . . 531
CanActivate . . . 532
CanActivateChild . . . 535
Guard function parameters . . . 536
CanDeactivate . . . 536
Summary . . . 538
Listing . . . 538
Routing Strategies . . . 547
Learning Objectives . . . 547
HashLocationStrategy . . . 547
PathLocationStrategy . . . 548
Summary . . . 550
Wrapping Up . . . 551
Activity . . . 552
Steps . . . 552
Solution . . . 553
Unit Testing . . . 554
Overview . . . 555
Jasmine & Karma . . . 556
Learning Objectives . . . 556
Jasmine . . . 556
Karma . . . 561
Angular CLI . . . 562
Angular Plunker . . . 562
Summary . . . 564
Listing . . . 564
Testing Classes & Pipes . . . 566
Learning Objectives . . . 566
Sample class & test suite . . . 566
Setup & teardown . . . 566
Creating test specs . . . 567
Running the tests . . . 568
Pipes . . . 568

Summary . . . 570
Listing . . . 570
Testing with Mocks & Spies . . . 574
Learning Objectives . . . 574
Sample code . . . 574
Testing with the real `AuthService` . . . 575
Mocking with fake classes . . . 576
Mocking by overriding functions . . . 578
Mock by using a real instance with Spy . . . 578
Summary . . . 580
Listing . . . 580
Angular Test Bed . . . 582
Learning Objectives . . . 582
Configuring . . . 582
Fixtures and DI . . . 583
Test specs . . . 583
When to use ATB . . . 584
Summary . . . 584
Listing . . . 584
Testing Change Detection . . . 586
Learning Objectives . . . 586
Setup . . . 586
Detect Changes . . . 588
Summary . . . 589
Listing . . . 589
Testing Asynchronous Code . . . 592
Learning Objectives . . . 592
Test setup . . . 592
No asynchronous handling . . . 593
Jasmines `done` function . . . 594
`async` and `whenStable` . . . 595
`fakeAsync` and `tick` . . . 596
Summary . . . 596
Listing . . . 597
Testing Dependency Injection . . . 601
Learning Objectives . . . 601
Resolving via `TestBed` . . . 601
Resolving via the `inject` function . . . 601
Overriding the components providers . . . 602
Resolving via the component injector . . . 602
Summary . . . 603

Listing . . . 603
Testing Components . . . 605
Learning Objectives . . . 605
Test setup . . . 605
Testing `@Inputs` . . . 607
Testing `@Outputs` . . . 608
Summary . . . 609
Listing . . . 609
Testing Directives . . . 613
Learning Objectives . . . 613
Test setup . . . 613
Test wrapper component . . . 614
Interacting and inspecting the the view . . . 615
Summary . . . 615
Listing . . . 615
Testing Model Driven Forms . . . 618
Learning Objectives . . . 618
Test setup . . . 618
Form validity . . . 620
Field validity . . . 621
Submitting a form . . . 622
Summary . . . 622
Listing . . . 623
Testing Http . . . 627
Learning Objectives . . . 627
Test setup . . . 627
Using the `MockBackend` to simulate a response . . . 630
Testing the response . . . 631
Summary . . . 632
Listing . . . 633
Testing Routing . . . 637
Learning Objectives . . . 637
Test setup . . . 637
Router setup . . . 638
Testing routing . . . 639
Summary . . . 640
Listing . . . 640
Wrapping Up . . . 643
Advanced Topics . . . 644
Custom Form Validators . . . 645
Learning Objectives . . . 645

Built in validators . . . 645
Custom model form validator . . . 646
Custom template driven form validator . . . 648
Summary . . . 650
Model Driven Listing . . . 650
Template Driven Listing . . . 655
Configurable Custom Form Validators . . . 660
Learning Objectives . . . 660
Configurable model driven validators . . . 660
Configurable template driven validators . . . 661
Bindable template driven validators . . . 664
Summary . . . 665
Model Driven Listing . . . 666
Template Driven Listing . . . 670

Copyright

Copyright © 2016 Daolrevo Ltd trading as Code Craft

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

codecraft.tv

Changelog

Versions

1.2.0 (2017-11-24)

- Sample code and plunkers updated to Angular 5.0.4.
- The signatute and usage of `CurrencyPipe` changed, the book has been updated to the latest version.
- `ng-template` has been depreciated, we now use `template` instead.
- Fixed error in the description of the `useExisting` provider type, this **does** return the same instance when requested.
- The plunker on page 35 pointed to the wrong plunker, this has been fixed.
- Various typos were corrected.

1.1.0 (2017-04-27)

- Sample code and plunkers updated to Angular 4.1.0
- Fixed error in course with `useExisting` parameter in Dependency Injection, the code and course materials now explain the concept correctly.
- `OpaqueToken` was marked as depreciated in v4, you can still use it but in v5 it will be dropped from the framework. Updated the course and code to use the new `InjectionToken` instead.

1.0.1 (2017-01-05)

- Sample code and plunkers updated to Angular 2.4.1
- Added practice activities to (most) sections.
- Bug fix for child routing: empty child paths now also require a `pathMatch: 'full'` param to the route config.
- Various spelling mistakes and typos corrected.

1.0.0 (2016-11-16)

- First Release supporting Angular 2.1.0

About

This book has been funded in large part through the generous support of my backers through kickstarter and is therefore released for **FREE**.

Please ensure you have the latest version by visiting angular.codecraft.tv.

This book is also available as a series of videos! Again please check angular.codecraft.tv for details.

Cheers,

Asim Hussain

@jawache

Quickstart

Overview

The course is going to begin with a quickstart.

We use a web editor called plunker and create a basic Angular application from scratch.

In this quickstart you get a 50,000 foot view of the major features of Angular and since we use a web editor it means you can get stuck in writing code ASAP without having to spend ages setting up your computer.

Let's get started!

Plunker

Later on I'll be showing you how to install Angular and run the required development tools locally on your computer.

But for now we are going to use an online editor called plunker.

The goal of this chapter is to explain how plunker works and the different files that make up an Angular plunker.

Learning Outcomes

- Understand what plunker is, why and how we use it.
- Understand the structure of an Angular plunker.
- Know what each of the included libraries do.

What is plunker and how to use it?

- It's a web based editor and development environment. With it you can create, edit and run HTML, css and JavaScript files directly from your browser.
- No setup required, everyone can code in the browser instantly.
- Each plunker has it's own unique URL which you can share with others so it's a useful way to show others your work.
- You *can't* edit someone else's plunker but you can *fork* it. Forking creates a new plunker that you own with all the files copied across.
- If your plunker has an `index.html` file pressing the *Run* button will preview the plunker in the preview pane.

Structure of an Angular Plunker

An Angular plunker is composed of:

index.html

The main HTML file which includes the required libraries and bootstraps our Angular application

script.ts

The main file in which we'll be placing our Angular code

system.config.js

Configuration for SystemJS which handles module loading (*) and compilation of TypeScript into JavaScript

tsconfig.json

Configuration for the TypeScript transpiler (*)

(*) We will be covering these topics in more detail later.

In our index.html we are including 4 javascript files, like so:

```
<script src="https://unpkg.com/core-js/client/shim.min.js"></script>
<script src="https://unpkg.com/zone.js@0.6.21?main=browser"></script>
<script src="https://unpkg.com/reflect-metadata@0.1.3"></script>
<script src="https://unpkg.com/systemjs@0.19.27/dist/system.src.js"></script>
```

core-js

The version of javascript that has the broadest support across all browsers is ES5.

Then next version of javascript is ES6. ES6 does not have full support across *all* browsers yet. (See https://kangax.github.io/compat-table/es6/).

This library enables a browser which doesn't have support for ES6 to run ES6 code, just maybe not in the most efficient way

For further details take a look at the project homepage: https://github.com/zloirock/core-js

zone

"Zone is a mechanism for intercepting and keeping track of asynchronous work" - https://github.com/angular/zone.js

You don't need to know details about this yet, it's an advanced topic, but to summarise.

One of the problems with Angular 1 was that Angular didn't know when things happened outside of Angular, for example in asynchronous callbacks.

When Angular 1 knew about callbacks it could do wonderful things, like automatically update the page. When it didn't know about callbacks it frustratingly didn't update the page.

No matter how experienced you were with Angular 1 bugs with these kinds of issues cropped up time and time again.

Zones solves this problem, it keeps track of all *pending* asynchronous work and tells Angular when things happen. So in Angular you don't have to worry about whether Angular knows about your callback or not, zones tracks this for you and notifies Angular when something happens.

reflect

Angular is written in TypeScript, this file lets us use a feature of TypeScript called annotations (or decorators).

You'll learn a lot more about annotations later on in this course.

SystemJS

Instead of including all the files we need as script tags in our index.html, in Angular we break up all our code into files called *modules*. We then leave it to a *module loader* to load the *modules* when they are needed, in the order they are needed.

It's a complicated problem, in a browser we can't make hundreds of requests to load JavaScript modules one at a time when a module loader requests them so a module loader needs to be clever.

It will become part of the core JavaScript language but until then we use a *module loader*, there are several available but the one we use in our plunker is SystemJS.

If we build an application locally with the Angular command line tools it will use another module loader called Webpack.

systemjs.config.js

SystemJS needs help to figure out when and how to load up certain modules, e.g. if we ask for a module called `@angular` what does that mean? What should it load up? Which version? This configuration is stored in the `systemjs.config.js` file.

System.import

Now we've loaded up the SystemJS library and configured it by loading up the `systemjs.config.js` file, we can use the `System.import` function to load up our `script.ts` file, like so:

```
System.import('script.ts').catch(function(err) {
  console.error(err);
});
```

Why not just add script.ts as a script tag?:

```
<script src="script.ts"></script>
```

Because in `script.ts` we include other modules, if we loaded via a script tag the browser doesn't know how to load those other dependant js files.

By loading with `System.import` we make SystemJS figure out which dependant modules are required and loads the files for those automatically.

Summary

We can code up Angular in the browser using an online editor called plunker. It gives us the ability to try our Angular quickly without requiring complex setup.

It also gives us a unique URL so:

1. We can quickly take a look at some code another person has written.
2. We can share our code with other people, which is especially useful when we are stuck with some broken code and need help.

Listing

http://plnkr.co/edit/NzQ1skgIrliMIGgEPkp8?p=preview

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.6.23?main=browser"></script>
  <script src="https://unpkg.com/reflect-metadata@0.1.3"></script>
  <script src="https://unpkg.com/systemjs@0.19.27/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body>
</body>
</html>
```

Intro to TypeScript

In this lecture we will explain what TypeScript is and how we can code a web application using TypeScript.

Learning Outcomes

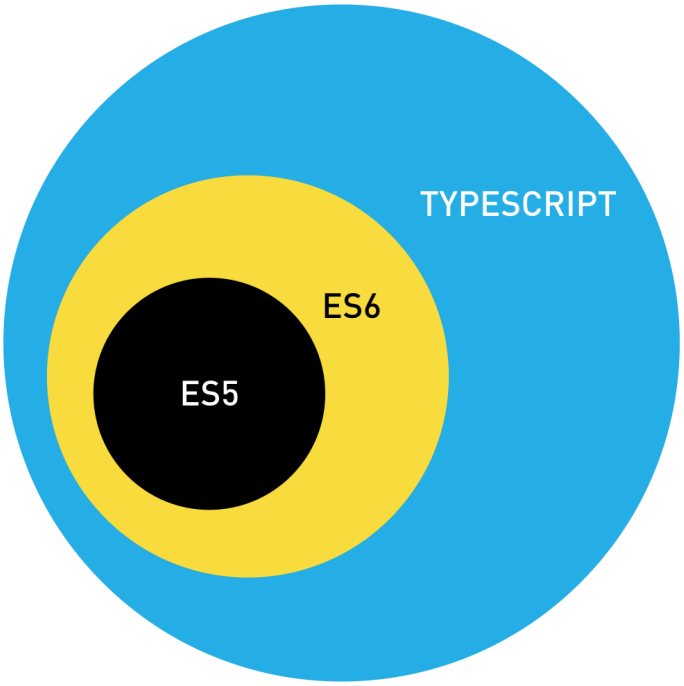
- Difference between TypeScript and JavaScript.
- How we convert from TypeScript into JavaScript.

TypeScript vs JavaScript

From the previous lecture we named the file we are going to type our code into `script.ts`

The reason the file ends in `.ts` instead of `.js` is that Angular is written in a superset of JavaScript called TypeScript.

TypeScript is the ES6 version of JavaScript plus a few other TypeScript only features which Angular needs in order to work.



You can write Angular applications in either *TypeScript*, *ES6* or even *ES5 JavaScript*.

However Angular itself is written in TypeScript, most examples on the web are written in TypeScript, most Angular jobs require you to write TypeScript so this book will be teaching in TypeScript.

Transpilation

Browsers don't support TypeScript. Browsers *barely* support ES6 JavaScript. So how can we write our code in TypeScript?

We use something called a *transpiler* which converts from one language to another.

We can write in TypeScript and have a transpiler convert to ES6 or ES5.

To understand why is it called *transpilation* and not *compilation* in more detail see https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/

Since most browsers don't support ES6 features yet we are going to *transpile* our TypeScript into ES5.

Later on in the book I'm going to show you how to transpile TypeScript into JavaScript locally on your computer, but since we are using plunker we are using a feature of SystemJS which lets us transpile in the browser.

If we look at the `tsconfig.json` file in our demo plunker we can see there are a few settings we are using to convert TypeScript into JavaScript.

```
{
  "compilerOptions": {
    "target": "es5", ①
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

① As you can see the target is set to ES5.

Summary

- TypeScript is just JavaScript with a few more advanced features.
- Browser can't run TypeScript so we first need to transpile it into JavaScript.
- The most common version of JavaScript is currently ES5 so we transpile TypeScript into ES5 JavaScript.

Writing our first app

We want to create a simple app that displays a joke to the user.

Learning Outcomes

- That *Components* are the building blocks of an Angular application.
- What annotations are and how to use them in TypeScript.
- How to import code from other files so we can use it in our file.
- How to package our application into an Angular Module.
- How to bootstrap an Angular application so is starts on a web page

Components

To begin writing our application we open up `script.ts` and create a class called `JokeComponent`.

```
class JokeComponent {
}
```

A `class` is a new feature of ES6 which we will explain in much more detail in the next section, but to summarise it's a blueprint for creating objects with specific functions and properties already attached to it.

The word `Component` isn't random. Components are a feature of Angular that let us create a new HTML *language* and they are how we structure Angular applications.

HTML comes with a bunch of pre-built tags like `<input>` and `<form>` which look and behave a certain way. In Angular we create new *custom* tags with their own look and behaviour.

An Angular application is therefore just a set of custom tags that interact with each other, we call these tags *Components*.

If you are coming from Angular 1 then Components are the same as Components in Angular 1.5 and the same as element directives in Angular <1.5

The code that controls a component we put into a class like the `JokeComponent` above, but how do we *link* this class with a HTML tag, say a tag called `<joke>`?

We use a new feature of TypeScript called *annotations*, and specifically an annotation called `@Component`, like so:

```
@Component({
  selector: 'joke'
})
class JokeComponent {
}
```

The `@Component` is an annotation, an annotation automatically adds some boilerplate code to the class, function or property its attached to.

You can write Angular without using annotations you would just have to write the boilerplate code yourself.

We are going to use other annotations later on, however the main one for working with components is `@Component`.

You can configure the @Component annotation by passing it an object with various parameters. In our example above @Component has one parameter called `selector`, this tells Angular which tag to link this class too.

By setting the selector to `joke` we've told angular that whenever it finds a tag in the HTML like `<joke></joke>` to use an instance of the JokeComponent class to control it.

Imports

Before we can use `@Component` though we need to import it, like so:

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

The line above is saying we want to import the `Component` code from the module `@angular/core`.

We leave to SystemJS to figure out *how* to load that component from @angular/core or even *where* @angular/core is.

The above might not look like javascript but it is, the `{ Component }` part is something called destructuring and that's a feature of ES6, more on that later.

If you are coming from a language like Python or Java you'll be used to the concept of imports. Basically we are pulling in dependencies from another file and making it available in this file.

Template

To use our brand new custom component we add the tag `<joke></joke>` to our HTML file, like so:

```
<body>
  <joke></joke>
</body>
```

So far this isn't doing much yet though, we want Angular to replace <joke></joke> with some template HTML. To do that we use another attribute of the Component decorator called template, like so:

```
@Component({
  selector: 'joke',
  template: '<h1>What did the cheese say when it looked in the mirror?</h1><p>Halloumi
(hello me)</p>'
})
```

That's hard to read though, the HTML is all written on one line but i'd like to read it on multiple lines.

There is a new feature of ES6 JavaScript called `template strings` which lets us define multi-line strings, lets use it:

```
@Component({
  selector: 'joke',
  template: `
  <h1>What did the cheese say when it looked in the mirror?</h1>
  <p>Halloumi (hello me)</p>
  `
})
```

The string uses a special character `` ` `` it's called a back-tick, we'll be digging into this in much more detail in the next section. For now just accept that it lets us define strings on multiple lines like the above.

Angular Modules

If we ran this code now we would see it's still not working!

We've defined a component with a custom tag, added the tag to our HTML but we haven't told Angular that we want to use Angular on this page.

To do that we need to do something called *bootstrapping*.

In Angular 1 when we added `np-app="module-name"` to the top of the HTML page it bootstrapped the application for us. When Angular 1 loaded it first checked for this tag, looked for the module that was associated with that tag and loaded the code from it. However with Angular we need to do all of this manually, for good reasons which we'll explain later.

In Angular your code is structured into `packages` called Angular Modules, or `NgModules` for short. Every app requires at least one module, the root module, that we call `AppModule` by convention.

We are using the term *module* for two different concepts. In JavaScript the term module generally refers to code which exists in a single file. An NgModule is a different concept, it combines code from different files together into one package. An NgModule therefore contains functionality from multiple files a module refers to functionality in a single file.

Lets create our root Angular Module, like so:

```
@NgModule({
  imports: [BrowserModule],
  declarations: [JokeComponent],
  bootstrap: [JokeComponent]
})
export class AppModule {
}
```

To define an Angular Module we first create a class and then annotate it with a decorator called `@NgModule`.

You'll notice this follows a similar pattern to when we created a component. We created a class and then annotated it with a decorator, this pattern is a common one in Angular.

`@NgModule` has a few params:

imports

The other Angular Modules that export material we need in this Angular Module. Almost every application's root module should import the BrowserModule.

declarations

The list of components or directives belonging to *this* module.

bootstrap

Identifies the root component that Angular should bootstrap when it starts the application.

We know `NgModule` but `BrowserModule` is the Angular Module that contains all the needed Angular bits and pieces to run our application in the browser.

Angular itself is split into separate Angular Modules so we only need to import the ones we really use. Some other common modules you'll see in the future are the `FormsModule`, `RouterModule` and `HttpModule`.

We also need to remember to import NgModule and BrowserModule, like so:

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

Bootstrapping

Now we have defined our root Angular Module called AppModule we need to bootstrap the application using it, like so:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
.
.
.
platformBrowserDynamic().bootstrapModule(AppModule);
```

You might be asking yourself why Angular doesn't do this for us like it did in Angular 1?

In Angular bootstrapping is *platform specific*.

Angular 1 assumed that Angular would only ever be run in a browser, Angular makes no such assumption. We could be writing Angular code for a mobile device using a solution like Ionic. We could be loading up Angular on a node server so we can render HTML for web crawlers that don't run JavaScript.

Angular isn't limited to only working in the browser which is why we need to tell Angular exactly *how* we want it to bootstrap itself, in our case we are running in the browser so we use the platformBrowserDynamic function to bootstrap our application.

Component Tree

An Angular application is architected as a tree of Components stemming from one root Component.

Your root component is the component you configured on your root `NgModule` in the bootstrap property, so in our case it's the `JokeComponent`.

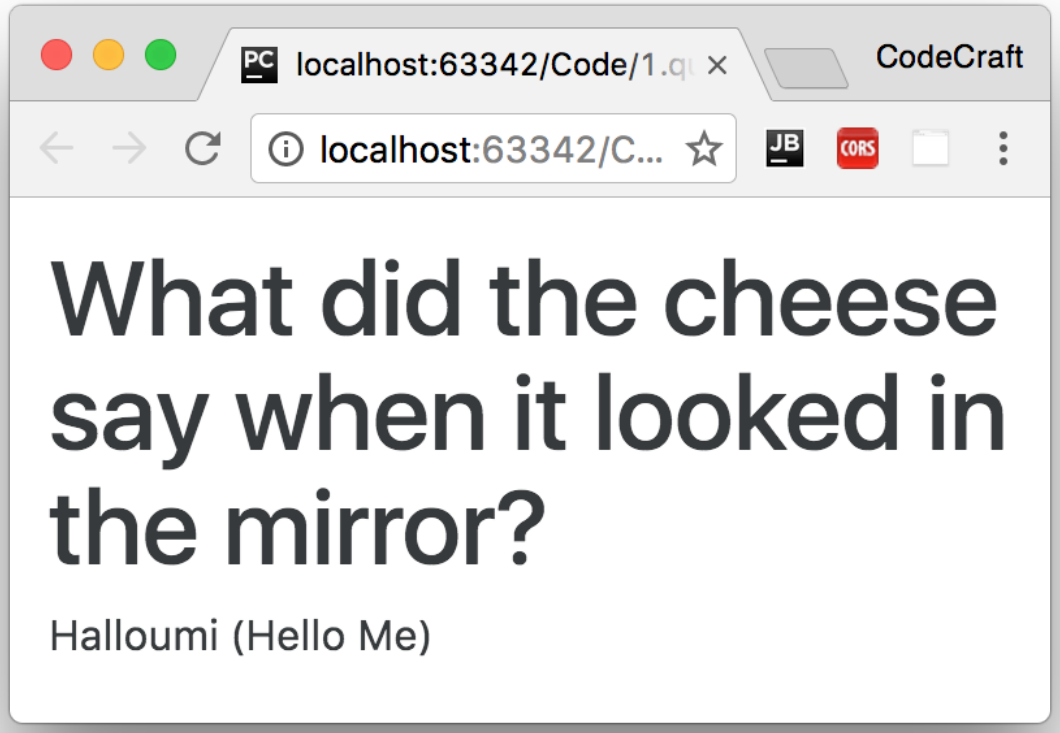
By bootstrapping with `JokeComponent` we are saying that it's the root component for our application.

In the template for our `JokeComponent` we would add tags for other Components, in the template for those Components we would add tags for others... and so on and so on.

However in our `index.html` file we will never see anything other than the tag for our root component, like so:

```
<body>
  <joke></joke>
</body>
```

Running the application in our browser we would see:



Troubleshooting

If when looking at the browser console you see see an error like the below:

```
> The selector "joke" did not match any elements
```

This means you forgot to add the tag for your root component to your `index.html` file.

Summary

A *Component* is the building block of an Angular application.

It lets us create a new HTML language of custom tags and link them with javascript classes which describe the behaviour of that tag.

An application is composed of a tree of such Components glued together all depending from one root component.

We package together related Components and supporting code into something called an Angular Module which we use to bootstrap Angular onto a webpage.

Listing

http://plnkr.co/edit/X1PpIFrTgBZb22YsvLeo?p=preview

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body class="container m-t-1">
<joke></joke>
</body>
</html>
```

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

@Component({
  selector: 'joke',
  template: `
    <h1>What did the cheese say when it looked in the mirror?</h1>
    <p>Halloumi (Hello Me)</p>
  `
})
class JokeComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [JokeComponent],
  bootstrap: [JokeComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

String Interpolation

Start to make our application a bit more re-usable by storing the setup and punchline of our joke as properties on our component class instead of hardcoded in the HTML.

Learning Outcomes

- Understand classes in more detail, how to define them and how to use them.
- What is string interpolation and the moustache syntax `{{ }}`

Motivation

In the previous chapter we created our first Angular application, a very simple one with only one component called `JokeComponent` with a tag of `joke`.

Now whenever we want to display that joke in our application we simply add the tag `<joke></joke>` in our HTML.

But it's not very re-usable, it just shows the same joke over and over again, a more re-usable `JokeComponent` would be one where the developer can use different jokes.

Firstly lets add some properties or our class, like so:

```
class JokeComponent {
  setup: string;
  punchline: string;
}
```

We are saying that this class has two properties, `setup` and `punchline`, both of them can only hold `strings`.

The code `:string` is something called a type, and it's a core part of TypeScript, something you probably are not used to if you've never worked with typed languages before.

We will discuss Types in more detail in the next section but in summary if we ever try to make these properties hold anything other than a string TypeScript will throw an error.

Classes

Remember I said that classes are blueprints for objects, or in other words *instructions* for how to create an object. To actually create an object using a `class` we use the `new` keyword, like so:

```
let joke = new JokeComponent()
```

`joke` is an object created using the `JokeComponent` class, another word for an object created using a class is a *class instance* or just *instance*.

Since we have added some properties to the `JokeComponent` class the class instance that is created also has those properties.

```
console.log(joke.setup);
console.log(joke.punchline);
```

If we really did try to run the above code we would see that nothing gets printed out to the console, that's because the `setup` and `punchline` properties have not been initialised, they are blank.

With classes how we initialise properties when we instantiate an object is via a special class function called a `constructor`.

```
class JokeComponent {
  setup: string;
  punchline: string;

  constructor() {
    this.setup = "What did the cheese say when it looked in the mirror?";
    this.punchline = "Halloumi (Hello Me)";
  }
}
```

Now when we instantiate the class the `constructor` function is called and this initialises the `setup` and `punchline` properties.

In the interest of brevity I'm initialising our properties in the constructor, the recommended approach with initialising a component is to use *Component Lifecycle Hooks*, again more on that later.

String Interpolation

Currently the setup and punchline is hardcoded into the HTML template of our `JokeComponent`. We need to have the template output the contents of our properties instead.

We can do that in the template by using the special `{{ }}` syntax, also known as *moustache syntax*.

The `{{ }}` contains JavaScript which is run by Angular and the output inserted in the HTML.

So if we put `{{ 1 + 1 }}` in the template the number `2` would be inserted into the HTML.

The template knows about the `JokeComponent` class it's attached to so in-between the `{{ }}` we can also read properties or even call functions on our `JokeComponent` and have the output inserted in the HTML.

We just want to display the values of the `setup` and `punchline` properties so we just use those, like so:

```
<h1>{{setup}}</h1>
<p>{{punchline}}</p>
```

Summary

We've explained how a class is a blueprint for an object and we can create a specific instance of a class using the `new` keyword.

The class instance can have properties and we can bind those properties to portions of our template by using string interpolation via the `{{ }}` syntax.

Listing

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body class="container m-t-1">
<joke></joke>
</body>
</html>
```

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

@Component({
  selector: 'joke',
  template: `
    <h1>{{setup}}</h1>
    <p>{{punchline}}</p>
  `
})
class JokeComponent {
  setup: string;
  punchline: string;

  constructor() {
    this.setup = "What did the cheese say when it looked in the mirror?";
    this.punchline = "Halloumi (Hello Me)";
  }
}

@NgModule({
  imports: [BrowserModule],
  declarations: [JokeComponent],
  bootstrap: [JokeComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Looping

Our goal in this lecture is to display a *list* of jokes instead of just one.

To add some visual jazz to our application we are going to be using the twitter bootstrap ui framework and specifically the *card* style for our `JokeComponent`.

Learning Outcomes

- Using Arrays in TypeScript.
- Using the `NgFor` directive to repeat an element.

JokeListComponent

We will create a new component called `JokeListComponent` with the following listing:

```
@Component({
  selector: 'joke-list',
  template: `
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text">{{joke.punchline}}</p>
</div>
  `
})
class JokeListComponent {
  jokes: Object[];

  constructor() {
    this.jokes = [
      {
        setup: "What did the cheese say when it looked in the mirror?",
        punchline: "Hello-Me (Halloumi)"
      },
      {
        setup: "What kind of cheese do you use to disguise a small horse?",
        punchline: "Mask-a-pony (Mascarpone)"
      },
      {
        setup: "A kid threw a lump of cheddar at me",
        punchline: "I thought 'That's not very mature'"
      },
    ];
  }
}
```

Arrays

The first change you'll notice is that we have a property called `jokes` and the type is `Object[]`.

The `[]` syntax in the type means *list of* or *Array*, so the jokes property holds a list of `Objects`.

Another perfectly legal way to write this would be `Array<Object>` but I prefer `Object[]` since for me it's easier to see the `[]` characters at a glance.

In the constructor we initialise this array with some hilarious cheese jokes.

Card Element

You might notice in the template we are using some classes called `card`, `card-block` etc... this is from twitter bootstrap and it's a style called a *card* which displays a rectangle with a border.

The basic HTML structure for a twitter bootstrap *card element* is like so:

```
<div class="card card-block">
 <h4 class="card-title"></h4>
 <p class="card-text"></p>
</div>
```

NgFor

We want to repeat this card element for each joke in our array of jokes.

So we add a special syntax called an `NgFor` on the card element, like so:

```
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title"></h4>
  <p class="card-text"></p>
</div>
```

`*ngFor="let joke of jokes"` will create a new HTML element, using the div element it's attached to as a template, for every joke in the jokes array.

It will also make available to the element a variable called `joke` which is he item in the joke array it's currently looping over.

The syntax translates to `let <name-i-want-to-call-each-item> of <array-property-on-component>`

This is what we call in Angular a *Directive*. Specifically it's a *structural directive* since it changes the structure of the DOM. We'll be going through more built-in directives later on and also you'll learn how to create your own.

So now we can display properties of this `joke` object in the HTML using `{{joke.setup}}` and

`{{joke.punchline}}`, like so:

```
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text">{{joke.punchline}}</p>
</div>
```

If you've worked with Angular 1 before, you probably used the `ng-repeat` directive. NgFor is the analogous directive In Angular. Its syntax is slightly different but they have the same purpose.

Configuring

In order to use our `JokeListComponent` we need to add it to the declarations on our `NgModule` and also mark it as the component we want to bootstrap the page with.

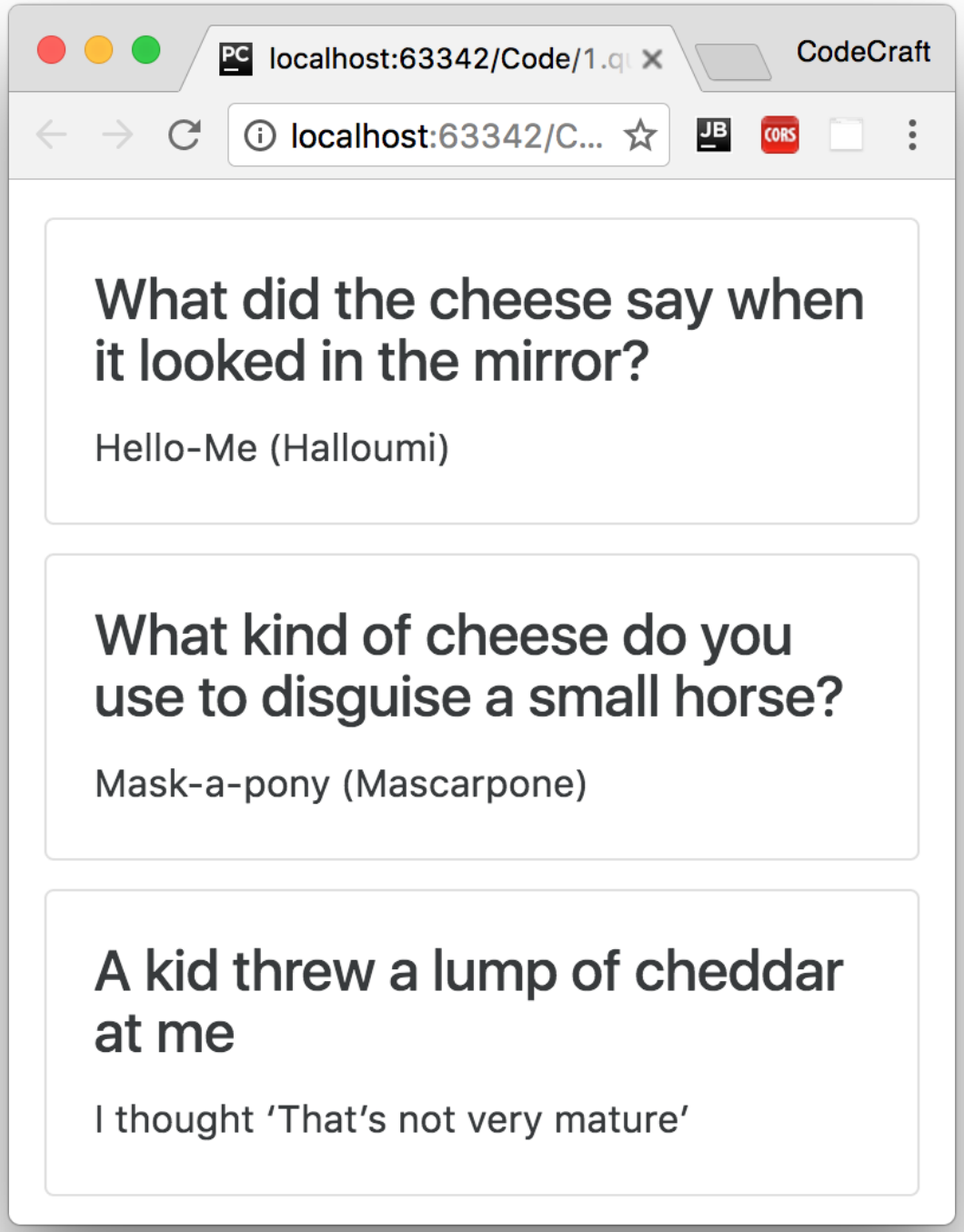
```
@NgModule({
  imports:[BrowserModule],
  declarations: [JokeComponent, JokeListComponent],
  bootstrap: [JokeListComponent]
})
```

Since we are now bootstrapping `JokeListComponent` and it's selector is `joke-list` we also need to change the root tag in our `index.html`, like so:

```
<body class="container m-t-1">
  <joke-list></joke-list>
</body>
```

The classes `container` and `m-t-1` are from twitter bootstrap and add some nice visual padding to the page.

Now if we run the application we see multiple jokes printed to the screen, like so:



Summary

When we declare an array in TypeScript we also tell it what *Type* of thing the array holds using `Type[]` or the `Array<Type>` syntax.

We can repeat the same element multiple times in Angular using the `NgFor` directive

Listing

http://plnkr.co/edit/6BGJzWKFuPLFeBdKZ6z2?p=preview

Since we are now using the JokeListComponent as our root component, our root components tag has changed from `<joke></joke>` to `<joke-list></joke-list>`

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body class="container m-t-1">
<joke-list></joke-list>
</body>
</html>
```

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

@Component({
  selector: 'joke-list',
  template: `
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text">{{joke.punchline}}</p>
</div>
  `
})
class JokeListComponent {
  jokes: Object[];

  constructor() {
    this.jokes = [
      {
        setup: "What did the cheese say when it looked in the mirror?",
        punchline: "Hello-Me (Halloumi)"
      },
      {
        setup: "What kind of cheese do you use to disguise a small horse?",
        punchline: "Mask-a-pony (Mascarpone)"
      },
      {
        setup: "A kid threw a lump of cheddar at me",
        punchline: "I thought ‘That’s not very mature’"
      },
    ];
  }
}

@NgModule({
  imports: [BrowserModule],
  declarations: [JokeListComponent],
  bootstrap: [JokeListComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Property & Event Binding

Our goal in this lecture is to hide the punchline of our jokes and only reveal them when the user clicks a button.

Learning Outcomes

- How to use the *hidden* DOM property to hide/show elements.
- Difference between DOM properties and HTML attributes.
- How to perform input property binding with `[]`
- How to perform output event binding with `()`

Hiding and showing elements

We can hide any element by adding an *attribute* called `hidden` to the element in HTML, so we could hide the punchline like so:

```
<p class="card-text" hidden>{{joke.punchline}}</p>
```

This is a core feature of HTML, not some functionality provided by Angular.

We want to add the `hidden` property to the element by default and then remove it when the user clicks a button.

So we add the following markup:

```
<p class="card-text" [hidden]="true">{{joke.punchline}}</p>
```

Specifically we added the markup `[hidden]="true"` and again it works, the element is hidden.

A few things:

1. We wrapped the attribute with a `[]`, more on that later.
2. We made the attribute equal to `true`, if we made it equal to `false` it shows the element.

We say we have bound the value `true` to the property called `hidden`.

This is called **Input Property Binding** and it's a very important concept in Angular.

HTML Attribute vs DOM Property

The distinction between an HTML attribute and a DOM property is important in understanding binding in Angular.

HTML is a set of written instructions for *how* to display a web page.

The browser reads the HTML and creates something called a *DOM*, a *Document Object Model*. This is the manifestation of those HTML instructions in memory.

Changing the HTML doesn't automatically update the webpage unless the user refreshes the browser, changing the DOM however instantly updates the webpage.

There is mostly a 1-to-1 mapping between the names and values of HTML attributes and their equivalent DOM properties, but not always.

The hidden HTML attribute is a good example, it only needs to `exist` on an HTML element to instruct the browser to hide the element.

So `hidden="true"` hides the element but confusingly so does `hidden="false"` in HTML we just need to add `hidden` to hide the element.

The DOM representation of the `hidden` attribute is a property also called `hidden`, which if set to `true` hides the element and `false` shows the element.

Angular doesn't manipulate HTML attributes, it manipulates DOM properties because the *DOM* is what actually gets displayed.

So when we write `[hidden]` we are manipulating the *DOM property* and not the *HTML attribute*.

This is why the above is called Input *Property* Binding and not Input Attribute Binding.

Input Property Binding

Looking back at our use of the hidden property:

```
<p class="card-text" [hidden]="true">{{joke.punchline}}</p>
```

The *target* inside `[]` is the name of the property. In the example above the target is the `hidden` DOM property.

The *text* to the right of `=` is javascript code that gets executed and the resulting value is assigned to the target.

`true` is still javascript code which if executed returns `true`.

So in summary, we are binding to the DOM property `hidden` and setting it to `true` so the element is hidden.

In other parts of the web you'll see this referred to as just *property binding*. However to distinguish it from the other type of binding in Angular I like to call this **input** property binding.

We can only use this type of binding to change the value of the target. We can't use it to get *notified*

when the target's value changes, to do that we need to use something called *Output Event Binding*, more on that soon.

Let's add a property called `hide` on each joke and set it to `true`, like so:

```
[
  {
    setup: "What did the cheese say when it looked in the mirror?",
    punchline: "Hello-Me (Halloumi)",
    hide: true
  },
  {
    setup: "What kind of cheese do you use to disguise a small horse?",
    punchline: "Mask-a-pony (Mascarpone)",
    hide: true
  },
  {
    setup: "A kid threw a lump of cheddar at me",
    punchline: "I thought 'That's not very mature'",
    hide: true
  },
]
```

Now we can set the hidden input property to `joke.hide` in the template, like so:

```
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text"
     [hidden]="joke.hide">{{joke.punchline}}</p>
</div>
```

Output Event Binding

We want to show or hide the punchline when a user clicks a button, so lets add a button with the label *Tell Me* to the bottom of each card, like so:

```
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text"
     [hidden]="joke.hide">{{joke.punchline}}</p>
  <a class="btn btn-primary">Tell Me</a>
</div>
```

We want to set `joke.hide` to `false` when the user clicks the button, and then back to `true` again when they click the button a second time.

To have Angular call some code every time someone clicks on the button we add some special markup to our button:

```
<a class="btn btn-primary"
   (click)="joke.hide = !joke.hide">Tell Me
</a>
```

We have some new syntax with `()`. The *target* inside the `()` is an event we want to listen for, we are listening for the click event.

The *text* to the right of `=` is some javascript which will be called every time a click event occurs.

`joke.hide = !joke.hide` toggles the value of `joke.hide`, so if it's false clicking the button will change it to true, if it's true clicking it will change it to false.

We can just as easily make the expression to the right of `=` call a function on our component instead, like so:

```
<a class="btn btn-primary"
   (click)="toggle(joke)">Tell Me
</a>
```

Now when the button gets clicked it calls the `toggle(⋯)` function on the `JokeListComponent` and passes it the joke object the card is bound to, like so:

```
toggle(joke) {
  joke.hide = !joke.hide;
}
```

Now when we click the button, we set the show property to true which then *unhides* the element.

Summary

The way think about these two different ways of binding is in terms of inputs and outputs.

- With the `[]` we are binding to an input of a Component.
- With the `()` we are binding to an output of a Component.

This is what we call *one-way data binding*, since data only flows one way, either into or out of a component.

It is possible to *simulate* two-way data binding in Angular and we'll cover that in a later section on Forms.

Listing

http://plnkr.co/edit/6BGJzWKFuPLFeBdKZ6z2?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

@Component({
  selector: 'joke-list',
  template: `
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text">{{joke.punchline}}</p>
</div>
  `
})
class JokeListComponent {
  jokes: Object[];

  constructor() {
    this.jokes = [
      {
        setup: "What did the cheese say when it looked in the mirror?",
        punchline: "Hello-Me (Halloumi)"
      },
      {
        setup: "What kind of cheese do you use to disguise a small horse?",
        punchline: "Mask-a-pony (Mascarpone)"
      },
      {
        setup: "A kid threw a lump of cheddar at me",
        punchline: "I thought ‘That’s not very mature’"
      },
    ];
  }
}

@NgModule({
  imports: [BrowserModule],
  declarations: [JokeListComponent],
  bootstrap: [JokeListComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Domain Model

The goal of this lecture is to use a domain model to store our data instead of plain old objects.

Learning Outcomes

- Why domain models are better for storing application state.
- How to structure our application to use a domain model.

What are Domain Models?

A good practice when writing Angular code is to try to isolate the data structures you are using from the component code.

Right now our data structure is just an array of objects which we initialised on the component.

We also have this `toggle` function which doesn't do anything other than modify a property of the object passed in, i.e. the function could exist outside of the component completely and still do it's job just fine.

Imagine if this was a much larger application, if all the data was stored inside components as plain objects it would be hard for a new developer to find out *where* the data is stored and *which* function to edit.

To solve this, let's create a class that represents a single joke and attach the toggle function to that class.

This class is what we call a *Domain Model*, it's just a plain class which we will use to store data and functions.

Our Domain Model

We create a simple class with 3 properties, a constructor and a toggle function, like so:

```
class Joke {
  setup: string;
  punchline: string;
  hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}
```

As we've mentioned before we can instantiate a class by using the `new` keyword and when that happens javascript calls the `constructor` function where we can have code that initialises the properties.

However the constructors we've used so far have not had any arguments, the one for the joke class above does. It initialises the properties of the joke instance from the arguments passed into the constructor.

We can pass those arguments in when we instantiate a joke like so:

```
let joke = new Joke("What did the cheese say when it looked in the mirror?", "Hello-Me
(Halloumi)");
```

We also added a toggle function which just flips the value of the `hide` property.

We use `this` in a class function to distinguish between properties that are *owned* by the class instance vs. arguments that are passed in or declared locally by the function.

So `this.joke` in our toggle function is explicitly saying the `joke` property on the class instance.

Now lets change our `JokeListComponent` so it uses our domain model, like so:

```
class JokeListComponent {
  jokes: Joke[]; ①

  constructor() {
    this.jokes = [ ②
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not very
mature'"),
    ];
  }

  toggle(joke) {
    joke.hide = !joke.hide;
  }
}
```

① We changed the type of our `jokes` property to `Joke[]`. Now TypeScript will throw an error if we accidentally insert something other than an instance of a Joke class in this array.

② We also converted our `jokes` array to now contain instance of the `Joke` class instead of plain objects.

The final thing to do is to change the markup in the template so we call the `joke.toggle()` function on the joke instance and not the `toggle(joke)` function on the component, like so:

```
<a class="btn btn-warning" (click)="joke.toggle()">Tell Me</a>
```

Summary

Although its possible to store all your data as objects it's both recommend and simple to encapsulate your data into domain models in Angular.

Using a domain model clarifies for all developers where you should put functions that need to act on the data, like our toggle function.

Listing

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

class Joke {
```

```
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}

@Component({
  selector: 'joke-list',
  template: `
<div class="card card-block"
     *ngFor="let joke of jokes">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text"
     [hidden]="joke.hide">{{joke.punchline}}</p>
  <a class="btn btn-warning" (click)="joke.toggle()">Tell Me</a>
</div>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}

@NgModule({
  imports: [BrowserModule],
  declarations: [JokeListComponent],
  bootstrap: [JokeListComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Nesting Components & Inputs

An application in Angular is a set of custom components glued together in HTML via inputs and outputs.

So far we've only built applications with a single component, our goal now is to start building applications that are *composed* of multiple components working together.

Breaking up an application into multiple logical components makes it easier to:

- Architect an application as it grows in complexity.
- Re-use common components in multiple places.

The goal of this lecture is to break up our small application into 3 components and start *gluing* them together.

Learning Outcomes

- How to create and configure multiple components in one application.
- How to enable Input Property Binding on a custom component so components can communicate with each other.

Create & Configure Multiple Components

If you think of a typical webpage we can normally break it down into a set of logical components each with its own view, for example most webpages can be broken up into a header, footer and perhaps a sidebar.

We are going to break up our application into a root `AppComponent`, this component won't have any functionality and will just contain other components.

This component will hold our `JokeListComponent` and our `JokeListComponent` will hold a list of `JokeComponents`.

Most Angular apps will have a root component called *AppRoot* or *AppComponent*, this typically just acts as a container to hold other components.

Our components will therefore nest something like the below image:

For the *convenience* of learning we are going to keep everything in one file. When building Angular apps the recommended approach is to have one component per file.

JokeComponent

This looks similar to our previous `JokeListComponent` we just removed the `NgFor` since this component will now render a single joke.

```
@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">{{joke.setup}}</h4>
  <p class="card-text"
     [hidden]="joke.hide">{{joke.punchline}}</p>
  <a (click)="joke.toggle()"
     class="btn btn-warning">Tell Me
  </a>
</div>
  `
})
class JokeComponent {
  joke: Joke;
}
```

JokeListComponent

We've broken out a joke into its own `JokeComponent` so now we change the `JokeListComponent` template to contain multiple `JokeComponent` components instead.

```
@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not very
mature'"),
    ];
  }
}
```

AppComponent

Our final component is our new top level `AppComponent`, this just holds an instance of the `JokeListComponent`.

```
@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}
```

Configuring multiple Components

In order to use our new components we need to add them to the declarations on our `NgModule`.

And since we've changed our top level component we need to set that in the bootstrap property as well as change our `index.html` to use the `<app></app>` root component instead.

```
@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

```
<body class="container m-t-1">
  <app></app>
</body>
```

In Angular we need to be explicit regarding what components & directives are going to use in our Angular Module by either adding them to the `imports` or `declarations` property.

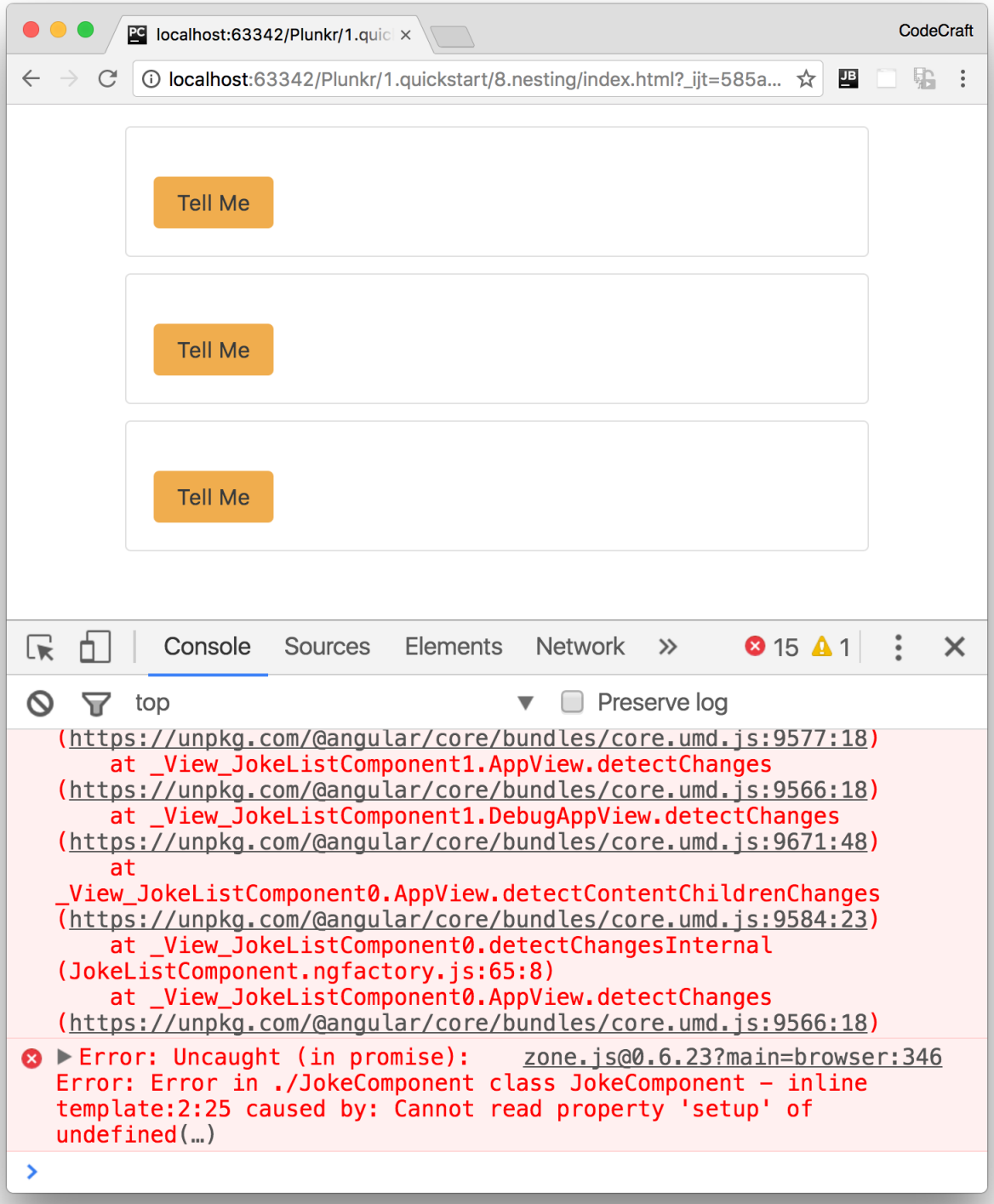
In Angular 1 each directive when added via a script tag was globally available, which made it convenient for smaller projects but a problem for larger ones. Issues like name clashes came up often as different third party libraries often used the same names.

With Angular two third party libraries can export the same name for components but only the version we explicitly include into our Angular Module will be used.

The built-in directives we are using such as `NgFor` are all defined in `CommonModule` which is again included in `BrowserModule` which we have already added to our `NgModule` imports.

Input Property Binding on a Custom Component

If we ran the application now we would see just some empty boxes with some errors in the console, like so:



The errors should read something like:

```
class JokeComponent - inline template:2:25 caused by: Cannot read property 'setup' of
undefined
```

The above should give you a hint about what's going on

1. It's something to do with the `JokeComponent`
2. It's something to do with the template.
3. It's something to do with the `setup` property.

So if we look at the offending part of our `JokeComponent` template:

```
<h4 class="card-title">{{joke.setup}}</h4>
```

Essentially `Cannot read property 'setup' of undefined` in the context of `joke.setup` means that `joke` is undefined, it's blank

If you remember from our `JokeComponent` class we do have a property called `joke`:

```
class JokeComponent {
  joke: Joke;
}
```

And we are looping and creating `JokeComponents` in our `JokeListComponent`, like so:

```
<joke *ngFor="let j of jokes"></joke>
```

But we are not setting the property `joke` of our `JokeComponent` to anything, which is why it's `undefined`.

Ideally we want to write something like this:

```
<joke *ngFor="let j of jokes" [joke]="j"></joke>
```

In just the same way as we bound to the `hidden` property of the `p` tag in the element above we want to bind to the `joke` property of our `JokeComponent`.

Even though our `JokeComponent` has a `joke` property we can't bind to it using the `[]` syntax, we need to explicitly mark it as an *Input* property on our `JokeComponent`.

We do this by pre-pending the `joke` property in the component with a new annotation called `@Input`, like so:

```
import { Input } from '@angular/core';
.
.
.
class JokeComponent {
  @Input() joke: Joke;
}
```

This tells Angular that the `joke` property is an *input* property and therefore in HTML we can bind to it using the `[]` input property binding syntax.

This `@Input` now becomes part of the *public interface* of our component.

Lets say at some future point we decided to change the `joke` property of our JokeComponent to perhaps just `data`, like so:

```
class JokeComponent {
  @Input() data: Joke;
}
```

Because this input is part of the public interface for our component we would also need to change all the input property bindings every where our component is used, like so:

```
<joke *ngFor="let j of jokes" [data]="j"></joke>
```

Not a great thing to ask the consumers of your component to have to do.

This is a common problem so to avoid expensive refactors the `@Input` annotation takes a parameter which is the name of the input property to the outside world, so if we changed our component like so:

```
class JokeComponent {
  @Input('joke') data: Joke;
}
```

To the outside world the input property name is still joke and we could keep the `JokeListComponent` template the same as before:

```
<joke *ngFor="let j of jokes" [joke]="j"></joke>
```

Summary

An Angular application should be broken down into small logical components which are glued together in HTML.

Its normal to have one root component called `AppComponent` which acts as the root node in the component tree.

We need to explicitly declare all components in the applications root `NgModule`.

We can make properties of our custom components input bindable with the `[]` syntax by prepending them with the `@Input` annotation.

Listing

http://plnkr.co/edit/LKj9OAoUMIUZhDS5HAiP?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {NgModule, Input}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Component} from '@angular/core';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [hidden]="data.hide">{{data.punchline}}</p>
  <a (click)="data.toggle()"
     class="btn btn-warning">Tell Me
  </a>
</div>
  `
})
class JokeComponent {
  @Input('joke') data: Joke;
}
```

```
@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}

@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
        System.import('script.ts').catch(function (err) {
            console.error(err);
        });
    </script>
</head>

<body class="container m-t-1">
    <app></app>
</body>
</html>
```

User Interaction & Outputs

In this lecture we are going to add an input form to our joke application so the user can add their own jokes to the list.

Learning Outcomes

- How to emit custom output events from components.
- How to create local variables in the templates.

The Joke Form

We *could* add a form in our `JokeListComponent` it would definitely be easier to setup since all the code would be contained in one component.

However we are going create a new component to support the form, perhaps we want to re-use this form in other places in our application or perhaps we want this functionality in one component so it's easier to test in isolation later on.

This component renders a bootstrap form on the page with two fields, setup and punchline, and a Create button.

When architecting an Angular application I often find it helps to first think about *how* your component is going to be used.

I believe our form component is best placed in the `JokeListComponent` template, like so:

```
<joke-form ①
  (jokeCreated)="addJoke($event)"> ②
</joke-form>
<joke *ngFor="let j of jokes" [joke]="j"></joke>
```

① The components tag, its selector, is going to be `joke-form`.

② The component is going to emit an event called `jokeCreated` whenever a user clicks the Create button.

We treat the component as a black-box, we don't care *how* the component is implemented or how the user interacts with it. The parent component just wants to know *when* the user has created a new joke.

When the `jokeCreated` event is emitted I want to call the `addJoke` function.

We'll go through what `$event` means a bit later on in this chapter.

The above syntax describes the behaviour we expect from our `JokeFormComponent`, lets now go and create it.

The JokeFormComponent

The component renders a twitter bootstrap formatted form with two fields and a submit button, like so:

```
@Component({
  selector: 'joke-form',
  template: `
<div class="card card-block">
  <h4 class="card-title">Create Joke</h4>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the setup">
  </div>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the punchline">
  </div>
  <button type="button"
          class="btn btn-primary">Create
  </button>
</div>
  `
})
class JokeFormComponent {
}
```

Remember to add `JokeFormComponent` to the declarations on the `NgModule`.

The component above just renders a form on the page, nothing happens when you click submit.

From the outside, all we care about this component is that it makes available an output event binding called `jokeCreated`.

To create a custom output event on our component we need to do two things:

1. Create an `EventEmitter` property on the `JokeFormComponent` class.
2. Similar to when we created a custom input property binding, we need to annotate that property with the `@Output` decorator.

An `EventEmitter` is a helper class which we can use to emit events when something happens, other components can then bind and react to these events.

```
import {Output, EventEmitter} from '@angular/core';
.
.
.
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();
}
```

We have now created an output event property on the component.

The name between the `<>` on the EventEmitter is the type of thing that will be output by this property. The syntax above is called `generics` and we'll cover them in more detail on the section on TypeScript.

When the `jokeCreated` event fires we are going to pass out an instance of a `Joke`.

We are initialising the property when it's defined, in one line. Previously we just declared the properties and initialised them in a constructor. The method you choose is up to you, with situations like this when the property will never need to change over time I like to initialise and declare on one line.

Lets now create a function called `createJoke()` and have it actually output an event, like so:

```
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke() {
    this.jokeCreated.emit(new Joke("A setup", "A punchline"));
  }
}
```

We output an event by calling the `emit` function on our `jokeCreated` property. Whatever we pass to the `emit` function is what will be output by the property. We are outputting an instance of a Joke with some dummy data.

Gluing it all together

We need to call the `createJoke` function when the user clicks the Create button, like so:

```
<button type="button"
      class="btn btn-primary"
      (click)="createJoke()">Create
</button>
```

We also need to *bind* to the output event property on our parent `JokeListComponent` so we can add the joke that gets output to the list of jokes, like so:

```
@Component({
  selector: 'joke-list',
  template: `
<joke-form (jokeCreated)="addJoke($event)"></joke-form>
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not very
mature'"),
    ];
  }

  addJoke(joke) {
    this.jokes.unshift(joke);
  }
}
```

Breaking this down, in the template we bind to the jokeCreated property like so:

```
<joke-form (jokeCreated)="addJoke($event)"></joke-form>
```

This calls the `addJoke` function when the `jokeCreated` property outputs an event.

`$event` is a special variable and holds whatever was emitted by the jokeCreated EventEmitter, in our case its an instance of a Joke.

```
addJoke(joke) {
  this.jokes.unshift(joke);
}
```

All the the addJoke function does is add the joke that got output to the front of the jokes array.

Angular automatically detects that the jokes array changed and then updated the view so it shows the new joke.

Now when we click the Create button we add the joke to the list, but it's just showing the dummy fixed joke.

Template Reference Variables

So we've got the mechanism we are going to use to send events, how do we actually get the values from the input fields?

We want to actually get the value of the setup and punchline input fields and pass them to the `createJoke` function when the user clicks the Create button.

One way we can solve this problem in Angular is to use something called a template reference variable.

We add the string `#setup` to our setup input control, like so:

```
<input type="text"
       class="form-control"
       placeholder="Enter the setup"
       #setup>
```

This tells Angular to bind this `<input>` control to the variable `setup`.

We can then use this variable `setup` in other places in the template.

`setup` is only available as a variable in the template, we don't automatically see the variable `setup` inside the javascript code of our `JokeFormComponent` class.

`setup` now points to the DOM element that represents an `<input>` control, which is an instance of the type `HTMLInputElement`.

Therefore to get the value of the setup input control we call `setup.value`.

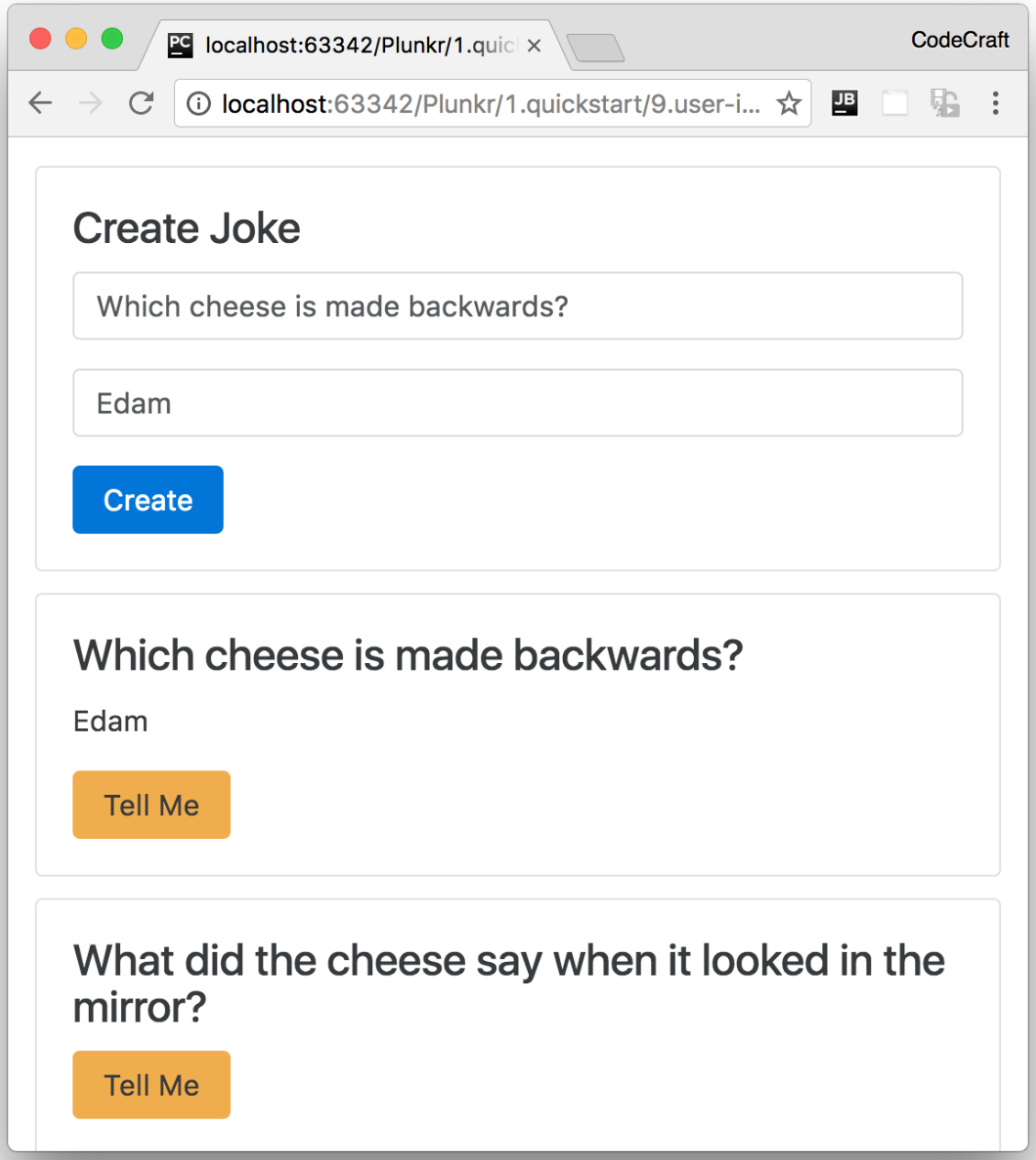
We do the same to the other punchline input control, then we can pass both the setup and punchline values to the `createJoke` function, like so:

```
<button type="button"
      class="btn btn-primary"
      (click)="createJoke(setup.value, punchline.value)">Create
</button>
```

Finally we change the `createJoke` function so it accepts these new arguments and uses them to construct an instance of a Joke, like so:

```
createJoke(setup: string, punchline: string) {
  this.jokeCreated.emit(new Joke(setup, punchline));
}
```

Now when we run the application the user can add a new joke via the form and have it appear in the list below.



Summary

Similar to input properties, we can also create output properties on our custom components using the `@Output` annotation.

These output properties are always instances of the `EventEmitter` class and we output events by calling the `emit` function. Whatever we pass in to the `emit` function is output as the `$event` template variable.

We can create local template variables by adding variables starting with the `#` character on any element in our template.

By default that variable is only visible in the template that it's declared in and points to the DOM element it's attached to.

Listing

http://plnkr.co/edit/v3vmZkOK4fxDXsrziqHx?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {Component, NgModule, Input, Output, EventEmitter} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
    public setup: string;
    public punchline: string;
    public hide: boolean;

    constructor(setup: string, punchline: string) {
        this.setup = setup;
        this.punchline = punchline;
        this.hide = true;
    }

    toggle() {
        this.hide = !this.hide;
    }
}


@Component({
    selector: 'joke-form',
    template: `
<div class="card card-block">
  <h4 class="card-title">Create Joke</h4>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the setup"
           #setup>
  </div>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the punchline"
           #punchline>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="createJoke(setup.value, punchline.value)">Create
```

```
  </button>
</div>
  `
})
class JokeFormComponent {
    @Output() jokeCreated = new EventEmitter<Joke>();

    createJoke(setup: string, punchline: string) {
        this.jokeCreated.emit(new Joke(setup, punchline));
    }
}


@Component({
    selector: 'joke',
    template: `
<div class="card card-block">
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [hidden]="data.hide">{{data.punchline}}</p>
  <a (click)="data.toggle()"
     class="btn btn-warning">Tell Me
  </a>
</div>
  `
})
class JokeComponent {
    @Input('joke') data: Joke;
}

@Component({
    selector: 'joke-list',
    template: `
<joke-form (jokeCreated)="addJoke($event)"></joke-form>
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
    jokes: Joke[];

    constructor() {
        this.jokes = [
            new Joke("What did the cheese say when it looked in the mirror?", "Hello-
me (Halloumi)"),
            new Joke("What kind of cheese do you use to disguise a small horse?",
"Mask-a-pony (Mascarpone)"),
            new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not very
mature'"),
        ];
    }
```

```
    addJoke(joke) {
        this.jokes.unshift(joke);
    }
}


@Component({
    selector: 'app',
    template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
    imports: [BrowserModule],
    declarations: [
        AppComponent,
        JokeComponent,
        JokeListComponent,
        JokeFormComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body class="container m-t-1">
  <app></app>
</body>
</html>
```

Wrapping Up

In this quick start section you created your first Angular Application!

You should now have at least a top level view of what an Angular application is, how it functions and how to go about architecting it.

We covered:

Environment

How to use plnkr.co to write Angular apps in a browser</dd>

Components

Which let us extend the HTML language to create new tags and forms the basis of an Angular application.

Bootstrapping

How to actually make Angular load and take control of a HTML page.

Binding

String interpolation with `{{ }}` and both input property binding as well as output event binding.

NgFor

How to render multiple elements using the the built-in `NgFor` **directive**.

Data Modelling

We touched on **data modelling**, by creating our own Joke class which encapsulated all our data and logic relate to a joke.

Template Local Variables

Capturing user input from users by adding `#` variables to input controls.

Architecture

We started to see how we go about building Angular apps by wiring together *inputs* and *outputs* in order to glue Components together.

We'll be going into *much* more detail into each of these topics, and others, in future chapters.

However since Angular is based on TypeScript it makes sense to get a good understanding of those features before we dive into the rest of this course so that's the topic of the next section.

Activity

Steps

Fork this plunker:

Extend the joke application in the above plunker so the user can delete a joke from the list.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/b0F6Dhb40Hm5zfiamAix?p=preview

ES6 JavaScript & TypeScript

TypeScript Setup

Transpilation

Since browsers don't know about TypeScript we need to convert our TypeScript code into something they do understand.

The current version of JavaScript with the broadest support across all browsers is still ES5, so that's what we currently convert our TypeScript code into.

The process of converting TypeScript into ES5 is called *transpilation* and we use a tool called `tsc` to compile on the command line.

It's useful to have the `tsc` tool so you can try out TypeScript yourself and transpile on the command line but it's *not* necessary to do this in your everyday Angular development.

Projects built using the Angular command line tools use the webpack bundler which handles transpilation of your TypeScript for you. When coding Angular via plunker SystemJS transpiles TypeScript for you in the browser.

It is good however to have `tsc` installed especially if you want quickly try our some TypeScript features and see how they are transpiled into ES5 or ES6.

Installing TypeScript

We can install typescript via `npm`

```
npm install -g typescript
```

The preceding command will install the TypeScript compiler and add its executable `tsc` to your path.

To make sure everything worked correctly type:

```
tsc -v
```

And it should print something like

```
Version 1.8.0
```

Running TypeScript

Create a file called `hello.ts`.

Inside that file add `console.log('hello world');`

We can compile a typescript file into a javascript file by calling:

```
tsc hello.ts
```

This generates a file called `hello.js`

And we can execute that file by using node, like so:

```
node hello.js
```

We can watch a typescript file for changes and compile it automatically with:

```
tsc -w hello.ts
```

We can provide configuration on the command line, like so:

```
tsc -t ES6 -w hello.ts
```

The above `-t ES6` is a flag to tell typescript to transpile into ES6 instead of the default ES5.

However if we are want to watch a whole directory we need to use a configuration file.

We can create a config file with the most common settings like so:

```
tsc --init
```

If we want to watch all files in a directory we can just type:

```
tsc -w
```

If there are errors, by default TypeScript will still output the javascript file if it can, this behaviour can be changed in the config.

Overview

TypeScript is an open source programming language that is developed and maintained by Microsoft.

It was initially released in 2012.

TypeScript is a super-set of ES6. That means whatever features we have in ES6 are also in TypeScript with some extra features on top, such as static typing and richer syntax.

In this section will cover both; we will go over some of the key features of ES6 and then we will cover some of the key features of TypeScript.

ES6 Support

ES6 is an *agreement* for how JavaScript should function in the future.

Its up to each browser manufacturer to implement those features according to that agreement.

Therefore all browsers don't support all the features of ES6 *yet*.

Chrome has the best support by far and so does Node. I will be providing plunkers where you can try out the code yourself, remember to use Chrome to run the plunker.

Sometimes ES6 features are only *switched on* when we are in "use strict" mode, so to test our ES6 code my recommendation is to always add "use strict" to the top of your files.

A list of all the ES6 features and current support across a variety of platforms can be found here: http://kangax.github.io/compat-table/es6/

We should be using at least node version 6 to support most of the ES6 features we will be demonstrating in this section.

TypeScript Support

TypeScript transpiles down into ES5 so we don't need to worry about browser support.

Code will be provided for all the examples, but also we'll explain how to how to compile typescript locally on your computer with the command line tools.

Let

Block Scope

A confusing point for developers coming from different languages is the way variable scope behaves in JavaScript.

Scope refers to the lifecycle of a variable, i.e. where in the code it's visible and for how long.

In Java and C++ there is the concept of block scope, a block is any code that is wrapped in { and }, like so:

```
{
    // This is a block
}
// This is not part a block
```

This means that in those languages if you declare a variable inside a block, it's only visible inside that block and any nested blocks inside that block.

But in ES5 JavaScript we only have two scopes, the *global scope* and *function scope*.

So if I wrote:

```
{
  var a = "block";
}
console.log(a);
```

Those coming from Java or C++ backgrounds might think that this would throw an error because we are trying to use the variable `a` outside of the block it was created in. But in JavaScript it doesn't, this code is perfectly legal.

The variable `a`, as we've declared it above, exists in *global scope* so this means it's visible from *everywhere* in our application.

In ES5 apart from global scope, the only other scope is *function scope*, so if we wrote.

```
function hello() {
    var a = "function";
}
hello();
console.log(a);
```

If we ran the above we would get an error, like so:

```
Uncaught ReferenceError: a is not defined(⋯)
```

This is because the `a` variable is declared *inside* a function and is therefore only visible inside *that* function, trying to access it outside the function results in an error.

But again this isn't *block level scope* as we can see if we add a *for loop* inside the function, like so:

```
function hello() {
    var a = "function";
    for (var i = 0; i < 10; i++) {
        var a = "block";
    }
    console.log(a);
}
hello();
```

What gets printed out here is `block` not `function` despite the fact we are outside the for loop, that's because the body of the for loop is not its own scope.

IIFE

This issue of *no block level scope* has plagued JavaScript developers since its inception.

One common workaround in the past has been to use something called an Immediately Invoked Function Expression (IIFE) like so:

```
function hello() {
    var a = "function";

    for (var i=0; i<5; i++) {
        (function() {
            var a = "block";
        })();
    }
    console.log(a);
}
hello();
```

This now prints out `function`.

If this syntax looks a bit strange to you:

```
(function() {
    var a = "block";
})();
```

Know its just a shorter way of writing:

```
function something() {
  var a = "block";
}
something();
```

It's a function that we call *immediately* after defining it.

Since functions have their own scope, using an IIFE has the same effect as if we had block level scope, the variable `a` inside the IIFE isn't visible *outside* the IIFE.

Let

IIFEs work but it's a pretty long winded way of solving this problem. So with ES6 we now have the new `let` keyword, we use it in place of the `var` keyword and it creates a variable **with** block level scope, like so:

```
function hello() {
    var a = "function";
    for (var i = 0; i < 5; i++) {
        let a = "block";
    }
    console.log(a);
}
hello();
```

Now the `a` declared in the for loop body only exists between the `{` and `}`, and the code snippet above prints out `function` as expected.

Using let in for loops

So a classic interview question to test JavaScript developers knowledge of the lack of block level scope is this one:

```
var funcs = [];
for (var i = 0; i < 5; i += 1) {
    var y = i;
    funcs.push(function () {
        console.log(y);
    })
}
funcs.forEach(function (func) {
    func()
});
```

What gets printed out?

You might expect

```
0
1
2
3
4
```

But in fact it's

```
5
5
5
5
5
```

The reason for this is that the variable `y` is not *block level*, it doesn't *only* exist inside its enclosing `{}` In fact it's a global variable and by the time any of the functions are called it's already been set to 5.

In the above example if we replace `var y = i` with `let y = i` then the variable `y` only exists inside it's block, like so:

```
var funcs = [];
for (var i = 0; i < 5; i += 1) {
    let y = i;
    funcs.push(function () {
        console.log(y);
    })
}
funcs.forEach(function (func) {
    func()
});
```

And so executing this now results in:

```
0
1
2
3
4
```

The for loop short-cut

The above construct is so common we have a shortcut, we can declare the index variable `i` with `let` in the for loop expression, like so:

```
var funcs = [];
for (let i = 0; i < 5; i += 1) {
    funcs.push(function () {
        console.log(i);
    })
}
funcs.forEach(function (func) {
    func()
});
```

Even though `let i = 0` is strictly declared *outside* of the for block `{ }`, any variables declared in the for loop expression with `let` has block level scope in the for loop.

Summary

`let` is a very important addition the javascript language in ES6.

It's not a replacement to `var`, `var` can still be used even in ES6 and has the same semantics as ES5.

However unless you have a specific reason to use `var` I would expect all variables you define from now on to use `let`.

Listing

http://plnkr.co/edit/k7150PqX1DML7EFHKuRc?p=preview

script.js

```
/* let */
var funcs = [];
for (let i = 0; i < 5; i += 1) {
  funcs.push(function () {
    console.log(i);
  })
}
funcs.forEach(function (func) {
  func()
});
```

index.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="script.js"></script>
</head>
</html>
```

Const

`const` is a new keyword which declares a variable as *constant over time*.

Declaring a const variable

We can use `const` to declare a variable but unlike `let` and `var` it must be *immediately* initialised, with a value that can't be changed afterwards.

If we try to declare it without initialising it we get a `SyntaxError`, like so:

```
const foo; // SyntaxError: Missing initializer in const declaration
```

If we try to change it after we have declared and initialised it we get a `TypeError`, like so:

```
const foo = 1;
foo = 2; // TypeError: Assignment to constant variable
```

Block scoping

Both `let` and `const` create variables that are block-scoped – they only exist within the innermost block that surrounds them.

```
function func() {
    if (true) {
        const tmp = 123;
    }
    console.log(tmp); // Uncaught ReferenceError: tmp is not defined
}
func();
```

Immutable variable

Variables created by `let` and `var` are *mutable*:

```
let foo = "foo";
foo = "moo";
console.log(foo);
```

Mutable in this case means *can change over time*.

Variables created by `const` however are *immutable*, they don't change over time, specifically the the const variable can't *point to* another thing later on.

```
const foo = 'abc';
foo = 'def'; // TypeError: Assignment to constant variable
```

Mutable Value

There is one big pitfall with `const` however.

When we say "`const` variables are immutable" it only means that the variable always has to *point to* the same thing. It does not mean than the *thing* it points to can't change over time.

For example, if the variable `foo` is a `const` that points to an object - we can't make `foo` point to another object later on:

```
const foo = {};
foo = {}; // TypeError: Assignment to constant variable.
```

But we can however *mutate*, make changes to, the object `foo` points to, like so:

```
const foo = {};
foo['prop'] = "Moo"; // This works!
console.log(foo);
```

If we want the *value* of `foo` to be immutable we have to freeze it using `Object.freeze(⋯)`.

When we freeze an object we can't change it, we can't add properties or change the values of properties, like so:

```
const foo = Object.freeze({});
foo.prop = 123;
console.log(foo.prop) // undefined
```

However by default the above doesn't throw an error, it just *silently* ignores the issue. So in the example above it didn't throw an error when we tried to set `foo.prop = 123` but when we try to print out the value on the next line we just get `undefined`.

To force `Object.freeze(⋯)` to throw an error we must remember to be in `"use strict"` mode, like so:

```
"use strict";
const foo = Object.freeze({});
foo.prop = 123; // SyntaxError: Identifier 'foo' has already been declared
```

Summary

`const` lets us declare variables which don't change over time, which are immutable.

The important *gotcha* with const is that the variable is immutable, but not the value, the thing the variable points to.

This means that if we declare an object as `const`, confusingly we can still change properties of the object later on.

To solve this and make an object immutable we use the `Object.freeze(⋯)` function which together with the `"use strict";` param throws an error if we try to change the object.

Listing

script.js

```
'use strict';


/* // Declaring a const variable
const foo; // SyntaxError: Missing initializer in const declaration
*/


/* // Declaring a const variable
const moo = 1;
moo = 2; // TypeError: `foo` is read-only
*/

/* // Block Scoping
function func() {
    if (true) {
        const tmp = 123;
    }
    console.log(tmp); // Uncaught ReferenceError: tmp is not defined
}
func();
*/


const foo = Object.freeze({});
foo.prop = 1;
console.log(foo.prop);
```

index.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="script.js"></script>
</head>
</html>
```

Template Strings

Multi-Line Strings

With ES5 and ES6 we can specify a string with either the `'` or `"` characters.

```
let single = "hello world";
```

But in ES5 if we wanted to make that string span *multiple lines* it quickly becomes a pain.

```
let single = 'hello ' +
    'world ' +
    'my ' +
    'name ' +
    'is ' +
    'asim';
```

If we wanted each line in the string to contain new line characters, matching how it was written, we had to remember to add `\n` to the end of each string.

```
let single = 'hello\n' +
    'world\n' +
    'my\n' +
    'name\n' +
    'is\n' +
    'asim\n';
```

In ES6 we have another way of defining strings, using the *back-tick* character `` ` ``

```
let multi = `
hello
world
my
name
is
asim`;
console.log(multi);
```

The above prints out:

```
hello
world
my
name
is
asim
```

With ` strings can now span *multiple lines* and they are also formatted with new line characters.

Variable Substitution

Another really interesting feature of declaring strings with ` is that they can now expand variables using the `${variable_name}` syntax, like so:

```
let name = "asim";

let multi = `
hello
world
my
name
is
${name}
`;
console.log(multi);
```

prints out:

```
hello
world
my
name
is
asim
```

Summary

Template strings are a small change to JavaScript in ES6 but the convenience of multi-line strings and variable substitution is substantial.

Listing

http://plnkr.co/edit/Io5rE4kUBp1xQgAphuih?p=preview

script.js

```
'use strict';

let name = "Asim";

let multi = `hello
world
my
name
is
${name}`;
console.log(multi);
```

Fat Arrow Functions

Syntax

JavaScript has *first class functions*.

This means that in JavaScript functions can be themselves be passed around like any other value, even as arguments to other functions.

E.g. we can pass to the `setTimeout` function, a function, like so:

```
setTimeout(function() {
    console.log("setTimeout called!");
}, 1000);
```

The function we pass as an argument to `setTimeout` is called an *anonymous function* because it doesn't have a name.

ES6 has introduced a slightly different syntax to define anonymous functions called the *fat arrow* syntax, with it we can re-write the above as:

```
setTimeout(() => {
    console.log("setTimeout called!")
}, 1000);
```

If the function only contains one expression we can drop the braces and shorten even further to:

```
setTimeout(() => console.log("setTimeout called!"), 1000);
```

Arguments

What if we wanted to pass an argument to the function?

We can re-write the below normal function to one that uses the fat arrow syntax:

```
let add = function(a,b) {
    return a + b;
};
```

Can now be written as:

```
let add = (a,b) => a + b;
```

In the first example we write `return a + b` but in the *fat arrow* version we just wrote `a + b`. That's because in the fat arrow syntax if it is on one line, the statement gets returned automatically without the need to use the `return` keyword.

this

Lets imagine we have an `object` with a function called `sayLater`, like so:

```
let obj = {
    name: "asim",
    sayLater: function() {
        console.log(`${this.name}`);
    }
};
obj.sayLater();
```

In the `sayLater` function `this` points to `obj`

So `console.log(`${this.name}`);` prints out `asim`.

Now lets imagine that we log the message using the `setTimeout` function.

```
let obj = {
    name: "asim",
    sayLater: function () {
        setTimeout(function () {
            console.log(`${this.name}`);
        }, 1000);
    }
};
obj.sayLater();
```

In the `sayLater()` function we call `setTimeout` and then log the value of `this.name`, which we expect to be *asim*.

In fact we get `undefined` printed out to the console.

Calling context

The reason for this is that the value of `this` in a function depends on *how* the function is called.

At it's most basic level if the function is called as `obj.sayLater()`, the value of `this` is the calling context which in this case is `obj`.

What is the calling context for the anonymous function we pass to `setTimeout`? What will `this` point to inside that function?

```
setTimeout(function () {
    console.log(`${this.name}`);
}, 1000);
```

The answer is *it depends*.

In the browser it's either `undefined` or the `global` object depending on if you are running in strict mode or not. In node it's an internal `timeout` object.

In all cases however it *isn't* going to be `obj`, so `this.name` is not going to return *asim*, it's going to return `undefined` or raise an error.

This *instability* of `this` is an incredibly common problem in javascript that has affected it since the early days.

In ES5 there are a number of methods we can use to stabilise the value of `this`. One common solution is to assign `this` to another variable at the top, usually called `self` or `vm`, and then always use `self` in the function body, like so:

```
let obj = {
    name: "asim",
    sayLater: function () {
        let self = this; // Assign to self
        console.log(self);
        setTimeout(function () {
            console.log(`${self.name}`); // Use self not this
        }, 1000);
    }
};
```

But in ES6 we can do better, if we use *fat arrow* functions the value of `this` inside a fat arrow function will be the same as the value of `this` *outside* the fat arrow function.

It uses the value of `this` from the surrounding code for its context. i.e. whatever `this` points to in the surrounding code, `this` will point to in the function body of the fat arrow function.

We can re-write our obj to use fat arrow syntax like so:

```
let obj = {
    name: "asim",
    sayLater: function () {
        console.log(this); // `this` points to obj
        setTimeout(() => {
            console.log(this); // `this` points to obj
            console.log(`${this.name}`); // `this` points to obj
        }, 1000);
    }
};
obj.sayLater();
```

If we ran the above code we would see that the value of `this` in the `setTimout` function is now `obj`; the same as the value of `this` outside the `setTimeout` function.

Summary

The new fat arrow function syntax in ES6 is far more than just a slightly shorter way of writing anonymous functions.

It has finally solved the thorny issue of stabilising the value of `this` that has affected javascript since the start and caused so many work arounds to be discussed and applied in code.

Listing

script.js

```
'use strict';

setTimeout(() => {
    console.log("setTimeout called!")
}, 1000);

setTimeout(() => console.log("setTimeout called!"), 1000);

let add = (a,b) => a + b;
console.log(add(1,2));

let obj = {
    name: "Asim",
    sayLater: function() {
        setTimeout(() => console.log(`${this.name}`), 1000)
    }
};
obj.sayLater();
```

Destructuring

Destructuring is a way of extracting values into variables from data stored in objects and arrays.

Object Destructuring

Let's imagine we have an object like so:

```
const obj = {first: 'Asim', last: 'Hussain', age: 39 };
```

We want to extract the `first` and `last` properties into local variables, prior to ES6 we would have to write something like this:

```
const f = obj.first;
const l = obj.last;
console.log(f); // Asim
console.log(l); // Hussain
```

With destructing we can do so in one line, like so:

```
const {first: f, last: l} = obj;
console.log(f); // Asim
console.log(l); // Hussain
```

`{first: f, last: l}` describes a *pattern*, a set of rules for *how* we want to destructure an object.

`const {first: f} = obj;` translates to *extract the property* `first` *and store in a constant called f.*

If we wanted to extract the properties into variables with the same name we would write it like so:

```
const {first: first, last: last} = obj;
console.log(first); // Asim
console.log(last); // Hussain
```

The above is quite a common use case for destructuring so it has a shortcut, like so

```
// {prop} is short for {prop: prop}
const {first, last} = obj;
console.log(first); // Asim
console.log(last); // Hussain
```

Array Destructuring

Array destructuring works in a similar way except it extracts based of the index in the array, like so:

```
const arr = ['a', 'b'];
const [x, y] = arr;
console.log(x); // a
console.log(y); // b
```

Function Parameter Destructuring

One really useful use case for destructuring is in function parameters.

Typically if we want to pass multiple params to a function, with maybe some optional parameters, we would pass it in as an object like so:

```
function f(options) {
  console.log(options.x);
}
f({x:1}); // 1
```

Now we can define the function parameter list as an *object destructure pattern*, like so:

```
function f({x}) {
  console.log(x); // Refer to x directly
}
f({x:1});
```

Notice that in the function body above we can refer to `x` directly, we don't have to refer to it through an object property like `options.x`.

In addition to that when using destructured function parameters we can also provide default values, like so:

```
function f({x=0}) {
  console.log(x);
}
f({}); // 0
```

In the above example `x` now has a default value of 0 even if it's not passed into the function when called.

Summary

Destructuring is a useful feature of ES6, with it we can extract values from objects and arrays with ease.

Through function parameter destructing we now have a built in syntax for providing optional parameters to functions, including giving them default values if none are provided.

Listing

http://plnkr.co/edit/v6BS0PekPcDZN0QpV8wN?p=preview

script.js

```
'use strict';

// Object Destructuring
const obj = {first: 'Asim', last: 'Hussain', age: 39};

function getObj() {
  return obj;
}

const {first, last} = getObj();

console.log(first);
console.log(last);

// Array Destructuring
const arr = ['a', 'b'];
const [x, y] = arr;
console.log(x);
console.log(y);

// Function Parameter Destructuring
function func({x = 1}) {
  console.log(x);
}
func({});
```

For Of

In this lecture we are going to examine how we can loop over arrays, using the methods available to us in the past with ES5 and also the new `for-of` looping mechanism in ES6.

For & ForEach

We have a couple of ways of looping through Arrays in ES5 javascript.

For one we have the classic `for` loop, like so:

```
let array = [1,2,3];
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```

With ES5 JavaScript we can also use the `forEach` method on the Array class, like so:

```
let array = [1,2,3];
array.forEach(function (value) {
  console.log(value);
});
// 1
// 2
// 3
```

It's slightly shorter but has a few downsides:

1. You can't break out of this loop using a `break` statement or move to the next iteration with `continue`.
2. You can't return from the enclosing function using a `return` statement.

For In

The `for-in` loop is designed for iterating over an *objects properties*, like so:

```
var obj = {a:1,b:2};
for (let prop in obj) {
    console.log(prop);
}
// a
// b
```

If we tried to use it with an array, it might initially look like it's working:

```
let array = [10,20,30];
for (let index in array) {
  console.log(index);
});
// 0
// 1
// 2
```

But if we tried to print out the type of `index` like so:

```
let array = [10,20,30];
for (let index in array) {
  console.log(typeof(index));
};
// string
// string
// string
```

The `index` variable is a *string* and not a *number*, using `for-in` with arrays converts the index to a string.

If you are expecting a number but in fact have a string this can cause problems, for example `"1" + "2"` is the string `"12"` and not the number `3`.

For-of loop

Rather than change the way the `for-in` loops work in ES6 and in the process create a breaking change, instead in ES6 we have a new syntax called `for-of`.

```
let array = [10,20,30];
for (var value of array) {
  console.log(value);
}
// 10
// 20
// 30
```

- This is the most concise way of looping through array elements.
- It avoids all the pitfalls of `for–in`.
- It works with `break`, `continue`, and `return`

Summary

The `for–in` loop is for looping over object properties.

The `for-of` loop is for looping over the values in an array.

`for-of` is not just for arrays. It also works on most array-like objects including the new `Set` and `Map` types which we will cover in the next lecture.

Listing

http://plnkr.co/edit/v6BS0PekPcDZN0QpV8wN?p=preview

script.js

```
'use strict';

// Object Destructuring
const obj = {first: 'Asim', last: 'Hussain', age: 39};

function getObj() {
  return obj;
}

const {first, last} = getObj();

console.log(first);
console.log(last);

// Array Destructuring
const arr = ['a', 'b'];
const [x, y] = arr;
console.log(x);
console.log(y);

// Function Parameter Destructuring
function func({x = 1}) {
  console.log(x);
}
func({});
```

Map & Set

Using Object as a Map

In ES5 and below the only data structure we had to map keys to values was an `Object`, like so:

```
let obj = {key: "value", a: 1}
console.log(obj.a); // 1
console.log(obj['key']); // "value"
```

However it does have a few pitfalls.

Inherited Objects

Looping over objects with `for-in` also iterated over the inherited properties as well as the objects own properties, like so:

```
let base = {a:1,b:2};
let obj = Object.create(base);
obj[c] = 3;
for (prop in obj) console.log(prop)
// a
// b
// c
```

`Object.create` creates a new objects who's *prototype* points to the passed in `base` object. If it can't find a requested property on `obj`, javascript then tries to search the `base` object for the same property.

Perhaps this is the behaviour you want? Or perhaps you only want to print out the keys that belong to the current object?

With ES5 JavaScript to ensure you were looking at a property of the current object we need to use the `hasOwnProperty` function, like so:

```
let base = {a:1,b:2};
let obj = Object.create(base);
obj[c] = 3;
for (prop in obj) {
    if (obj.hasOwnProperty(prop)) {
        console.log(prop)
    }
}
// c
```

Overriding Functions

If we are using Object as a dictionary then we could theoretically store a key of `hasOwnProperty` which then leads to the code in the example above failing, like so:

```
let obj = {hasOwnProperty: 1};
obj.hasOwnProperty("test");
// TypeError: Property 'hasOwnProperty' is not a function
```

`proto` property

`proto` holds a special meaning with respect to an objects prototype chain so we can't use it as the name of a key.

```
let base = {__proto__:1,b:2};
for (prop in obj) console.log(prop)
// b
```

Map

Map is a new data structure introduced in ES6 which lets you map keys to values without the drawbacks of using Objects.

Creating, getting and setting

We create a map using the `new` keyword, like so

```
let map = new Map();
```

We can then add entries by using the `set` method, like so:

```
let map = new Map();
map.set("A",1);
map.set("B",2);
map.set("C",3);
```

The set method is also chainable, like so:

```
let map = new Map()
    .set("A",1)
    .set("B",2)
    .set("C",3);
```

Or we could initialise the `Map` with a an array of key-value pairs, like so:

```
let map = new Map([
    [ "A", 1 ],
    [ "B", 2 ],
    [ "C", 3 ]
]);
```

We can extract a value by using the `get` method:

```
map.get("A");
// 1
```

We can check to see if a key is present using the `has` method:

```
map.has("A");
// true
```

We can delete entries using the `delete` method:

```
map.delete("A");
// true
```

We can check for the size (number of entries) using the `size` property:

```
map.size
// 2
```

We can empty an entire `Map` by using the `clear` method:

```
map.clear()
map.size
// 0
```

Looping over a Map

We use the `for-of` looping operator to loop over entries in a Map.

There are a couple of different method we can employ, we'll go over each one using the below map as the example:

```
let map = new Map([
    [ "APPLE", 1 ],
    [ "ORANGE", 2 ],
    [ "MANGO", 3 ]
]);
```

Using keys()

The `keys` method returns the keys in the map as an array which we can loop over using `for-of` like so:

```
for (let key of map.keys()) {
    console.log(key);
}
// APPLE
// ORANGE
// MANGO
```

Using values()

The `values` method returns the values in the map as an array which we can loop over using `for-of` like so:

```
for (let value of map.values()) {
    console.log(value);
}
// 1:
// 2
// 3
```

Using entries()

The `entries` method returns the [key,value] pairs in the map as an array which we can loop over using `for-of` like so:

```
for (let entry of map.entries()) {
    console.log(entry[0], entry[1]);
}
// "APPLE" 1
// "ORANGE" 2
// "MANGO" 3
```

Using *destructuring* we can access the keys and values directly, like so:

```
for (let [key, value] of map.entries()) {
    console.log(key, value);
}
// "APPLE" 1
// "ORANGE" 2
// "MANGO" 3
```

Looping over key-value pairs via `entries` is so common that this is the default for a Map.

Therefore we don't even need to call `entries()` on a map instance, like so:

```
for (let [key, value] of map) {
    console.log(key, value);
}
// "APPLE" 1
// "ORANGE" 2
// "MANGO" 3
```

A distinction between Object and Map is that Maps record the *order in which elements are inserted*. It then replays that order when looping over keys, values or entries.

Set

Sets are a bit like maps but they only store keys not key-value pairs.

They are common in other programming languages but are a new addition to JavaScript in ES6.

Creating, getting and setting

We create a Set using the `new` keyword, like so

```
let set = new Set();
```

We can then add entries by using the `add` method, like so:

```
let set = new Set();
set.add('APPLE');
set.add('ORANGE');
set.add('MANGO');
```

The `add` method is *chainable*, like so:

```
let set = new Set()
    .add('APPLE')
    .add('ORANGE')
    .add('MANGO');
```

Or we can initialise the Set with an array, like so:

```
let set = new Set(['APPLE', 'ORANGE', 'MANGO']);
```

We can check to see if a value is in a set like so:

```
set.has('APPLE')
// true
```

We can delete a value from the set:

```
set.delete('APPLE')
```

We can count the number of entries in the set like so:

```
set.size
// 2
```

We can empty the entire set with the `clear` method:

```
set.clear();
set.size
// 0
```

Sets can only store *unique* values, so adding a value a second time has no effect:

```
let set = new Set();
set.add('Moo');
set.size
// 1
set.add('Moo');
set.size
// 1
```

Looping over a Set

We can use the `for-of` loop to loop over items in our set, like so:

```
let set = new Set(['APPLE', 'ORANGE', 'MANGO']);
for (let entry of set) {
    console.log(entry);
}
// APPLE
// ORANGE
// MANGO
```

Similar to Maps, Sets also record the *order in which elements are inserted*, it then replays that order when looping.

Summary

Map and Set are great additions to JavaScript in ES6.

We no longer have to deal with Map and Sets *poor cousin* the Object and it's many drawbacks.

Listing

http://plnkr.co/edit/IXnaorsOWB1XuaePtyf4?p=preview

script.js

```
'use strict';

// Map
let map = new Map();
map.set("A", 1);
map.set("B", 2);
map.set("C", 3);

let map2 = new Map()
  .set("A", 1)
  .set("B", 2)
  .set("C", 3);

let map3 = new Map([
  ["A", 1],
  ["B", 2],
  ["C", 3]
]);

for (let [key, value] of map) {
  console.log(key, value);
}

console.log(map.get("A"));
console.log(map.has("A"));
```

```
console.log(map.size);

map.delete("A");
console.log(map.size);

map.clear();
console.log(map.size);


// Set
let set = new Set();
set.add('APPLE');
set.add('ORANGE');
set.add('MANGO');


let set2 = new Set()
  .add('APPLE')
  .add('ORANGE')
  .add('MANGO');

let set3 = new Set(['APPLE', 'ORANGE', 'MANGO']);

console.log(set.has('APPLE'));

set.delete('APPLE');

console.log(set.size);

set.clear();
console.log(set.size);


let set4 = new Set();
set3.add('Moo');
console.log(set3.size);
// 1
set4.add('Moo');
console.log(set4.size);
// 1

for (let entry of set2) {
  console.log(entry);
}
```

Promises

When you execute a task *synchronously*, you wait for it to finish before moving on to the next line of code.

When you execute a task *asynchronously*, the program moves to the next line of code before the task finishes.

Think of synchronous programming like waiting in line and asynchronous programming like taking a ticket. When you take a ticket you can go do other things and then be notified when ready.

Callbacks

One way to program asynchronously is to use *callbacks*. We pass to an asynchronous function a function which it will call when the task is completed.

```
function doAsyncTask(cb) {
  setTimeout(() => {
    console.log("Async Task Calling Callback");
    cb();
  }, 1000);
}

doAsyncTask(() => console.log("Callback Called"));
```

The `doAsyncTask` function when called kicks of an asynchronous task and *returns immediately*.

To get notified when the async task completes we pass to `doAsyncTask` a function which it will call when the task completes.

It's called a *callback* function, `cb` for short, because it *calls-you-back*.

Promise API

In ES6 we have an alternative mechanism built into the language called a *promise*.

A *promise* is a *placeholder* for a future value.

It serves the same function as callbacks but has a nicer syntax and makes it easier to handle errors.

Creating a Promise

We create an instance of a promise by calling `new` on the `Promise` class, like so:

```
var promise = new Promise((resolve, reject) => {
});
```

We pass to Promise an inner function that takes two arguments `(resolve, reject)`.

Since we are defining the function we can call these arguments whatever we want but the convention is to call them `resolve` and `reject`.

`resolve` and `reject` are in fact functions themselves.

Inside this inner function we perform our asynchronous processing and then when we are ready we call `resolve()`, like so:

```
var promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log("Async Work Complete");
    resolve();
  }, 1000);
});
```

We usually return this promise from a function, like so:

```
function doAsyncTask() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Async Work Complete");
      resolve();
    }, 1000);
  });
  return promise;
}
```

If there was an error in the async task then we call the `reject()` function like so:

```
function doAsyncTask() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Async Work Complete");
      if (error) {
        reject();
      } else {
        resolve();
      }
    }, 1000);
  });
  return promise;
}
```

Promise Notifications

We can get notified when a promise `resolves` by attaching a *success* handler to its `then` function, like so:

```
doAsyncTask().then(() => console.log("Task Complete!"));
```

`then` can take two arguments, the second argument is a *error* handler that gets called if the promise is `rejected`, like so:

```
doAsyncTask().then(
  () => console.log("Task Complete!"),
  () => console.log("Task Errored!"),
);
```

Any values we pass to the `resolve` and `reject` functions are passed along to the *error* and *success* handlers, like so:

```
let error = true;
function doAsyncTask() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (error) {
        reject('error'); // pass values
      } else {
        resolve('done'); // pass values
      }
    }, 1000);
  });
}

doAsyncTask().then(
  (val) => console.log(val),
  (err) => console.error(err)
);
```

Immediate Resolution or Rejection

We can create an immediately *resolved* Promise by using the `Promise.resolve()` method, like so:

```
let promise = Promise.resolve('done');
```

And an immediately *rejected* Promise by using the `Promise.reject()` method, like so:

```
let promise = Promise.reject('fail');
```

One of the nice things about Promises is that if we add a `then` handler **after** the promise resolves or rejects the handler **still** gets called.

```
let promise = Promise.resolve('done');
promise.then((val) => console.log(val)); // 'done'
```

In the above example, even though the Promise has resolved *before* we added the success handler, the promise framework still calls the success handler.

Chaining

We can also connect a series of `then` handlers together in a chain, like so:

```
Promise.resolve("done")
  .then(
    (val) => {
      console.log(val);
      return 'done2';
    },
    (err) => console.error(err)
  )
  .then(
    (val) => console.log(val),
    (err) => console.error(err)
  );
// 'done'
// 'done2'
```

Promises pass an error along the chain till it finds an error handler. So we don't need to define an error handler for each `then` function, we can just add one at the end like so:

```
Promise.reject('fail')
  .then((val) => console.log(val))
  .then(
    (val) => console.log(val),
    (err) => console.error(err)
  );
```

If we throw an exception from our promise function or one of the success handlers, the promise gets rejected and the error handler is called, like so:

```
Promise.resolve('done')
  .then((val) => {
    throw new Error("fail")
  })
  .then(
    (val) => console.log(val),
    (err) => console.error(err)
  );
// [Error: fail]
```

Catch

The `catch` function works exactly the same way as the `then` error handler, it's just clearer and more explicitly describes our intent to handle errors.

```
Promise.resolve('done')
  .then((val) => {throw new Error("fail")})
  .then((val) => console.log(val))
  .catch((err) => console.error(err));
```

Summary

Promises are a far cleaner solution to writing asynchronous code than callbacks.

The resulting code that's created is easier to read and is often written the order the application will execute.

So it can be easier to trace through code in your head.

With the `catch` handler it also gives us a single place where we can handle errors.

Listing

http://plnkr.co/edit/OMGgN6qpED6wTEEP2XYN?p=preview

script.js

```
'use strict';

// Via callbacks
/*
 function doAsyncTask(cb) {
 setTimeout(() => {
 console.log("Async Task Calling Callback");
 cb();
 }, 1000);
 }

 doAsyncTask(() => console.log("Callback Called"));
 */



// Via Promise
let error = false;
function doAsyncTask() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (error) {
        reject('error');
      } else {
        resolve('done');
      }
    }, 1000);
  });
}

doAsyncTask().then(
    (val) => console.log(val),
    (err) => console.error(err)
);

// Immediately Resolved Promise
let promise = Promise.resolve('done');
promise.then((val) => console.log(val)); // 'done'

// Handling Errors
Promise.resolve('done')
    .then((val) => {throw new Error("fail")})
    .then((val) => console.log(val))
    .catch((err) => console.error(err));
```

Class & Interface

How do we write object orientated code in ES6?

Object Orientation in JavaScript

JavaScript has a prototype-based, object-oriented programming model. It creates objects using other objects as blueprints and to implement inheritance it manipulates what's called a *prototype chain*.

We normally call the way object orientation is implemented in C++ or Java as the *Classical OO Pattern* and the way it's implemented in JavaScript as the *Prototype Pattern*.

Although the prototype pattern is a valid way to implement object orientation it can be confusing for newer javascript developers or developers used to the classical pattern.

So in ES6 we have an alternative syntax, one that closer matches the classical object orientated pattern as is seen in other languages.

Under the hood the new syntax still uses the prototype pattern with constructor functions and the prototype-chain. However, it provides a more common and convenient syntax with less boilerplate code.

TypeScript supports the ES6 `class` syntax but also adds some other feature like access modifiers and interfaces, so in this lecture we'll be writing TypeScript rather than pure ES6.

Class

A `class` is a blueprint for creating objects with specific functions and properties already attached to it, lets go through a simple example line by line:

```
class Person { ①
    firstName = ""; ②
    lastName = "";
    constructor(firstName, lastName) {  ③
        this.firstName = firstName;
        this.lastName = lastName;
    }

    name() { ④
        return `${this.firstName} ${this.lastName}`;
    }

    whoAreYou() {
        return `Hi i'm ${this.name()}`; ⑤
    }
}
```

① We define a class using then `class` keyword.

② We describe the properties we want on our class instance.

③ Each class has a special `constructor` function, this is called when we create an instance of a class with `new`

④ We describe the functions, also known as methods, that we want on our class instance.

⑤ `this` in a method points to the class instance, the object that is created using this class.

Class Instance

A class is a *blueprint* for creating an object, we call that created object an *instance of a class*, or a *class instance* or just *instance* for short.

We instantiate a class by using the `new` keyword and when that happens javascript calls the `constructor` function. We can pass to the constructer arguments which it uses to initialise properties or call other function, like so:

```
let asim = new Person("Asim","Hussain");
```

The above creates an instance of the *Person* class called *asim*.

The *asim* instance has the same properties and functions that are described on the Person class:

```
let asim = new Person("Asim","Hussain");
asim.whoAreYou()
// "Hi i'm Asim Hussain"
```

Inheritance

A class can inherit from another class. We can create a class blue-print that *extends* an existing class blue-print by adding other methods or properties.

We do this by using the `extends` keyword, like so:

```
class Student extends Person { ①
    course; ②

    constructor(firstName, lastName, course) {
        super(firstName, lastName); ③
        this.course = course;
    }

    whoAreYou() {   ④
        return `${super.whoAreYou()} and i'm studying ${this.course}`; ⑤
    }
}
```

① We use the `extends` keyword to signal that this class inherits all the properties and methods from the *parent* Person class.

② We can describe additional properties.

③ We use the `super` function to call the constructor of the parent class

④ We can *override* member functions of the parent class with our own versions.

⑤ In member functions `super` refers to the `parent` instance.

We can then instantiate this derived class like so:

```
let asim = new Student("Asim", "Hussain", "Angular 2");
console.log(asim.whoAreYou());
// Hi i'm Asim Hussain and i'm studying Angular 2
```

Access Modifiers

Everything we have learned so far about classes is pure ES6 JavaScript.

However TypeScript adds some nice functionality on top of ES6 classes, namely function and property *visibility* via *access modifiers*.

For example we can define the properties of our Person class as `private`, like so:

```
class Person {
    private firstName = "";
    private lastName = "";

    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

And we create a function on our Student class called `test()` which tries to access one of these properties, like so:

```
class Student extends Person {
  .
  .
  .
  test() {
    console.log(this.firstName);
  }
}
```

And we tried to call this function from our `Student` instance, like so:

```
let asim = new Student("Asim", "Hussain", "Angular 2");
console.log(asim.test());
```

Compiling the above with typescript prints out this error:

```
error TS2341: Property 'firstName' is private and only accessible within class
'Person'.
```

By marking the `firstName` property as `private` it is now only visible from one of the methods on `Person` class.

We can also define *class methods* as `private` with the same effect. If we tried to call a `private` method from *outside* of a Person class, the typescript transpiler throws an error.

There are 3 access modifiers:

public

This is the default and means its visible everywhere.

private

Only member functions of the class it's declared in can see this.

protected

Only the class it's declared in and any class that *inherits* from that class can see this.

Constructor shortcut

A really common pattern in constructors is to use them to initialise properties via arguments you pass into the constructor, like in our example:

```
class Person {
    private firstName = "";
    private lastName = "";

    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

As long as you are using an *access modifier* TypeScript lets us shorten this to:

```
class Person {
    constructor(private firstName, private lastName) {
    }
}
```

Interfaces

TypeScript has another feature called an `interface`. An interface can be used in a number of scenarios but by far the most common is when used with classes.

When used with classes the syntax looks like this:

```
class Person implements Human {
}
```

`Human` in the example above is an *interface*. An interface lets you describe the *minimum* set of **public** facing properties or methods that a `class` has.

Another way interfaces are explained is that they describe a set of rules the class has to follow, a *contract* it has to adhere to.

So for us a `Human` interface might look like:

```
interface Human {
    firstName: string;
    lastName: string;
}
```

Since interfaces are all about the **public** interface of a class they can't have access modifiers, the properties above have to be *public*.

If the Person class then doesn't implement at least a `firstName` and a `lastName` then typescript throws an error like so:

```
error TS2420: Class 'Person' incorrectly implements interface 'Human'. Property
'firstName' is missing in type 'Person'.
```

Sometimes however we might want an interface to describe an optional contract. We can append `?` to the *name* of the property or function to mark it as *optional*, like so:

```
interface Human {
    firstName: string;
    lastName: string;
        name?: Function;
        isLate?(time: Date): Function;
}
```

Summary

In ES6 we now have a new way of writing object oriented code with the `class` syntax.

We can inherit methods and properties of one class into another by using the `extends` keyword.

Under the hood we are still using prototype based inheritance but the syntax is easier to understand and more familiar for developers who are coming from other languages.

TypeScript adds some extra functionality on-top of ES6 classes such as *access modifiers* and *interfaces*

Listing

https://github.com/codecraftpro/angular2-sample-code/blob/master/2.es6-typescript/10.class-interface/script.ts

script.ts

```
interface Human {
  firstName: string;
  lastName: string;
  name?: Function;
  isLate?(time: Date): Function;
}

class Person implements Human {
  constructor(public firstName, public lastName) {
  }

  public name() {
    return `${this.firstName} ${this.lastName}`;
  }

  protected whoAreYou() {
    return `Hi i'm ${this.name()}`;
  }
}

class Student extends Person {
  constructor(public firstName, public lastName, public course) {
    super(firstName, lastName);
  }

  whoAreYou() {
    return `${super.whoAreYou()} and i'm studying ${this.course}`;
  }
}

let asim = new Student("Asim", "Hussain", "typescript");
console.log(asim.whoAreYou());
```

Decorators

We've seen in the quickstart that in Angular we can *decorate* a class with extra info using the `@` syntax, like so:

```
@Component({
    selector: "thingy",
    template: `foo`
})
class MyComponent {
}
```

This is a new feature that will *probably* make it into the *ES7* version of JavaScript, it's not available right now however even in the ES6 version.

However the functionality is available in TypeScript, so we can already make use it.

It allows us to *decorate* classes and functions, similar to annotations in java and decorators in python.

Specific Angular implementations might be more complex and harder to read and understand but the concept is actually quite simple.

Simple no-argument decorator

I'm going to explain by creating a decorator called `@course` for our Person class

```
@course
class Person {
    firstName;
    lastName;

    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

`@course` is just a *function*, like so:

```
function course(target) {
    Object.defineProperty(target.prototype, 'course', {value: () => "Angular 2"})
}
```

The first argument to the `course` function is the `target`.

This is the *thing* the decorator is attached to, so for a class it's going to be the *function constructor* for that class, the *under-the-hood* implementation of a class.

Knowing this we can actually dynamically add a function to our Person class by using the `Object.defineProperty` function,

The details of the `Object.defineProperty` function are beyond the scope of this chapter. We use it to add a function called `course` onto the class it decorates and for now this function just returns the string "Angular 2".

We can now call `asim.course()` and this prints out `"Angular 2"`:

```
let asim = new Person("Asim", "Hussain");
console.log(asim.course()); // Angular 2
```

Decorators with arguments

But how do we pass arguments to our decorator, like the way the `@Component` decorator works?

We create a function that *returns* a decorator, like so:

```
function Course(config) { // 1
  return function (target) {
    Object.defineProperty(
        target.prototype,
        'course',
        {value: () => config.course} // 2
    )
  }
}
```

1. We pass a `config` object to the *outer* `Course` function.
2. Then use that `config` in the returned *inner* decorator function.

Now we can use this decorator like so:

```
@Student({
    course: "Angular 2"
})
class Person {
}
```

Summary

Decorators are a new feature of TypeScript and used throughout the Angular code, but they are nothing to be scared of.

With decorators we can configure and customise our classes at design time.

They are just functions that can be used to add meta-data, properties or functions to the thing they are attached to.

A collection of useful decorators, for use in your projects or just to read and learn, can be found here: https://github.com/jayphelps/core-decorators.js

Listing

https://github.com/codecraftpro/angular2-sample-code/blob/master/2.es6-typescript/11.decorators/script.ts

script.js

```
function Student(config) {
    return function (target) {
        Object.defineProperty(target.prototype, 'course', {value: () =>
config.course})
    }
}



@Student({
    course: "angular3"
})
class Person {
    constructor(private firstName, private lastName) {
    }

    public name() {
        return `${this.firstName} ${this.lastName}`;
    }

    protected whoAreYou() {
        return `Hi i'm ${this.name()}`;
    }
}

let asim = new Person("Asim", "Hussain");
//noinspection TypeScriptUnresolvedFunction
console.log(asim.course());
```

Modules

Module Loading

By default javascript doesn't have a module system like other languages, e.g. Java or Python.

This means that if you wanted to call a function in some other file, you have to remember to *explicitly* load that file via script tags *before* you call the function.

If you tried to use code that you forgot to add via a script tag, then javascript would complain.

Other languages have a module loading system e.g. in python if you wanted to use some code from another file you would type something like

```
import foo from bar;
foo();
```

The language itself figured out *where* `bar` was, loaded it up from the filesystem, extracted the function `foo` and made it available to you in your file to use.

This feature was missing in JavaScript so the community developed their own solutions, such as *CommonJS* which is used in node.

ES6 Modules

ES6 took the best of the existing module systems and introduced this concept on a language level.

Although it's made it into the ES6 standard it's up to the javascript engine makers to *actually* implement it natively and they *haven't... yet*.

So until that happens we code using the ES6 module syntax in TypeScript. When typescript transpiles the code to ES5 it uses the CommonJS module loading system which we touched on above.

We can configure TypeScript to use other module loaders, but the default is CommonJS.

Exporting

```
// utils.ts
function square(x) {
    return Math.pow(x,2)
}

function cow() {
    console.log("Mooooo!!!")
}

export {square, cow};
```

We declare some functions in a file.

By using the `export` keyword we say *which* of those functions can be exported, and therefore imported and used in other modules.

`{square, cow}` is just destructuring syntax and is short for `{square: square, cow: cow}`.

Importing

```
// script.ts
import {square, cow} from './utils';

console.log(square(2));
cow();
```

To compile with typescript we need to provide both files on the command line `tsc -t ES5 -w utils.ts script.ts`

We again use that destructuring syntax to import the functions we want from the utils module, we provide a path relative to *this* module.

Aliases

We may want to import a function with one name but then use it via another name. Perhaps to avoid name collisions or just to have a more convenient naming, like so:

```
import {square as sqr} from './utils';
sqr(2);
```

Or we can import everything in a file like so:

```
import * as utils from './utils';
console.log(utils.square(2));
utils.cow();
```

Alternative export syntax

As well as describing the exports by using the export keyword, like so:

```
export {square, cow};
```

We can also export functions or variables as they are defined by prepended the word `export` to the front of their declarations:

```
export function square(x) {
    return Math.pow(x,2)
}
```

Default exports

If a module defines one export which is the most common, we can take advantage of the *default export* syntax, like so:

```
export default function square(x) {
    return Math.pow(x,2)
}
```

And then when we import it we don't need to use { }, like so:

```
import square from './utils';
```

Or, if we want to import the default export as well as some other exports, we can use:

```
import square, { cow } from './utils';
```

Summary

With ES6 modules we finally have a mechanism for letting the language deal with loading of dependant files for us.

This isn't baked into javascript engines yet. So to solve this problem in Angular we still use the ES6 module loading syntax but leave it to TypeScript to transpile to CommonJS.

Listing

https://github.com/codecraftpro/angular2-sample-code/tree/master/2.es6-typescript/12.modules

script.js

```
import * as utils from './utils';
console.log(utils.square(4));
utils.cow();
```

utils.js

```
export function square(x) {
  return Math.pow(x, 2)
}

export function cow() {
  console.log("Mooooo!!!")
}
```

Types

Transpile-time type checking

Some of the most common mistakes we make while writing JavaScript code is to misspell a property or a method name.

We only find out about the mistake when we get a *runtime error*, that is to say when we are running our application.

This is why testing is so important for javascript applications, only *running* them can surface errors.

Other languages like C++ or Java have something called *type checking*, during a compilation phase it first checks for some simple mistakes in the code and raises an error if it finds any.

TypeScripts transpilation mechanism also performs *type checking*, however it only works when we tell TypeScript the *type* of things.

e.g. in javascript we might write:

```
let a;
a = 1;
a = '2';
```

We expected the type of the `a` variable to remain a number over its life. But by mistake we assigned a string `'2'` to `a`, this doesn't cause an error in JavaScript.

Only if one of the functions which depends on `a` starts misbehaving do we even get a clue that something might be wrong.

However with typescript we can clearly state that we want the `a` variable to always hold a number, like so:

```
let a: number = 1;
```

Then if somewhere else in our code we wrote:

```
a = "1";
```

We get this error:

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

If we take advantage of types in typescript the transpiler will warn us with a transpile-time errors.

Supported Types

Basic Types

We can support boolean, number and string.

```
let decimal: number = 6;
let done: boolean = false;
let color: string = "blue";
```

Arrays

We have two ways of describing the types of arrays.

The first is to use the brackets notation `[]`, like so:

```
let list: number[] = [1, 2, 3];
```

The above indicates to TypeScript that this array should only hold numbers.

The second way uses a generic type specifically `Array<Type>`, like so:

```
let list: Array<number> = [1, 2, 3];
```

We cover *generics* in more detail later on in this lecture.

Functions

We can describe a variable as one that will only point to a function, like so:

```
let fun: Function = () => console.log("Hello");
```

With TypeScript we can also define the *expected return type* of a function, like so:

```
function returnNumber(): number {
  return 1;
}
```

The above lets TypeScript know that the function should only return a number.

Enum

An Enum is a datatype consisting of a set of named values. The names are usually identifiers that behave as constants. Enums were introduced in ES6.

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}

let go: Direction;
go = Direction.Up;
```

Class & Interface

Classes and Interfaces, whether created by you or others, are also types of things. So we can use them where we would use any of the built-in types:

```
class Person {};
let person: Person;
let people: Person[];
```

Any

If we don't know the type of something we can fall back to using `any`.

If used it doesn't provide any type checking but does make it clear that we intentionally *don't know* the type, rather than we forgot to add the type.

```
let notsure: any = 1;
notsure = "hello"; // This is fine since we don't do type checking with any
```

Void

`void` means no type, it's typically used to indicate that a function will not return anything at all, like so:

```
function returnNothing(): void {
  console.log("Moo");
}
```

Type Assertion

Sometimes we end up in a situation where we know more than TypeScript about the type of something. In those situations we can use *type assertions* as a hint to the transpiler about the type of something, like so:

```
let value: any = "Asim Hussain";
let length: number = (<string>value).length;
```

`(<string>value)` lets TypeScript know that we believe the type of value to be string.

Generics

How can we create re-usable bits of code which can still take advantage of typescripts transpile-time type checking?

Take for instance this example:

```
class Audio {}
class Video {}
class Link {}
class Text {}

class Post {
    content: any;
}
```

We have a class called Post which has a `content` property. The content might be an instance of Audio, Video, Link or Text classes.

We could ignore the type of content and just use `any` like the example above but this doesn't give us the benefit of type checking.

We can create seperate `AudioPost`, `VideoPost`, `LinkPost` and `TextPost` types, like so:

```
class AudioPost {
    content: Audio;
}

class VideoPost {
    content: Video;
}

class LinkPost {
    content: Link;
}

class TextPost {
    content: Text;
}
```

But apart from being verbose and time-consuming it assumes we know all the content types upfront. Maybe a consumer later on would like to create a `VRPost` type.

With *generics* we can dynamically generate a new type by passing into the Post type a *type variable*, like so:

```
class Audio {}
class Video {}
class Link {}
class Text {}

class Post<T> {
    content: any;
}
```

`<T>` above is the *generic* syntax and `T` is the *type variable*. We could name it anything but the convention if there is only one is to call it just `T`.

Then we can use `T` wherever we would use a type, like so:

```
class Post<T> {
    content: T;
}
```

Finally when we want to create a specific type of Post we can declare it like so:

```
let videoPost: Post<Video>;
```

Optional Types

By default we don't *have* to add types when using TypeScript, we could just leave them out.

```
let answer;
answer = 42;
```

TypeScript won't perform any type checking for the above and assumes a type of `any`.

This behaviour is controlled by the `noImplicitAny` flag in `tsconfig.json`. If set to `false` then TypeScript assumes `any` for missing types, of set to `true` then TypeScript throws an error if no type is present.

Opinion is split as to whether to keep this to `true` or `false`, by default it's set to `false` which is more forgiving but then doesn't force developers to use types which is the point of TypeScript.

Type Inference.

If a variable is *declared and initialised on one line* TypeScript will try to infer, *guess*, the type of a variable by how it's declared, for example:

```
let answer = 42;
answer = "42";
```

TypeScript inferred that the type of answer was number by the fact we initialised it with a number. When we later on try to assign a string to `answer` it throws an error.

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

Types for external libraries

This is all fine for code that's written in TypeScript and has types.

What if you wanted to use code that wasn't written in TypeScript or which you are not going to include as TypeScript and compile yourself.

We can use something called an *ambient type definition*.

This is a file that contains meta-data about the types in another library, a *meta-type* file.

This repository (https://github.com/DefinitelyTyped/DefinitelyTyped) contains type definitions for some of the most popular 3rd party javascript libraries.

To use them in a file we simply:-

1. Download the associated type file, for example `jquery.d.ts`.
2. In the typescript file where we want to *use* the 3rd party library, add this to the top of the file:

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

typings

Thats a bit cumbersome so there is also a command line tool you can use called `typings` which handles the leg-work for you.

We install it via `npm`, like so:

```
npm install -g typings
```

In our project folder we then initialise a config file by typing `typings init` this creates a `typings.json` file.

Then we can install type definition files like so:

```
typings install jquery --save --source dt --global
```

This downloads and installs the type definition files to the `./typings` folder and conveniently bundles all of the downloaded files into a single file called `index.d.ts`.

So then to use jQuery with typescript type support we just need to add a reference to `index.d.ts` to the top the file where we use jQuery.

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

Summary

With transpile-time type checking TypeScript can help uncover bugs much earlier and faster than if they were to manifest at run-time.

Using types is optional but highly recommended by the Angular team.

If using 3rd party libraries that have already been transpiled into javascript, typescript can still perform transpile-time type checking if we include the type definition file for the library.

Listing

https://github.com/codecraftpro/angular2-sample-code/blob/master/2.es6-typescript/13.types/script.ts

script.js

```
"use strict";

// Core
let decimal: number = 6;
let done: boolean = false;
let color: string = "blue";
let list: number[] = [1, 2, 3];
let list2: Array<number> = [1, 2, 3];

// Function
let fun: Function = () => console.log("Hello");
function returnNumber(): number {
  return 1;
}

// Void
function returnNothing(): void {
  console.log("Moo");
}

// Enum
enum Direction {
  Up,
  Down,
  Left,
  Right
}
let go: Direction;
go = Direction.Up;


// Class
class Person {
}
let person: Person;
let people: Person[];

// Any
let notsure: any = 1;
notsure = "hello"; // This is fine since we don't do type checking with any

// Type Assertion
let value: any = "Asim Hussain";
let length: number = (<string>value).length;


// Generics
class Audio {
}
```

```
class Video {
}

class Post<T> {
  content: T;
}

let audioPost: Post<Audio>;
let videoPost: Post<Video>;
```

Wrapping Up

In this section we covered the basics of ES6 JavaScript and TypeScript.

We discussed how TypeScript is a super-set of ES6 JavaScript.

We explained how to install the command line tools so you can transpile TypeScript locally on your computer.

We covered the core features of ES6 such as `let`, `const`, template strings, fat arrow functions; `for-of` loops, Map and Set; as well as how to deal with asynchronous programming by using Promises.

With TypeScript we covered classes, class access modifiers, interfaces; decorators, modules and types including generic types.

We've covered a lot of topics but still this is just the *essentials* required to build Angular applications, there is more to learn in each of these areas.

Further Reading

If you would like to learn more about ES6 I recommend reading this book (it's free): https://github.com/getify/You-Dont-Know-JS/tree/master/es6%20%26%20beyond

If you would like to learn more about TypeScript the official documentation is a good place to start: https://www.typescriptlang.org/docs/tutorial.html

Angular CLI

Angular CLI

Angular now comes with a command line interface (CLI) to make it easier and faster to build Angular applications.

The Angular CLI at the time of writing (09/2016) is still in a prototype stage and has a number of issues that still need to be addressed. The biggest is that it doesn't currently work with the Angular router.

Even without support for the router however the CLI is a useful if not essential tool.

Features

The Angular CLI helps with:

Bootstrapping a project

It creates the initial project structure with a root `NgModule` and a root component and bootstraps it using the `platformBootstrapDynamic` method.

The project is also configured to use the *webpack loader* which handles things like module loading, bundling and minification of dependant code.

In the course we've used SystemJS for this since webpack doesn't work with Plunker *yet*. We'll continue to use SystemJS for the code samples in Plunker and WebPack for any applications created with the Angular CLI.

Serving and live reloading

The CLI starts a local web-server so we can view our application in the browser via localhost:4000.

The CLI also watches for any changes to our files and automatically reloads the webpage if there are any.

Code generation

Using the CLI we can create components directives, services, pipes etc... all from the command line with all the necessary files, folders and boilerplate code included.

All the generated code adheres to the official Angular style guide.

In Angular 1 the Angular team never supported an official style guide. This meant that most projects ended up looked pretty different to each other. A developer moving teams would have to figure out from scratch how *this* team likes to write Angular 1 code.

Testing

The generated code also comes with bootstrapped jasmine test spec files, we can use the CLI to

compile and run all the tests with a single command.

Whenever the CLI detects changes to any file it re-runs all the tests automatically in the background.

Packaging and releasing

The CLI doesn't just stop with development, using it we can also package our application ready for release to a server.

Installing the Angular CLI

To install the CLI we use Node and npm.

```
npm install -g angular-cli
```

If the above ran successfully it will have made available to you a new application called `ng`, to test this installed correctly run this command:

```
ng -v
```

It *should* output the version of the application that was installed, like so:

```
angular-cli: 1.0.0-beta.15
node: 6.4.0
os: darwin x64
```

Start an application with `ng new`

Lets create a new project called `codecraft`.

To bootstrap our new project with `ng` we run this command:

```
ng new codecraft
```

This command might take sometime to run, be patient.

This outputs something like the below:

```
➜ Scratch ng new codecraft
installing ng2
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/app/index.ts
  create src/app/shared/index.ts
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create src/typings.d.ts
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
```

The command generates a number of new files and folders for us:

```
codecraft
// production or development builds of our applicaiton go here.
 ├─── dist

// main application code goes here.
 ├─── src
 |    ├─── app
 |    |    ├─── app.component.css
 |    |    ├─── app.component.html
 |    |    ├─── app.component.spec.ts
 |    |    ├─── app.component.ts
 |    |    ├─── app.module.ts
 |    |    ├─── index.ts
 |    |    └─── shared
 |    |         └─── index.ts

// settings for the different environments, dev, qa, prod.
 |    ├─── environments
 |    |    ├─── environment.prod.ts
 |    |    └─── environment.ts

// main html and typescript file
 |    ├─── index.html
 |    ├─── main.ts

 |    ├─── favicon.ico
 |    ├─── polyfills.ts
 |    ├─── styles.css
// prepares test environment and runs all the unit tests
 |    ├─── test.ts

// typescript configuration file
 |    ├─── tsconfig.json

// typescript type definition file
 |    └─── typings.d.ts

// The E2E tests for our application go here
 ├─── e2e

 ├─── angular-cli.json
 ├─── karma.conf.js
 ├─── package.json
 ├─── protractor.conf.js
 ├─── README.md
 └─── tslint.json
```

The directory structure follows the **recommended** app structure and style guide.

As well as creating the files and folders for us; we can see from `package.json` that it installed the correct versions of all the required npm dependencies for us also.

```
{
  "name": "activity",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^5.0.0",
    "@angular/common": "^5.0.0",
    "@angular/compiler": "^5.0.0",
    "@angular/core": "^5.0.0",
    "@angular/forms": "^5.0.0",
    "@angular/http": "^5.0.0",
    "@angular/platform-browser": "^5.0.0",
    "@angular/platform-browser-dynamic": "^5.0.0",
    "@angular/router": "^5.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.5.2",
    "zone.js": "^0.8.14"
  },
  "devDependencies": {
    "@angular/cli": "1.5.0",
    "@angular/compiler-cli": "^5.0.0",
    "@angular/language-service": "^5.0.0",
    "@types/jasmine": "~2.5.53",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codelyzer": "~3.2.0",
    "jasmine-core": "~2.6.2",
    "jasmine-spec-reporter": "~4.1.0",
    "karma": "~1.7.0",
    "karma-chrome-launcher": "~2.1.1",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~3.2.0",
    "tslint": "~5.7.0",
    "typescript": "~2.4.2"
  }
}
```

So far in this course we have bundled all our code into one file on plunker for convenience.

Lets see how the Angular CLI breaks up the code into multiple files and where those files are located.

src/app/app.component.ts

The new project is bootstrapped with one component, our root component which it called `AppComponent` and has a selector of `app-root`.

src/app/app.component.ts

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

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

src/index.html

`app-root` component has been added to our `index.html` file already.

There are no script tags present yet, that's fine the angular build process adds all the required script and link tags for us.

src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Activity</title>
  <base href="/">

    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

src/app/app.module.ts

Our top level module configuration is stored in this file.

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

import { AppComponent } from './app.component';
import { JokeComponent } from './joke/joke.component';
import { JokeListComponent } from './joke-list/joke-list.component';
import { JokeFormComponent } from './joke-form/joke-form.component';
import { HeaderComponent } from './header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    JokeFormComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

src/main.ts

The actual act of importing our main module and boostrapping our Angular web application is left to the `main.ts` file.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

Serve an application with `ng serve`

With the CLI we can also easily serve our application using a local web-server.

We just run:

```
ng serve
```

This builds our application, bundles all our code using webpack and makes it all available through `localhost:4200`.

`ng serve` also watches for any changes to files in our project and auto-reloads the browser for us.

The command runs the application through a web-server that support HTML5 *push-state* routing.

The above will make sense once we cover Routing later on in this book.

Generate code with `ng generate`

The ability to generate stub code is one of the most useful features of the CLI.

The most exciting part of this is that it automatically generates code that adheres to the official style guide.

Projects built using the Angular CLI should look like each other. Developers who are used to the way Angular CLI generates files are going to find it easier to work on multiple different projects, as long as they all use the Angular CLI.

With the generate command we can create new components, directives, ~~routes~~ *not available in version 1.0.0-beta.15*, pipes, services, classes, interfaces and enums.

Each of the above *types of things* it can create is called a **scaffold**.

We can run this command using `ng generate <scaffold> <name>`

If we wanted to generate a component called `HeaderComponent` we would write:

```
ng generate component Header
```

This creates a number of files in a folder called `header` in `src/app`, like so:

```
app
 ├─── header
 |    ├─── header.component.css // The css for this component
 |    ├─── header.component.html // The template for this component
 |    ├─── header.component.spec.ts // The unit test for this component
 |    └─── header.component.ts // The component typescript file
```

Taking a look at `header.component.ts`:

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

Don't name your component `HeaderComponent`. Angular CLI automatically appends `Component` to the name, so you component class would end up being `HeaderComponentComponent`.

The command above can be shortened to:

```
ng g c Header
```

If we run the command in an app folder, the `generate` command will create files **relative to the current folder you are in**. So if we are in `src/app/header` and we run `ng g c LoginButton` it will generate the files in `src/app/header/login-button/`

We can also be explicit about where we want the generated files to go by running `ng g component ./src/app/foo/bar` this will create a component called `BarComponent` in the folder `./src/app/foo/bar`.

Available Scaffolds

Component

```
ng g component My // Creates MyComponent
```

By default all generated files go in into `src\app\my-component`, a folder called `my-component` is created for us.

Directive

```
ng g directive My // Creates MyDirective
```

By default all generated files go in into `src\app`, *no folder is created.*

Pipe

```
ng g pipe My // Creates MyPipe
```

By default all generated files go in into `src\app`, *no folder is created.*

Service

```
ng g service MyService // Creates MyService
```

By default all generated files go in into `src\app`, *no folder is created.*

Class

```
ng g class MyClass // Creates MyClass
```

By default all generated files go in into `src\app`, *no folder is created.*

Interface

```
ng g interface MyInterface // Creates MyInterface
```

By default all generated files go in into `src\app`, *no folder is created.*

Enum

```
ng g enum MyEnum // Creates MyEnum
```

By default all generated files go in into `src\app`, *no folder is created.*

Create a build with `ng build`

The `ng serve` command does a great job of enabling development locally.

However *eventually* we will want some code which we can host on another server somewhere.

The Angular CLI again has us covered in this regard, if we want to create a development build we simply type

```
ng build
```

This bundles all our javascript, css, html into a smaller set of files which we can host on another site simply.

It outputs these files into the `dist` folder:

```
.
├──── assets
├──── index.html
├──── inline.js
├──── inline.map
├──── main.bundle.js
├──── main.map
├──── styles.bundle.js
└──── styles.map
```

To serve our built application site we just need to serve this folder. For example if using python we could simply run `python -m SimpleHTTPServer` from the `dist` folder and view the application from `0.0.0.0:8000`.

Production Builds

By default the `ng build` command creates a development build, no effort is made to optimise the code.

To create a production build we just run

```
ng build --prod
```

This might generate an output like the below:

```
.
├──── assets
├──── index.html
├──── inline.js
├──── main.3f26904b701596b6d90a.bundle.js
├──── main.3f26904b701596b6d90a.bundle.js.gz
└──── styles.b52d2076048963e7cbfd.bundle.js
```

Running with `--prod` changes a few things:

- The bundles now have random strings appended to them to enable **cache busting**.

This ensures that a browser doesn't try to load up previously cached versions of the files and instead load the new ones from the server.

- The file sizes are much smaller. The files have been processed through a minifier and uglifier.
- There is a much small `.gz` file, this is a compressed version of the equivalent javascript file. Browsers will automatically try to download the `.gz` version of files if they are present.

Adding a third party module

The build system simplifies the process of serving and releasing your application considerably. It works only because Angular knows about all the files used by your application.

So when we include 3rd party libraries into our application we need to do so in such a way that Angular knows about the libraries and includes them in the build process.

Bundled with the main application javascript files

If we want to include a module to use in our Angular javascript code, perhaps we want to to use the moment.js library, we just need to install it via npm like so:

```
npm install moment --save
```

If we also want to include the typescript type definition file for our module we can install it via:

```
npm install @types/moment --save
```

Now when Angular create a build either when releasing or serving locally, the moment library is automatically added to the bundle.

Global Library Installation

Some javascript libraries need to be added to the global scope, and loaded as if they were in a script tag.

We can do this by editing the `angular-cli.json` file in our project root.

The twitter bootstrap library is a great example of this, we need to incldue css and script files in the global scope.

First we install the bootstrap library via npm like so:

```
npm install bootstrap@next
```

Then we add the required javascript files to the `app.scripts` section or the `app.styles` in `angular-cli.json` like so:

```
{
  .
  .
  .
  "apps": [
    {
      .
      .
      .
      "styles": [
        "styles.css",
        "../node_modules/bootstrap/dist/css/bootstrap.css"
      ],
      "scripts": [
        "../node_modules/jquery/dist/jquery.js",
        "../node_modules/tether/dist/js/tether.js",
        "../node_modules/bootstrap/dist/js/bootstrap.js"
      ],
      .
      .
      .
    }
  ],
  .
  .
  .
}
```

Now when the build runs the CLI includes those files in the bundle and injects then in the global scope.

Testing Angular

Angular has always been synonymous with testing and so there should be no surprise that the command line tool comes with features to make Angular testing easier.

The default mechanism for unit testing in Angular is via jasmine and karma.

Whenever we generate code via scaffolds it also generates a `.spec.ts`. The code the CLI bootstraps inside this file depends on the scaffold type but essentially is a jasmine test spec which you can flesh out with more test cases.

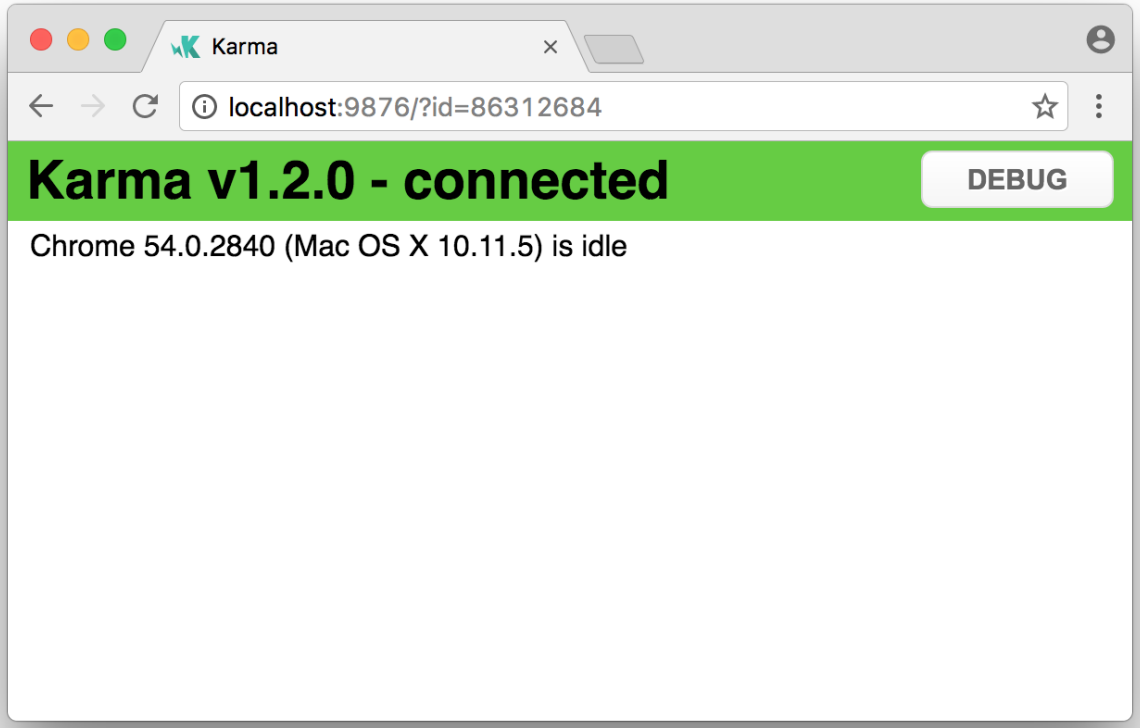
These types of tests are called *unit tests* because we should be writing the tests so we only test one *unit* of code and each test case is independent of the others.

We can run all our unit tests with one command:

```
ng test
```

This builds our project and then runs all the tests, any errors are output to the terminal.

This command also watches for any changes in our files and, if it detects any, re-runs the tests automatically.



When running the tests it opens up a browser window like the example above.

It needs this browser windows to run the tests, do not close it!

Summary

The above is just an overview of the main commands and their default features.

To find out more details about each command and how we can customise the behaviour via flags we can run `ng help` in the terminal.

By handling the setup for us the CLI has made working with Angular much easier.

By standardising setup and structure it's also made Angular projects fungible. Angular developers used to the Angular CLI should feel comfortable on **all** Angular CLI projects and be able to hit the floor running.

Activity

In this activity you will re-create the Joke application we've ran using Plunker as an Angular CLI project instead.

Steps

1. Ensure you have *node* installed on your computer. If you are unsure how to install node then follow the instructions in the Node lecture in the Appendix to this course.
2. Follow the instructions in the previous lectures and on the Angular CLI repository site to install the Angular CLI on your computer.
3. Crete a project locally on your computer.
4. Mirror the functionality of the Joke application in the plunker below in your Angular CLI powered project.

http://plnkr.co/edit/b0F6Dhb40Hm5zfiamAix?p=preview

Solution

When you are ready compare your answer to the solution in this repository: https://github.com/codecraftpro/angular-2-book-cli-activity

Components

Overview

Components are the fundamental building block of Angular applications.

Components are *composable*, we can build larger Components from smaller ones.

An Angular application is therefore just a tree of such Components, when each Component renders, it recursively renders its children Components.

At the root of that tree is the top level Component, the *root* Component.

When we *bootstrap* an Angular application we are telling the browser to render that top level *root* Component which renders it's child Components and so on.

In this section we build upon the knowledge learned in the quickstart and cover:

- How to architect an application using Components.
- The Component decorator in more depth.
- Content Projection
- Component Lifecycle Hooks
- View Children vs. Content Children

Architecting with Components

Learning Objectives

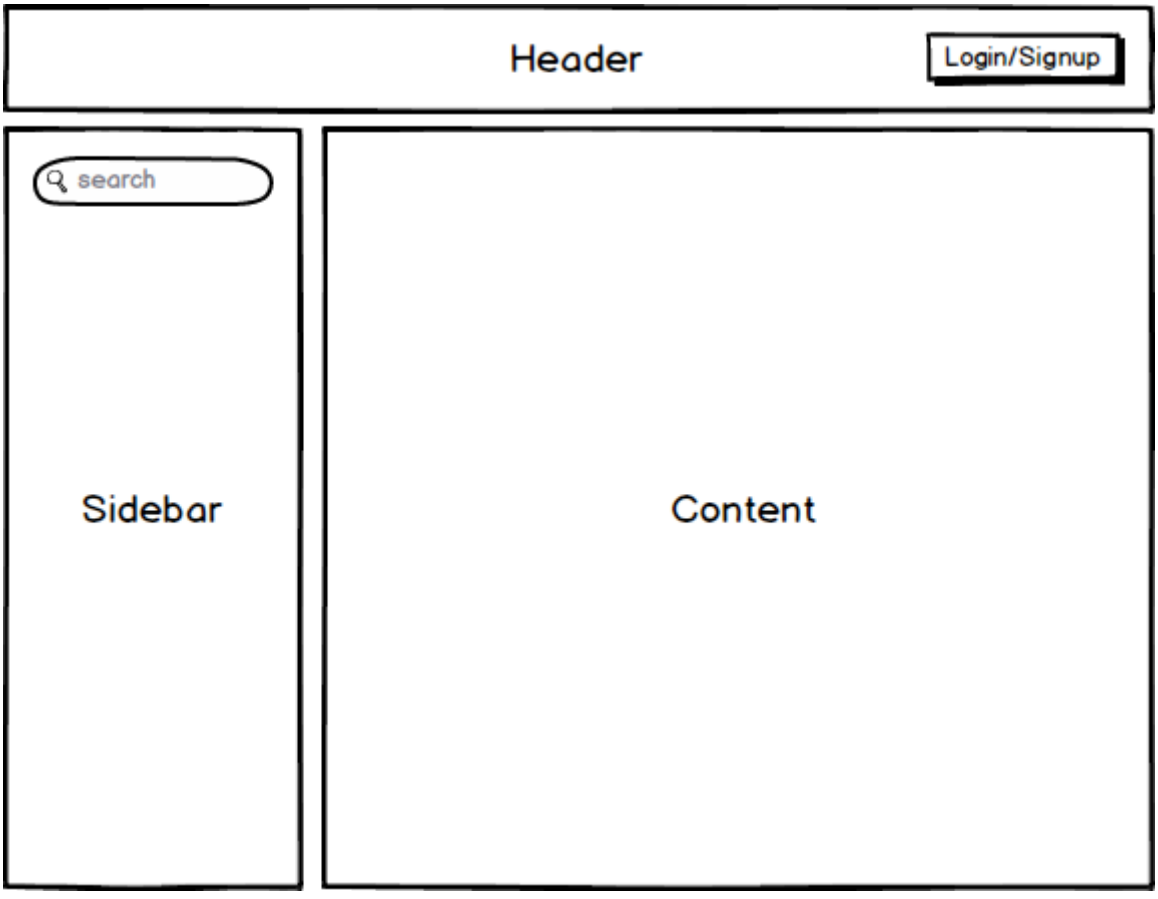
- Know how to *architect* an angular Application.

Process

When building a new Angular application, we start by:

1. Breaking down an applications into seperate Components.
2. For each component we describe it's *resonsibilities*.
3. Once we've desribed the responsibilites we then describe it's *inputs & outputs,* it's public facing interface.

Take for instance this simple example.



We have a basic application with a Header, Sidebar and Content area, we would implement each of these as their own Component. The next stage is for each component to list out the responsibilities, inputs and outputs, like so:

HeaderComponent

Responsibilities

All aspects of authentication. Letting the user login/signup and logout.

Inputs

None

Outputs

- `LoginChanged` - An output event that is fired when the users login state changes.

SidebarComponent

Responsibilities

Performing searches

Inputs

None

Outputs

- `SearchTermChanged` - An output event that is fired when a user performs a search, `$event` contains the search term.

ContentComponent

Responsibilities

Showing the search results.

Inputs

- `SearchTerm` the search term that we want to filter the results by.

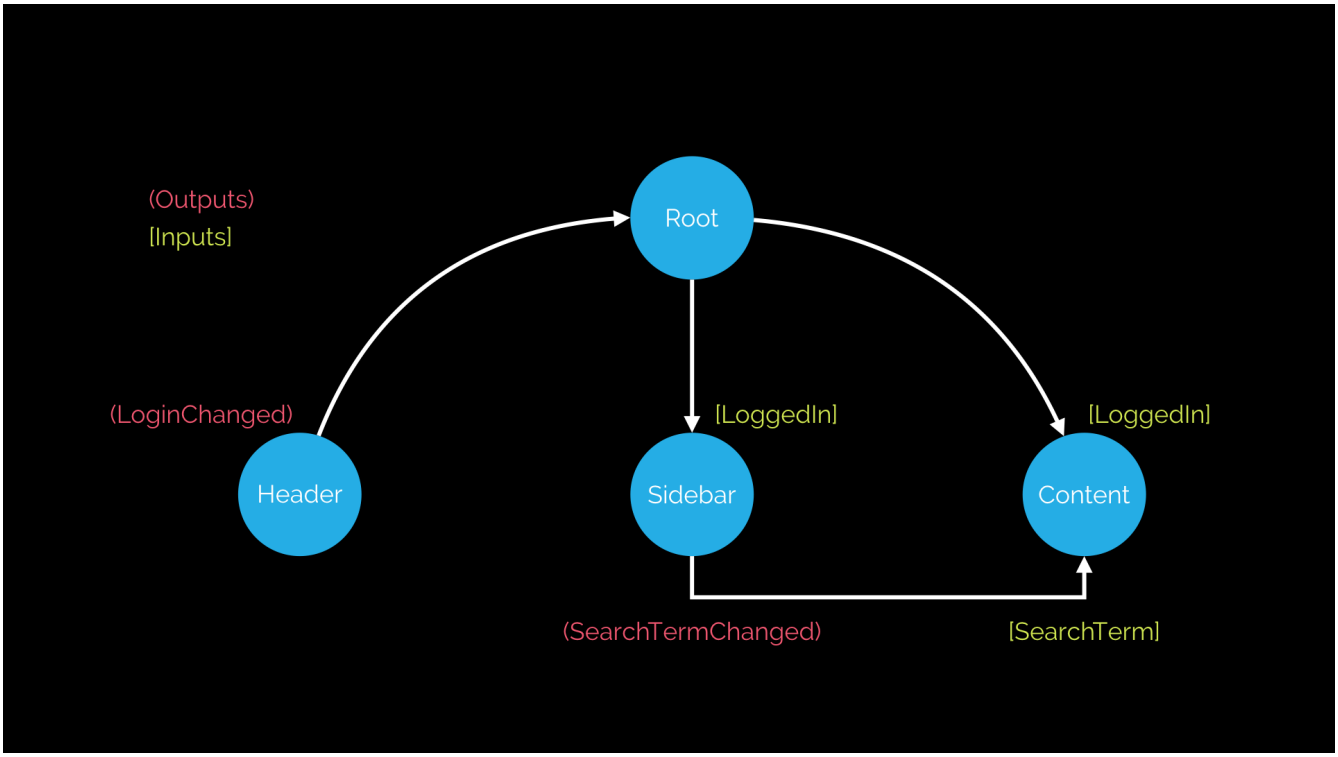
Outputs

None

Listing the inputs and outputs and what area this Component is responsible for helps to ensure the Components are architected correctly. The goal is for each Component to have a well defined *boundary*.

Data Flow

When we link up the Components above with our applications root Component the flow of data between all the inputs and outputs might look something like this:



The actual binding of inputs and outputs happens in HTML, in the templates of Components. The template for our root Component might end up looking like so:

```
<header (loginChanged)="loggedIn = $event"></header>
<sidebar (searchTermChanged)="searchTerm = $event"></sidebar>
<content [searchTerm]="searchTerm"></content>
```

Data flow describes how we **glue** Components together through their inputs and outputs.

Closely looking at the diagram above an interesting fact occurs; with one way data binding, *inputs go down* the tree, *outputs go up* the tree. With one way data binding reasoning about your application becomes a lot simpler, we can trace through the flow of events in our application easily.

Summary

Architecting an Angular application means understanding that it's just a tree of Components, each Component has some inputs and outputs and should have a clear responsibility with respect to the rest of the application.

Components are *composable*, so we might go a step further and include a `LoginButtonComponent` in our `HeaderComponent`. But only if we would want to re-use the `LoginButtonComponent` independently of the `HeaderComponent`.

Templates, Styles & View Encapsulation

Learning Objectives

We've covered the basics of the `@Component` decorator in the quickstart. We explained how decorators work and both the `template` and `selector` configuration properties of the `@Component` decorator.

In this lecture we will go through a number of other configuration properties including `templateUrl`, `styles`, `styleUrls` and `encapsulation`.

In the section on Dependency Injection we will cover two other ways of configuring Components with the `providers` and `viewProviders` properties.

templateURL

We don't have to write our template code inline with our component code. We can store our HTML template files separately and just refer to them in our component by using the `templateUrl` property.

Using the joke application we built in the quickstart, lets move the template for the `JokeFormComponent` to a file called `joke-form-component.html`, like so:

```
@Component({
  selector: 'joke-form',
  templateUrl: 'joke-form-component.html' ①
})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}
```

① We point our component to an external template file by using the `templateUrl` property.

styles

We can also specify any custom css styles for our component with the `styles` property.

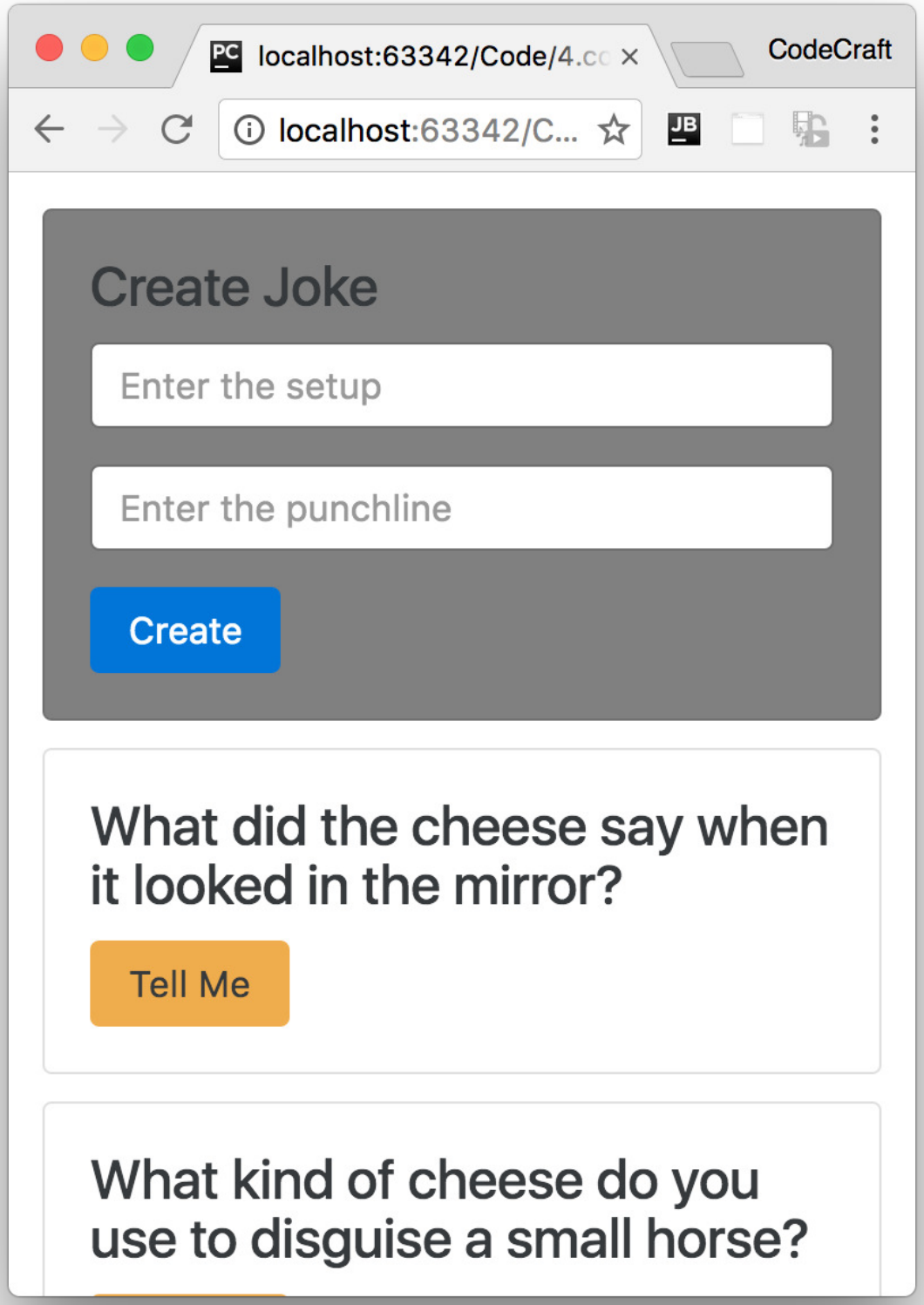
`styles` takes an *array of strings* and just like `template` we can use multi-line strings with back-ticks.

Let's define a style for the `JokeFormComponent` so it gives it a background color of gray.

```
@Component({
  selector: 'joke-form',
  template: 'joke-form-component.html',
  styles: [
    `
    .card {
      background-color: gray;
    }
    `
  ],
})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}
```

The `styles` property above takes an **array** of strings, each string can contain any number of CSS declarations.

The form component in our application now turns gray, like so:



View Encapsulation

Even though we changed the background color of `.card` and we have multiple cards on the page only the form component card was rendered with a gray background.

Normally if we change a css class the effect is seen throughout an application, something special is happening here and it's called *View Encapsulation*.

Angular is inspired from Web Components, a core feature of which is the shadow DOM.

The shadow DOM lets us include styles into Web Components without letting them *leak* outside the component's scope.

Angular also provides this feature for Components and we can control it with the `encapsulation` property.

The valid values for this config property are:

- `ViewEncapsulation.Native`
- `ViewEncapsulation.Emulated`
- `ViewEncapsulation.None`.

The default value is `ViewEncapsulation.Emulated` and that is the behaviour we are currently seeing.

ViewEncapsulation.Emulated

Let's inspect the form element with our browsers developer tools to investigate what's going on.

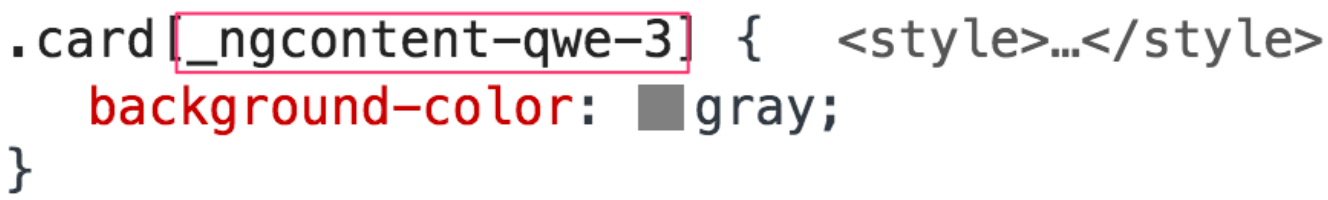
By looking at the DOM for our `JokeFormComponent` we can see that Angular added some *automatically* generated attributes, like so.

```
▼<joke-form _nghost-qwe-3>
  ▼<div _ngcontent-qwe-3 class="card card-block">
      <h4 _ngcontent-qwe-3 class="card-title">Create
      Joke</h4>
    ▶<div _ngcontent-qwe-3 class="form-group">…</div>
    ▶<div _ngcontent-qwe-3 class="form-group">…</div>
      <button _ngcontent-qwe-3 class="btn btn-primary"
      type="button">Create
        </button>
      ::after
    </div>
  </joke-form>
```

Specifically it added an attribute called `_ngcontent-qwe-3`.

The other components on the page don't have these automatically generated attributes, only the `JokeFormComponent` which is the only component where we specified some styles.

Again by looking at the styles tab in our developer tools we can see a style is set for `_ngcontent-qwe-3` like so:



The css selector `.card[_ngcontent-qwe-3]` targets *only* the `JokeFormComponent` since that is the only component with a html attribute of `_ngcontent-qwe-3`.

In the `ViewEncapsulation.Emulated` mode Angular changes our generic css class selector to one that target just a single component type by using automatically generated attributes.

This is the reason that *only* the `JokeFormComponent` is gray despite the fact that we use the same card class for all the other `JokeComponents` as well.

Any styles we define on a component *don't leak out* to the rest of the application but with `ViewEncapsulation.Emulated` our component still inherits global styles from twitter bootstrap.

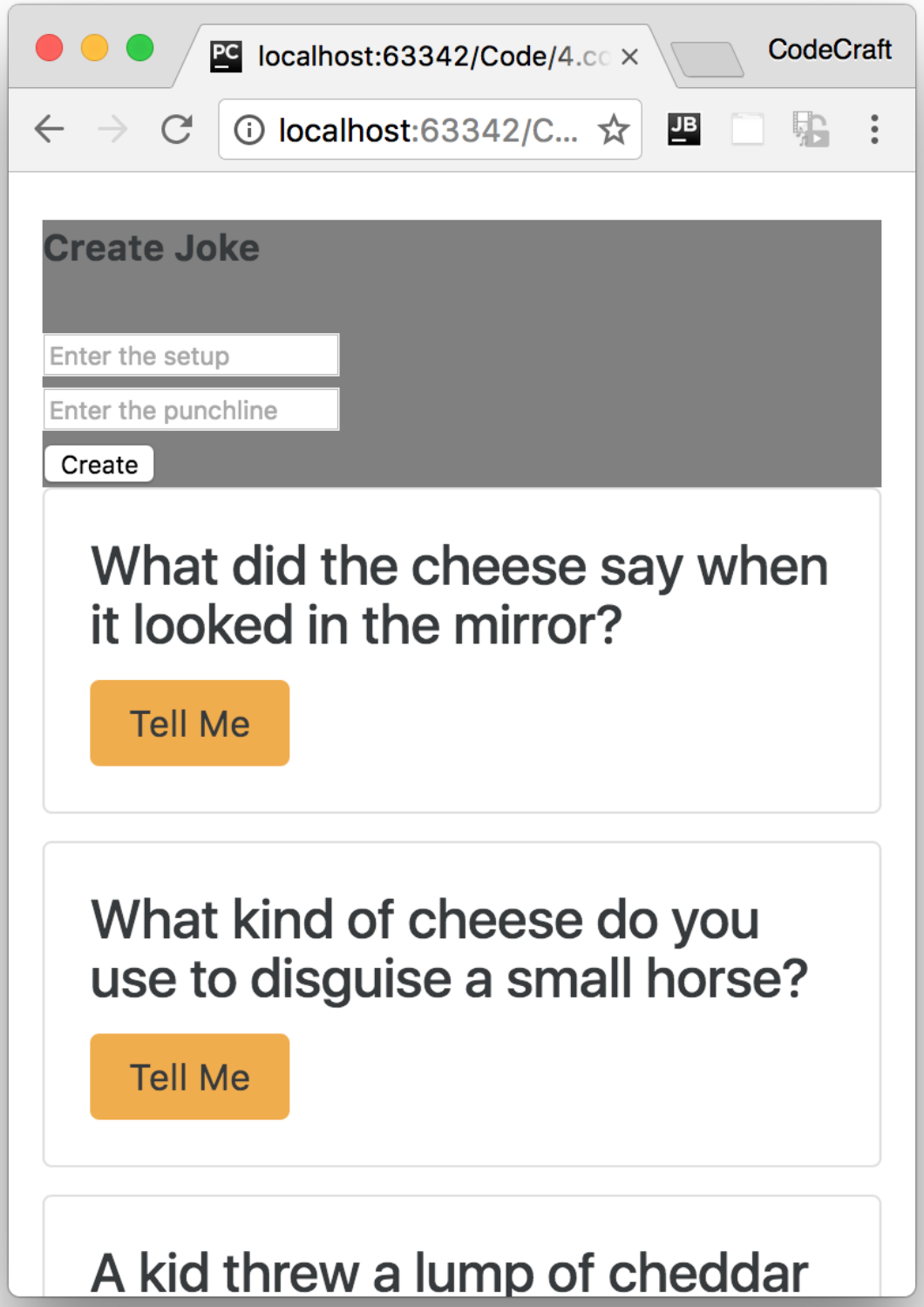
Our `JokeFormComponent` still gets the global card styles from twitter bootstrap and the encapsulated style from the component itself.

ViewEncapsulation.Native

If we want Angular to use the *shadow DOM* we can set the encapsulation parameter to use `ViewEncapsulation.Native` like so:

```
@Component({
  selector: 'joke-form',
  templateUrl: 'joke-form-component.html',
  styles: [
    `
    .card {
      background-color: gray;
    }
    `
  ],
  encapsulation: ViewEncapsulation.Native # <!>
})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}
```

But now if we look at the application although the background color of the `JokeFormComponent` is still gray, we've *lost* the global twitter bootstrap styles.



With `ViewEncapsulation.Native` styles we set on a component *do not leak outside* of the components scope. The other cards in our application do not have a gray background despite the fact they all still use the card class.

This is great if we are defining a 3rd party component which we want people to use in isolation. We

can describe the look for our component using css styles without any fear that our styles are going to leak out and affect the rest of the application.

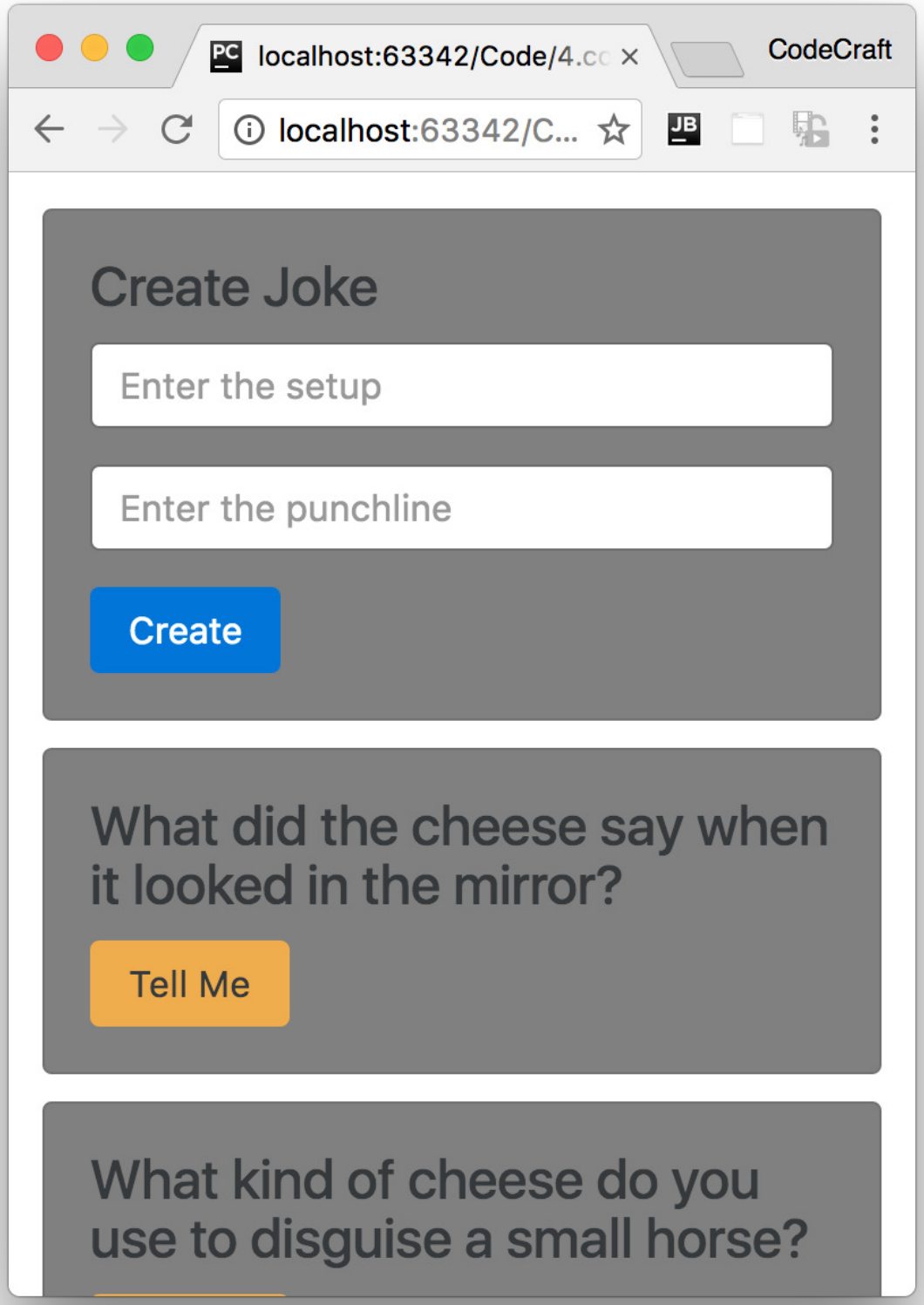
However with `ViewEncapsulation.Native` our component is also isolated from the global styles we've defined for our application. So we don't inherit the twitter bootstrap styles and have to define all the required styles on our component decorator.

Finally `ViewEncapsulation.Native` requires a feature called the *shadow DOM* which is not supported by all browsers.

ViewEncapsulation.None

And If we don't want to have any encapsulation at all, we can use `ViewEncapsulation.None`.

The resulting application looks like so:



By doing this all the cards are now gray.

If we investigate with our browser's developer tools we'll see that Angular added the `.card` class as a *global style* in the head section of our HTML.

```
<head>
  <link rel="stylesheet" href="https://
  maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.4/css/
  bootstrap.min.css">
  <script src="https://unpkg.com/core-js/client/
  shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.6.23?
  main=browser"></script>
  <script src="https://unpkg.com/reflect-
  metadata@0.1.3"></script>
  <script src="https://unpkg.com/systemjs@0.19.27/dist/
  system.src.js"></script>
  <script src="systemjs.config.js"></script>
▶<script>…</script>
  <style>
          .card {
              background-color: gray;
          }
          </style>
</head>
```

We are not encapsulating anything, the style we defined in our card form component has leaked out and started affecting the other components.

styleURLs

Like the `templateUrl` property, with the `styleUrls` property we can externalise the css for our component into another file and include it in.

However like the `styles` parameter, the `styleUrls` param takes an *array* of css files, like so:

```
@Component({
  selector: 'joke-form',
  templateUrl: 'joke-form-component.html',
  styleUrls: [
    'joke-form-component.css'
  ]
})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}
```

Depreciated Properties

If you have seen code that discusses the additional `@Component` properties; `directives`, `pipes`, `inputs` and `outputs` these were in the *beta* version of Angular and were removed in the final 2.0 release. So the information you've read is unfortunately outdated.

Summary

We can externalise our HTML template into a separate file and include it in with the `templateUrl` property.

We can also define styles for our component via the `styles` and `styleUrls` property.

By default styles for our components are *encapsulated*, that means that they don't *leak* out and affect the rest of the application.

We can explicitly set the encapsulation strategy using the `encapsulation` property.

By default, the renderer uses ViewEncapsulation.Emulated if the view has styles, otherwise ViewEncapsulation.None. There is also a ViewEncapsulation.Native method which uses the *shadow DOM* to encapsulate the view.

Listing

http://plnkr.co/edit/2MUcs44WUo2G1cS2JvX4?p=preview

index.html

```
<!DOCTYPE html>
<!--suppress ALL -->
<html>
<head>
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.4/css/bootstrap.min.css">

  <script src="https://unpkg.com/core-js/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
  <script src="https://unpkg.com/reflect-metadata@0.1.8"></script>
  <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('script.ts').catch(function (err) {
      console.error(err);
    });
  </script>
</head>

<body class="container">
    <app>Loading...</app>
</body>
</html>
```

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {Component, NgModule, Input, Output, EventEmitter, ViewEncapsulation} from
'@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}
```

```
@Component({
  selector: 'joke-form',
  templateUrl: 'joke-form-component.html',
  styleUrls: [
    'joke-form-component.css'
  ],
  encapsulation: ViewEncapsulation.Emulated
  // encapsulation: ViewEncapsulation.Native
  // encapsulation: ViewEncapsulation.None

})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [hidden]="data.hide">{{data.punchline}}</p>
  <a (click)="data.toggle()"
     class="btn btn-warning">Tell Me
  </a>
</div>
`
})
class JokeComponent {
  @Input('joke') data: Joke;
}

@Component({
  selector: 'joke-list',
  template: `
<joke-form (jokeCreated)="addJoke($event)"></joke-form>
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
```

```
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }

  addJoke(joke) {
    this.jokes.unshift(joke);
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    JokeFormComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

joke-form-component.css

```
.card {
  background-color: gray;
}
```

joke-form-component.html

```
<div class="card card-block">
  <h4 class="card-title">Create Joke</h4>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the setup"
           #setup>
  </div>
  <div class="form-group">
    <input type="text"
           class="form-control"
           placeholder="Enter the punchline"
           #punchline>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="createJoke(setup.value, punchline.value)">Create
  </button>
</div>
```

Content Projection

Goals

Change our application so that a developer who is re-using our `JokeComponent` can also configure *how* a joke will be rendered on the screen.

Learning Objectives

- What is content projection and why might we want to use it.
- How to project content using the `ng-content` tag.
- How to project multiple pieces of content using css selectors.

Motivation

Lets say someone else wanted to use our `JokeComponent` but instead of displaying the punchline in a `<p>` tag they wanted to display it in a larger `<h1>` tag.

Our component right now doesn't let itself be *reused* like that.

However we can design our component with something called *content projection* to enable it to be customised by the component or developer who is using it.

Content projection

If we add the tag `<ng-content></ng-content>` anywhere in our template HTML for our component. The inner content of the tags that define our component are then *projected* into this space.

So if we changed the template for our `JokeComponent` to be something like:

```
<div class="card card-block">
  <h4 class="card-title">{{ data.setup }}</h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content></ng-content> ①
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
```

① We've replaced `{{data.punchline}}` with `<ng-content></ng-content>`

Then if we changed our `JokeListComponent` template from this:

```
<joke *ngFor="let j of jokes" [joke]="j"></joke>
```

To this:

```
<joke *ngFor="let j of jokes" [joke]="j">
  <h1>{{ j.punchline }}</h1> ①
</joke>
```

① In-between the opening and closing `joke` tags we've added some HTML to describe *how* we want the punchline to be presented on the screen, with a `<h1>` tag.

The `<h1>{{ j.punchline }}</h1>` defined in the parent `JokeListComponent`, replaces the `<ng-content></ng-content>` tag in the `JokeComponent`.

This is called *Content Projection* we *project* content from the parent Component to our Component.

If we create our components to support content projection then it enables the consumer of our component to configure *exactly* how they want the component to be rendered.

The downside of content projection is that the `JokeListComponent` doesn't have access to the properties or methods on the JokeComponent.

So the content we are projecting we can't bind to properties or methods of our `JokeComponent`, only the `JokeListComponent`.

Multi-content projection

What if we wanted to define *multiple* content areas, we've got a setup and a punchline lets make both of those content projectable.

Specifically want the setup line to always end with a `?` character.

We are using this example for demonstration purposes only. This problem could easily be solved in a number of other ways, all of them easier than using content projection.

We might change our `JokeListComponent` template to be something like:

```
<joke *ngFor="let j of jokes" [joke]="j">
  <span>{{ j.setup }} ?</span>
  <h1>{{ j.punchline }}</h1> ①
</joke>
```

But in our JokeComponent template we *can't* simply add two `<ng-content></ng-content>` tags, like so:

```
<div class="card card-block">
  <h4 class="card-title">
    <ng-content></ng-content>
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content></ng-content>
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
```

Angular doesn't know *which* content from the parent `JokeListComponent` to project into which `ng-content` tag in `JokeComponent`.

To solve this `ng-content` has another attribute called `select`.

If you pass to `select` a css matching selector, it will extract *only* the elements matching the selector from the passed in content to be projected.

Let's explain with an example:

```
<div class="card card-block">
  <h4 class="card-title">
    <ng-content select="span"></ng-content> ①
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content select="h1"></ng-content> ②
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
```

① `<ng-content select="span"></ng-content>` will match `<span>{{ j.setup }} ?</span>`.

② `<ng-content select="h1"></ng-content>` will match `<h1>{{ j.punchline }}</h1>`.

That however can be a bit tricky to manage, lets use some more meaningful rules matching perhaps by class name.

```
<div class="card card-block">
  <h4 class="card-title">
    <ng-content select=".setup"></ng-content> ①
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content select=".punchline"></ng-content> ②
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
```

To support this lets change our parent components content to identify the different elements by classnames, like so:

```
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }} ?</span>
  <h1 class="punchline">{{ j.punchline }}</h1> ①
</joke>
```

Summary

Sometimes the nature of a component means that the consumer would like to customise the view, the presentation of data, in a unique way for each use case.

Rather than trying to predict all the different configuration properties to support all the use cases we can instead use content projection. Giving the consumer the power to configure the presentation of the component as they want.

Listing

http://plnkr.co/edit/g5nuoD8m4e1tTp7gDW4N?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {Component, NgModule, Input, Output, EventEmitter, ViewEncapsulation} from
'@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
```

```
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}


@Component({
  selector: 'joke-form',
  templateUrl: 'joke-form-component.html',
  styleUrls: [
    'joke-form-component.css'
  ],
  encapsulation: ViewEncapsulation.Emulated
  // encapsulation: ViewEncapsulation.Native
  // encapsulation: ViewEncapsulation.None

})
class JokeFormComponent {
  @Output() jokeCreated = new EventEmitter<Joke>();

  createJoke(setup: string, punchline: string) {
    this.jokeCreated.emit(new Joke(setup, punchline));
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">
    <ng-content select=".setup"></ng-content>
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content select=".punchline"></ng-content>
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
`
})
class JokeComponent {
  @Input('joke') data: Joke;
}
```

```
@Component({
  selector: 'joke-list',
  template: `
<joke-form (jokeCreated)="addJoke($event)"></joke-form>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }

  addJoke(joke) {
    this.jokes.unshift(joke);
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    JokeFormComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Lifecycle Hooks

Learning Objectives

- Understand the different phases an Angular component goes through from being created to being destroyed.
- Know how to hook into those phases and run your own code.
- Know the order in which the different phases happen and what triggers each phase.

Phases

A component in Angular has a life-cycle, a number of different phases it goes through from birth to death.

We can hook into those different phases to get some pretty fine grained control of our application.

To do this we add some specific methods to our component class which get called during each of these life-cycle phases, we call those methods *hooks*.

The hooks are executed in this order:

These phases are broadly split up into phases that are linked to the component itself and phases that are linked to the *children* of that component.

Hooks for the component

constructor

This is invoked when Angular creates a component or directive by calling `new` on the class.

ngOnChanges

Invoked **every** time there is a change in one of th *input* properties of the component.

ngOnInit

Invoked when given component has been initialized.
This hook is only called **once** after the first `ngOnChanges`

ngDoCheck

Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.

`ngDoCheck` and `ngOnChanges` should not be implemented together on the same component.

We will cover this hook in more detail in the *Advanced Components* section at the end of this course.

ngOnDestroy

This method will be invoked just before Angular destroys the component.
Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

Hooks for the components children

These hooks are only called for components and not directives.

We will cover the difference between Components and Directives in the next section.

ngAfterContentInit

Invoked *after* Angular performs any content projection into the components view (see the previous lecture on *Content Projection* for more info).

ngAfterContentChecked

Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

ngAfterViewInit

Invoked when the component's view has been fully initialized.

ngAfterViewChecked

Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

We'll dig into the children hooks in more detail in the next lecture.

Adding hooks

In order to demonstrate how the hooks work we'll adjust the joke application we've been working with so far.

Firstly lets change the `JokeComponent` so it hooks into all the phases.

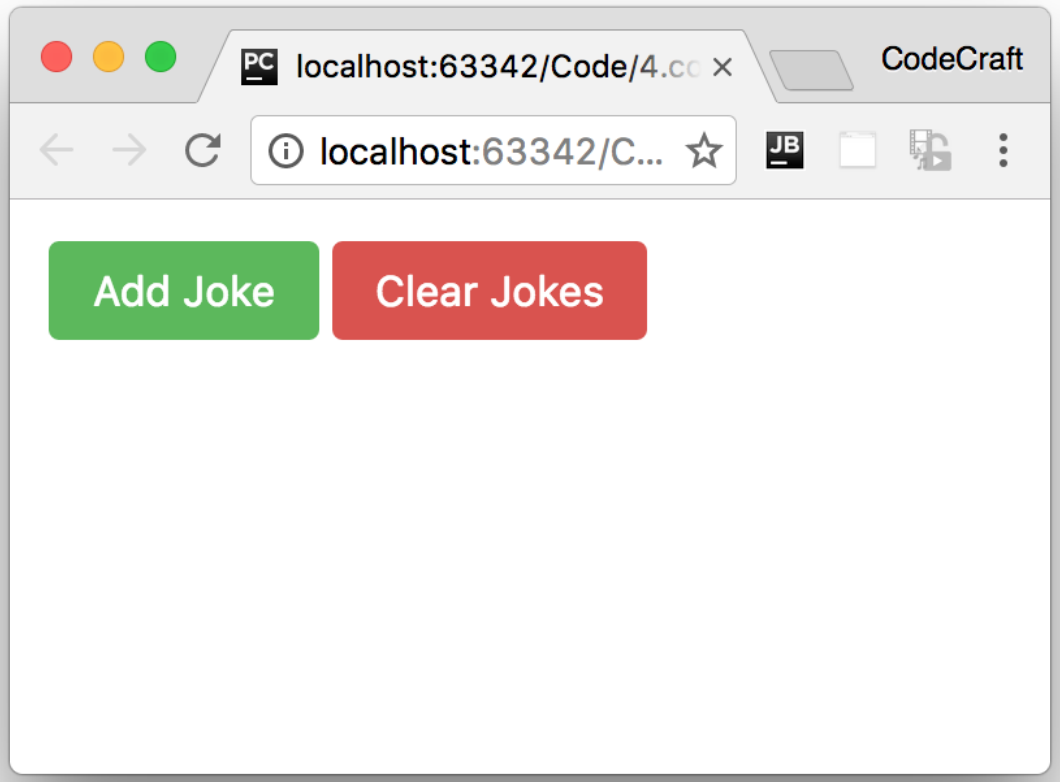
All we need to do is to add functions to the component class matching the hook names above, like so:

```
class JokeComponent {
  @Input('joke') data: Joke;

  constructor() {
    console.log(`new - data is ${this.data}`);
  }

  ngOnChanges() {
    console.log(`ngOnChanges - data is ${this.data}`);
  }

  ngOnInit() {
    console.log(`ngOnInit  - data is ${this.data}`);
  }

  ngDoCheck() {
    console.log("ngDoCheck")
  }

  ngAfterContentInit() {
    console.log("ngAfterContentInit");
  }

  ngAfterContentChecked() {
    console.log("ngAfterContentChecked");
  }

  ngAfterViewInit() {
    console.log("ngAfterViewInit");
  }

  ngAfterViewChecked() {
    console.log("ngAfterViewChecked");
  }

  ngOnDestroy() {
    console.log("ngOnDestroy");
  }
}
```

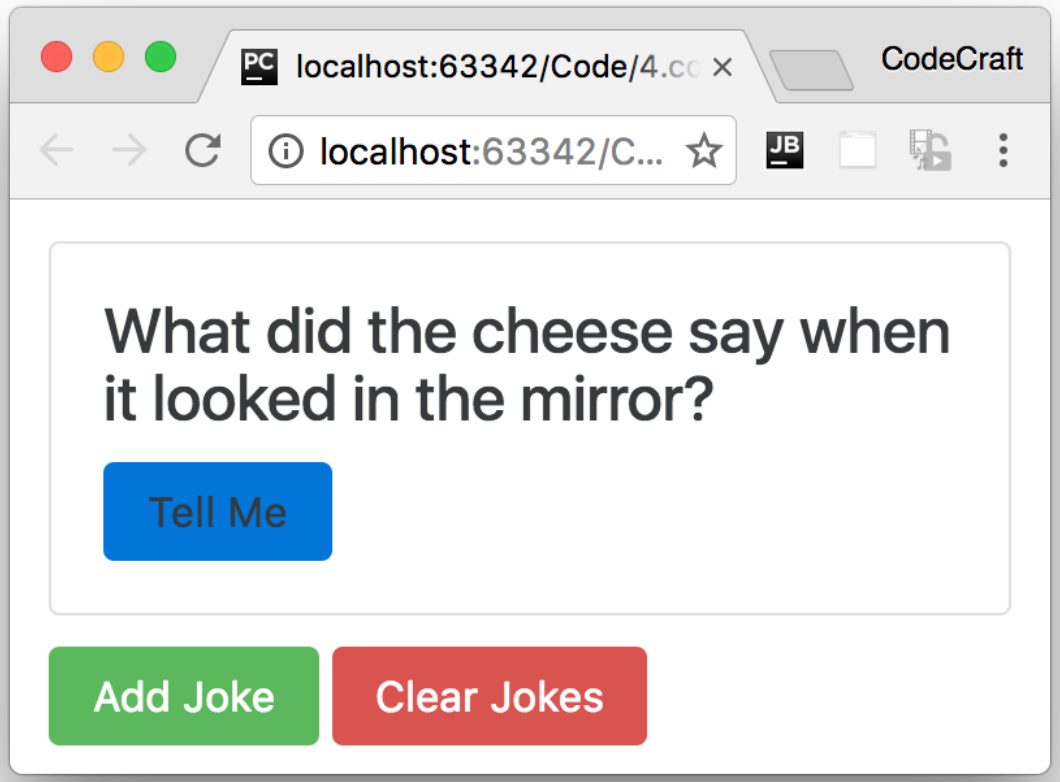
To easily trigger these hooks lets change the rest of the application. We remove the form and change the parent `JokeListComponent` so it has two buttons. One that adds a joke triggering Angular to create a new `JokeComponent` instance. Another button to clear the list of jokes triggering Angular to delete the `JokeComponents`.

```
@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }} ?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>

<button type="button"
        class="btn btn-success"
        (click)="addJoke()">Add Joke
</button>
<button type="button"
        class="btn btn-danger"
        (click)="deleteJoke()">Clear Jokes
</button>
`
})
class JokeListComponent {
  jokes: Joke[] = [];

  addJoke() {
    this.jokes.unshift(new Joke("What did the cheese say when it looked in the
mirror", "Hello-me (Halloumi)"));
  }

  deleteJoke() {
    this.jokes = []
  }
}
```

When we run this application we now see two buttons:



If we click "Add Joke" then a joke is added to the list and Angular creates an instance of the `JokeComponent` triggering the lifecycle hooks.



Looking at the console at this time we would see these logs:

```
new - data is undefined
ngOnChanges - data is [object Object]
ngOnInit  - data is [object Object]
ngDoCheck
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
```

For the first 3 hooks we are also printing out the value of the components `joke` input property.

We can see that in the `constructor` the input property is undefined.

However by the time the `ngOnChanges` hook is called we can see that the input property is now set to the joke.

The best place to initialise your components is in the `ngOnInit` lifecycle hook and not the constructor because only at this point have any input property bindings been processed.
The reason we use `ngOnInit` and not `ngOnChanges` to initialise a component is that `ngOnInit` is only called *once* whereas `ngOnChanges` is called for every change to the input properties.

When we press the *Clear Jokes* button, Angular deletes the `JokeComponent` and calls the `ngOnDestroy` hook which we can see in the logs like so:

```
ngOnDestroy
```

Detecting what has changed

We can actually tap into the exact changes to the input properties by examining the first param to the `ngOnChanges` function, which we typically call `changes`.

The type of `changes` is a map of the input property name to an instance of `SimpleChange`:

```
class SimpleChange {
  constructor(previousValue: any, currentValue: any)
  previousValue : any
  currentValue : any
  isFirstChange() : boolean
}
```

Using the above we can find out in our `ngOnChanges` function which input properties changed (if we have more than one) and also what the previous and current values are.

We change our `ngOnChanges` function to take the `changes` argument and loop through it to print out the `SimpleChange.currentValue` and `previousValue`.

```
ngOnChanges(changes: SimpleChanges) {
  console.log(`ngOnChanges - data is ${this.data}`);
  for (let key in changes) {
    console.log(`${key} changed.
Current: ${changes[key].currentValue}.
Previous: ${changes[key].previousValue}`);
  }
}
```

This prints out the below to the console:

```
ngOnChanges - data is [object Object]
data changed.
Current: [object Object].
Previous: CD_INIT_VALUE
```

The current value is the joke object that was bound to the `data` input property.

When no value has been set for an input property it gets defaulted to the string 'CD_INIT_VALUE' rather than `null` or `undefined`.

Interfaces

In the sample code so far we are just defining the hook functions directly on the class, but we can take advantage of a feature of TypeScript, interfaces, and be more explicit regarding our intentions.

Each of these lifecycle hooks has an associated typescript `interface` of the same name but without the `ng` prefix. So `ngOnChanges` has an interface called `OnChanges`.

Each interface defines just one hook, by making a class implement an interface we are saying we expect the class to have implemented that member function, if it doesn't then TypeScript should throw an error.

Adding the interfaces for our life-cycle hooks to our `JokeComponent` class would look something like so:

```
import {
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy
} from '@angular/core';


class JokeComponent implements
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy {
  ...
}
```

The browser based TypeScript compiler doesn't trigger a compilation error when we don't implement interface functions so we actually can't see the benefit of this in the browser, but doing this using the offline local compiler will throw an error.

Summary

Using life-cycle hooks we can fine tune the behaviour of our components during creation, update and destruction.

We use the `ngOnInit` hook most often, this is where we place any initialisation logic for our component. It's preferred over initialising via the constructor since in the constructor we don't yet have access to the input properties whereas by the time ngOnInit is called they have been bound to and are available to use.

`ngOnChanges` is the second most common hook, this is where we can find out details about which input properties have changed and how they have changed.

The third most common hook is `ngOnDestroy` which is where we place any cleanup logic for our component.

Listing

http://plnkr.co/edit/gIQCdWyb4sL4ZpDIyrFV?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
    Component,
    NgModule,
    Input,
    Output,
    EventEmitter,
    ViewEncapsulation,
    SimpleChanges,
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">
    <ng-content select=".setup"></ng-content>
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content select=".punchline"></ng-content>
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
```

```
  </a>
</div>
`
})
class JokeComponent implements OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy {
  @Input('joke') data: Joke;

  constructor() {
    console.log(`new - data is ${this.data}`);
  }

  ngOnChanges(changes: SimpleChanges) {
    console.log(`ngOnChanges - data is ${this.data}`);

    for (let key in changes) {
      console.log(`${key} changed.
Current: ${changes[key].currentValue}.
Previous: ${changes[key].previousValue}`);
    }
  }

  ngOnInit() {
    console.log(`ngOnInit  - data is ${this.data}`);
  }

  ngDoCheck() {
    console.log("ngDoCheck")
  }

  ngAfterContentInit() {
    console.log("ngAfterContentInit");
  }

  ngAfterContentChecked() {
    console.log("ngAfterContentChecked");
  }

  ngAfterViewInit() {
    console.log("ngAfterViewInit");
  }

  ngAfterViewChecked() {
    console.log("ngAfterViewChecked");
  }
```

```
  ngOnDestroy() {
    console.log("ngOnDestroy");
  }
}

@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>

<button type="button"
        class="btn btn-success"
        (click)="addJoke()">Add Joke
</button>
  <button type="button"
        class="btn btn-danger"
        (click)="deleteJoke()">Clear Jokes
</button>
`
})
class JokeListComponent {
  jokes: Joke[] = [];

  addJoke() {
    this.jokes.unshift(new Joke("What did the cheese say when it looked in the
mirror", "Hello-me (Halloumi)"));
  }

  deleteJoke() {
    this.jokes = []
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
`
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
```

```
    JokeComponent,
    JokeListComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

ViewChildren & ContentChildren

Learning Objectives

- Understand the difference between view children and content children of a component.
- Know how to get references to child components in host components.

Example application

The view children of a given component are the elements used *within* its template, its view.

We can get a reference to these view children in our component class by using the `@ViewChild` decorator.

We'll explain how all this works using the joke application we've been working with so far in this course.

We've changed the application so that the `JokeListComponent` shows two jokes in it's own view and one joke which is content projected from it's host `AppComponent`, like so:

JokeListComponent

```
@Component({
  selector: 'joke-list',
  template: `
<h4>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j"> ①
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>

<h4>Content Jokes</h4>
<ng-content></ng-content>  ②
`
})
class JokeListComponent  {
  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];
}
```

① The component renders jokes in it's *own* view.

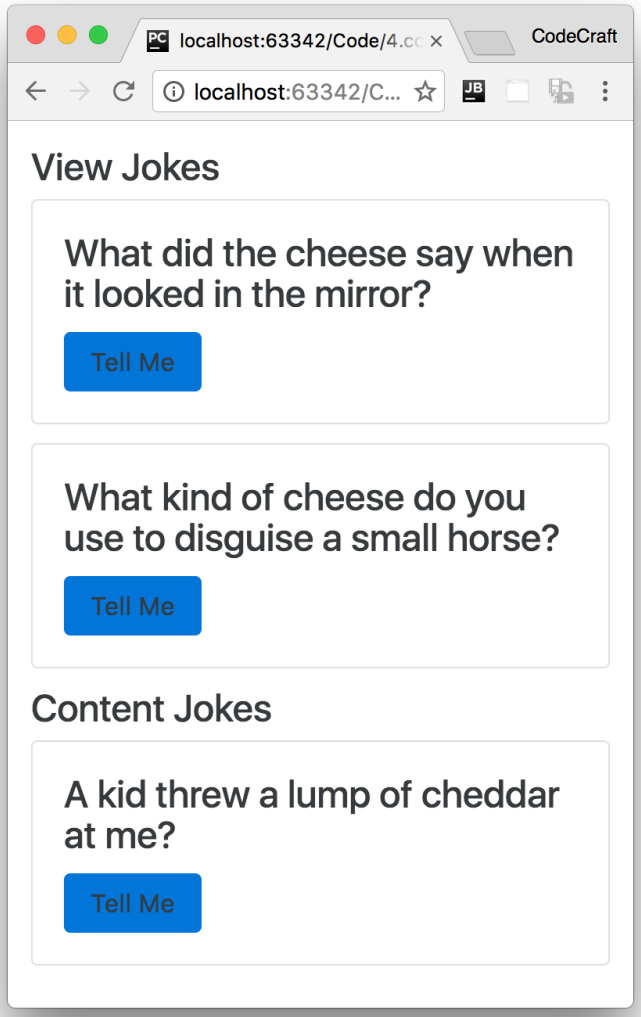
② It also projects some content from it's host component, in our example the other content is going to be a third joke.

AppComponent

```
@Component({
  selector: 'app',
  template: `
<joke-list>
  <joke [joke]="joke"> ①
    <span class="setup">{{ joke.setup }}?</span>
    <h1 class="punchline">{{ joke.punchline }}</h1>
  </joke>
</joke-list>
`
})
class AppComponent {
  joke: Joke = new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not
very mature'");
}
```

① Use content projection to inject a third joke into the `JokeListComponent`.

When we view this app now we see 3 jokes, two of which are from the JokeListComponent and the third is projected in from the AppComponent.



ViewChild

In our `JokeListComponent` lets add a reference to the child `JokeComponents` that exists in it's view.

We do this by using the `@ViewChild` decorator like so:

```
import { ViewChild } from '@angular/core';
.
.
.
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent {

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent; ①

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
  }
}
```

① We are storing a reference to the child `JokeComponent` in a property called `jokeViewChild`.

`jokeViewChild` isn't an instance of a `Joke` class, it is the actual instance of the child `JokeComponent` that exists inside *this* components view.

We create a new property called `jokeViewChild` and we pre-pend this with a decorator of `@ViewChild`. This decorator tells Angular *how* to find the child component that we want to bind to this property.

A `@ViewChild` decorator means, search inside this components template, it's view, for this child component.

The parameter we pass as the first argument to `@ViewChild` is the *type* of the component we want to search for, if it finds more than one it will just give us the first one it finds.

If we try to print out the reference in the constructor, like the code sample above, `undefined` will be printed out.

That's because by the time the constructor is called we haven't rendered the children yet. We

render in a *tree down approach* so when a parent component is getting constructed it means the children are not yet created.

We can however hook into the lifecycle of the component at the point the view children have been created and that's with the `ngAfterViewInit` hook.

To use this we need to make our component implement the interface `AfterViewInit`.

```
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);
  }
}
```

In the `ngAfterViewInit` function `jokeViewChild` has been initialised and we can see it logged in the console.

ViewChildren

The above isn't so useful in our case since we have *multiple* joke children components. We can solve that by using the alternative `@ViewChildren` decorator along side the `QueryList` generic type.

```
import { ViewChildren, QueryList } from '@angular/core';
.
.
.
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>; ①

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);
    let jokes: JokeComponent[] = this.jokeViewChildren.toArray(); ②
    console.log(jokes);
  }
}
```

① We use the `@ViewChildren` decorator which matches *all* `JokeComponent`s and stores them in a `QueryList` called jokeViewChildren.

② We can convert our `QueryList` of `JokeComponent`s into an array by calling `toArray()`

When we run the above application we see two `JokeComponents` printed to the console, like so:

```
Array[2]
> 0: JokeComponent
> 1: JokeComponent
```

The reason we see 2 jokes printed out and 3 is because only two of the jokes are *view children* the other joke is a *content child*. We cover content children in the end of ths lecture.

ViewChild referencing a template local variable

One practical application of `@ViewChild` is to get access to template local variables in our component class.

In the past we've said that template local variables are just that, *local* to the template.

But as the first param to the `@ViewChild` decorator we can also pass the name of a template local variable and have Angular store a reference to that variable on our component, like so:

```
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef; ①

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    //noinspection TypeScriptUnresolvedVariable
    this.headerEl.nativeElement.textContent = "Best Joke Machine"; ②
  }
}
```

① The type of our template variable is an `ElementRef`, which is a low level reference to any element in the DOM. We are requesting a reference to the header template variable which points to the first `<h4>` element in the template.

② Since `headerEl` is an `ElementRef` we can interact with the DOM directly and change the title of our header to *Best Joke Machine*.

It's not recommended to interact with the DOM directly with an `ElementRef` since that results in code that's not very portable.

ContentChild & ContentChildren

The concept of a *content child* is similar to that of a *view child* but the content children of the given component are the child elements that are *projected* into the component from the host component.

In our example application we are projecting one joke in from the host `AppComponent`.

To get a reference to that child we can use either the `@ContentChild` or the `@ContentChildren` decorators. They work in similar ways to the view child counterparts, `@ContentChild` returns one child and `@ContentChildren` returns a `QueryList`.

Lets use `@ContentChild` to get a reference to the third joke that is projected in, like so:

```
import { ContentChildren, ContentChild } from '@angular/core';
.
.
.
class JokeListComponent implements AfterContentInit, AfterViewInit { ①

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef;

  @ContentChild(JokeComponent) jokeContentChild: JokeComponent; ②

  ngAfterContentInit() { ③
    console.log(`ngAfterContentInit - jokeContentChild is ${this.jokeContentChild}`);
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    this.headerEl.nativeElement.textContent = "Best Joke Machine";
  }
}
```

① Just like before we need to tap into one of the component lifecycle hooks, this time it's `AfterContentInit`

② We create a `jokeContentChild` property and bind it to the content child by using the `@ContentChild`

decorator.

③ By the time the `ngAfterContentInit` hook is run the `jokeContentChild` property is set to the content child.

You can implement multiple interfaces just by separating them with a `,`.

If we logged `jokeContentChild` in our constructor it would again log undefined, since it's not actually initialised at that point.

Content children are only visible by the time the `AfterContentInit` lifecycle hook has run.

Summary

An Angular application is composed from a number of components nested together.

These components can nest in two ways, as view children, in the template for that component. Or they can nest as content children, via content projection from a host component.

As developers of our components we can get access to these child components via the `@ViewChild` and `@ContentChild` (and `@ViewChildren` and `@ContentChildren`) decorators.

View children of a component are the components and elements in *this* components view.

Content children of a component are the components and elements that are *projected* into *this* components view by a host component.

View children are only initialised by the time the `AfterViewInit` lifecycle phase has been run.

Content children are only initialised by the time the `AfterContentInit` lifecycle phase has been run.

Listing

http://plnkr.co/edit/V6tqGnvNiiOOMMLmajKP?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
    Component,
    NgModule,
    Input,
    Output,
    EventEmitter,
    ViewEncapsulation,
    SimpleChanges,
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
```

```
    AfterViewInit,
    AfterViewChecked,
    OnDestroy,
    ViewChild,
    ViewChildren,
    ContentChild,
    ContentChildren,
    ElementRef,
    QueryList
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">
    <ng-content select=".setup"></ng-content>
  </h4>
  <p class="card-text"
     [hidden]="data.hide">
    <ng-content select=".punchline"></ng-content>
  </p>
  <a class="btn btn-primary"
     (click)="data.toggle()">Tell Me
  </a>
</div>
`
})
class JokeComponent {

  @Input('joke') data: Joke;
}

@Component({
```

```
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements OnInit,
    AfterContentInit,
    AfterViewInit {

  jokes: Joke[] = [
    new Joke("What did the cheese say when it looked in the mirror", "Hello-me
(Halloumi)"),
    new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-pony
(Mascarpone)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef;
  @ContentChild(JokeComponent) jokeContentChild: JokeComponent;

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
    console.log(`new - jokeContentChild is ${this.jokeContentChild}`);
  }

  ngAfterContentInit() {
    console.log(`ngAfterContentInit - jokeContentChild is ${this.jokeContentChild}`);
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    this.headerEl.nativeElement.textContent = "Best Joke Machine";
  }
}


@Component({
  selector: 'app',
  template: `
```

```
<joke-list>
  <joke [joke]="joke">
    <span class="setup">{{ joke.setup }}?</span>
    <h1 class="punchline">{{ joke.punchline }}</h1>
  </joke>
</joke-list>
`
})
class AppComponent {
  joke: Joke = new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not
very mature’");
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

In this section we covered Components in much more detail than the quickstart.

We discussed how to architect your application using components in Angular. We explained one method of breaking down your applications design into components with clear responsibilities, inputs and outputs.

We covered some of of the ways we can configure a component via the `@Component` decorator, specifically the `templateUrl`, `styles`, `styleUrls` and `encapsulation` configuration properties.

We covered the mechanism of content projection in Angular with the `ng-content` tag.

We explained the various component lifecycle phases, when they are run and how to hook into them with functions on the component.

Finally we covered the idea of view and content children and how to get references to each in a component.

Activity

Create a set of components with work together to create a *Carousel* which we can use to display different images in rotation.

We also want the carousel delay between rotations to be configurable.

The markup for our carousel will look like so:

```
<carousel [delay]="2000">
  <carousel-item>
    <img src="https://unsplash.it/200?image=0" alt="">
  </carousel-item>
  <carousel-item>
    <img src="https://unsplash.it/200?image=100" alt="">
  </carousel-item>
  <carousel-item>
    <img src="https://unsplash.it/200?image=200" alt="">
  </carousel-item>
</carousel>
```

Steps

Fork this plunker:

http://plnkr.co/edit/oajylErBv8adMGXywNTC?p=preview

Finish off the components to implement the carousel.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

Built-in Directives

Overview

We've covered Components in the last section, in this section and the next we'll cover the concept of *Directives*.

We've touched on a few directives already, such as `NgFor`.

Directives are components *without* a view. They are components without a template. Or to put it another way, components are directives *with* a view.

Everything you can do with a directive you can also do with a component. But not everything you can do with a component you can do with a directive.

We typically *associate* directives to existing elements by use *attribute* selectors, like so:

```
<elemenent aDirective></element>
```

We capitalise the name of directives when we are talking about the directive *class*. For example when we say `NgFor` we mean the class which defines the `NgFor` directive.

When we are talking about either an *instance* of a directive or the *attribute* we use to associate a directive to an element we lowercase the first letter. So `ngFor` refers to both the *instance* of a directive and the _attribut_e name used to associate a directive with an element.

In this section we are going to cover the built in directives that come bundled with Angular.

NgFor

Learning Objectives

- Know how to use the `NgFor` directive in your application.
- Know how to get the index in the array of the item you are looping over.
- Know how to nest multiple `NgFor` directives together.

Basics

We've covered this directive before in the quickstart.

`NgFor` is a structural directive, meaning that it changes the structure of the DOM.

It's point is to repeat a given HTML template once for each value in an array, each time passing it the array value as context for string interpolation or binding.

This directive is the successor of Angular 1s `ng-repeat` directive.

Let's take a look at an example:

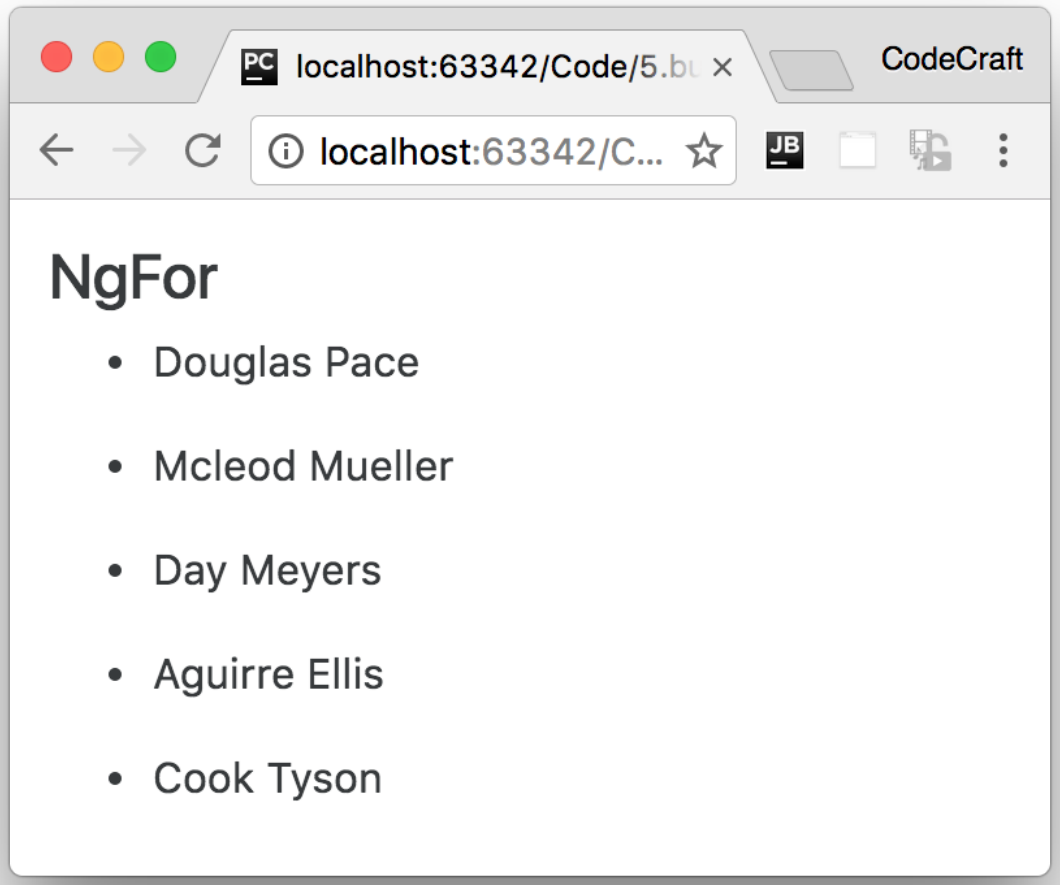
```
@Component({
  selector: 'ngfor-example',
  template: `
 <ul>
  <li *ngFor="let person of people"> ①
    {{ person.name }}
  </li>
 </ul>
 `
})
class NgForExampleComponent {
  people: any[] = [
    {
      "name": "Douglas  Pace"
    },
    {
      "name": "Mcleod  Mueller"
    },
    {
      "name": "Day  Meyers"
    },
    {
      "name": "Aguirre  Ellis"
    },
    {
      "name": "Cook  Tyson"
    }
  ];
}
```

① We loop over each `person` in the `people` array and print out the persons name.

The syntax is `*ngFor="let <value> of <collection>"`.

<*value*> is a variable name of your choosing, <*collection*> is a property on your component which holds a collection, usually an array but anything that can be iterated over in a `for-of` loop.

If we ran the above we would see this:



Index

Sometimes we also want to get the *index* of the item in the array we are iterating over.

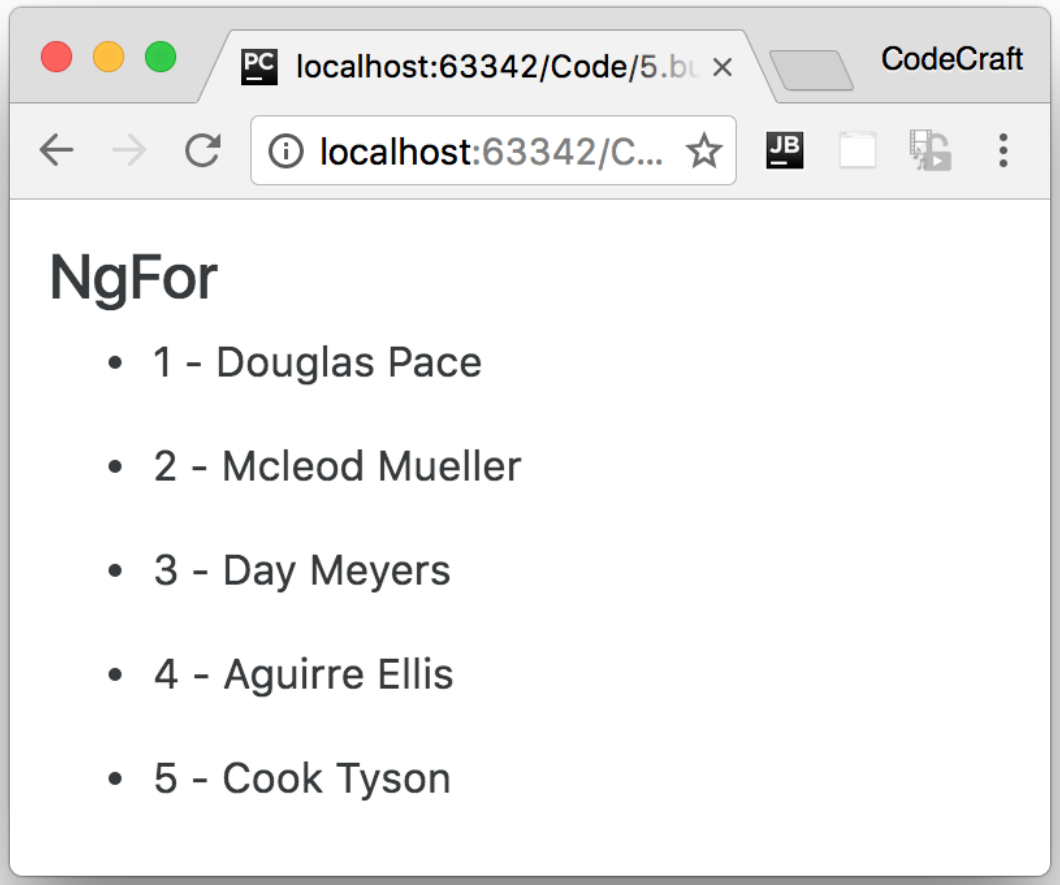
We can do this by adding another variable to our `ngFor` expression and making it equal to `index`, like so:

```
<ul> ①
  <li *ngFor="let person of people; let i = index"> ①
    {{ i + 1 }} - {{ person.name }} ②
  </li>
</ul>
```

① We create another variable called `i` and make it equal to the special keyword `index`.

② We can use the variable `i` just like we can use the variable `person` in our template.

If we ran the above we would now see this:



The index is always zero based, so starts at 0 then 1,2,3,4 etc..

In Angular 1 the variable `$index` would automatically be available for us to use in an `ng-repeat` directive. In Angular we now need to provide this variable explicitly.

Grouping

If our data structure was in fact grouped by country we can use two `ngFor` directives, like so:

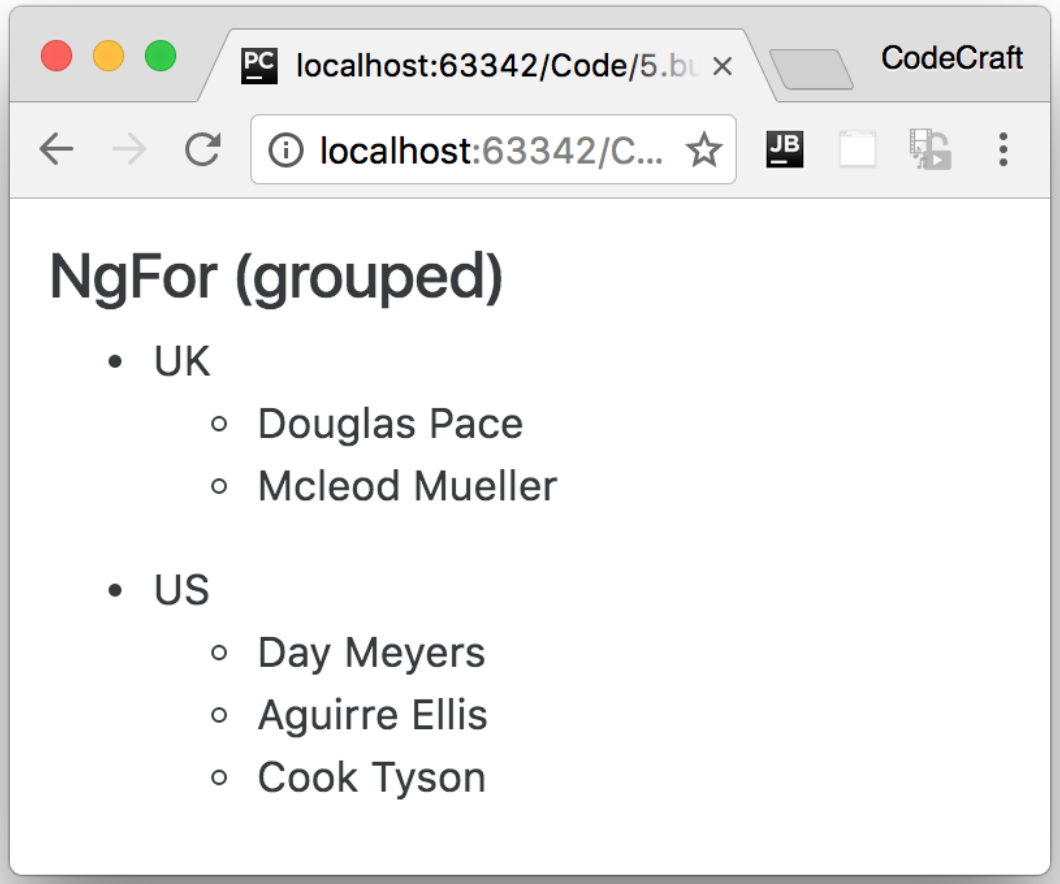
script.ts

```
@Component({
  selector: 'ngfor-grouped-example',
  template: `
 <h4>NgFor (grouped)</h4>
 <ul *ngFor="let group of peopleByCountry"> ①
   <li>{{ group.country }}</li>
   <ul>
    <li *ngFor="let person of group.people"> ②
      {{ person.name }}
    </li>
   </ul>
 </ul>
 `
})
class NgForGroupedExampleComponent {

  peopleByCountry: any[] = [
    {
      'country': 'UK',
      'people': [
        {
          "name": "Douglas  Pace"
        },
        {
          "name": "Mcleod  Mueller"
        },
      ]
    },
    {
      'country': 'US',
      'people': [
        {
          "name": "Day  Meyers"
        },
        {
          "name": "Aguirre  Ellis"
        },
        {
          "name": "Cook  Tyson"
        }
      ]
    }
  ];
}
```

① The first `ngFor` loops over the groups, each grop contains a `country` property which we render out on the next line and a `people` array property.

② To loop over the `people` array we create a second *nested* `ngFor` directive.

If we ran the above we would see:



Summary

We use the `NgFor` directive to loop over an array of items and create multiple elements dynamically from a template element.

The *template* element is the element the directive is attached to.

We can nest muliple `NgFor` directives together.

We can get the index of the item we are looping over by assigning `index` to a variable in the `NgFor` expression.

Listing

http://plnkr.co/edit/n76eLP4d1SubfA2u9FeD?p=preview

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';


@Component({
  selector: 'ngfor-example',
  template: `<h4>NgFor</h4>
<ul>
  <li *ngFor="let person of people; let i = index">
    {{ i + 1 }} - {{ person.name }}
  </li>
</ul>
 `
})
class NgForExampleComponent {
  people: any[] = [
    {
      "name": "Douglas  Pace"
    },
    {
      "name": "Mcleod  Mueller"
    },
    {
      "name": "Day  Meyers"
    },
    {
      "name": "Aguirre  Ellis"
    },
    {
      "name": "Cook  Tyson"
    }
  ];
}


@Component({
  selector: 'ngfor-grouped-example',
  template: `<h4>NgFor (grouped)</h4>
<ul *ngFor="let group of peopleByCountry">
  <li>{{ group.country }}</li>
  <ul>
    <li *ngFor="let person of group.people">
      {{ person.name }}
    </li>
  </ul>
</ul>
 `
})
class NgForGroupedExampleComponent {
```

```
  peopleByCountry: any[] = [
    {
      'country': 'UK',
      'people': [
        {
          "name": "Douglas  Pace"
        },
        {
          "name": "Mcleod  Mueller"
        },
      ]
    },
    {
      'country': 'US',
      'people': [
        {
          "name": "Day  Meyers"
        },
        {
          "name": "Aguirre  Ellis"
        },
        {
          "name": "Cook  Tyson"
        }
      ]
    }
  ];
}

@Component({
  selector: 'directives-app',
  template: `
  <ngfor-grouped-example></ngfor-grouped-example>
  <ngfor-example></ngfor-example>
 `
})
class DirectivesAppComponent {
}


@NgModule({
  imports: [BrowserModule],
  declarations: [
    NgForExampleComponent,
    NgForGroupedExampleComponent,
    DirectivesAppComponent],
  bootstrap: [DirectivesAppComponent],
})
class AppModule {

}
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

NgIf & NgSwitch

Learning Objectives

- Know how to conditionally add or remove an element from the DOM using the `NgIf` directive.
- Know how to conditionally add or remove elements from the DOM using the `NgSwitch` directive.

NgIf

The `NgIf` directive is used when you want to display or remove an element based on a condition.

If the condition is `false` the element the directive is *attached to* will be *removed* from the DOM.

The difference between `[hidden]='false'` and `*ngIf='false'` is that the first method simply *hides* the element. The second method with `ngIf` *removes* the element completely from the DOM.

We define the condition by passing an expression to the directive which is evaluated in the context of it's host component.

The syntax is: `*ngIf="<condition>"`

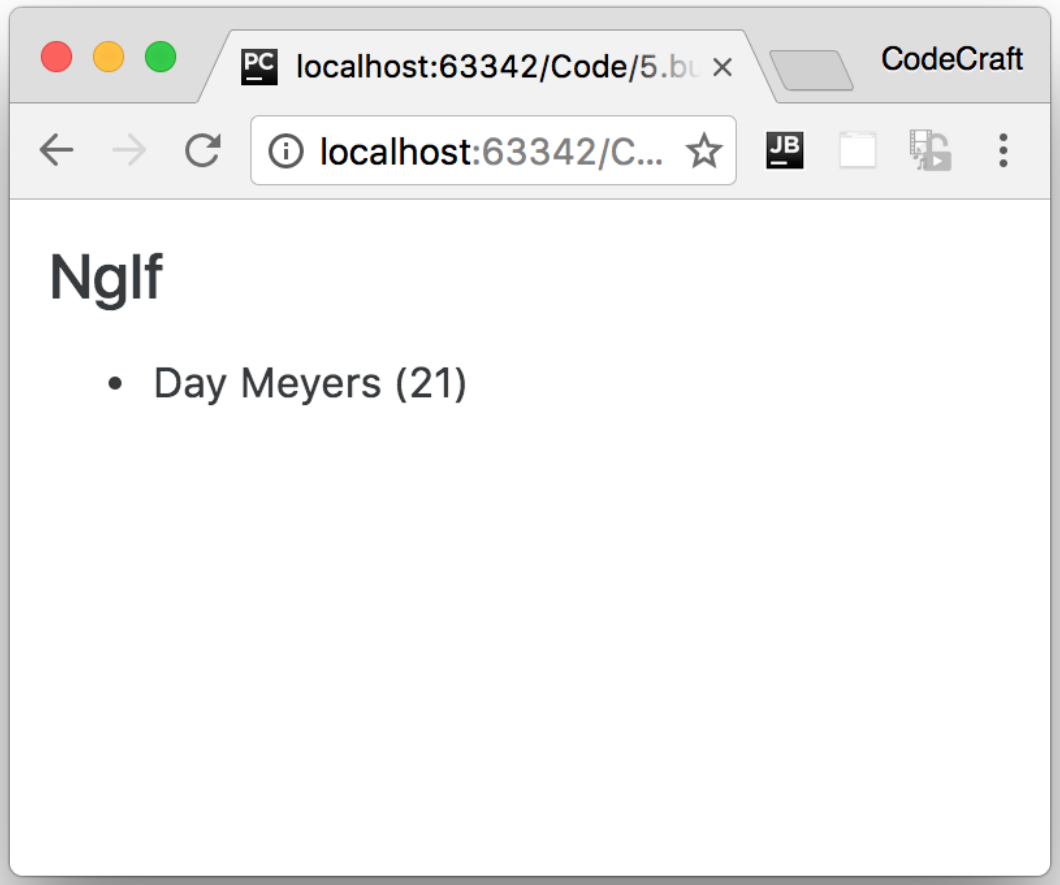
Lets use this in an example, we've taken the same code sample as we used for the `NgFor` lecture but changed it slightly. Each person now has an age as well as a name.

Lets add an `NgIf` directive to the template so we only show the element if the age is less than 30, like so:

```
@Component({
  selector: 'ngif-example',
  template: `
<h4>NgIf</h4>
<ul *ngFor="let person of people">
  <li *ngIf="person.age < 30"> ①
  {{ person.name }} ({{ person.age }})
  </li>
</ul>
`
})
class NgIfExampleComponent {

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32
    },
    {
      "name": "Day  Meyers",
      "age": 21
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34
    },
    {
      "name": "Cook  Tyson",
      "age": 32
    }
  ];
}
```

① The `NgIf` directive *removes* the `li` element from the DOM if `person.age` is less than 30.

If we ran the above we would see:



We *can't* have two structural directives, directives starting with a *, attached to the *same* element.

The below code would **not** work:

```
<ul *ngFor="let person of people" *ngIf="person.age < 30">
  <li>{{ person.name }}</li>
</ul>
```

This is exactly the same as Angular 1's `ng-if` directive however Angular doesn't have a built-in alternative for `ng-show`. To achieve something similar you can bind to the `[hidden]` property or use the `NgStyle` or `NgClass` directives we cover later in this section.

NgSwitch

Lets imagine we wanted to print peoples names in different colours depending on *where* they are

from. Green for UK, Blue for USA, Red for HK.

With bootstrap we can change the text color by using the `text-danger`, `text-success`, `text-warning` and `text-primary` classes.

We *could* solve this by having a series of *ngIf statements, like so:

```
<ul *ngFor="let person of people">
  <li *ngIf="person.country ==='UK'"
      class="text-success">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country === 'USA'"
      class="text-primary">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country === 'HK'"
      class="text-danger">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngIf="person.country !== 'HK' && person.country !== 'UK' && person.country !==
'USA'"
      class="text-warning">{{ person.name }} ({{ person.country }})
  </li>
</ul>
```

This initially seems to make sense until we try to create our *else* style element. We have to check to see if the person is not from any of the countries we have specified before. Resulting in a pretty long `ngIf` expression and it will only get worse the more countries we add.

Most languages, including javascript, have a language construct called a `switch` statement to solve this kind of problem. Angular also provides us with similar functionality via something called the `NgSwitch` directive.

This directive allows us to render different elements depending on a given condition, in fact the `NgSwitch` directive is actually a number of directives working in conjunction, like so:

script.ts

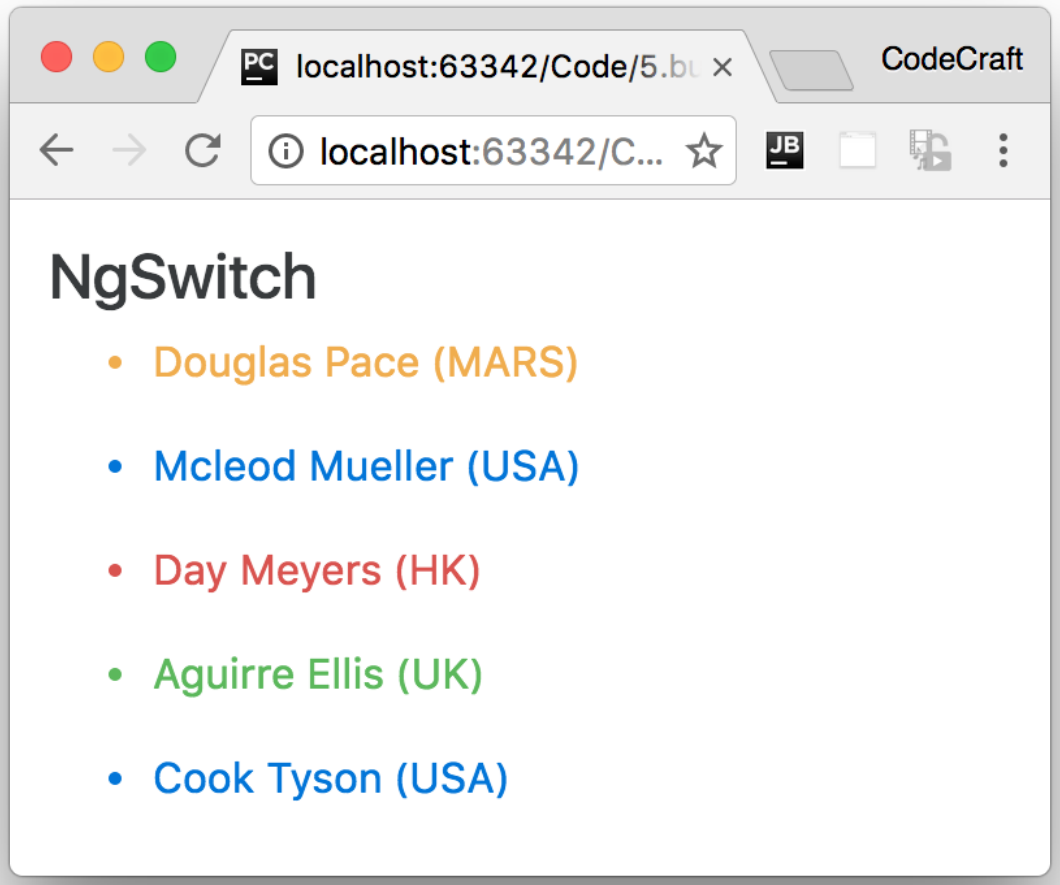
```
@Component({
  selector: 'ngswitch-example',
  template: `<h4>NgSwitch</h4>
<ul *ngFor="let person of people"
    [ngSwitch]="person.country"> ①

  <li *ngSwitchCase="'UK'" ②
      class="text-success">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
      class="text-primary">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
      class="text-danger">{{ person.name }} ({{ person.country }})
```

```
  </li>
  <li *ngSwitchDefault ③
      class="text-warning">{{ person.name }} ({{ person.country }})
  </li>
</ul>`
})
class NgSwitchExampleComponent {

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35,
      "country": 'MARS'
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32,
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "age": 21,
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34,
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "age": 32,
      "country": 'USA'
    }
  ];
}
```

1. We bind an expression to the `ngSwitch` directive.
2. The `ngSwitchCase` directive lets us define a condition which if it matches the expression in (1) will render the element it's attached to.
3. If no conditions are met in the switch statement it will check to see if there is an ngSwitchDefault directive, if there is it will render the element thats attached to, however it is optional - if it's not present it simply won't display anything if no matching ngSwitchCase directive is found.

The key difference between the `ngIf` solution is that by using `NgSwitch` we evaluate the expression only once and then choose the element to display based on the result.

If we ran the above we would see:



The use of `NgSwitch` here is just for example and isn't an efficient way of solving this problem. We would use either the `NgStyle` or `NgClass` directives which we'll cover in the next lecture.

Summary

With `NgIf` we can conditionally add or remove an element from the DOM.

If we are faced with multiple conditions a cleaner alternative to multiple nested `NgIf` statements is the `NgSwitch` series of directives.

Listing

http://plnkr.co/edit/1mDlqE56ZxdiZbusUrva?p=preview

script.ts

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

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

@Component({
  selector: 'ngif-example',
  template: `
<h4>NgIf</h4>
<ul *ngFor="let person of people">
  <li *ngIf="person.age < 30">
    {{ person.name }} ({{ person.age }})
  </li>
</ul>
`
})
class NgIfExampleComponent {

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32
    },
    {
      "name": "Day  Meyers",
      "age": 21
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34
    },
    {
      "name": "Cook  Tyson",
      "age": 32
    }
  ];
}


@Component({
  selector: 'ngswitch-example',
  template: `<h4>NgSwitch</h4>
<ul *ngFor="let person of people"
    [ngSwitch]="person.country">

  <li *ngSwitchCase="'UK'"
      class="text-success">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
```

```
      class="text-primary">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
      class="text-danger">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchDefault
      class="text-warning">
    {{ person.name }} ({{ person.country }})
  </li>
</ul>`
})
class NgSwitchExampleComponent {

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35,
      "country": 'MARS'
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32,
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "age": 21,
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34,
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "age": 32,
      "country": 'USA'
    }
  ];
}


@Component({
  selector: 'directives-app',
  template: `
  <ngswitch-example></ngswitch-example>
  <ngif-example></ngif-example>
 `
```

```
})
class DirectivesAppComponent {
}


@NgModule({
  imports: [BrowserModule],
  declarations: [
    NgIfExampleComponent,
    NgSwitchExampleComponent,
    DirectivesAppComponent],
  bootstrap: [DirectivesAppComponent]
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

NgStyle & NgClass

Learning Objectives

- Understand when and how to use the `NgStyle` directive to set an elements style.
- Understand when and how to use the `NgClass` directive to set an elements classes.

NgStyle

The `NgStyle` directive lets you set a given DOM elements style properties.

One way to set styles is by using the `NgStyle` directive and assigning it an *object literal*, like so:

```
<div [ngStyle]="{'background-color':'green'}"></<div>
```

This sets the background color of the `div` to green.

`ngStyle` becomes much more useful when the value is *dynamic*. The *values* in the object literal that we assign to `ngStyle` can be javascript expressions which are evaluated and the result of that expression is used as the value of the css property, like this:

```
<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green' : 'red'
}"></<div>
```

The above code uses the ternary operator to set the background color to green if the persons country is the UK else red.

But the expression doesn't have to be *inline*, we can call a function on the component instead.

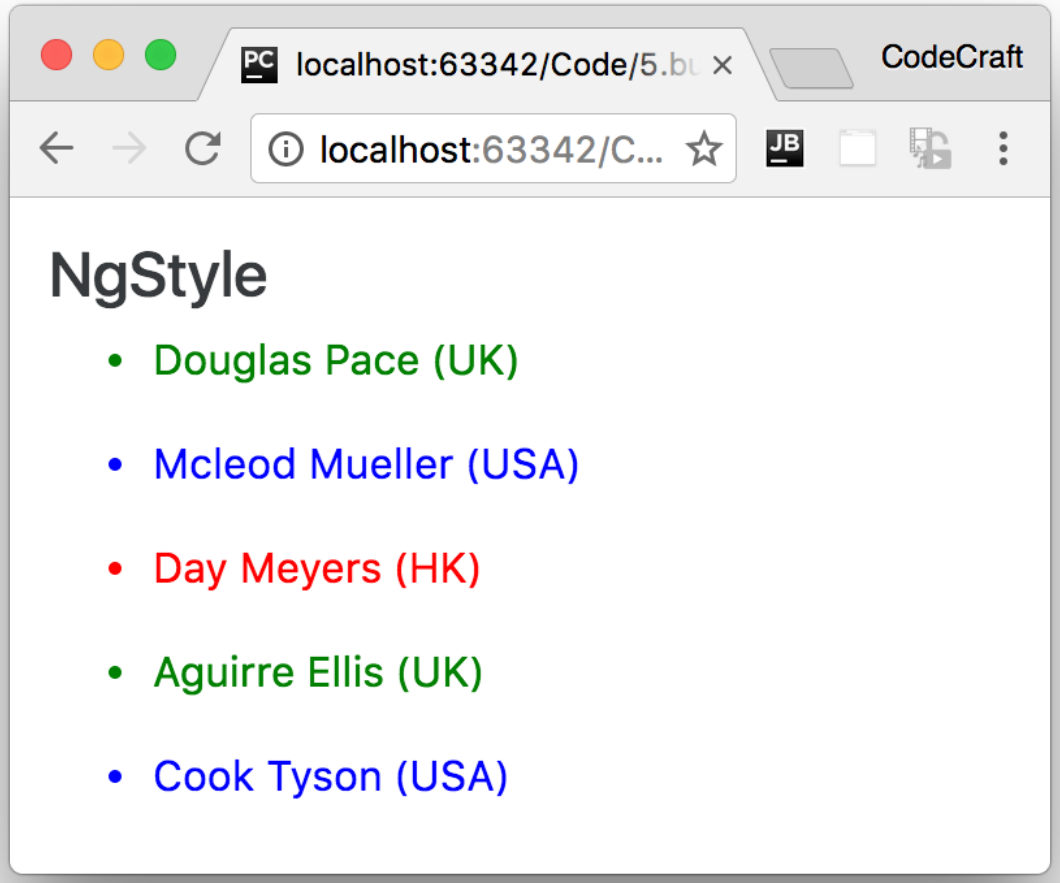
To demonstrate this lets flesh out a full example. Similar to the ones we've created before lets loop through an array of people and print out there names in different colors depending on the country they are from.

```
@Component({
  selector: 'ngstyle-example',
  template: `<h4>NgStyle</h4>
<ul *ngFor="let person of people">
  <li [ngStyle]="{'color':getColor(person.country)}"> {{ person.name }} ({{
person.country }}) ①
  </li>
</ul>
 `
})
class NgStyleExampleComponent {

  getColor(country) { ②
    switch (country) {
      case 'UK':
        return 'green';
      case 'USA':
        return 'blue';
      case 'HK':
        return 'red';
    }
  }

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "country": 'UK'
    },
    {
      "name": "Mcleod  Mueller",
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "country": 'USA'
    }
  ];
}
```

① We set the color of the text according to the value that's returned from the `getColor` function.

② Our `getColor` function returns different colors depending on the country passed in.

If we ran the above code we would see:



Alternative syntax

As well as using the `nyStyle` directive we can also set individual style properties using the `[style.<property>]` syntax, for example `[style.color]="getColor(person.country)"`

```
<ul *ngFor="let person of people">
  <li [style.color]="getColor(person.country)">{{ person.name }} ({{ person.country
}})
  </li>
</ul>
```

Points and pixels

Lets imagine we wanted to set the font size to 24, we could use:

```
[ngStyle]="{'font-size':24}"
```

But this wouldn't work, it isn't valid CSS to just set the font size to 24. We also have to specify a *unit* such as *px* or *em*.

Angular comes to the rescue with a special syntax, `<property>.<unit>`. So for the above if we wanted the size to be 24 pixels we would write `[ngStyle]="{'font-size.px':24}"`

The property name is `font-size.px` and not just `font-size`

The `.px` suffix says that we are setting the font-size in pixels. You could `.em` to express the font size in ems or even in percentage using `.%`

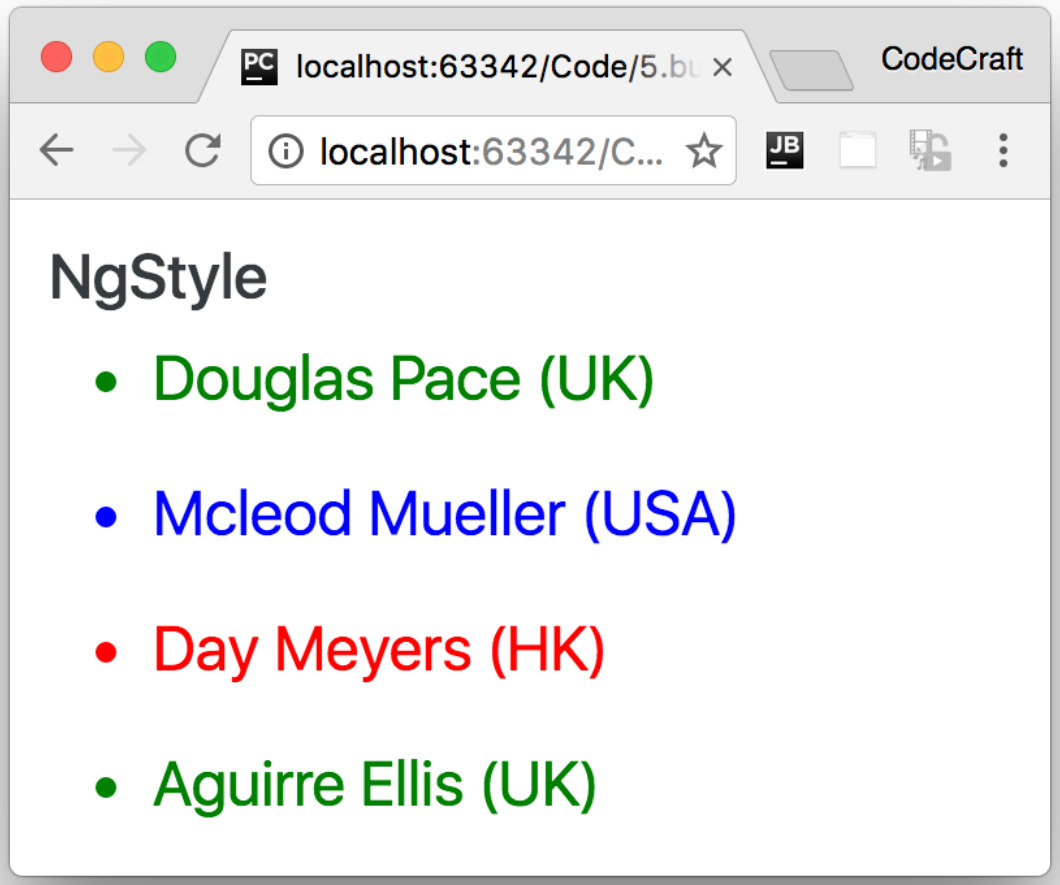
This is also applicable with the alternative syntax, e.g:-

`[style.font-size.px]="24"`

Let's change our demo application to display the names in a font size of 24 pixels, like so:

```
<ul *ngFor="let person of people">
  <li [ngStyle]="{'font-size.px':24}"
      [style.color]="getColor(person.country)">{{ person.name }} ({{ person.country
}})
  </li>
</ul>
```

Running the above would display:



NgClass

The `NgClass` directive allows you to set the CSS class dynamically for a DOM element.

The `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

There are two ways to use this directive, the first is by passing an object literal to the directive, like so:

```
[ngClass]="{'text-success':true}"
```

When using an object literal, the keys are the classes which are added to the element if the value of the key evaluates to true.

So in the above example, since the value is `true` this will set the class `text-success` onto the element the directive is attached to.

The value can also be an *expression*, so we can re-write the above to be.

```
[ngClass]="{'text-success':person.country === 'UK'}"
```

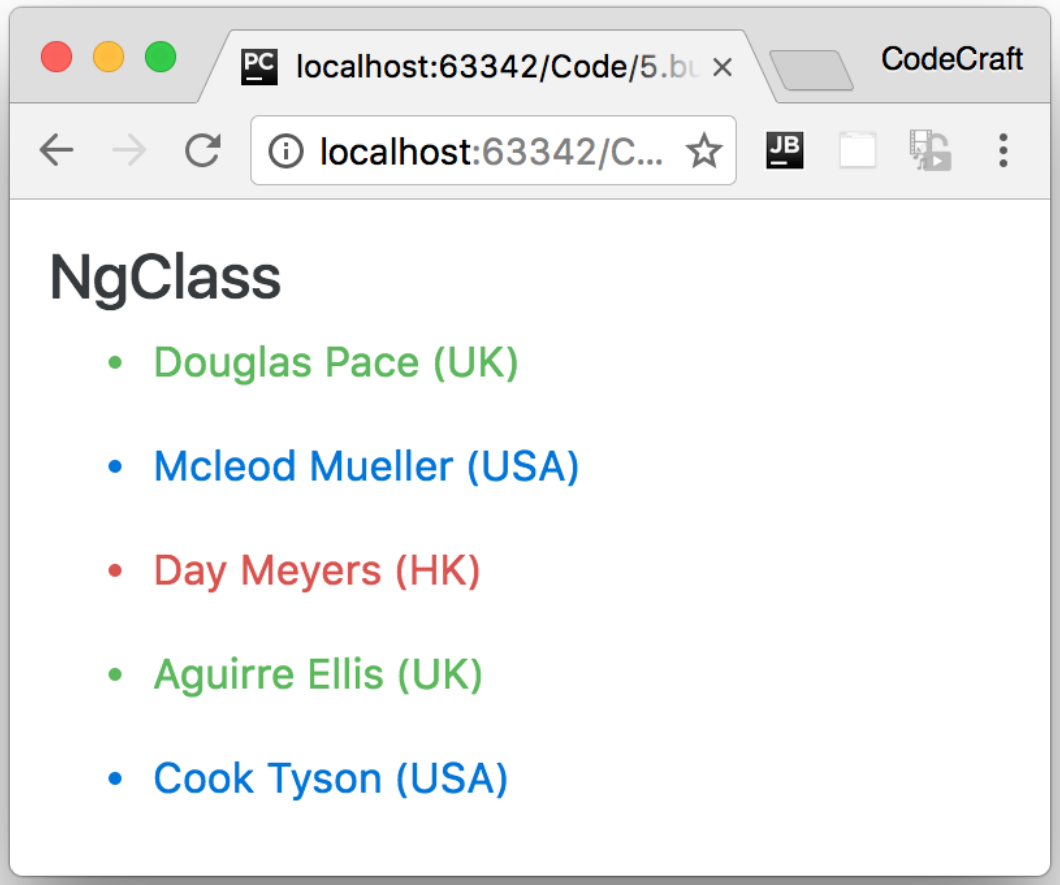
Lets implement the colored names demo app using `ngClass` instead of `ngStyle`.

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]="{
    'text-success':person.country === 'UK',
    'text-primary':person.country === 'USA',
    'text-danger':person.country === 'HK'
  }">{{ person.name }} ({{ person.country }})
  </li>
</ul>
```

Since the object literal can contain many keys we can also set many class names.

We can now color our text with different colors for each country with one statement.

If we ran the above code we would see:



Alternative syntax

We can also set a class on an element by binding to the input property binding called `class`, like so `[class]="'text-success'"`

The `'text-success'` is wrapped with single quotes so when it's evaluated as javascript it doesn't try to treat `text-success` as a variable.

The above syntax removes **all** the existing classes for that element and replaces them with just 'text-success'.

If we want to just add `text-success` to the list of classes already set on the element we can use the extended syntax `[class.<class-name>]='truthy expression'`

So for instanace to add text-success to the list of classes for an element we can use:

```
[class.text-success]="true"
```

Or just like before we can use an expression as the value, like so:

```
[class.card-success]="person.country === 'UK'"
```

And we can actually specify multiple values at one time, like so:

```
<ul *ngFor="let person of people">
  <li [class.text-success]="person.country === 'UK'"
      [class.text-primary]="person.country === 'USA'"
      [class.text-danger]="person.country === 'HK'">{{ person.name }} ({{
person.country }})
  </li>
</ul>
```

Summary

Both the `NgStyle` and `NgClass` directives can be used to conditionally set the look and feel of your application.

`NgStyle` gives you fine grained control on individual properties. But if you want to make changes to multiple properties at once, creating a class which bundles those properties and adding the class with `NgClass` makes more sense.

The object literal syntax is better for when you want to set multiple classes or styles in one statement. The shortcut syntax, `[style|class.<property>]`, is better for setting a single class or style.

Listing

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

@Component({
  selector: 'ngstyle-example',
  template: `<h4>NgStyle</h4>
<ul *ngFor="let person of people">
  <li [ngStyle]="{'font-size.px':24}"
      [style.color]="getColor(person.country)">
    {{ person.name }} ({{ person.country }})
  </li>
</ul>
 `
})
class NgStyleExampleComponent {
```

```
  getColor(country) {
    switch (country) {
      case 'UK':
        return 'green';
      case 'USA':
        return 'blue';
      case 'HK':
        return 'red';
    }
  }

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "country": 'UK'
    },
    {
      "name": "Mcleod  Mueller",
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "country": 'USA'
    }
  ];
}


@Component({
  selector: 'ngclass-example',
  template: `<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]="{
  'text-success':person.country === 'UK',
  'text-primary':person.country === 'USA',
  'text-danger':person.country === 'HK'
  }">
  {{ person.name }} ({{ person.country }})
  </li>
</ul>

<!--
<ul *ngFor="let person of people">
```

```
  <li [class.text-success]="person.country === 'UK'"
      [class.text-primary]="person.country === 'USA'"
      [class.text-danger]="person.country === 'HK'">
    {{ person.name }} ({{ person.country }})
  </li>
</ul>
-->
 `
})
class NgClassExampleComponent {

  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35,
      "country": 'UK'
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32,
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "age": 21,
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34,
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "age": 32,
      "country": 'USA'
    }
  ];
}


@Component({
  selector: 'directives-app',
  template: `
<ngclass-example></ngclass-example>
<ngstyle-example></ngstyle-example>`
})
class DirectivesAppComponent {
}
```

```
@NgModule({
  imports: [BrowserModule],
  declarations: [
    NgClassExampleComponent,
    NgStyleExampleComponent,
    DirectivesAppComponent],
  bootstrap: [DirectivesAppComponent]
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

NgNonBindable

Learning Objectives

- Understand when and how to use the `NgNonBindable` directive.

Description

We use `ngNonBindable` when we want tell Angular not to compile, or bind, a particular section of our page.

The most common example of this is if we wanted to write out some Angular code on the page, for example if we wanted to render out the text `{{ name }}` on our page, like so:

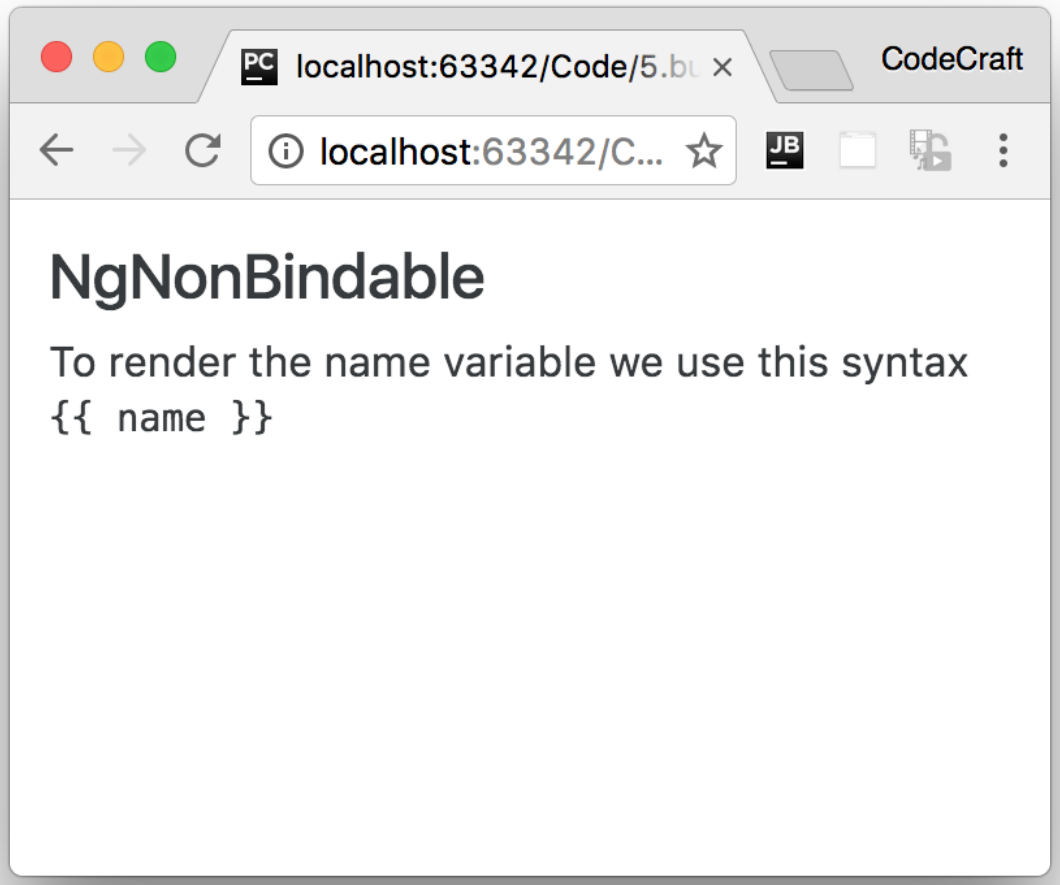
```
<div>
  To render the name variable we use this syntax <pre>{{ name }}</pre>
</div>
```

Normally Angular will try to find a variable called `name` on the component and print out the value of the `name` variable instead of just printing out `{{ name }}`.

To make angular ignore an element we simply add the `ngNonBindable` directive to the element, like so:

```
<div>
  To render the name variable we use this syntax <pre ngNonBindable>{{ name }}</pre>
</div>
```

If we run this in the browser we would see:



Summary

We use `NgNonBindable` when we want to tell Angular not to perform any binding for an element.

Listing

http://plnkr.co/edit/1u12vtOPFA9K00jvUjlO?p=preview

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

@Component({
  selector: 'ngnonbindable-example',
  template: `<h4>NgNonBindable</h4>
<div>
  To render the name variable we use this syntax
  <pre ngNonBindable>{{ name }}</pre>
</div>
 `
})
class NgNonBindableExampleComponent {
}

@Component({
  selector: 'directives-app',
  template: `<ngnonbindable-example></ngnonbindable-example>`
})
class DirectivesAppComponent {
}


@NgModule({
  imports: [BrowserModule],
  declarations: [NgNonBindableExampleComponent, DirectivesAppComponent],
  bootstrap: [DirectivesAppComponent],
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Structural Directives

Learning Objectives

- Understand what a structural directive is.
- Know why we use the * character for some directives only.

Long form structural directives

Structural Directives are directives which *change* the structure of the DOM by adding or removing elements.

There are three built in structural directives, `NgIf`, `NgFor` and `NgSwitch`.

These directives work by using the HTML5 `<template>` tag. This is a new tag in HTML which is specifically designed to hold *template* code. It can sit under the `body` element but any content inside it is not shown in the browser.

Using `template` we can write an `ngIf` expression as:

```
<template [ngIf]='condition'>
  <p>I am the content to show</p>
</template>
```

If we go back to our joke app example and replace the hiding and showing of a joke with this *template* version of `ngIf` we would end up with:

```
<template [ngIf]="!data.hide">
  <p class="card-text">
  {{ data.punchline }}
  </p>
</template>
```

The `NgFor` version is slightly more complex:

```
<template ngFor ①
          let-j ②
          [ngForOf]="jokes"> ③
  <joke [joke]="j"></joke>
</template>
```

① This is the `NgFor` directive itself.

② This is another way of declaring a template local reference variable, equivalent to `#j`.

③ `[ngForOf]` is an *input property* of the `NgFor` directive.

Syntax sugar and *

So if we can write `ngIf` with `template` what is al the fuss about `*`.

When we *prepend* a directive with `*` we are telling it to use the element it's attached to *as* the template.

Looking at the `NgIf` example from above, these two snippets of code are equivalent:

```
<template [ngIf]="!data.hide">
  <p class="card-text">
    {{ data.punchline }}
  </p>
</template>
```

```
<p class="card-text"
   *ngIf="!data.hide">
  {{ data.punchline }}
</p>
```

Finally looking at the more complex `NgFor` example from above, these two snippets of code are also equivalent:

```
<template ngFor
          let-j
          [ngForOf]="jokes">
  <joke [joke]="j"></joke>
</template>
```

```
<joke *ngFor="let j of jokes"
      [joke]="j">
</joke>
```

Summary

Structural directives are a type of directive which changes the structure of the DOM.

They take advantage of the HTML5 `<template>` tag to define the element they want to insert into the DOM.

We can prepend the directive name with `*` to skip having to define a `<template>` and have the directive use the element it's attached to as the template.

Listing

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
    Component,
    Directive,
    NgModule,
    Input,
    Output,
    EventEmitter,
    TemplateRef,
    ViewContainerRef
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {Directive, Input} from '@angular/core';

//
// Domain Model
//

class Joke {
  public hide: boolean;

  constructor(public setup: string, public punchline: string) {
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}

//
// Structural Directives
//

@Directive({
  selector: '[ccIf]'
})
export class CodeCraftIfDirective {
  constructor(private templateRef: TemplateRef<any>,
              private viewContainer: ViewContainerRef) {
  }

  @Input() set ccIf(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
```

```
    }
  }
}


@Directive({
  selector: '[ccFor]'
})
export class CodeCraftForOfDirective {
  constructor(private templateRef: TemplateRef<any>,
              private viewContainer: ViewContainerRef) {
  }

  @Input() set ccForOf(collection: any) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}


//
// Components
//
//

@Component({
  selector: 'joke',
  template: `
<div class="card card-block">
  <h4 class="card-title">
    {{ data.setup }}
  </h4>
  <ng-template [ngIf]="!data.hide">
    <p class="card-text">
    {{ data.punchline }}
  </p>
  </ng-template>
  <button class="btn btn-primary"
          (click)="data.toggle()">Tell Me
  </button>
</div>
`
})
class JokeComponent implements OnInit {
  @Input('joke') data: Joke;
}

@Component({
```

```
  selector: 'joke-list',
  template: `
<ng-template ngFor
          let-j
          [ngForOf]="jokes">
  <joke [joke]="j"></joke>
</ng-template>
`
})
class JokeListComponent {
  jokes: Joke[] = [];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
`
})
class AppComponent {
}

//
// Bootstrap
//

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    CodeCraftIfDirective
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

Directives are just components but without views.

We *attach* directives to elements by adding them as attributes.

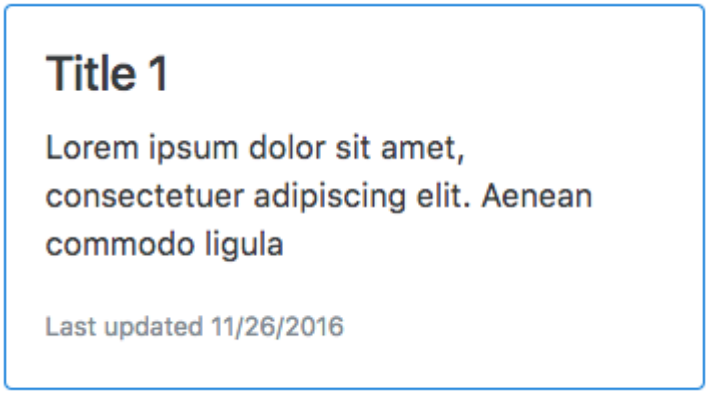
Angular provides a small number of built-in directives and in this section we covered all of them, including:

- `NgFor`
- `NgIf`
- `NgSwitch`
- `NgClass`
- `NgNonBindable`

In the next section we will create our own custom directives.

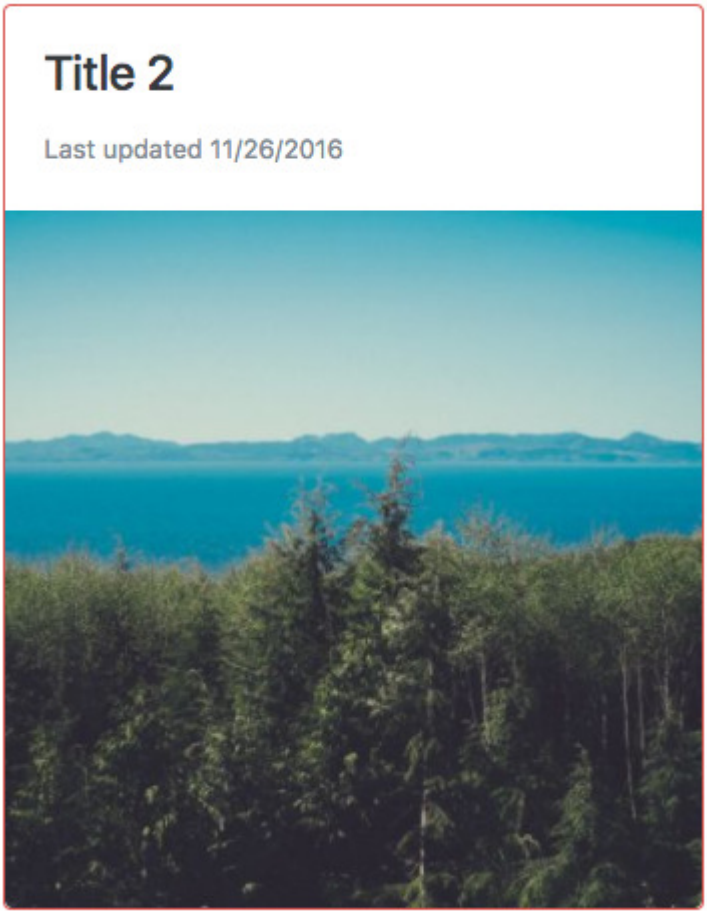
Activity

Create a component which shows the articles from a list. If the article is a text article show a component like this:



To get the blue border use the class `card-outline-primary`

If the article is an image article show a component like this:



To get the red border use the class `card-outline-danger`

Use the `shortDate` date format string to get the date formatted as we want.

Steps

Fork this plunker:

http://plnkr.co/edit/i6Lc92Of0DHjFy5xnmCX?p=preview

Use the built-in directives you've just learned to finish off the `RecentArticlesComponent`.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/aNeU1SVei02wYfjdNefY?p=preview

Custom Directives

Overview

By this point you should be able to create your own custom component as well as use a set of *built-in* Angular directives.

Now it's time to learn how to build your own *custom* directives.

But you might be surprised to hear that you've *already* created a custom directive. That's because *Components are Directives*.

Components have all the features of Directives but also have a view, that is to say they have a template and some HTML that is injected into the DOM when we use it.

Another difference is that a single HTML element can only have a *single component* associated with it. However a single element can have *multiple directives* associated with it.

Lets continue with our joke example app and create a directive which shows the punchline of the joke when the user hovers over the card.

In this section you will learn:

- How to create custom directives using the `@Directive` decorator.
- How directives can both listen to events and change properties of the host element they are associated with.
- How we can configure a directive so that it can take `inputs` when it's defined on an element, like `[aDirective]={config:'value'}`

Creating a custom directive

In this lecture we are going to create our very own custom directive.

Learning Objectives

- Know how to create a basic directive using the `@Directive` decorator.
- Know how to use selectors to associate an element with a directive based on an *attribute*.
- Know how to interact with the raw DOM element of the associated element from the directive.

Directive decorator

We'll call our directive `ccCardHover` and we'll attach it to the card block like so:

```
<div class="card card-block" ccCardHover>...</div>
```

The Angular team recommends using directives as attributes, prefixed with a namespace. We've prefixed our directive with the namespace 'cc'.

We create directives by annotating a class with the `@Directive` decorator.

Lets create a class called `CardHoverDirective` and use the `@Directive` dectorator to associate this class with our attribute `ccCardHover`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:"[ccCardHover]"
})
class CardHoverDirective { }
```

Attribute selector

The above code is very similar to what we would write if this was a component, the first striking difference is that *the selector is wrapped with `[]`*.

To understand why we do this we first need to understand that the selector attribute uses *CSS matching rules* to match a component/directive to a HTML element.

In CSS to match to a specific element we would just type in the name of the element, so `input {⋯}`or `p {⋯}`.

This is why previously when we defined the selector in the `@Component` directive we just wrote the

name of the element, which matches onto an element of the same name.

If we wrote the selector as `.ccCardHover`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:".ccCardHover"
})
class CardHoverDirective { }
```

Then this would associate the directive with any element that has a *class* of ccCardHover, like so:

```
<div class="card card-block ccCardHover">...</div>
```

We want to associate the directive to an element which has a certain attribute.

To do that in CSS we wrap the name of the attribute with `[]`, and this is why the selector is called `[ccCardHover]`.

Directive constructor

The next thing we do is add a constructor to our directive, like so:

```
import { ElementRef } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef) {
  }
}
```

When the directive gets created Angular can inject an instance of something called `ElementRef` into its constructor.

How this works is called *Dependency Injection*, it's a really important aspect of Angular and we discuss this in detail in a later section.

The `ElementRef` gives the directive *direct access* to the DOM element upon which it's attached.

Let's use it to change the background color of our card to gray.

`ElementRef` itself is a wrapper for the actual DOM element which we can access via the property `nativeElement`, like so:

```
el.nativeElement.style.backgroundColor = "gray";
```

This however assumes that our application will always be running in the environment of a browser.

Angular has been built from the ground up to work in a number of different environments, including server side via node and on a native mobile device. So the Angular team has provided a *platform independent* way of setting properties on our elements via something called a `Renderer`.

script.ts

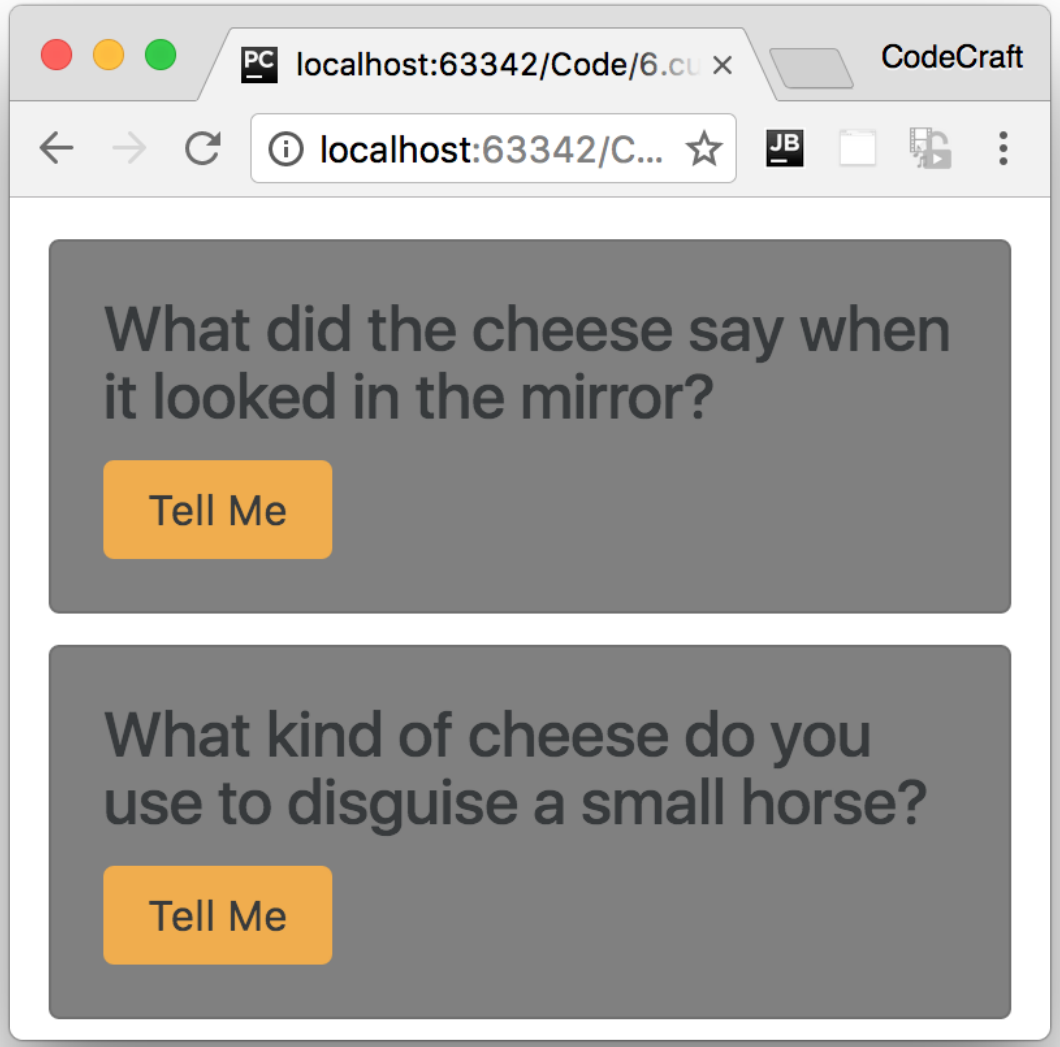
```
import { Renderer } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) { ①
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray'); ②
  }
}
```

① We use *Dependency Injection* (DI) to inject the `renderer` into our directives constructor.

② Instead of setting the background color directly via the DOM element we do it by going through the `renderer`.

In the future if we wanted to render our application on a platform other than a web browser then the `Renderer` calls the appropriate functions to change the background color on that platform. We are not limited to only being run in a web browser with a DOM.

Running the application now show this:



Summary

We create a directive by decorating a class with the `@Directive` decorator.

The convention is to associate a directive to an element via an *attribute selector*, that is the name of the attribute wrapped in `[]`.

We can inject a reference to the element the directive is associated with to the constructor of the directive. Then via a `renderer` we can interact with and change certain properties of that element.

The above is a very basic example of a custom directive, in the next lecture we'll show you how you can detect when the user hovers over the card and a *better* way of interacting with the host element.

Listing

http://plnkr.co/edit/Aj5s1koYdGKlK5vkyGCn?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
    Component,
    Directive,
    Renderer,
    ElementRef,
    NgModule,
    Input,
    Output,
    EventEmitter
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}


@Directive({
  selector: "[ccCardHover]"
})
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) {
    //noinspection TypeScriptUnresolvedVariable,TypeScriptUnresolvedFunction
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block" ccCardHover>
```

```
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [hidden]="data.hide">{{data.punchline}}</p>
  <button (click)="data.toggle()"
     class="btn btn-primary">Tell Me
  </button>
</div>
  `
})
class JokeComponent {
  @Input('joke') data: Joke;
}

@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    CardHoverDirective
```

```
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

HostListener & HostBinding

Learning Objectives

- How to respond to output events that occur on the host element the directive is attached to.
- How to manipulate the host element by binding to it's input properties.

HostListener

In the previous lecture we created our first directive, `ccCardHover`, which simply turned the background color gray for every element it's attached to.

But as the name of the directive implies we need a way of *detecting* if the user is hovering over the host element.

Angular makes this easy with the `@HostListener` decorator.

This is a *function* decorator that accepts an *event name* as an argument. When that event gets fired on the *host* element it calls the associated function.

So if we add this function to our directive class:

```
@HostListener('mouseover') onHover() {
  window.alert("hover");
}
```

Hovering over the host element would trigger an alert popup.

Lets change our directive to take advantage of the `@HostListener`.

```
import { HostListener } from '@angular/core'
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) {
    ①
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() { ②
    let part = this.el.nativeElement.querySelector('.card-text') ③
    this.renderer.setElementStyle(part, 'display', 'block'); ④
  }
}
```

① We've removed the code to render the background color to gray.

② We decorate a class method with `@HostListener` configuring it to call the function on every `mouseover` events.

③ We get a reference to the DOM element that holds the jokes punchline.

④ We set the display to block so that element is shown.

For the above to work we need to change our `JokeComponent` template so the joke is hidden using the display style property, like so:

```
<div class="card card-block" ccCardHover>
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [style.display]="'none'">{{data.punchline}}</p> ①
  ②
</div>
```

① We change the `p` element so it's hidden via a style of `display: none`, this is so we can *show* the element by setting the style to `display: block`.

② We've also removed the button, so the only way to show the punchline is via hovering over the card.

As well as showing the punchline on a `mouseover` event we also want to *hide* the punchline on a `mouse*out*` event, like so:

```
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'block');
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'none');
  }
}
```

Now when we hover over the card we show the joke and when we move out of the card the punchline is hidden again.

HostBinding

As well as listening to output events from the host element a directive can also bind to *input properties* in the host element with `@HostBinding`.

This directive can *change* the properties of the host element, such as the list of classes that are set on the host element as well as a number of other properties.

Using the `@HostBinding` decorator a directive can link an internal property to an input property on the host element. So if the internal property changed the input property on the host element would also change.

We first need something, a property on our directive which we can use as a source for binding.

We'll create a boolean called `ishovering` and in our `onMouseOver()` and `onMouseOut()` functions we'll set this to true and false accordingly, like so:

```
class CardHoverDirective {
  private ishovering: boolean; ①

  constructor(private el: ElementRef,
              private renderer: Renderer) {
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true; ②
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'none');
    this.ishovering = false; ②
  }
}
```

① We've created a boolean called `ishovering`.

② We set our boolean to `true` when we are being hovered over and `false` when we are not.

Now we need to *link* this source property to an input property on the host element, we do this by decorating our `ishovering` boolean with the `@HostBinding` decorator.

The `@HostBinding` decorator takes one parameter, the name of the property on the host element which we want to bind to.

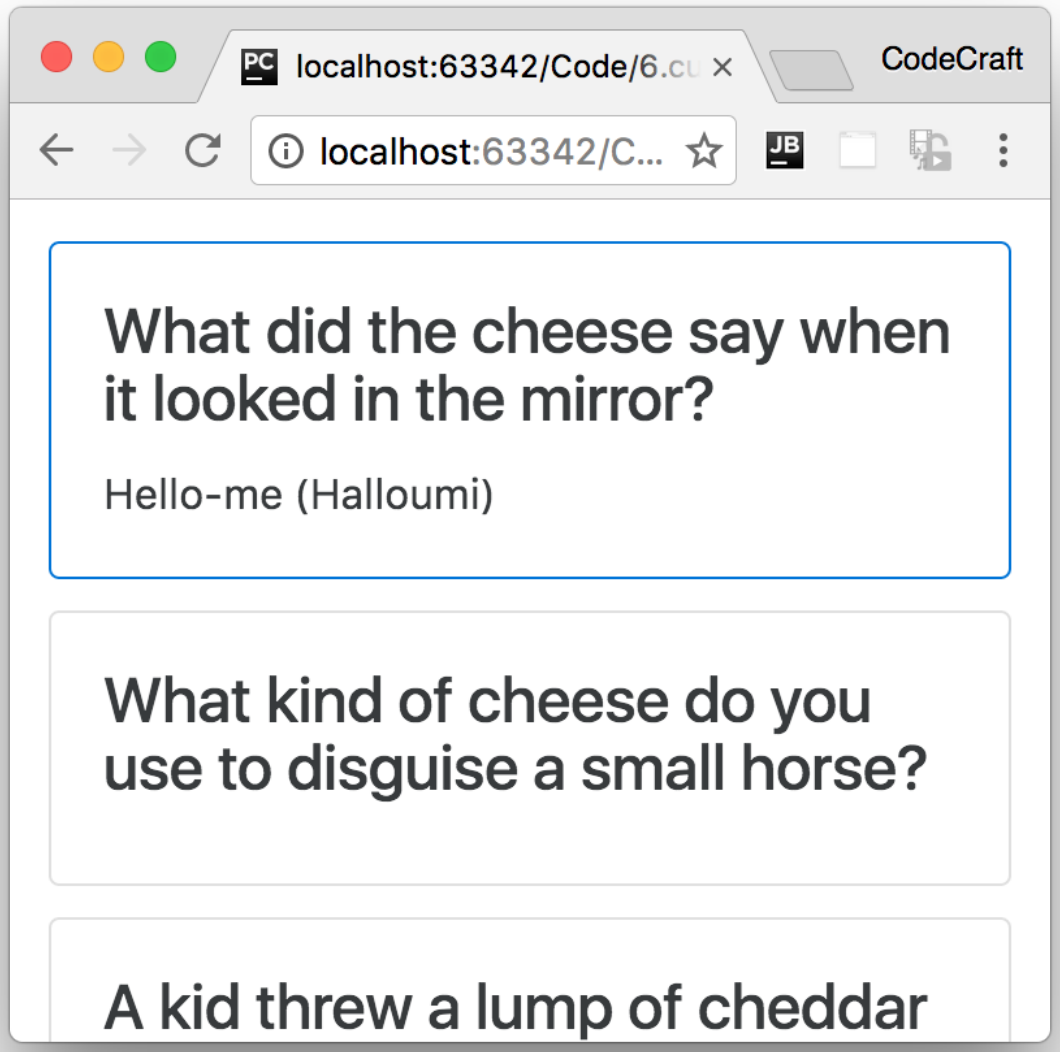
If you remember we can use the alternative `ngClass` syntax by binding to the `[class.<class-name>]` property. Lets add the `card-outline-primary` class to our host element when the `ishovering` boolean is `true`.

script.ts

```
import { HostBinding } from '@angular/core'
.
.
.
class CardHoverDirective {
  @HostBinding('class.card-outline-primary') private ishovering: boolean; ①

  constructor(private el: ElementRef,
              private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'none');
    this.ishovering = false;
  }
}
```

① We've added the `@HostBinding` decorator to the `ishovering` property.

Now when we hover over a card as well as the punchline showing we adding a blue border to the card, like so:



Summary

By using the `@HostListener` and `@HostBinding` decorators we can both listen to output events from our host element and also bind to input properties on our host element as well.

In the next lecture we will cover how to provide inputs and configuration to our directives so they can be easily re-used.

Listing

http://plnkr.co/edit/EgsmbXMN7s7YYDYIu9N8?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
```

```
import {
    Component,
    Directive,
    Renderer,
    HostListener,
    HostBinding,
    ElementRef,
    NgModule,
    Input,
    Output,
    EventEmitter
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}


@Directive({
  selector: "[ccCardHover]"
})
class CardHoverDirective {
  @HostBinding('class.card-outline-primary')private ishovering: boolean;

  constructor(private el: ElementRef,
              private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'none');
```

```
    this.ishovering = false;
  }
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block" ccCardHover>
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [style.display]="'none'">{{data.punchline}}</p>
</div>
  `
})
class JokeComponent {
  @Input('joke') data: Joke;
}

@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

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

```
  declarations: [
    AppComponent,
    JokeComponent,
    JokeListComponent,
    CardHoverDirective
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Inputs & Configuration

Learning Objectives

- How to make a directive configurable.

Configuration

In the last lecture we finished off our `ccCardHover` directive. But its not very re-usable; we now want to be able to *configure* it so that it can be used in other situations.

One such configuration parameter is the query selector for the elemenent we want to hide or show, currently it's hard coded to `.card-text`, like so:

```
let part = this.el.nativeElement.querySelector('.card-text');
```

The first thing to do is move the query selector to a *property* of our directive, but to future-proof ourselves i'm going to set it to a property of an *object*, like so:

```
config: Object = {
  querySelector: '.card-text'
}
```

This way if we wanted to add further config params in the future we can just add them as properties to our config object.

Next up lets use this config object instead of our hard coded selector.

```
let part = this.el.nativeElement.querySelector(this.config.querySelector);
```

Finally lets make our `config` property an *input binding* on the directive.

```
@Input() config: Object = {
  querySelector: '.card-text'
}
```

Now to configure our directive we can add an input property binding on the *same element* the directive, like so:

```
<div class="card card-block"
     ccCardHover
     [config]="{querySelector:'p'}"> ①
     ...
<div>
```

① We've configured the `querySelector` to select on `.card-text` again, just like before but this time it's configurable.

But what if we wanted to use our directive like this:

```
<div class="card card-block"
     [ccCardHover]="{querySelector:'.card-text'}"> ①
     ...
<div>
```

Just like how we've seen other built-in directives work.

That's pretty simple to do, we just need to add an *alias* to the input decorator which matches this decorators *selector*, like so:

```
@Input('ccCardHover') config: Object = {
  querySelector: '.card-text'
}
```

Now we can use and configure our directive in one statement!

The code for our completed directive looks like this:

```
@Directive({
  selector: "[ccCardHover]"
})
class CardHoverDirective {
  @HostBinding('class.card-outline-primary') private ishovering: boolean;

  @Input('ccCardHover') config: Object = {
    querySelector: '.card-text'
  };

  constructor(private el: ElementRef,
              private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector(this.config.querySelector);
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector(this.config.querySelector);
    this.renderer.setElementStyle(part, 'display', 'none');
    this.ishovering = false;
  }
}
```

Summary

We can configure our directives with standard input property bindings.

To make the syntax look similar to the built-in directives we use an *alias* for the `@Input` decorator to match the directives selector.

Listing

http://plnkr.co/edit/HMY7tUIztJ79WEmQaj2f?p=preview

script.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
    Component,
    Directive,
    Renderer,
    HostListener,
    HostBinding,
```

```
    ElementRef,
    NgModule,
    Input,
    Output,
    EventEmitter
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

class Joke {
  public setup: string;
  public punchline: string;
  public hide: boolean;

  constructor(setup: string, punchline: string) {
    this.setup = setup;
    this.punchline = punchline;
    this.hide = true;
  }

  toggle() {
    this.hide = !this.hide;
  }
}


@Directive({
  selector: "[ccCardHover]"
})
class CardHoverDirective {
  @HostBinding('class.card-outline-primary') private ishovering: boolean;

  @Input('ccCardHover') config: Object = {
    querySelector: '.card-text'
  };

  constructor(private el: ElementRef,
              private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector(this.config.querySelector);
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector(this.config.querySelector);
    this.renderer.setElementStyle(part, 'display', 'none');
    this.ishovering = false;
  }
```

```
}

@Component({
  selector: 'joke',
  template: `
<div class="card card-block"
     [ccCardHover]="{querySelector:'.card-text'}">
  <h4 class="card-title">{{data.setup}}</h4>
  <p class="card-text"
     [style.display]="'none'">{{data.punchline}}</p>
</div>
  `
})
class JokeComponent {
  @Input('joke') data: Joke;
}

@Component({
  selector: 'joke-list',
  template: `
<joke *ngFor="let j of jokes" [joke]="j"></joke>
  `
})
class JokeListComponent {
  jokes: Joke[];

  constructor() {
    this.jokes = [
      new Joke("What did the cheese say when it looked in the mirror?", "Hello-me
(Halloumi)"),
      new Joke("What kind of cheese do you use to disguise a small horse?", "Mask-a-
pony (Mascarpone)"),
      new Joke("A kid threw a lump of cheddar at me", "I thought ‘That’s not very
mature’"),
    ];
  }
}


@Component({
  selector: 'app',
  template: `
<joke-list></joke-list>
  `
})
class AppComponent {
}

@NgModule({
  imports: [BrowserModule],
  declarations: [
```

```
    AppComponent,
    JokeComponent,
    JokeListComponent,
    CardHoverDirective
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

In this section we covered how to create a custom directive.

We learned:

- That components *are* directives, but directives with views and templates.
- How to define a directive with the `@Directive` decorator.
- How to listen to the output events and bind to the input properties of a host element from within a directive.
- How to configure a directive to make it more re-usable.

Activity

Create a custom *rollover* image directive which shows one image normally and another when the user hovers over it.

The directive will be used like so:

```
<img [ccRollover]="{
  'initial':'https://unsplash.it/200/300?image=201',
  'over':'https://unsplash.it/200/300?image=202'
}"/>
```

Steps

Fork this plunker:

http://plnkr.co/edit/DcGlVKGc0UDKzqNkPH00?p=preview

Flesh out the `RolloverImageDirective` class to implement the functionality you need.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

Reactive Programming with RxJS

Overview

This is a quick introduction into the concept of Reactive Programming in Angular with RxJS.

The subject of *Reactive Programming* is large enough to warrant it's own course and indeed there are a number of books and courses which deal with nothing other than reactive programming with RxJS.

So the goal of this section *isn't* to give you a *complete* understanding, the goal is to *demystify* the concepts so you understand what it is and *where* you can learn more.

Angular uses RxJS for some parts of it's internal functioning. If you want you *can* also choose to use RxJS but you don't need to at all.

You do *not* need to know Reactive Programming or RxJS in order to build even the *most complex* applications with Angular. It can however make some types of applications easier to architect.

At the end of this section you will:

- Understand the terms *Stream* & *Reactive Programming*.
- Know what *Observables* are and how they are related to *RxJS*.
- Know how to write reactive code using pure RxJS.
- Know what operators are, where to read up on the breadth of available operators and how to understand them by using marble diagrams.
- Know the places you can use reactive programming in Angular and code up a simple form using RxJS.

Streams & Reactive Programming

Learning Objectives

- Know what streams are and how to think about things that happen in an application as streams.
- Know what reactive programming is and how to start transforming your way of thinking from imperative to reactive.

What are Streams?

Streams are a *sequence of values over time*, that's it.

For example a number that goes up by 1 every second might have a *stream* that looks like

`[0,1,2,3,4]`

Another stream might be a sequence of x and y positions of mouse click events, like so:

`[(12,34), (345,22), (1,993)]`

We could even have a stream to represent a user filling in a form on a website.

We could have a stream to represent each keypress, like so:

```
[
  "A",
  "s",
  "i",
  "m"
]
```

Or we could have a stream which contains a json representation of the whole form as the user enters data, like so:

```
[
  { "name": "A" },
  { "name": "As" },
  { "name": "Asi" },
  { "name": "Asim" }
]
```

We could have a stream for:

- The x,y position of the mouse as it moves around the screen in a HTML5 game.
- The data returned from a real-time websockets connection.
- The chat windows opened by this user in a browser.

The more you think about it the more *everything* we do with a web application can be thought of as a stream.

What is Reactive Programming?

Reactive programming is the idea that you can create your entire program just by defining the different streams and the *operations* that are performed on those streams.

As a concept that is easy to write, but how can we actually *train* our mind to program *reactively*?

To explain this lets convert a simple imperative function into a reactive one.

Imperative programming is a programming paradigm that you probably have been using so far in your career, it's by far the most common and it's involves executing statements that change a programs state, i.e. call functions that change the values of variables.

To get a good overview of the different programming paradigms read this article

```
add ( A , B ) {
  return A + B;
}

C = add(1, 2);
C = add(1, 4);
```

We have a function called `add` and some state variables, `A`, `B` and `C`.

To add `A` and `B` together and *change the state* of `C` to be the sum we call the function `add`.

Later on the value of `B` changes to 4.

1. First we need a way of simply *knowing* that `B` has changed, that's hard enough to figure out by

itself.

2. Secondly we need to know that because `B` has changed we need to recalculate `C`.

In a web application our inputs are constantly changing over time, via things like network events or a user interacting with a mouse.

Most of the logic we end up writing is just to figure out *what* functions need to be called for each of these changes to our inputs.

Applications can be thought of as just a huge pile of *variables* (which we call application state) as well *logic* to decide which *functions* to call, and in what order, when any of those variables change.

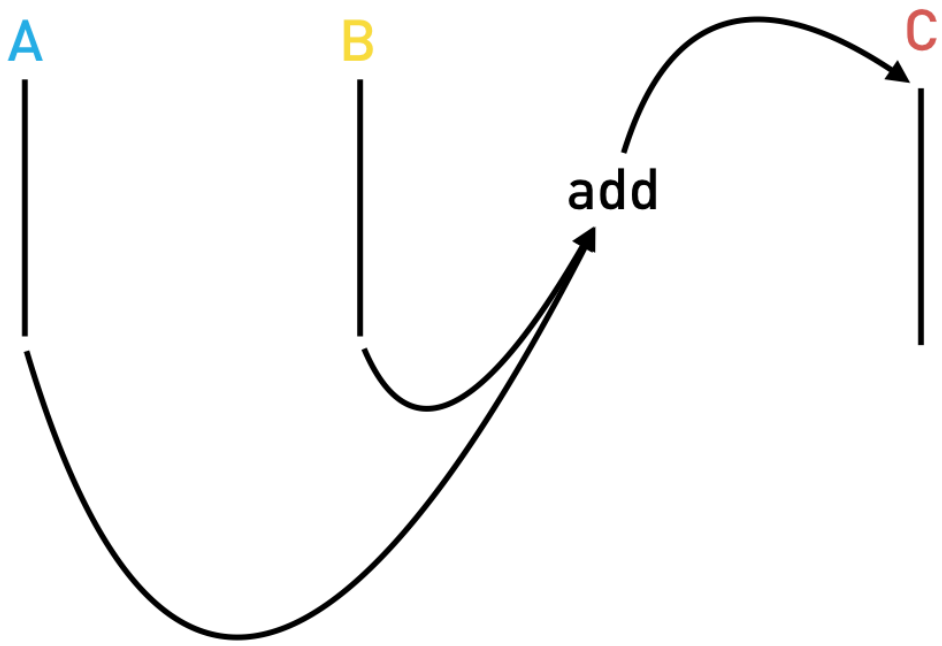
Calling those functions then also changes the values of variables, for which we need additional logic to figure out what *other* functions to call... it's endless!

With reactive programming we *stop* thinking about variables, instead we think in terms of *streams* and how those streams are *connected* together.

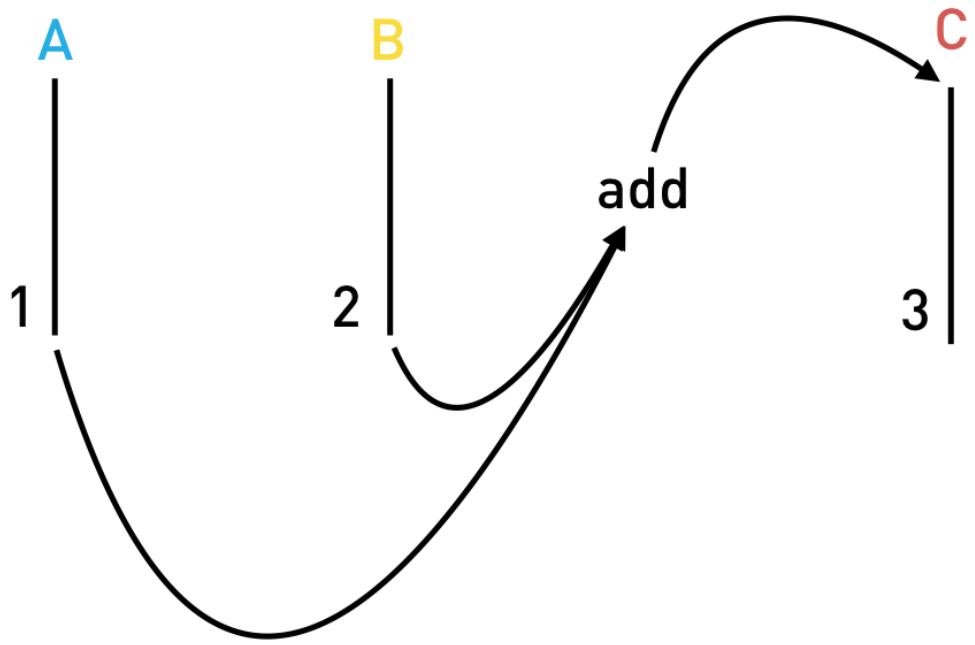
Going back to our example, we convert the variables `A`, `B` and `C` into *streams*.

`A` is now *not* an individual value at one point in time, it's a *stream* of values over time.

The function `add` we think of an an *operation* we perform which connects the output of streams `A` and `B` to the input of stream `C`, a visual representation would be something like the below:



Now if we push some numbers onto stream `A` and `B`, the `add` operation is *automatically* called, calculating the sum of 4 and pushing it onto stream `C`.

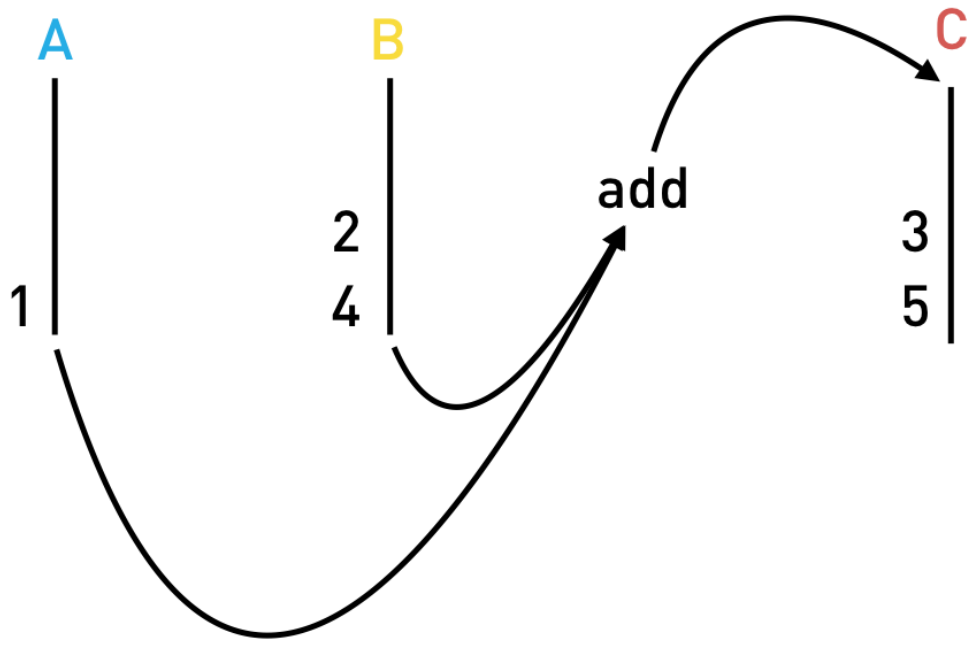


If stream `C` was connected to another stream via another operation, that operation would then be called automatically as well.

An analogy which works for me is to think about reactive programming as *plumbing*. We decide which *pipes* we need in our application, we decide *how* those pipes are connected together and then we *turn on* the water and sit back.

With reactive programming *we don't call functions*, we just define how our application is plumbed together and start pushing values onto streams and let the plumbing and operations handle the rest.

So if later on the value of `B` changes, we simply push the new value onto the stream `B` and then let the plumbing handle the rest, like so:



Summary

Streams are just a *sequence of values* over time.

Reactive programming is the idea we can define an application as a series of different streams with operations that connect the different streams together and which are automatically called when new values are pushed onto those streams.

In this this lecture we just covered the idea, the concept of reactive programming, in the next lecture we'll cover how to actually program reactively using observables and the RxJS library.

Observables & RxJS

Learning Objectives

- Know what *Observables* are.
- Know what *RxJS* is and how it relates to *Observables*.
- Know what *operators* are, how to find out about the list of operators and how to understand an operators function by using marble diagrams.
- Know how to build a simple application using *RxJS*.

Observables

Streams so far are just a *concept*, an idea.

We link streams together using operators, so in in our previous example the `add` function is an operation, specifically it's an operation which combines two streams to create a third.

Observables is a new primitive type which acts as a *blueprint* for how we want to create streams, subscribe to them, react to new values, and combine streams together to build new ones.

It's currently in discussion whether or not Observables make it into the *ES7* version of JavaScript.

We are still trying to roll out ES6 so even if it makes it, it will be many years before ES7 becomes something we can code with natively.

Until then we need to use a library that gives us the *Observable* primitive and that's where *RxJS* comes in.

RxJS

RxJS stands for *R*eactive E*x*tensions for *J*ava*S*cript, and its a library that gives us an implementation of Observables for JavaScript.

Observables might become a core part of the JavaScript language in the future, so we can think of RxJS as a placeholder for when that arrives.

RxJS is the JavaScript *implementation* of the ReactiveX API, which can be found here.

The API has *multiple* implementations in *different languages*, so if you learn RxJS you'll know how to write RxJAVA, Rx.NET, RxPY etc...

Library

Let's explain RxJS by working through a simple example.

To reduce file size the RxJS library is broken up into many different parts, one main one and one

for each operation you want to use.

For our example we'll add the `rx.all.js` library which contains *all* the operators.

We create a simple index.html file and add the `rx.all.js` library in via a `script` tag.

In Angular since we are using modules we'll be adding in RxJS using `import` statements. We are using script tags here just for simplicity of setup.

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/rx.all.js"></script>
  <script src="main.js"></script>
</head>
<body>
</body>
</html>
```

We also create a `main.js` where we will start adding our RxJS code.

interval

The first thing we need to to is get an instance of an RxJS Observable, we do this like so:

```
let obs = Rx.Observable;
```

An observable **isn't** a stream. An observable is a *blueprint* which describes a *set* of streams and how they are connected together with `operations`.

I want our observable to create a *single* stream and push onto that stream a number every second, incremented by 1.

With RxJS to define an observable to achieve the above we would use the operator `interval`, like so:

```
let obs = Rx.Observable
    .interval(1000);
```

The operation `interval` takes as the first param the number of milliseconds between each *push* of the number onto the stream.

In RxJS operators act on an observable and return an observable with the operator applied, so we can `chain` operators together creating an *Observable Chain*, like so:

```
let obs = Rx.Observable
    .operator1();
    .operator2();
    .operator3();
    .operator4();
    .operator5();
```

subscribe

In RxJS land no one can hear you *stream*, unless you subscribe.

This observable is *cold*, that means it's not currently pushing out numbers.

The observable will become *hot* and start pushing numbers onto it's first stream, when it gets it's first *subscriber*, like so:

```
let obs = Rx.Observable
    .interval(1000);

obs.subscribe(value => console.log("Subscriber: " + value));
```

By calling `subscribe` onto an observable it:

1. Turns the observable *hot* so it starts producing.
2. Lets us pass in a callback function so we react when anything is pushed onto the *final stream* in the observable chain.

Our application now starts printing out:

```
Subscriber: 0
Subscriber: 1
Subscriber: 2
Subscriber: 3
Subscriber: 4
Subscriber: 5
Subscriber: 6
Subscriber: 7
Subscriber: 8
Subscriber: 9
Subscriber: 10
```

take

But it just keeps on printing, forever, we just want the first 3 items so we use another operator called `take`.

We pass to that operator the *number of items* we want to take from the first stream. It creates a second stream and only pushes onto it the number of items we've requested, like so:

```
let obs = Rx.Observable
    .interval(1000)
    .take(3);

obs.subscribe(value => console.log("Subscriber: " + value));
```

This now prints out the below, and then stops:

```
Subscriber: 0
Subscriber: 1
Subscriber: 2
```

map

Finally I want to add another operator called `map`, this takes as input the output stream from `take`, convert each value to a date and pushes that out onto a third stream like so:

```
let obs = Rx.Observable
    .interval(1000)
    .take(3)
    .map((v) => Date.now());

obs.subscribe(value => console.log("Subscriber: " + value));
```

This now prints out the time in milliseconds, every second, like so:

```
Subscriber: 1475506794287
Subscriber: 1475506795286
Subscriber: 1475506796285
```

Other operators

The above example showed a very very small subset of the total number of operators available to you when using RxJS.

The hardest part of *learning* RxJS is understanding each of these operators and how to use them.

In that regard even though you are writing in JavaScript learning RxJS is closer to learning another language altogether.

You can find a list of the operators by looking at the official documentation here.

The documentation for the operators we just used above is:

- Interval
- Take
- Map

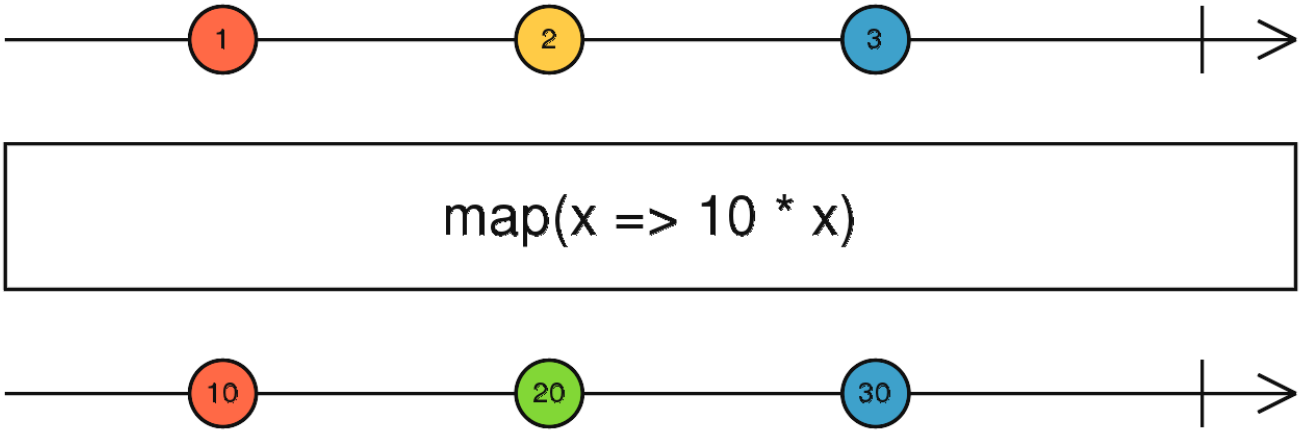
Marble Diagrams

Trying to understand an operator by just reading some words is pretty difficult.

This is why in this lecture I've tried to use animations as much as possible.

The Rx team use something called a *marble* diagram to visually describe an operators function.

This is the official marble diagram for the `map` operator:

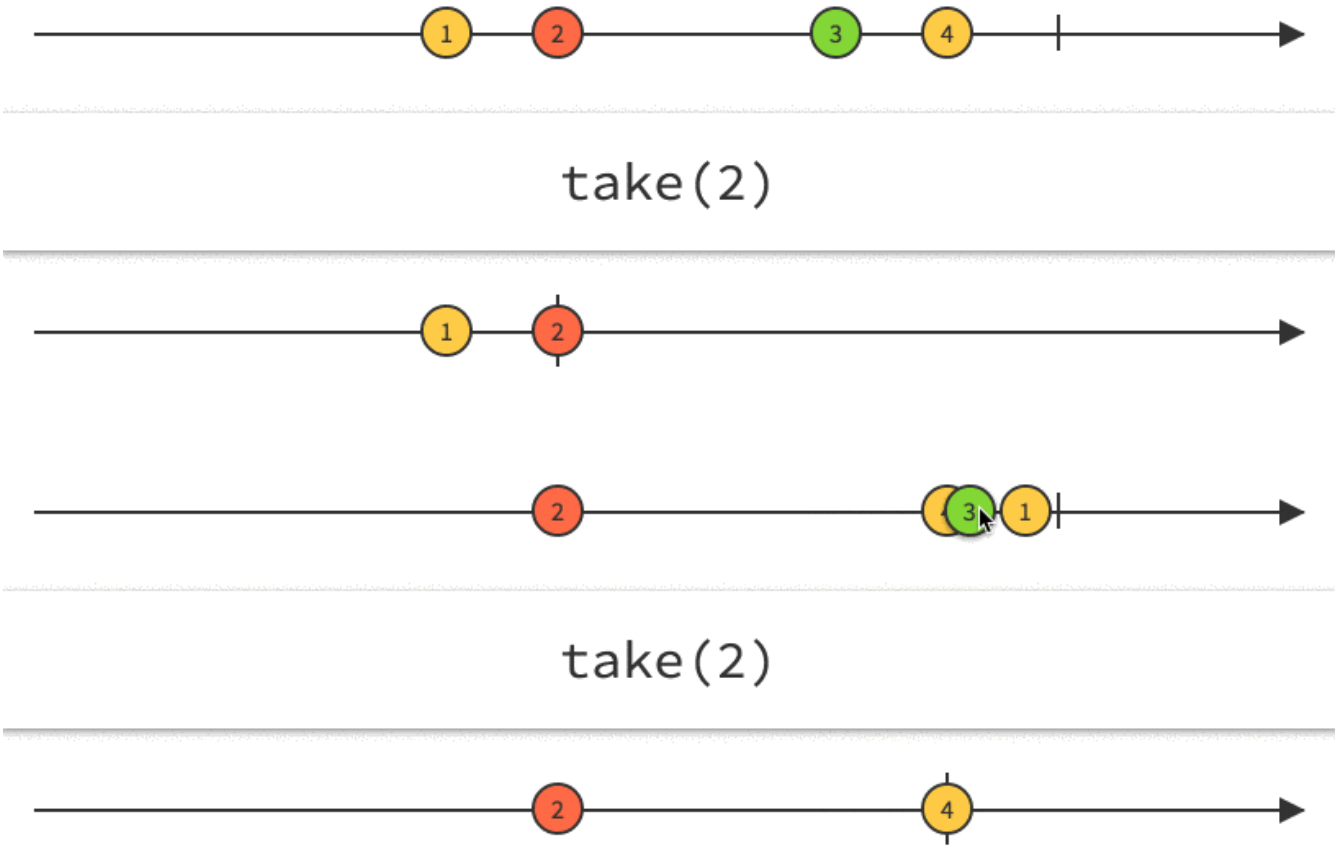


- The line at the top represents *time* and the *marbles* with numbers 1, 2 and 3 represent the *input* stream over time.
- The line at the bottom represents the *output* stream after each of the marbles has been processed through the operator.
- The bit in the middle is the operator, in this example the operator is a `map` function which multiplies each *marble* in the input stream by 10 and pushes them to the output stream.

So in the above the value 1 gets pushed out onto the output stream as 10.

These diagrams are actually *interactive*.

To understand how an operator works we move the marbles around in the input stream and see how this affects the output stream, like so:



Marbles for the above operators are Take and Map

Summary

Observables are a blueprint for creating streams and *plumbing* them together with *operators* to create *observable chains*.

RxJS is a library that lets us create and work with observables.

We can *subscribe* to an observable chain and get a callback every time something is pushed onto the *last* stream.

By default observables are *cold* and only become *hot* when they get their first subscriber.

Learning RxJS involves understanding all the operators that are available and how to use them together.

We use marble diagrams to help explain how an operator works.

In this lecture we used RxJS in isolation, in the next lecture we will look at how to use RxJS in Angular.

Listing

http://plnkr.co/edit/jfs3Y2WcXXGPVPsS5BM9?p=preview

main.js

```
let obs = Rx.Observable
    .interval(1000)
    .take(3)
    .map((v) => Date.now());

obs.subscribe(value => console.log("Subscriber: " + value));
```

index.html

```
<!DOCTYPE html>
<html>

<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/rx.all.js"></script>
  <script src="main.js"></script>
</head>

<body>
</body>

</html>
```

RxJS & Angular

Learning Objectives

- Know which parts of Angular expose observables.
- Know how to *practically* use RxJS with Angular Forms.
- Know how to chain common RxJS operators together.
- Know that you don't need to use observables in Angular and how to implement the same solution without an observable chain.

Angular observables

There are a few places in Angular where reactive programming and observables are in use.

EventEmitter

Under the hood this works via Observables.

HTTP

We've not covered this yet but HTTP requests in Angular are all handled via Observables.

Forms

Reactive forms in Angular expose an observable, a stream of all the input fields in the form combined.

In this lecture we'll use Forms as an example of how to do Reactive Programming in Angular.

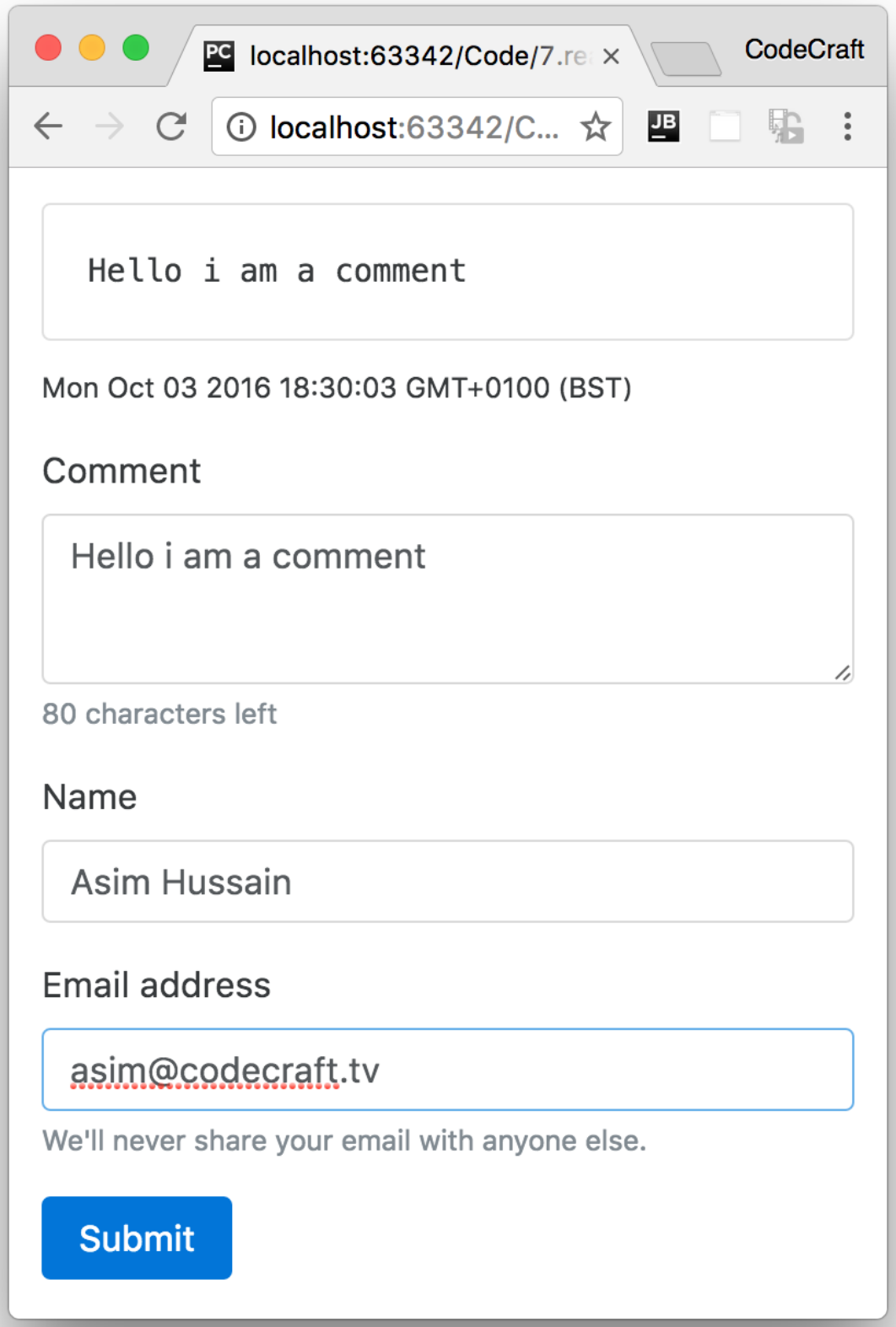
In order to teach you a practical example of RxJS in Angular I have to jump ahead and use something we have not covered yet, *Forms* and specifically the *Reactive Forms* module.

This is not the only way to implement forms in Angular. This is the reactive *model driven* approach, there is also the template driven approach, will go through both in the course.

We'll cover forms in *much* more detail later on in this course so for now I'm going to *gloss* over a lot of the details and focus *only* on the code that deals with RxJS.

Reactive form example

In our example we have a `FormAppComponent` which renders a form like so:



We have a comment input area, a name and email input field and also a submit button.

We won't go through the listing for the template HTML, since it's complex and won't make sense

until we cover forms later on, but we will go through the code for the component class, like so:

form-app-component.ts

```
import 'rxjs/Rx'; ①
.
.
.
class FormAppComponent {
  form: FormGroup; ②
  comment = new FormControl("", Validators.required); ③
  name = new FormControl("", Validators.required); ③
  email = new FormControl("", [ ③
    Validators.required,
    Validators.pattern("[^ @]*@[^ @]*")
  ]);

  constructor(fb: FormBuilder) {
    this.form = fb.group({ ④
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
  }

  onSubmit() {
    console.log("Form submitted!");
  }
}
```

① We import the *full* RxJS library into our application (this import includes *all* the operators).

② We create a `form` property which will hold a representation of our form so we can interact with it from code.

③ We create individual instances of controls and rules for when user input is valid or invalid.

④ We then link our `form` with the controls we created in the constructor using something called a `FormBuilder`

We create a `form` instance on our component, this instance exposes an observable, a stream of all the input fields combined into a object, via it's `valueChanges` property.

We can subscribe to that observable and print our the current value of the form, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

Then as I type into the form, this below gets printed to the console.

```
{"comment":"f","name":"","email":""}
{"comment":"fo","name":"","email":""}
{"comment":"foo","name":"","email":""}
{"comment":"foo","name":"a","email":""}
{"comment":"foo","name":"as","email":""}
{"comment":"foo","name":"asi","email":""}
{"comment":"foo","name":"asim","email":""}
{"comment":"foo","name":"asim","email":"a"}
{"comment":"foo","name":"asim","email":"as"}
{"comment":"foo","name":"asim","email":"asi"}
{"comment":"foo","name":"asim","email":"asim"}
{"comment":"foo","name":"asim","email":"asim@"}
```

Processing only valid form values

Looking at the stream above we can see that *most* of the stream items are for invalid forms; comment, name and email are required so any form without a value for those is invalid. Also we have some special validation logic for the email field, it's only valid if it contains an @ character.

In fact the only valid stream item is the last one `{"comment":"foo","name":"asim","email":"asim@"}`.

That's a common issue when dealing with forms, we only want to *bother* processing the results of a *valid* form, there really isn't any point processing invalid form entries.

We can solve this by using another RxJS operator called `filter`. `filter` accepts a function and passes to it each item in the stream, if the function returns true `filter` publishes the input item to the output stream.

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

`this.form.valid` is true when the whole form is valid. So while the form is *invalid* `.filter(data => this.form.valid)` *doesn't* push items to the output stream, when the form is *valid* it does start pushing items to the output stream.

The end result of the above is that when we type into the form the same data as before, the only item that gets published in our subscribe callback is:

```
{"comment":"foo","name":"Asim","email":"asim@"}
```

Cleaning form data

A comment input box is a dangerous place, hackers try to input things like `<script>` tags and if we are not careful we open ourselves to the possibility of hackers gaming our applications.

So one common safety measure for comment forms is to strip our app html tags from the message before we post it anywhere.

We can solve that again via. a simple `map` operator on our form processing stream.

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .map(data => {
          data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
          return data
        })
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

We add another operator to our stream, specifically we added a `map` operator.

We added this *after* the `filter` operator, so this `map` operator only gets called when the previous `filter` operator publishes to it's output stream. To put it another way, this map operator only gets called on *valid* form values.

```
.map(data => { ①
  data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, ''); ②
  return data ③
})
```

① The `map` operator gets passed the form object as a parameter called `data`.

② We apply a regular expression on the comment property to replace everything that could be a HTML tag with an empty string.

③ What we return from the map function is what gets pushed to the `map` operators output stream.

Now when we type into our form a comment of `<script>` this is what gets printed out:

```
{"comment":"<","name":"Asim","email":"asim@"}
{"comment":"<s","name":"Asim","email":"asim@"}
{"comment":"<sc","name":"Asim","email":"asim@"}
{"comment":"<scr","name":"Asim","email":"asim@"}
{"comment":"<scri","name":"Asim","email":"asim@"}
{"comment":"<scrip","name":"Asim","email":"asim@"}
{"comment":"<script","name":"Asim","email":"asim@"}
{"comment":"","name":"Asim","email":"asim@"}
```

Focusing on the last line `{"comment":"","name":"Asim","email":"asim@"}` we can see that the `<script>` tag the user typed in, is stripped from the comment property.

Adding form values

A useful feature would be if we could let the user know the last time the form was updated.

We can solve this again by adding another map operator to our observable chain, this time we just add the current time to another property to our data object called `lastUpdateTS`, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .map(data => {
          data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
          return data
        })
        .map(data => {
          data.lastUpdateTS = new Date();
          return data
        })
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

This second map operator simply adds a property to our data object and then pushes the data object onto its output stream.

If we now ran our application we would see this printed out:

```
{"comment":"f","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:45.980Z"}
{"comment":"fo","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:46.187Z"}
{"comment":"foo","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:46.364Z"}
```

Not using observables

In Angular we don't *need* to use observables, and therefore reactive programming, if we don't want to.

Instead of adding operators to the observable chain we can choose to *just* subscribe and do the processing in the callback, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .subscribe( data => {
          if (this.form.valid) {
            data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
            data.lastUpdateTS = new Date();
            console.log(JSON.stringify(data))
          }
        });
  }
```

In the above example there doesn't seem to be much advantage to reactive programming via an observable chain vs. just coding up as you are used to in the subscribe callback.

The advantage of RxJS and Observables come to play when we start using more of the complex operators like `debounce` and `distinctUntilChanged`. Implementing the same functionality as those operators via standard imperative coding techniques would take many more lines of code than the equivalent RxJS solution.

Summary

Angular exposes RxJS observables in a small but important number of places in Angular. The EventEmitter, HTTP and Reactive Forms.

We use `operators` to add to the observable chain and then subscribe to the output and perform *actual* real life actions in our application, either change the state of variables or call functions.

We can choose to take advantage of that and code reactively, or we can just subscribe to the observable and code imperatively.

Listing

http://plnkr.co/edit/FbBMirJsnerQqnUXGZel?p=preview

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {ReactiveFormsModule, FormGroup, FormControl, Validators, FormBuilder} from
"@angular/forms";
import 'rxjs/Rx';
```

```
@Component({
  selector: 'form-app',
  template: `<form [formGroup]="form"
      (ngSubmit)="onSubmit()">

  <!-- Output comment -->
  <div class="card card-block">
    <pre class="card-text">{{ form.value.comment }}</pre>
  </div>
  <p class="small">{{ form.value.lastUpdateTS }}</p>

  <!-- Comment text area -->
  <div class="form-group">
    <label for="comment">Comment</label>
    <textarea class="form-control"
              formControlName="comment"
              rows="3"></textarea>
    <small class="form-text text-muted">
           <span>{{ 100 - form.value.comment.length }}</span> characters left
    </small>
  </div>

  <!-- Name input -->
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text"
           class="form-control"
           formControlName="name"
           placeholder="Enter name">
  </div>

  <!-- Email input -->
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email"
           class="form-control"
           formControlName="email"
           placeholder="Enter email">
    <small class="form-text text-muted">
           We'll never share your email with anyone else.
    </small>
  </div>

  <button type="submit"
          class="btn btn-primary"
          [disabled]="!form.valid">Submit
  </button>
</form>
 `
})
```

```
class FormAppComponent {
  form: FormGroup;
  comment = new FormControl("", Validators.required);
  name = new FormControl("", Validators.required);
  email = new FormControl("", [
    Validators.required,
    Validators.pattern("[^ @]*@[^ @]*")
  ]);

  /* Observable Solution */
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .map(data => {
          data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
          return data
        })
        .map(data => {
          data.lastUpdateTS = new Date();
          return data
        })
        .subscribe( data => console.log(JSON.stringify(data)));
  }

  /* None Observable Solution */
  // constructor(fb: FormBuilder) {
  //   this.form = fb.group({
  //     "comment": this.comment,
  //     "name": this.name,
  //     "email": this.email
  //   });
  //   this.form.valueChanges
  //       .subscribe( data => {
  //         if (this.form.valid) {
  //           data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
  //           data.lastUpdateTS = new Date();
  //           console.log(JSON.stringify(data))
  //         }
  //       });
  // }

  onSubmit() {
    console.log("Form submitted!");
  }
}
```

```
@Component({
  selector: 'app',
  template: `
<form-app></form-app>
  `
})
class AppComponent {
}


@NgModule({
  imports: [BrowserModule, ReactiveFormsModule],
  declarations: [AppComponent, FormAppComponent],
  bootstrap: [AppComponent],
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

Reactive programming is about conceptualising our application as *streams* and *operations* that are performed on those streams.

Observables are a blueprint for creating streams and *plumbing* them together with *operators* to create *observable chains*.

RxJS is a library that lets us create and work with observables.

Angular has a few public facing APIs that use Observables, the EventEmitter, Forms and HTTP.

You are not forced to use observables in Angular, you can use as much RxJS or as little as you want.

Activity

There are two more very useful RxJS operators we could use in our sample Angular RxJS application.

`distinctUntilChanged` - http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-distinctUntilChanged

`debounceTime` - http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-debounceTime

Research both of these operators and extend the application so use them.

Steps

Fork this plunker:

http://plnkr.co/edit/pLIOd5ogcvfScLrtIrvV?p=preview

Implement both the operators.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/Bnsjqc9Br3CbeVIhnBeU?p=preview

Pipes

Overview

Pipes are used to transform data, when we *only* need that data transformed in a template.

If we need the data transformed *generally* we would implement it in our model, for example we have a number `1234.56` and want to display it as a currency such as `$1,234.56`.

We could convert the number into a string and store that string in the model but if the only place we want to show that number is in a view we can use a pipe instead.

We use a pipe with the `|` syntax in the template, the `|` character is called the *pipe* character, like so:

```
{{ 1234.56 | }}
{{ 1234.56 | currency : 'USD' }}
```

This would take the number `1234.56` and convert it into a *currency string* for display in the template like `USD1,234.56`.

We can even *chain* pipes together like so:

```
{{ 1234.56 | currency: 'USD' | lowercase }}
```

The above would print out `usd1,234.56`.

Pipes are just like `filters` in Angular 1

In this section you will learn:

- How to use the set of built-in pipes provided by Angular.
- How to create your own custom pipes.

Built-in Pipes

In this lecture we will cover all of the built-in pipes provided by Angular appart from the *async pipe* which we will cover in detail in a later lecture.

Learning Objectives

- Know the different built-in pipes provided by Angular and how to use them.

Pipes provided by Angular

Angular provides the following set of built-in pipes.

CurrencyPipe

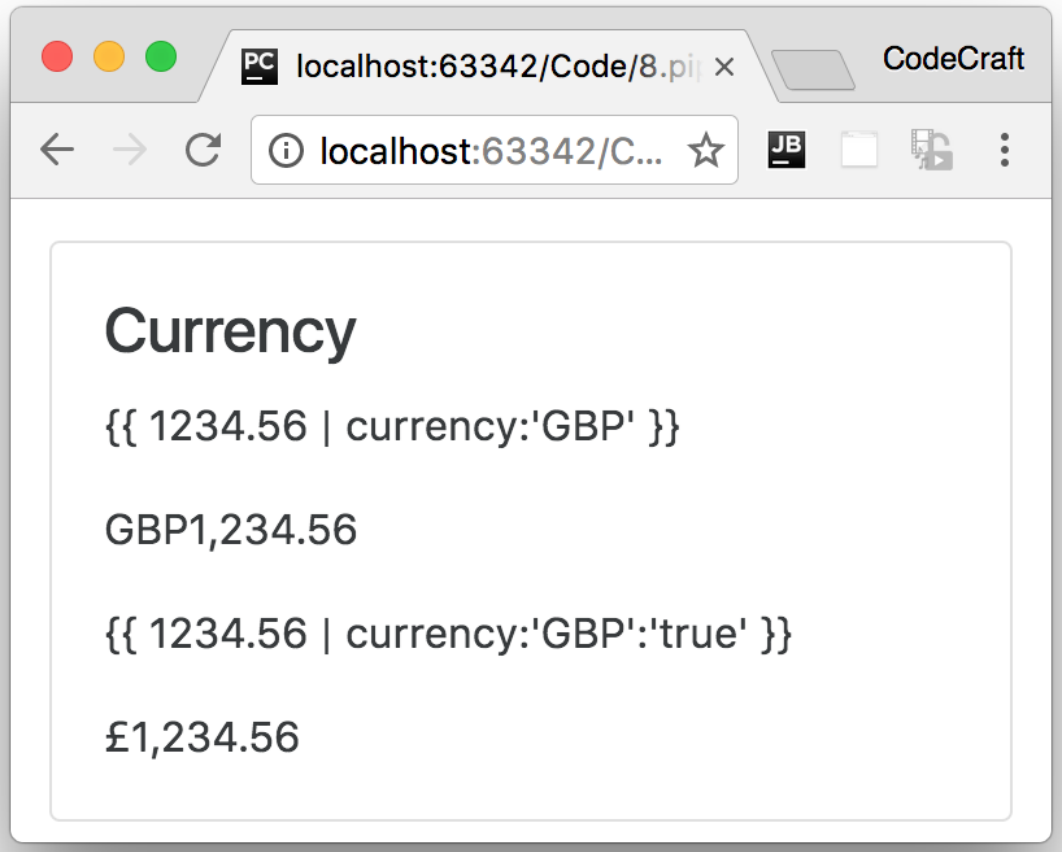
This pipe is used for formatting currencies. Its first argument is an abbreviation of the currency type (e.g. "EUR", "USD", and so on), like so:

```
{{ 1234.56 | currency:'GBP' }}
```

The above prints out `GBP1,234.56`, if instead of the abbreviation of `GBP` we want the currency symbol to be printed out we pass as a second parameter the boolean `true`, like so:

```
{{ 1234.56 | currency:"GBP":true }}
```

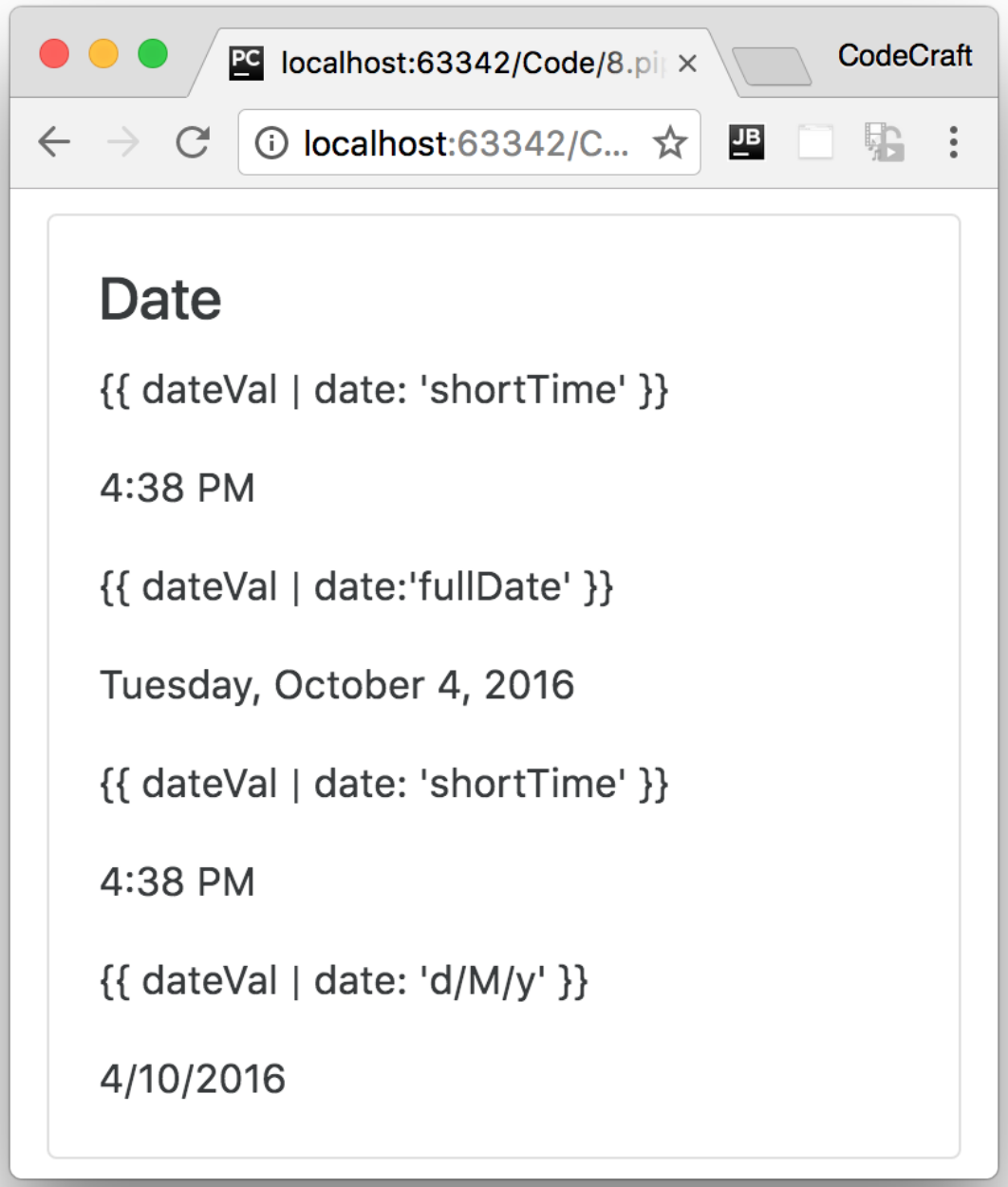
The above prints out `£1,234.56`.




```
<div class="card card-block">
  <h4 class="card-title">Currency</h4>
  <div class="card-text">

    <p ngNonBindable>{{ 1234.56 | currency:'GBP' }}</p>
    <p>{{ 1234.56 | currency:"GBP" }}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'GBP':'true' }}</p>
    <p>{{ 1234.56 | currency:"GBP":true }}</p>
  </div>
</div>
```

DatePipe

This pipe is used for the transformation of dates. The first argument is a format string, like so:


localhost:63342/Code/8.pi
CodeCraft
localhost:63342/C...
Date
{{ dateVal | date: 'shortTime' }}
4:38 PM
{{ dateVal | date:'fullDate' }}
Tuesday, October 4, 2016
{{ dateVal | date: 'shortTime' }}
4:38 PM
{{ dateVal | date: 'd/M/y' }}
4/10/2016

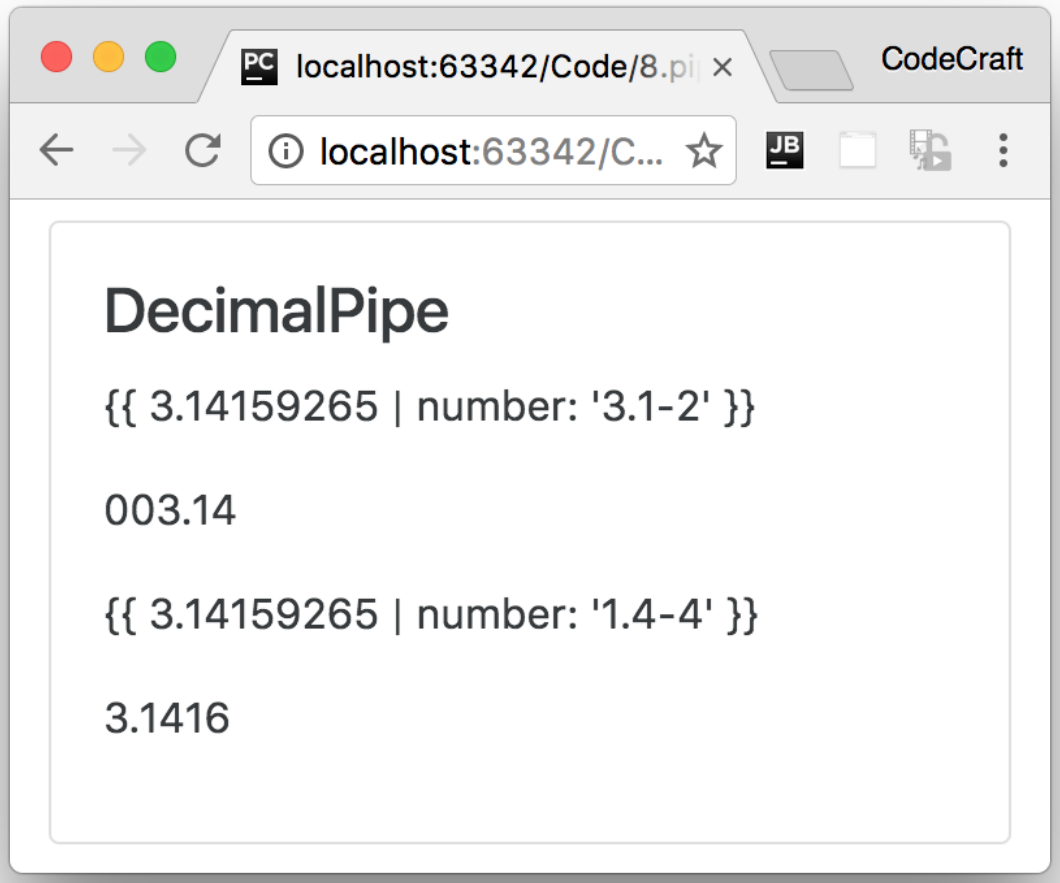
```
<div class="card card-block">
  <h4 class="card-title">Date</h4>
  <div class="card-text">
    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p> ①
    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>

    <p ngNonBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>
```

① `dateVal` is an instance of `new Date()`.

DecimalPipe

This pipe is used for transformation of decimal numbers.

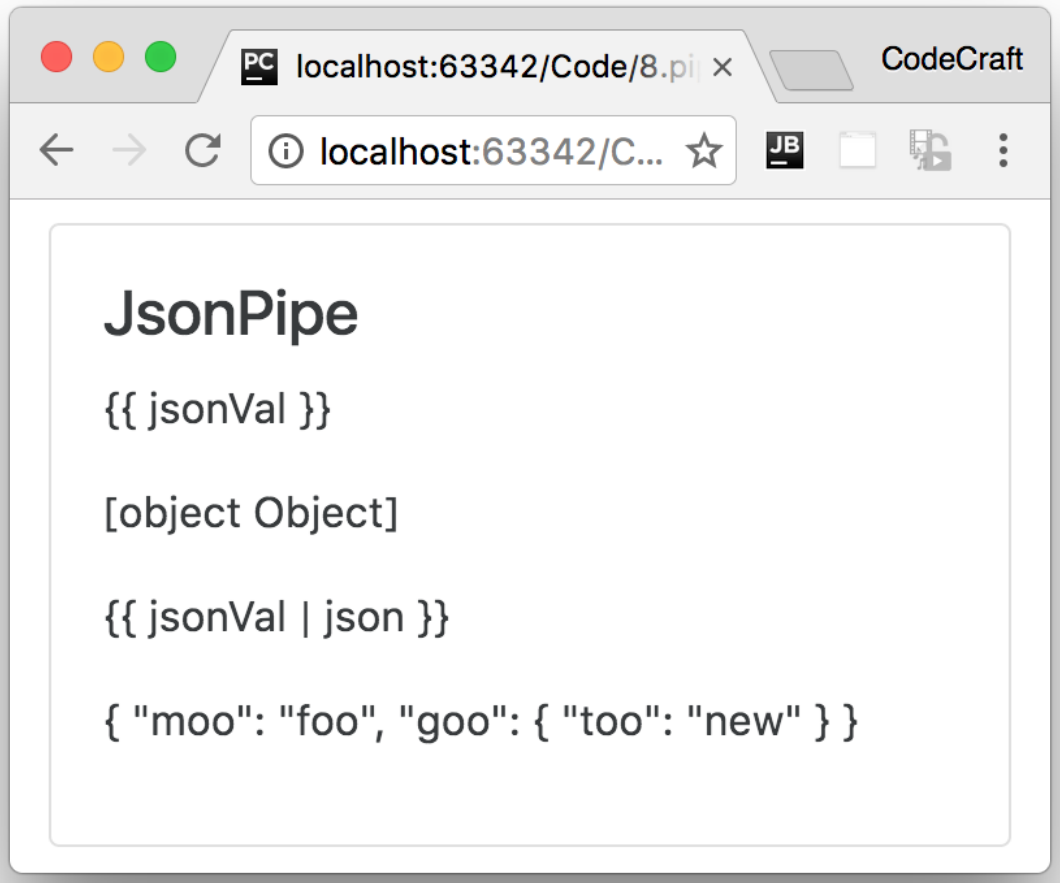
The first argument is a format string of the form "{minIntegerDigits}. {minFractionDigits}-{maxFractionDigits}", like so:




```
<div class="card card-block">
  <div class="card-text">
    <h4 class="card-title">DecimalPipe</h4>
    <p ngNonBindable>{{ 3.14159265 | number: '3.1-2' }}</p>
    <p>{{ 3.14159265 | number: '3.1-2' }}</p>

    <p ngNonBindable>{{ 3.14159265 | number: '1.4-4' }}</p>
    <p>{{ 3.14159265 | number: '1.4-4' }}</p>
  </div>
</div>
```

JsonPipe

This transforms a JavaScript object into a JSON string, like so:

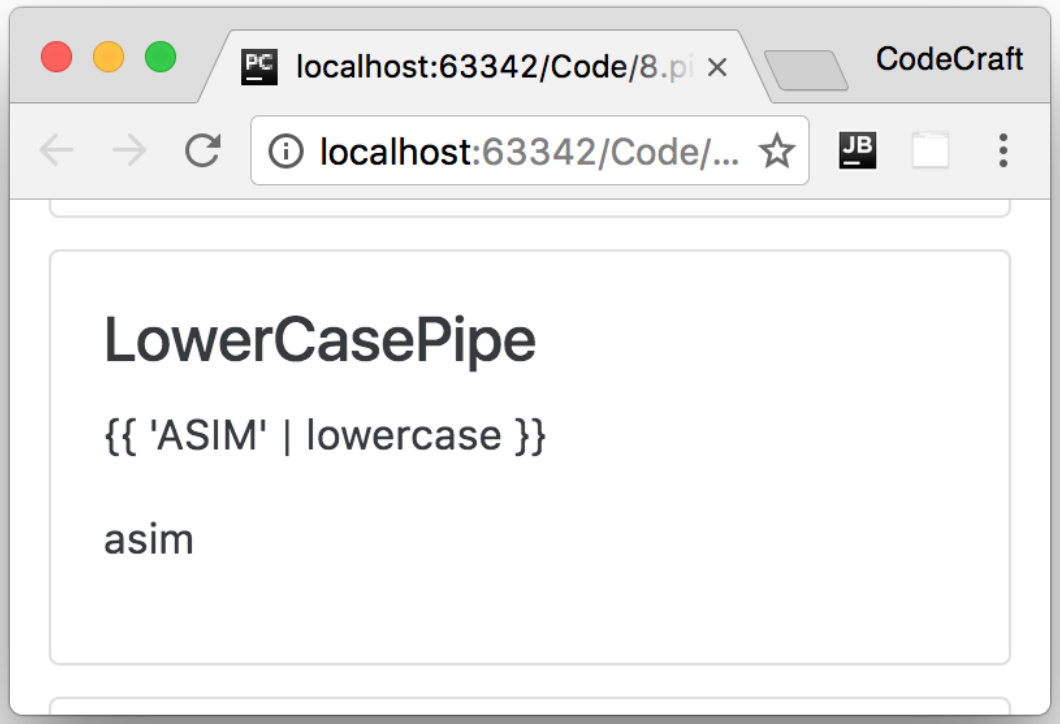



```
<div class="card card-block">
  <h4 class="card-title">JsonPipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ jsonVal }}</p> ①
    <p>{{ jsonVal }}</p>

    <p ngNonBindable>{{ jsonVal | json }}</p>
    <p>{{ jsonVal | json }}</p>
  </div>
</div>
```

① `jsonVal` is an object declared as `{ moo: 'foo', goo: { too: 'new' }}`.

LowerCasePipe

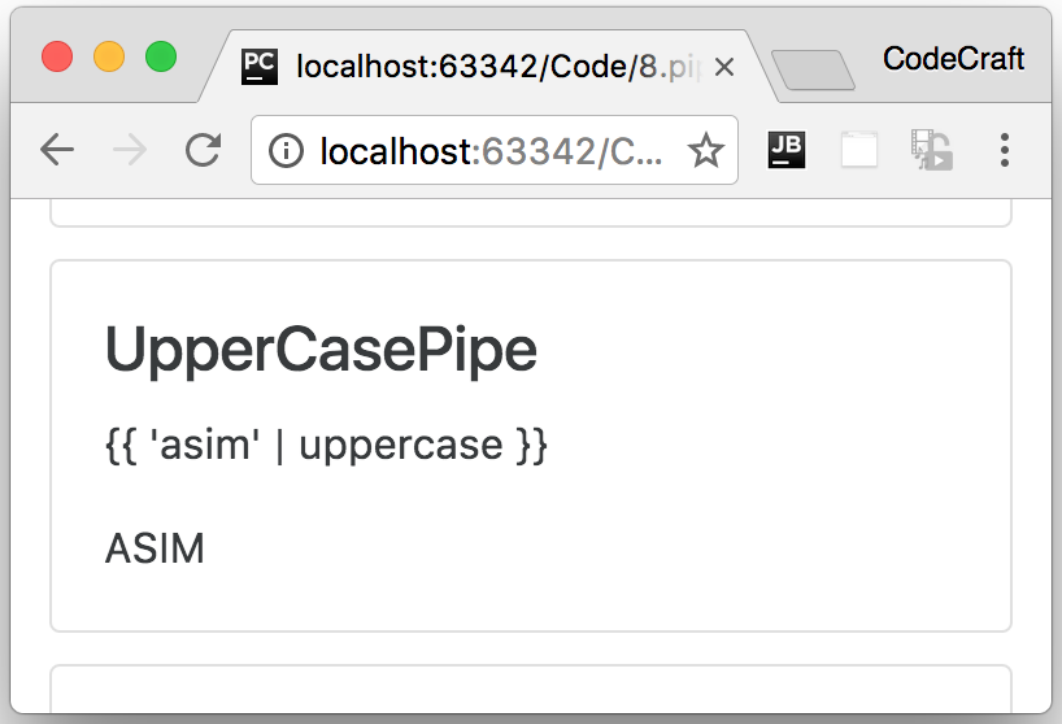
This transforms a string to lowercase, like so:




```
<div class="card card-block">
  <h4 class="card-title">LowerCasePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 'ASIM' | lowercase }}</p>
    <p>{{ 'ASIM' | lowercase }}</p>
  </div>
</div>
```

UpperCasePipe

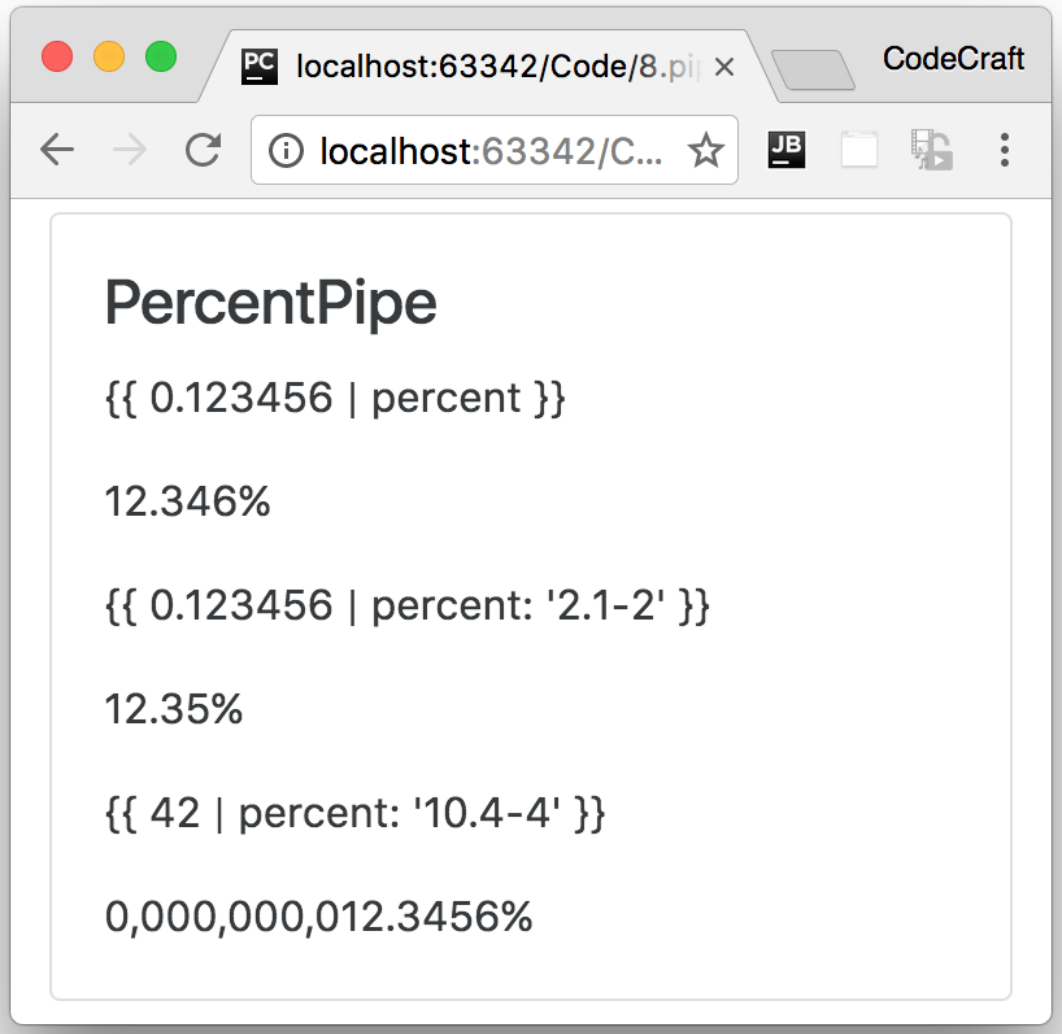
This transforms a string to uppercase, like so:




```
<div class="card card-block">
  <h4 class="card-title">UpperCasePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 'asim' | uppercase }}</p>
    <p>{{ 'asim' | uppercase }}</p>
  </div>
</div>
```

PercentPipe

Formats a number as a percent, like so:



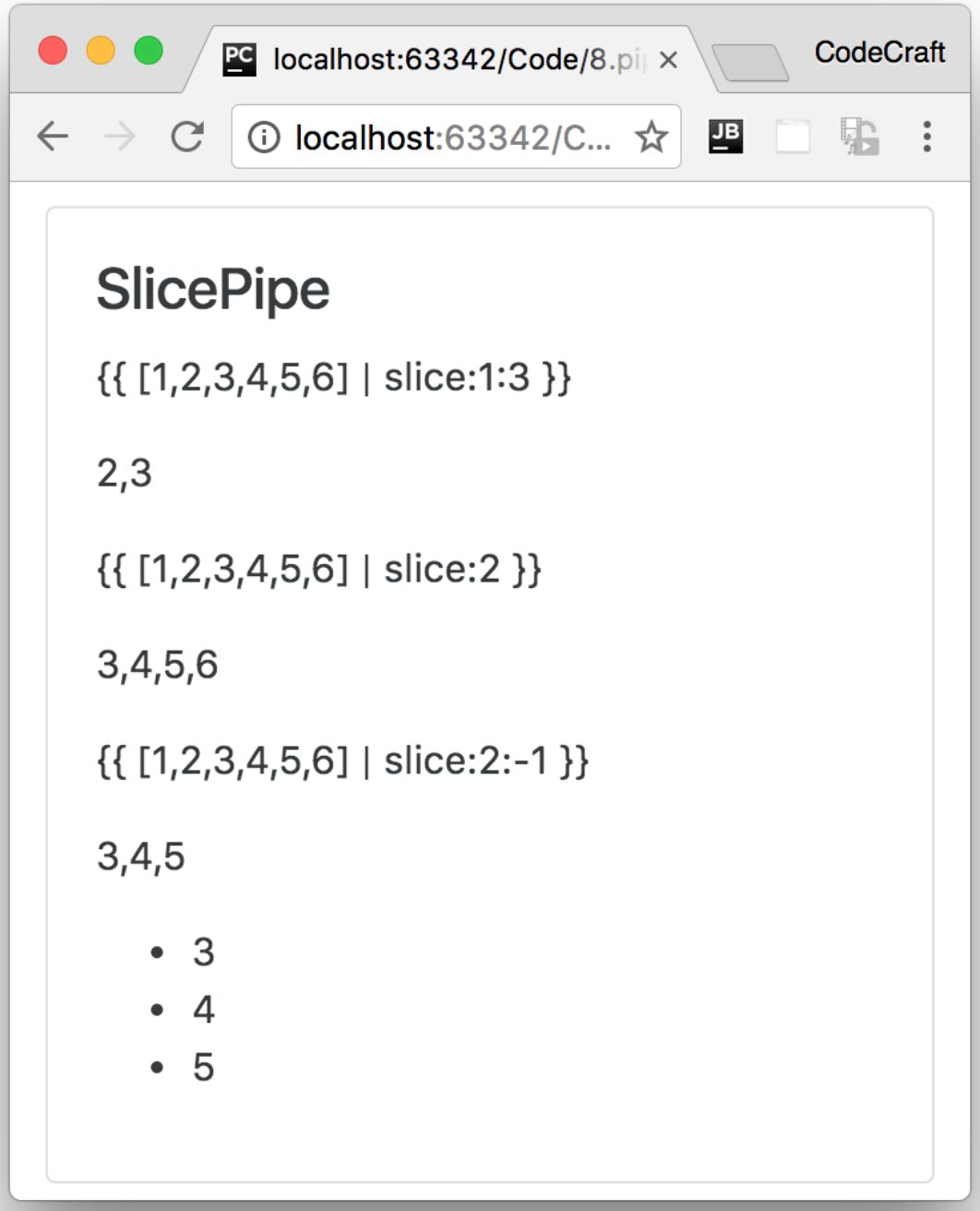

```
<div class="card card-block">
  <h4 class="card-title">PercentPipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 0.123456 | percent }}</p>
    <p>{{ 0.123456 | percent }}</p>

    <p ngNonBindable>{{ 0.123456 | percent: '2.1-2' }}</p> ①
    <p>{{ 0.123456 | percent: '2.1-2' }}</p>

    <p ngNonBindable>{{ 42 | percent: '10.4-4' }}</p>
    <p>{{ 0.123456 | percent : "10.4-4" }}</p>
  </div>
</div>
```

① Percent can be passed a format string similar to the format passed to the `DecimalPipe`.

SlicePipe

This returns a *slice* of an array. The first argument is the start index of the slice and the second argument is the end index.

If either indexes are not provided it assumes the start or the end of the array and we can use negative indexes to indicate an offset from the end, like so:



```
<div class="card card-block">
  <h4 class="card-title">SlicePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:1:3 }}</p> ①
    <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>

    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2 }}</p> ②
    <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>

    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p> ③
    <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>

    <pre ngNonBindable>
&lt;ul&gt;
  &lt;li *ngFor=&quot;let v of [1,2,3,4,5,6] | slice:2:-1&quot;&gt;
    {{v}}
  &lt;/li&gt;
&lt;/ul&gt;
    </pre>

    <ul>
      <li *ngFor="let v of [1,2,3,4,5,6] | slice:2:-1"> ④
        {{v}}
      </li>
    </ul>
  </div>
</div>
```

① `slice:1:3` means return the items from the 1st to the 3rd index inclusive (indexes start at 0).

② `slice:2` means return the items from the 2nd index to the end of the array.

③ `slice:2:-1` means return the items from the 2nd index to one from the end of the array.

④ We can use slice inside for loops to only loop over a subset of the array items.

AsyncPipe

This pipe accepts an observable or a promise and lets us render the output of an observable or promise without having to call `then` or `subscribe`.

We are going to take a much deeper look at this pipe at the end of this section.

Summary

Pipes enables you to easily transform data for display purposes in templates.

Angular comes with a very useful set of pre-built pipes to handle most of the common transformations.

One of the more complex pipes to understand in Angular is the async pipe that's what we'll cover

next.

Listing

http://plnkr.co/edit/UG4SwlJ0DQEGjkhbbQz9?p=preview

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

@Component({
  selector: 'pipe-builtins',
  template: `<div class="card card-block">
  <h4 class="card-title">Currency</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 1234.56 | currency:'CAD' }}</p>
    <p>{{ 1234.56 | currency:"CAD" }}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'code' }}</p>
    <p>{{ 1234.56 | currency:'CAD':'code'}}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'symbol' }}</p>
    <p>{{ 1234.56 | currency:'CAD':'symbol'}}</p>

    <p ngNonBindable>{{ 1234.56 | currency:'CAD':'symbol-narrow' }}</p>
    <p>{{ 1234.56 | currency:'CAD':'symbol-narrow'}}</p>
  </div>
</div>

<div class="card card-block">
  <h4 class="card-title">Date</h4>
  <div class="card-text">
    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p>
    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>

    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p>
    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>

<div class="card card-block">
  <div class="card-text">
    <h4 class="card-title">DecimalPipe</h4>
```

```
    <p ngNonBindable>{{ 3.14159265 | number: '3.1-2' }}</p>
    <p>{{ 3.14159265 | number: '3.1-2' }}</p>

    <p ngNonBindable>{{ 3.14159265 | number: '1.4-4' }}</p>
    <p>{{ 3.14159265 | number: '1.4-4' }}</p>
  </div>
</div>

<div class="card card-block">
  <h4 class="card-title">JsonPipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ jsonVal }}</p>
    <p>{{ jsonVal }}</p>

    <p ngNonBindable>{{ jsonVal | json }}</p>
    <p>{{ jsonVal | json }}</p>
  </div>
</div>


<div class="card card-block">
  <h4 class="card-title">LowerCasePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 'ASIM' | lowercase }}</p>
    <p>{{ 'ASIM' | lowercase }}</p>
  </div>
</div>

<div class="card card-block">
  <h4 class="card-title">UpperCasePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 'asim' | uppercase }}</p>
    <p>{{ 'asim' | uppercase }}</p>
  </div>
</div>

<div class="card card-block">
  <h4 class="card-title">PercentPipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 0.123456 | percent }}</p>
    <p>{{ 0.123456 | percent }}</p>

    <p ngNonBindable>{{ 0.123456 | percent: '2.1-2' }}</p>
    <p>{{ 0.123456 | percent: '2.1-2' }}</p>

    <p ngNonBindable>{{ 42 | percent: '10.4-4' }}</p>
    <p>{{ 0.123456 | percent : "10.4-4" }}</p>
  </div>
</div>

<div class="card card-block">
```

```
  <h4 class="card-title">SlicePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>
    <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>

    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2 }}</p>
    <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>

    <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>
    <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>

    <pre ngNonBindable>
&lt;ul&gt;
  &lt;li *ngFor=&quot;let v of [1,2,3,4,5,6] | slice:2:-1&quot;&gt;
    {{v}}
  &lt;/li&gt;
&lt;/ul&gt;
    </pre>

    <ul>
      <li *ngFor="let v of [1,2,3,4,5,6] | slice:2:-1">
        {{v}}
      </li>
    </ul>
  </div>
</div>

 `
})
class PipeBuiltinsComponent {
  private dateVal: Date = new Date();
  private jsonVal: Object = {moo: 'foo', goo: {too: 'new'}};

}

@Component({
  selector: 'app',
  template: `
<pipe-builtins></pipe-builtins>
 `
})
class AppComponent {
}


@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent,
    PipeBuiltinsComponent
  ],
  bootstrap: [AppComponent],
```

```
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Async Pipe

Learning Objectives

- When to use the *async* pipe.
- How to use async pipe with Promises and also Observables.

Overview

Normally to render the result of a promise or an observable we have to:

1. Wait for a *callback*.
2. Store the result of the callback is a *variable*.
3. *Bind* to that variable in the template.

With `AsyncPipe` we can use promises and observables directly in our template, without having to store the result on an intermediate property or variable.

`AsyncPipe` accepts as argument an observable or a promise, calls `subcribe` or attaches a `then` handler, then waits for the asynchronous result before passing it through to the caller.

AsyncPipe with promises

Lets first create a component with a promise as a property.

```
@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>
  <p class="card-text" ngNonBindable>{{ promiseData }}</p> ①
  <p class="card-text">{{ promiseData }}</p> ②
 </div>
  `
})
class AsyncPipeComponent {
  promiseData: string;
  constructor() {
        this.getPromise().then(v => this.promiseData = v); ③
  }

  getPromise() {  ④
     return new Promise((resolve, reject) => {
       setTimeout(() => resolve("Promise complete!"), 3000);
     });
  }
}
```

① We use `ngNonBindable` so we can render out `{{ promiseData }}` as is without trying to bind to to the property `promiseData`

② We bind to the property `promiseData`

③ When the promise resolves we store the data onto the `promiseData` property

④ `getPromise` returns a promise which 3 seconds later resolves with the value `"Promise complete!"`

In the constructor we wait for the promise to resolve and store the result on a property called `promiseData` on our component and then bind to that property in the template.

To save time we can use the `async` pipe in the template and bind to the promise *directly*, like so:

[source,javascript]cript.ts

```
@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>
  <p class="card-text" ngNonBindable>{{ promise }}</p>
  <p class="card-text">{{ promise | async }}</p> ①
 </div>
  `
})
class AsyncPipeComponent {
  promise: Promise<string>;
  constructor() {
        this.promise = this.getPromise(); ②
  }

  getPromise() {
     return new Promise((resolve, reject) => {
       setTimeout(() => resolve("Promise complete!"), 3000);
     });
  }
}
```

① We pipe the output of our `promise` to the `async` pipe.

② The property `promise` is the actual unresolved *promise* that gets returned from `getPromise` without `then` being called on it.

The above results in the same behaviour as before, we just saved ourselves from writing a `then` callback and storing intermediate data on the component.

AsyncPipe with observables

To demonstrate how this works with *observables* we first need to setup our component with a simple *observable*, like so:

```
import { Observable } from 'rxjs/Rx';
.
.
.
@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>
  <p class="card-text" ngNonBindable>{{ observableData }}
  <p class="card-text">{{ observableData }}</p> ①
 </div>
`
})
class AsyncPipeComponent {
  observableData: number;
  subscription: Object = null;

  constructor() {
    this.subscribeObservable();
  }

  getObservable() { ②
    return Observable
        .interval(1000)
        .take(10)
        .map((v) => v * v);
  }

  subscribeObservable() { ③
    this.subscription = this.getObservable()
        .subscribe( v => this.observableData = v);
  }

  ngOnDestroy() { ④
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }

}
```

① We render the value of `observableData` in our template.

② We create an observable which publishes out a number which increments by one every second then squares that number.

③ We subscribe to the output of this observable chain and store the number on the property `observableData`. We also store a reference to the subscription so we can unsubscribe to it later.

④ On destruction of the component we unsubscribe from the observable to avoid memory leaks.

We should also be destroying the subscription when the component is destroyed. Otherwise we will start leaking data as the old observable, which isn't used any more, will still be producing results.

Again by using `AsyncPipe` we don't need to perform the `subscribe` and store any intermediate data on our component, like so:

```
@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>
  <p class="card-text" ngNonBindable>{{ observable | async }}
  <p class="card-text">{{ observable | async }}</p> ①
 </div>
`
})
class AsyncPipeComponent {
  observable: Observable<number>;

  constructor() {
    this.observable = this.getObservable();
  }

  getObservable() {
    return Observable
      .interval(1000)
      .take(10)
      .map((v) => v*v)
  }
}
```

① We pipe our `observable` directly to the `async` pipe, it performs a subscription for us and then returns whatever gets passed to it.

By using `AsyncPipe` we: 1. Don't need to call `subscribe` on our observable and store the intermediate data on our component. 2. Don't need to remember to `unsubscribe` from the observable when the component is destroyed.

Summary

`AsyncPipe` is a convenience function which makes rendering data from observables and promises much easier.

For promises it automatically adds a `then` callback and renders the response.

For Observables it automatically `subscribes` to the observable, renders the output and then also `unsubscribes` when the component is destroyed so we don't need to handle the clean up logic ourselves.

That's it for the built-in pipes, next up we will look at creating out own custom pipes.

Listing

http://plnkr.co/edit/gHIalfn10CfocBwCE6UG?p=preview

script.ts

```
import {NgModule, Component, OnDestroy} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import { Observable } from 'rxjs/Rx';


@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>

  <p class="card-text" ngNonBindable>{{ promise | async }}  </p>
  <p class="card-text">{{ promise | async }}  </p>

  <p class="card-text" ngNonBindable>{{ observable | async }}  </p>
  <p class="card-text">{{ observable | async }}</p>

  <p class="card-text" ngNonBindable>{{ observableData }}  </p>
  <p class="card-text">{{ observableData }}</p>
 </div>
  `
})
class AsyncPipeComponent implements OnDestroy {
  promise: Promise<string>;
  observable: Observable<number>;
  subscription: Object = null;
  observableData: number;

  constructor() {
    this.promise = this.getPromise();
    this.observable = this.getObservable();
    this.subscribeObservable();

  }

  getObservable() {
    return Observable
        .interval(1000)
        .take(10)
        .map((v) => v * v);
  }
```

```
  // AsyncPipe subscribes to the observable automatically
  subscribeObservable() {
    this.subscription = this.getObservable()
        .subscribe((v) => this.observableData = v);
  }

  getPromise() {
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve("Promise complete!"), 3000);
    });
  }

  // AsyncPipe unsubscribes from the observable automatically
  ngOnDestroy() {
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }
}


@Component({
  selector: 'app',
  template: `
  <async-pipe></async-pipe>
 `
})
class AppComponent {
  imageUrl: string = "";

}


@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent,
    AsyncPipeComponent
  ],
  bootstrap: [AppComponent],
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Custom Pipes

Learning Objectives

- Know how to use the `@Pipe` decorator to create pipes.
- Know how to pass in parameters to custom pipes.

Pipe decorator

One pipe I really find useful when building web applications is a *default* pipe, which I use for things like avatar images.

I use this pipe in an image field, like so:

```
<img [src]="imageUrl | default:'<default-image-url>'"/>
```

The pipe is called `default` and we pass to it a default image to use if the `imageUrl` variable is blank.

To create a pipe we use the `@Pipe` decorator and annotate a class like so:

```
import { Pipe } from '@angular/core';
.
.
.
@Pipe({
  name:"default"
})
class DefaultPipe { }
```

The name parameter for the `Pipe` decorator is how the pipe will be called in templates.

Transform function

The actual logic for the pipe is put in a function called `transform` on the class, like so:

```
class DefaultPipe {
  transform(value: string, fallback: string): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
     return image;
  }
}
```

The first argument to the transform function is the *value* that is passed *into* the pipe, i.e. the thing that goes *before* the `|` in the expression.

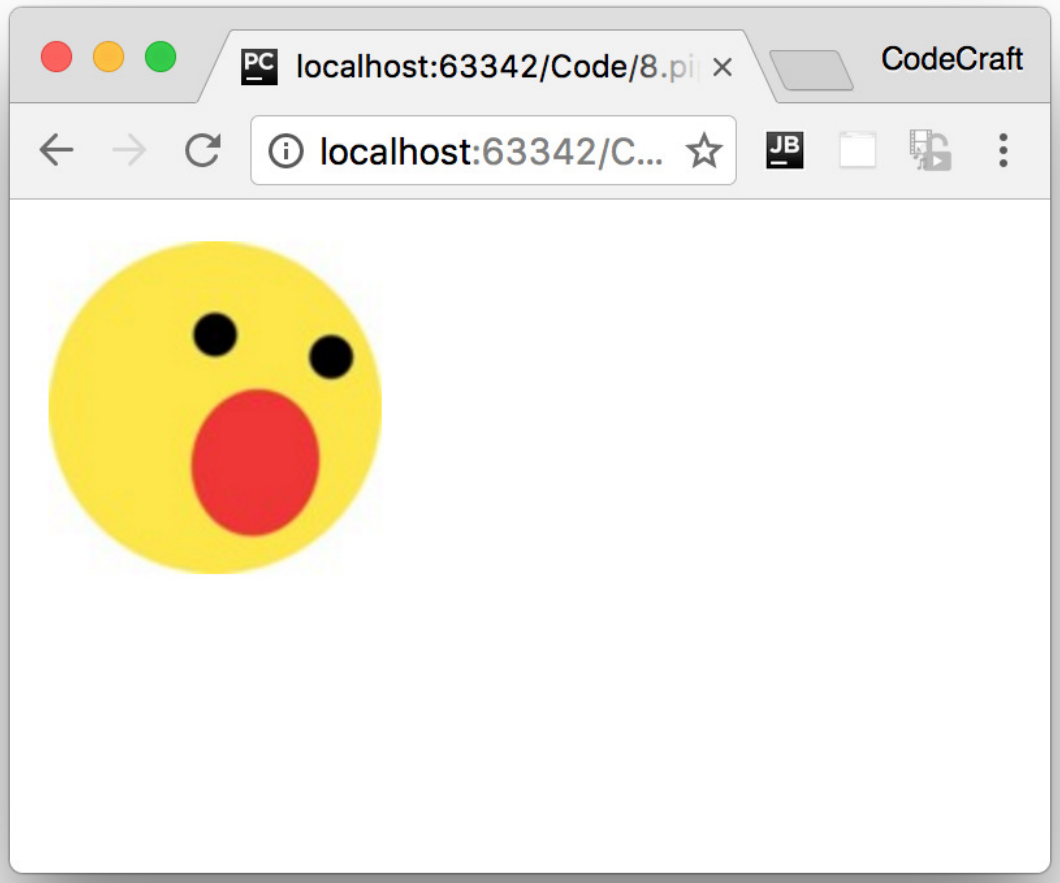
The second parameter to the transform function is the first param we pass into our pipe, i.e. the thing that goes after the `:` in the expression.

Specifically with this example:

```
@Component({
  selector: 'app',
  template: `
  <img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'"/>
 `
})
class AppComponent {
  imageUrl: string = "";
}
```

- `value` gets passed `imageUrl` which is blank.
- `fallback` gets passed 'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'

When we run the above, since `imageUrl` is blank the `default` pipe uses the default image which is passed in, like so:



Multiple parameters

Finally we want to support an *optional* third param to our pipe called `forceHttps`, if the image selected doesn't use *https* the pipe will convert the url to one that does use *https*.

To support additional parameters in pipes we just add more parameters to our *transform* function.

Because this one is optional and we are using TypeScript we can define the new param and also give it a default value of `false`.

```
class DefaultPipe {
  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }
    return image;
  }
}
```

And to use this optional param we just extend the pipe syntax in our template with another `:`, like so:-

```
<img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg':true"/> ①
```

① Notice the last `:true` at the end of the pipe expression.

Now we force the image url to use the https protocol.

Summary

Creating a pipe is very simple in Angular. We just decorate a class with the `@Pipe` decorator, provide a name and a transform function and we are done.

Listing

script.ts

```
import {NgModule, Component, Pipe} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {Observable} from 'rxjs/Rx';


@Pipe({
  name: "default"
})
class DefaultPipe {
  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
```

```
    if (value) {
      image = value;
    } else {
      image = fallback;
    }

    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }

    return image;
  }
}

@Component({
  selector: 'app',
  template: `
  <img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg':true"/>
 `
})
class AppComponent {
  imageUrl: string = "";
}


@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent,
    DefaultPipe
  ],
  bootstrap: [AppComponent],
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

Pipes are a way of having a different *visual representation* for the same piece of data without storing unnecessary intermediate data on the component.

We could solve the same problems without using pipes. We can transform the data ourselves on the component into different property, in all the different permutations they are required in all the views they are consumed in.

But that's wasteful and a fertile feeding ground for bugs.

Instead we use pipes which are used in templates and transform the data passed to it on demand in the format the consumer of the data wants without needing to store any intermediate values on our component.

Activity

Create a custom *Pipe* called `CleanPipe` for use in the Joke application which replaces instances of a *bad* word with `$%#@!` instead.

We want to pass into the pipe the list of bad words which we want replace, like so:

```
<p>{{data.punchline | clean:'boo,damn,hell'}}</p>
```

Steps

Fork this plunker:

http://plnkr.co/edit/nlRZRJnB8jEScijgZ9J6?p=preview

Flesh out the `CleanPipe` class to implement the functionality.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/CWPCpzzIjkPdYet6prLj?p=preview

Forms

Overview

In this chapter we will cover both the *template driven* and *model driven* approach to creating forms in Angular.

We are going to create a signup form with a first name, last name, email, password and language select box.

We'll be using the twitter bootstrap UI framework so the HTML markup and classes will match the layout and styles needed for bootstrap.

We will add *visual feedback* to the users so they know if the individual form fields contain valid values or not. If they are invalid we'll also show the user helpful validation error messages so they can fix their inputs.

Although the template driven approach is simpler, behind the scenes it's actually the model driven approach with the models automatically creating for you.

So we'll first explain the model driven approach, then the template driven approach and finally we'll show a method of implementing a *reactive model driven form* with RxJS.

Model Driven Forms

Learning Objectives

- How to create a HTML form with a *dynamic* select box control.
- How to define a form *model* on your component and link it to existing HTML form controls.

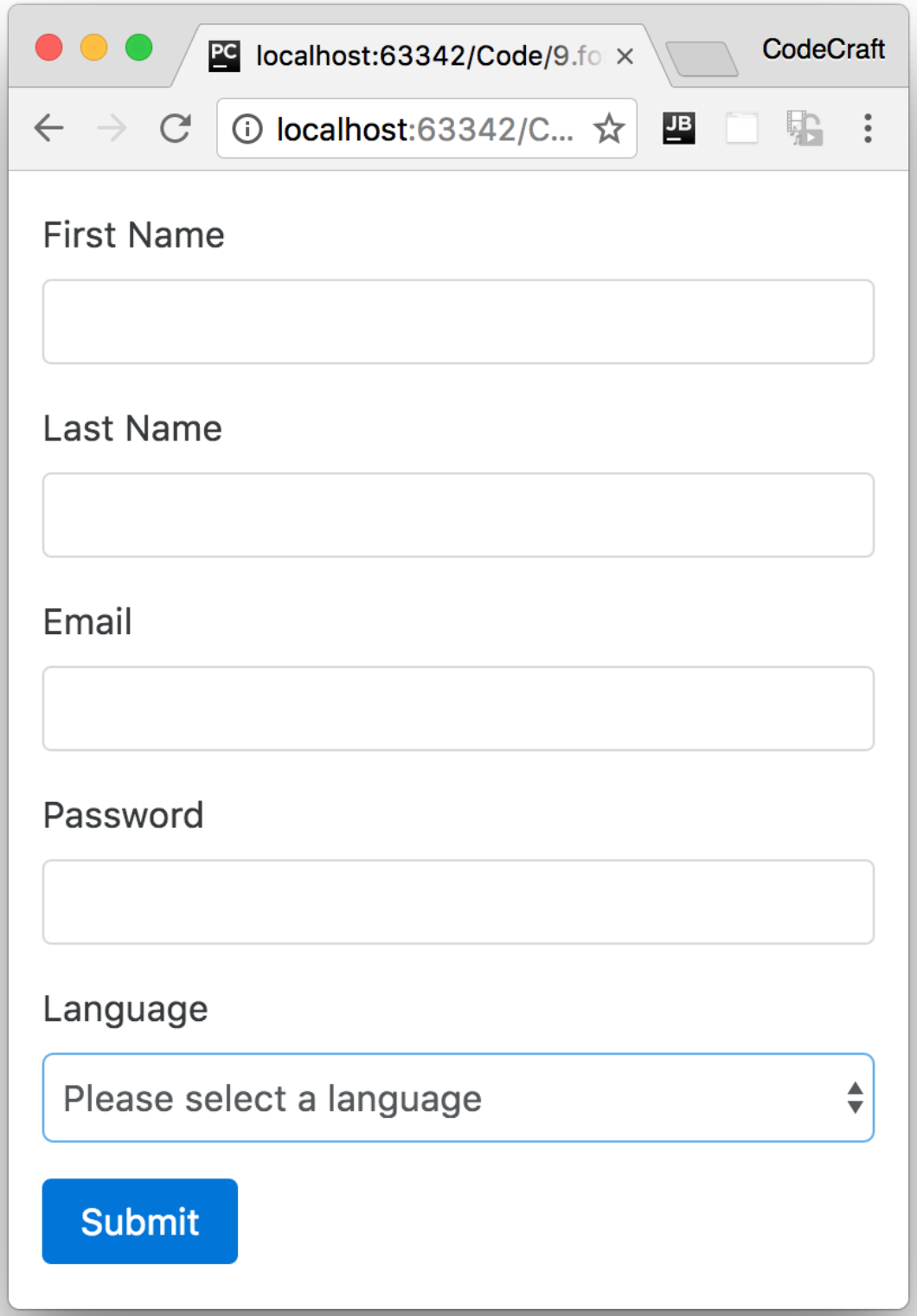
Form setup

Whether we are template driven or model driven we need some basic form HTML to begin with.

In model driven forms contrary to what you might think the HTML for our form isn't automatically created for us.

We *still* need to write the HTML that represents our form and then explicitly link the HTML form elements to code on our component.

We create a simple bootstrap form with a first name, last name, email, password and language select box controls.


localhost:63342/Code/9.fo
CodeCraft
localhost:63342/C...
First Name
Last Name
Email
Password
Language
Please select a language
Submit

```
<form novalidate>
  <fieldset>
    <div class="form-group">
      <label>First Name</label>
      <input type="text"
             class="form-control">
    </div>

    <div class="form-group">
      <label>Last Name</label>
      <input type="text"
             class="form-control">
    </div>
  </fieldset>

  <div class="form-group">
    <label>Email</label>
    <input type="email"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Password</label>
    <input type="password"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Language</label>
    <select class="form-control">
      <option value="">Please select a language</option>
    </select>
  </div>
</form>
```

We've added the `novalidate` attribute to the `form` element, by default browsers perform their own validation and show their own error popups. Since we want to handle the form validation ourselves we can switch off this behaviour by adding novalidate to the form element.

We are using the markup and styles from the twitter bootstrap UI framework to structure our form.

Dynamic select controls

We want the select box to have a list of languages for the user to select.

We could hard code the languages in the HTML with `<option>` tags but we want to make the list *dynamic* so we can easily add more languages later on. so we:

1. Add an array of languages to our component.

```
langs: string[] = [
  'English',
  'French',
  'German',
]
```

2. Use an `NgFor` loop to render these as options in the template.

```
<select class="form-control">
  <option value="">Please select a language</option>
  <option *ngFor="let lang of langs"
          [value]="lang"> ①
          {{lang}} ②
  </option>
</select>
```

1. The options *value*
2. The options *label*

An option has a *label* and a *value*. The label is the text the user sees in the select box and the *value* is the data that's stored for that label.

If we ask a select box what option has been selected it returns us the *value*, not the *label*.

To set the value of our select box we just bind to the input property of our option using the `[value]` syntax.

Form model

We represent the form as a *model* composed of instances of `FormGroups` and `FormControls`.

Lets create the model for our form on our component, like so:

```
import { FormGroup, FormControl } from '@angular/forms';
.
.
.
class ModelFormComponent implements OnInit {
  myform: FormGroup; ①

  ngOnInit() {
    myform = new FormGroup({
        name: new FormGroup({ ②
            firstName: new FormControl(), ③
            lastName: new FormControl(),
        }),
        email: new FormControl(),
        password: new FormControl(),
        language: new FormControl()
    });
  }
}
```

① `myform` is an instance of `FormGroup` and represents the form itself.

② `FormGroups` can nest inside other `FormGroups`.

③ We create a `FormControl` for each template form control.

The `myform` property is an instance of a `FormGroup` class and this represents our form itself.

Each *form control* in the template is represented by an instance of `FormControl`. This encapsulates the state of the control, such as if it's valid or invalid and even it's current value.

These instances of `FormControls` nest inside our top level `myform: FormGroup`, but what's interesting is that we can *nest* `FormGroups` inside other `FormGroups`.

In our model we've grouped the `firstName` and `lastName` controls under a `FormGroup` called `name` which itself is nested under our top level `myform: FormGroup`.

Like the `FormControl` instance, `FormGroup` instances encapsulates the state of all of it's inner controls, for example an instance of a `FormGroup` is valid only if *all* of it's inner controls are also valid.

Linking the form model to the form template

We have the HTML template for our form and the form model on our component, next up we need to link the two together.

We do this using a number of directives which are found in the `ReactiveFormsModule`, so lets import that and add it to the imports on our `NgModule`.

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

formGroup

Firstly we bind the `<form>` element to our top level `myform` property using the `formGroup` directive, like so:

```
<form [formGroup]="myform"> ... </form>
```

Now we've linked the `myform` model to the form template we have access to our `myform` model *in* our template.

The value property of the `myform` model returns the values of *all* of the controls as an object. We can use that with the `json` pipe to output some useful debug information about our form, like so:

```
<pre>{{myform.value | json}}</pre>
```

Running our application now prints out the below in the debug `pre` tag:

```
{
  "name": {
    "firstName": null,
    "lastName": null
  },
  "email": null,
  "password": null,
  "language": null
}
```

Initially this seems quite exciting but as we enter values into each of the input fields in our form we would see that the model *isn't* getting updated, the values remain null.

That's because although we've linked the `form` element to the `myform` model this doesn't automatically link each form control in the model with each form control in the template, we need to do this explicitly with the `formControlName` and `formGroupName` directives.

formGroupName & formControlName

We use the `formControlName` directive to map each form control in the template with a named form control in the model, like so:

```
<div class="form-group">
  <label>Email</label>
  <input type="email"
         class="form-control"
         formControlName="email" ①
         required>
</div>
```

① This looks for a model form control called email in the top level of our `myform` model and links the element to that.

We can also associate a group of template form controls to an instance of a form group on our model using `formGroupName` directive.

Since our `firstName` and `lastName` controls are grouped under a form group called `name` we'll do just that.

The only caveat is that in our template the controls we want to group must be surrounded by another element, we've surrounded our controls with a fieldset element but it doesn't need to be called fieldset, could for example be a div.

We then associate the `fieldset` element with the form group called `name` in our model like so:

```
<fieldset formGroupName="name"> ... </fieldset>
```

Then inside our fieldset element we again use the `formControlName` directive to map individual form controls in the template to form controls under the form group `name` in our model.

In the end the template should look like this:

```
<form [formGroup]="myform"> ①

    <fieldset formGroupName="name">  ②
        <div class="form-group">
            <label>First Name</label>
            <input type="text"
                   class="form-control"
                   formControlName="firstName" ③
                   required>
        </div>

        <div class="form-group">
            <label>Last Name</label>
            <input type="text"
                   class="form-control"
                   formControlName="lastName" ③
                   required>
        </div>
    </fieldset>


    <div class="form-group">
        <label>Email</label>
        <input type="email"
               class="form-control"
               formControlName="email" ④
               required>
    </div>

    <div class="form-group">
        <label>Password</label>
        <input type="password"
               class="form-control"
               formControlName="password" ④
               required>
    </div>

    <div class="form-group">
        <label>Language</label>
        <select class="form-control"
                formControlName="language" ④
            <option value="">Please select a language</option>
            <option *ngFor="let lang of langs"
                    [value]="lang">{{lang}}
            </option>
        </select>
    </div>

    <pre>{{myform.value | json}}</pre>
</form>
```

① Use `formGroup` to bind the form to an instance of `FormGroup` on our component.

② Use `formGroupName` to map to a *child* `FormGroup` of `myform`.

③ Use `formControlName` to bind to an instance of a `FormControl`, since these form controls are under a `formGroupName` of `name`, Angular will try and find the control in under `myform['name']`.

④ Use `formControlName` to bind to an instance of a `FormControl` directly under `myform`.

Now each form control in the template is mapped to form controls in our model and so as we type into the input elements `myform.value` updates and our debug section at the bottom prints out the current value of the form.

Summary

In this lecture we created a simple HTML form. We created a form *model* on our component using the `FormGroup` and `FormControl` classes.

Then by using directives such as `formGroup`, `formControlName` and `formGroupName` we linked our HTML form to our form *model*.

In the next lecture we will learn how to add validators to our forms to give visual feedback when the data entered is invalid.

Listing

http://plnkr.co/edit/MyHYNKJiB5ruiH1AOauL?p=preview

script.ts

```
import {
    NgModule,
    Component,
    Pipe,
    OnInit
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormsModule,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';


@Component({
  selector: 'model-form',
  template: `<!--suppress ALL -->
```

```
<form novalidate
      [formGroup]="myform">

  <fieldset formGroupName="name">
    <div class="form-group">
      <label>First Name</label>
      <input type="text"
             class="form-control"
             formControlName="firstName">
    </div>

    <div class="form-group">
      <label>Last Name</label>
      <input type="text"
             class="form-control"
             formControlName="lastName">
    </div>
  </fieldset>


  <div class="form-group">
    <label>Email</label>
    <input type="email"
           class="form-control"
           formControlName="email">
  </div>

  <div class="form-group">
    <label>Password</label>
    <input type="password"
           class="form-control"
           formControlName="password">
  </div>

  <div class="form-group">
    <label>Language</label>
    <select class="form-control"
            formControlName="language">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">{{lang}}
      </option>
    </select>
  </div>

  <pre>{{myform.value | json}}</pre>
</form>
`
})
class ModelFormComponent implements OnInit {
  langs: string[] = [
```

```
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;


  ngOnInit() {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: new FormControl('', Validators.required),
        lastName: new FormControl('', Validators.required),
      }),
      email: new FormControl('', [
        Validators.required,
        Validators.pattern("[^ @]*@[^ @]*")
      ]),
      password: new FormControl('', [
        Validators.required,
        Validators.minLength(8)
      ]),
      language: new FormControl()
    });
  }
}


@Component({
  selector: 'app',
  template: `<model-form></model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule],
  declarations: [
    AppComponent,
    ModelFormComponent
  ],
  bootstrap: [
    AppComponent
  ],
})
class AppModule {
}
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Model Driven Form Validation

Learning Objectives

- How to add validation checks to our form via the form *model*.
- How to *style* our form in order to give visual feedback to the user so they know when the fields don't pass the validation checks.
- How to add validation error messages to the form to give hints to the user about why the field isn't passing a validation check.

Validators

Carrying on from the model driven form we started in the previous lecture.

Our form is valid *all* the time, regardless of what input the user types into the controls.

Validators are rules which an input control has to follow. If the input doesn't match the rule then the control is said to be invalid.

Since it's a signup form most of the fields should be *required* and I would want to specify some more complex validators on the password field to make sure the user is entering a good strong password.

We can apply validators either by adding attributes to the template or by defining them on our `FormControls` in our model.

To stick to the theme of being *model driven* we are going to add *validators* to the form model directly.

Angular comes with a small set of pre-built validators to match the ones we can define via standard HTML5 attributes, namely `required`, `minlegth`, `maxlength` and `pattern` which we can access from the `Validators` module.

The first parameter of a `FormControl` constructor is the initial value of the control, we'll leave that as empty string. The second parameter contains either a single validator if we only want to apply one, or a list of validators if we want to apply multiple validators to a single control.

Our model then looks something like this:

```
import { FormGroup, FormControl, Validators } from '@angular/forms';
.
.
.
class ModelFormComponent implements OnInit {
  myform: FormGroup;

  ngOnInit() {
    myform = new FormGroup({
        name: new FormGroup({
            firstName: new FormControl('', Validators.required), ①
            lastName: new FormControl('', Validators.required),
        }),
        email: new FormControl('', [ ②
            Validators.required,
            Validators.pattern("[^ @]*@[^ @]*") ③
        ]),
        password: new FormControl('', [
            Validators.minLength(8), ④
            Validators.required
        ]),
        language: new FormControl() ⑤
    });
  }
}
```

① We add a single `required` validator to mark this control as required.

② We can also provide an array of validators.

③ We specify a pattern validator which checks whether the email contains a `@` character.

④ The `minlength` validator checks to see if the password is a minimum of 8 characters long.

⑤ We don't add any validators to the language select box.

Form control state

The form control instance on our model encapsulates state about the control itself, such as if it is currently valid or if it's been touched.

Dirty & Pristine

We can get a reference to these form control instances in our template through the `controls` property of our `myform` model, for example we can print out the the dirty state of the email field like so:

```
<pre>Dirty? {{ myform.controls.email.dirty }}</pre>
```

`dirty` is `true` if the user has *changed* the value of the control.

The opposite of `dirty` is `pristine` so if we wrote:

```
<pre>Pristine? {{ myform.controls.email.pristine }}</pre>
```

This would be `true` if the user hasn't changed the value, and `false` if the user has.

Touched & Untouched

A controls is said to be *touched* if the the user focused on the control and then focused on something else. For example by clicking into the control and then pressing *tab* or clicking on another control in the form.

The difference between `touched` and `dirty` is that with touched the user doesn't need to actually change the value of the input control.

```
<pre>Touched? {{ myform.controls.email.touched }}</pre>
```

`touched` is `true` of the field has been touched by the user, otherwise it's false.

The opposite of `touched` is the property `untouched`.

Valid & Invalid

We can also check the `valid` state of the control with:

```
<pre>Valid? {{ myform.controls.email.valid }}</pre>
```

`valid` is `true` of the field doesn't have any validators or if *all* the validators are passing.

Again the opposite of `valid` is `invalid`, so we could write:

```
<pre>Invalid? {{ myform.controls.email.invalid }}</pre>
```

This would be `true` if the control was invalid and `false` if it was valid.

Validation styling

Bootstrap has classes for showing visual feedback for form controls when they are invalid.

For instance if we add the `has-danger` class to the parent `div` of the input control with the class of `form-group` it adds a *red* border.

Conversely if we add the `has-success` class it adds a *green* border.

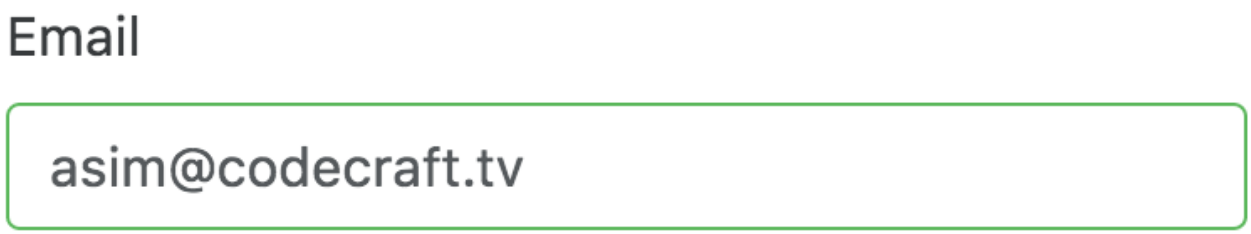


Figure 1. Valid Form Control

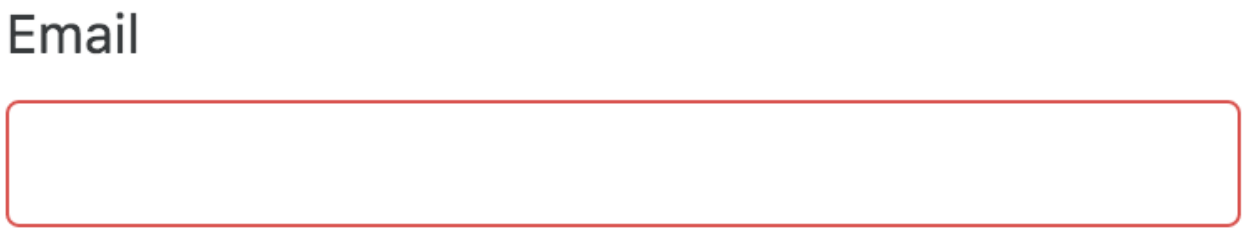


Figure 2. Invalid Form Control

We can combine bootstrap classes with `dirty` and `invalid FormControl` properties and the `ngClass` directive to give the user some nice visual feedback, like so:

```
<div class="form-group" [ngClass]="{
  'has-danger': myform.controls.email.invalid && myform.controls.email.dirty, ①
  'has-success': myform.controls.email.valid && myform.controls.email.dirty ②
}">
```

① If the email is invalid and it's been touched by the user then we add the `has-danger` class giving the control a red border.

② If the email is valid and it's been touched by the user then we add the `has-success` class giving the control a red border.

The reason we check for the `dirty` property being true is so we don't show the user visual feedback when the form is first displayed. Instead we only show the user feedback when they have had a chance to edit the field.

Now the input control shows the *green* border when it's `valid` and `dirty` and *red* if it's `invalid` and `dirty`.

Writing shorter validation expressions

The above can quickly become cumbersome to use in our templates, especially for things like the nested `firstName` and `lastName` controls.

Since the `firstName` and `lastName FormControls` exist *under* the `name FormGroup` to access those from the template we need to use syntax like this:

```
<div class="form-group"
       [ngClass]="{
        'has-danger': myform.controls.name.controls.firstName.invalid &&
myform.controls.name.controls.firstName.dirty,
        'has-success': myform.controls.name.controls.firstName.valid &&
myform.controls.name.controls.firstName.dirty
}">
```

The length of the expression quickly becomes *unwieldy*.

We can help ourselves here by creating local properties on our component to reflect the individual `FormControls` and binding directly to them in our template, like so:

script.ts

```
class ModelFormComponent implements OnInit {
  langs: string[] = [
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;
  firstName: FormControl; ①
  lastName: FormControl;
  email: FormControl;
  password: FormControl;
  language: FormControl;

  ngOnInit() {
    this.createFormControls();
    this.createForm();
  }

  createFormControls() { ②
    this.firstName = new FormControl('', Validators.required);
    this.lastName = new FormControl('', Validators.required);
    this.email = new FormControl('', [
      Validators.required,
      Validators.pattern("[^ @]*@[^ @]*")
    ]);
    this.password = new FormControl('', [
      Validators.required,
      Validators.minLength(8)
    ]);
    this.language = new FormControl('', Validators.required);
  }

  createForm() { ③
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName,
      }),
      email: this.email,
      password: this.password,
      language: this.language
    });
  }
}
```

① We declare the `FormControls` as properties of our component. So we can *bind* to them directly in our tempalte without having to go through the top level *myform* model.

② We first create the `FormControls`.

③ We then construct the *myform* model from the form controls we created previously and stored as properties on our component.

Now we can bind directly to our individual form controls in our template without having to traverse the tree from the *myform* instance.

We can therefore re-write the *wordy* firstName `ngClass` expression to something much more succinct, like so:

```
<div class="form-group"
 [ngClass]="{
  'has-danger': firstName.invalid && firstName.dirty,
  'has-success': firstName.valid && firstName.dirty
}">
```

Validation messages

As well as styling a form when it's invalid it's also useful to show the user error messages with helpful hints about how they can make the form valid again.

Taking what we have learnt about form validation styling we can apply the same method to conditionally show or hide an error message.

Bootstrap conveniently has some markup and classes for form controls which we can use to show these error messages, lets add them to our password form control, like so:

```
<div class="form-group">
  <label>Password</label>
  <input type="password"
         class="form-control"
         formControlName="password">
  <div class="form-control-feedback" ①
       *ngIf="password.invalid && password.dirty"> ②
    Field is invalid
  </div>
</div>
```

1. The class `form-control-feedback` shows a message in red if the parent `form-group` div also has the `has-danger` class, i.e. when the field is invalid any text under this div will show as red.
2. We only show the message when the password field is both `invalid` and `dirty`.

Now when the input control is both `dirty` and `invalid` we show the validation error message `"Field is invalid"`.

However this field has two validators associated with it, the required validator and the minlength validator but with the above solution we only show one *generic* validation error message. We can't tell the user what they need to do in order to make the field valid.

How to do we show a separate validation error message for each of the validators?

We can do that by checking another property on our form control called `errors`.

This is an object which has one entry per validator, the key is the name of the validator and if the value is *not* null then the validator is *failing*.

```
<div class="form-control-feedback" *ngIf="password.errors && (password.dirty ||
password.touched)">
  <p *ngIf="password.errors.required">Password is required</p>
  <p *ngIf="password.errors.minlength">Password must be 8 characters long</p>
</div>
```

If the `errors` object has a key of `required` it means the control is failing because it's `required` and the user hasn't entered any value into the input field.

Digging a bit deeper into the `errors` property. The value can contain useful bits of information which we can show the user, for example the `minlength` validator gives us the `requiredLength` and `actualLength` properties.

```
{
  "minlength": {
    "requiredLength": 8,
    "actualLength": 1
  }
}
```

We can use this in our validation error message to give the user a bit more help in resolving the issue, like so:

```
<div class="form-control-feedback"
     *ngIf="password.errors && (password.dirty || password.touched)">
  <p *ngIf="password.errors.required">Password is required</p>
  <p *ngIf="password.errors.minlength">Password must be 8 characters long, we need
another {{password.errors.minlength.requiredLength -
password.errors.minlength.actualLength}} characters </p>
</div>
```

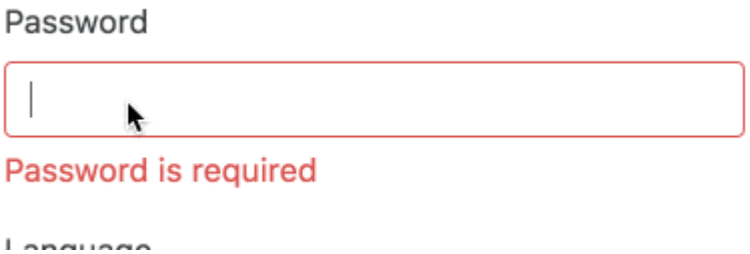


Figure 3. Form Validation Messages

Summary

We can add validators to our model form which check each field for validity.

Can render the controls with styling to show the user when fields are invalid.

Finally we can add validation error messages so the user knows how to make the form valid again.

Next up we'll look at how to submit and reset a model driven form.

Listing

http://plnkr.co/edit/whZd2BzwHhVocyxSkPia?p=preview

script.ts

```
import {
    NgModule,
    Component,
    Pipe,
    OnInit
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormsModule,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';


@Component({
  selector: 'model-form',
  template: `<form novalidate
      [formGroup]="myform">

  <fieldset formGroupName="name">
    <div class="form-group"
         [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
      }">
      <label>First Name</label>
      <input type="text"
             class="form-control"
             formControlName="firstName"
             required>
```

```
      <div class="form-control-feedback"
           *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
        <p *ngIf="firstName.errors.required">First Name is required</p>
      </div>

      <!--
        <pre>Valid? {{ myform.controls.name.controls.firstName.valid }}</pre>
        <pre>Dirty? {{ myform.controls.name.controls.firstName.dirty }}</pre>
      -->
    </div>

    <div class="form-group"
         [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
      }">
      <label>Last Name</label>
      <input type="text"
             class="form-control"
             formControlName="lastName"
             required>
      <div class="form-control-feedback"
           *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
        <p *ngIf="lastName.errors.required">Last Name is required</p>
      </div>
    </div>
  </fieldset>


  <div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
    <label>Email</label>
    <input type="email"
           class="form-control"
           formControlName="email"
           required>
    <div class="form-control-feedback"
         *ngIf="email.errors && (email.dirty || email.touched)">
      <p *ngIf="email.errors.required">Email is required</p>
      <p *ngIf="password.errors.pattern">The email address must contain at least the @
character</p>
    </div>

    <!--
      <pre>Valid? {{ myform.controls.email.valid }}</pre>
      <pre>Dirty? {{ myform.controls.email.dirty }}</pre>
    -->
```

```
  </div>

  <div class="form-group"
       [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
   }">
    <label>Password</label>
    <input type="password"
           class="form-control"
           formControlName="password"
           required>
    <div class="form-control-feedback"
         *ngIf="password.errors && (password.dirty || password.touched)">
      <p *ngIf="password.errors.required">Password is required</p>
      <p *ngIf="password.errors.minlength">Password must be 8 characters long, we need
another {{password.errors.minlength.requiredLength -
password.errors.minlength.actualLength}} characters </p>
    </div>
  </div>

  <!--
    <pre>{{ password.errors | json }}</pre>
  -->

  <div class="form-group"
       [ngClass]="{
        'has-danger': language.invalid && (language.dirty || language.touched),
        'has-success': language.valid && (language.dirty || language.touched)
      }">
    <label>Language</label>
    <select class="form-control"
            formControlName="language">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">{{lang}}
      </option>
    </select>
  </div>

  <pre>{{myform.value | json}}</pre>
</form>`
})
class ModelFormComponent implements OnInit {
  langs: string[] = [
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;
  firstName: FormControl;
```

```
  lastName: FormControl;
  email: FormControl;
  password: FormControl;
  language: FormControl;


  ngOnInit() {
    this.createFormControls();
    this.createForm();
  }

  createFormControls() {
    this.firstName = new FormControl('', Validators.required);
    this.lastName = new FormControl('', Validators.required);
    this.email = new FormControl('', [
      Validators.required,
      Validators.pattern("[^ @]*@[^ @]*")
    ]);
    this.password = new FormControl('', [
      Validators.required,
      Validators.minLength(8)
    ]);
    this.language = new FormControl('');
  }

  createForm() {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName,
      }),
      email: this.email,
      password: this.password,
      language: this.language
    });
  }
}


@Component({
  selector: 'app',
  template: `<model-form></model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
```

```
    ReactiveFormsModule],
  declarations: [
    AppComponent,
    ModelFormComponent
  ],
  bootstrap: [
    AppComponent
  ],
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Submitting & Resetting

Learning Objectives

- How to call a function on our component when the user clicks submit.
- How to reset the form back to it's original state.

Submitting

To submit a form in Angular we need a button with a `type` of `submit` in our form markup in between the `<form> ⋯ </form>` tags, like so:

```
<form>
  .
  .
  .
  <button type="submit" class="btn btn-primary" >Submit</button>
</form>
```

When we press this submit button this triggers the normal HTML5 form submission mechanism, so it tries to *POST* the form to the current URL.

However instead of issuing a standard POST we want to call a function on our component instead, to do that we use the `ngSubmit` directive and add it to the form element, like so:

```
<form (ngSubmit)="onSubmit()">
  .
  .
  .
  <button type="submit" class="btn btn-primary" >Submit</button>
</form>
```

This *hijacks* the normal form submission mechanism and instead calls the function `onSubmit()` on our component. Let's implement `onSubmit()` with a simple console.log line like so:

script.ts

```
onSubmit() {
  if (this.myform.valid) {
    console.log("Form Submitted!");
  }
}
```

Now when we press the *Submit* button `Form Submitted!` gets printed to the console.

We can do anything we want in this function.

To get the current value of the form model we can call `myform.value`.

In a later section we cover how to make HTTP requests, after learning that we will be able to make an API request passing in values from our form model.

Resetting

In a model driven form to *reset* the form we just need to call the function `reset()` on our `myform` model.

For our sample form lets reset the form in the `onSubmit()` function, like so:

script.ts

```
onSubmit() {
  if (this.myform.valid) {
    console.log("Form Submitted!");
    this.myform.reset();
  }
}
```

The form now resets, all the input fields go back to their initial state and any `valid`, `touched` or `dirty` properties are also reset to their starting values.

Summary

We can bind to the *ngSubmit* directives output event to call a function on our component when the user submits a form.

Calling the `reset` function on a form model resets the form back to it's original pristine state.

We can call functions on our component to process a form. However Angular gives us *another* way to process a forms values, by using reactive programming and RxJS, we'll cover that in the next lecture.

Listing

http://plnkr.co/edit/0srkPnZ96g2MYqwbY33k?p=preview

script.ts

```
import {
    NgModule,
    Component,
    Pipe,
    OnInit
```

```
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormsModule,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';


@Component({
  selector: 'model-form',
  template: `<form novalidate
      [formGroup]="myform"
      (ngSubmit)="onSubmit()">

  <fieldset formGroupName="name">
    <div class="form-group"
         [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
      }">
      <label>First Name</label>
      <input type="text"
             class="form-control"
             formControlName="firstName"
             required>
      <div class="form-control-feedback"
           *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
        <p *ngIf="firstName.errors.required">Last Name is required</p>
      </div>

      <!--
        <pre>Valid? {{ myform.controls.name.controls.firstName.valid }}</pre>
        <pre>Dirty? {{ myform.controls.name.controls.firstName.dirty }}</pre>
      -->
    </div>

    <div class="form-group"
         [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
      }">
      <label>Last Name</label>
      <input type="text"
             class="form-control"
             formControlName="lastName"
             required>
```

```
      <div class="form-control-feedback"
           *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
        <p *ngIf="lastName.errors.required">Last Name is required</p>
      </div>
    </div>
  </fieldset>


  <div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
    <label>Email</label>
    <input type="email"
           class="form-control"
           formControlName="email"
           required>
    <div class="form-control-feedback"
         *ngIf="email.errors && (email.dirty || email.touched)">
      <p *ngIf="email.errors.required">Email is required</p>
      <p *ngIf="password.errors.pattern">The email address must contain at least the @
character</p>
    </div>

    <!--
      <pre>Valid? {{ myform.controls.email.valid }}</pre>
      <pre>Dirty? {{ myform.controls.email.dirty }}</pre>
    -->

  </div>

  <div class="form-group"
       [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
   }">
    <label>Password</label>
    <input type="password"
           class="form-control"
           formControlName="password"
           required>
    <div class="form-control-feedback"
         *ngIf="password.errors && (password.dirty || password.touched)">
      <p *ngIf="password.errors.required">Password is required</p>
      <p *ngIf="password.errors.minlength">Password must be 8 characters long, we need
another {{password.errors.minlength.requiredLength -
password.errors.minlength.actualLength}} characters </p>
    </div>
  </div>
```

```
  <!--
    <pre>{{ language.errors | json }}</pre>
  -->

  <div class="form-group"
       [ngClass]="{
        'has-danger': language.invalid && (language.dirty || language.touched),
        'has-success': language.valid && (language.dirty || language.touched)
      }">
    <label>Language</label>
    <select class="form-control"
            formControlName="language">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">{{lang}}
      </option>
    </select>
  </div>

  <button type="submit"
          class="btn btn-primary">Submit
  </button>

  <pre>{{myform.value | json}}</pre>
</form>`
})
class ModelFormComponent implements OnInit {
  langs: string[] = [
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;
  firstName: FormControl;
  lastName: FormControl;
  email: FormControl;
  password: FormControl;
  language: FormControl;


  ngOnInit() {
    this.createFormControls();
    this.createForm();
  }

  createFormControls() {
    this.firstName = new FormControl('', Validators.required);
    this.lastName = new FormControl('', Validators.required);
    this.email = new FormControl('', [
      Validators.required,
      Validators.pattern("[^ @]*@[^ @]*")
```

```
    ]);
    this.password = new FormControl('', [
      Validators.required,
      Validators.minLength(8)
    ]);
    this.language = new FormControl('');
  }

  createForm() {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName,
      }),
      email: this.email,
      password: this.password,
      language: this.language
    });
  }

  onSubmit() {
    if (this.myform.valid) {
      console.log("Form Submitted!");
      this.myform.reset();
    }
  }
}


@Component({
  selector: 'app',
  template: `<model-form></model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule],
  declarations: [
    AppComponent,
    ModelFormComponent
  ],
  bootstrap: [
    AppComponent
  ],
})
class AppModule {
```

```
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Reactive Model Form

Learning Objectives

- How to use the `formControl` directive.
- How to subscribe to a `FormGroup` or `FormControl` observable.
- Learn how to use the `debounceTime` and `distinctUntilChanged` RxJS operators.

Setting up a reactive form

Both `FormControls` and `FormGroups` expose an *observable* called `valuesChanged`. By subscribing to this observable we can *react* in real-time to changing values of an individual form control, or a group of form controls.

One use case could be implementing a search field in an application, as the user types into the search field we may want to call an API.

Since calling an API is relatively expensive we want to limit the number of API calls to only when absolutely necessary.

Our component might have a template like so:

```
<input type="search"
       class="form-control"
       placeholder="Please enter search term">
<hr/>
<ul>
  <li *ngFor="let search of searches">{{ search }}</li>
</ul>
```

Just a single search input field and then underneath that we render out a list of search terms from an array called `searches`.

The initial component class for the above template looks like so:

```
class ReactiveModelFormComponent implements OnInit {

  searchField: FormControl; ①
  searches: string[] = []; ②

  ngOnInit() {
    this.searchField = new FormControl();
  }
}
```

① We declare a `searchField` property which is a `FormControl`, we initialise this later in our `ngOnInit`

function.

② We declare an array of `searches`, as we perform searches we'll push the individual search terms onto this array so we can see them printed out on the page.

To link our `searchField FormControl` to our template form control we use another directive called `formControl`, like so:

```
<input type="search"
       class="form-control"
       placeholder="Please enter search term"
       [formControl]="searchField"> ①
<hr/>
<ul>
  <li *ngFor="let search of searches">{{ search }}</li>
</ul>
```

① We use the `formControl` directive to link the `searchField` FormControl to the template form control.

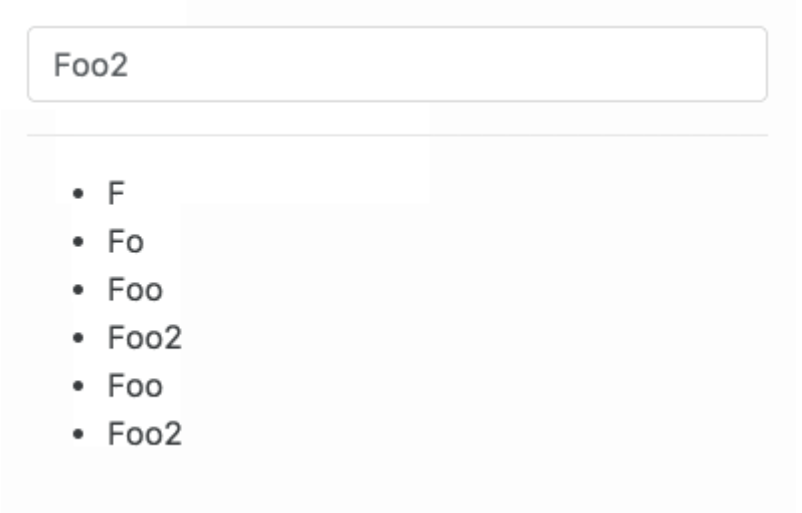
Previously we used a top level `FormGroup` instance to hold our entire form and then used the `formControlName` directive in the template to link individual template controls to controls on our `FormGroup` instance.
But in this example we just have a `FormControl` on it's own, this is why we use the `formControl` directive instead of the `formControlName` directive.

React to changes in our form

To react to changes on this form we need to subscribe to the `valueChanges` observable on our `searchField`, like so:

```
  ngOnInit() {
    this.searchField = new FormControl();
    this.searchField.valueChanges
        .subscribe(term => {
          this.searches.push(term);
        });
  }
```

As we type into the search control, each search term is pushed onto the searches array and through data binding we see the array printed on the screen, like so:



Looking at the search terms as they get printed to the screen:

```
F
Fo
Foo
Foo2
Foo
Foo2
```

We can see a search term printed for *every* keypress, if we were making API calls in response to this observable chain we would be making quite a few unnecessary API calls.

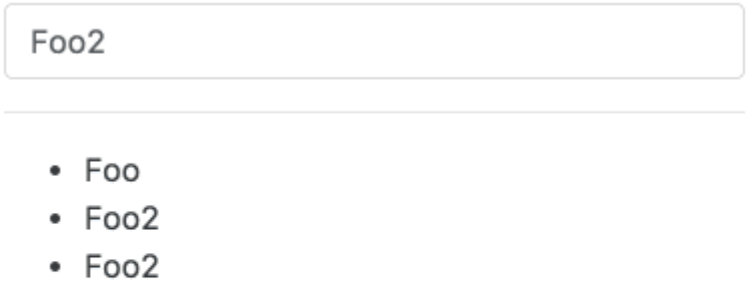
Ideally we want to only make a request when the user has stopped typing. This is a common use case with RxJS so there is an operator that implements it called `debounceTime` and we use it like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
      .debounceTime(400) ①
      .subscribe(term => {
        this.searches.push(term);
      });
}
```

① `debounceTime` takes as a first parameter a number of milliseconds, it will then only publish to the output stream if there has been no more input for *that* number of milliseconds.

Now it will only print to the console if the user has stopped typing for 400ms. If this was connected to an API then we would only be sending in one API request instead of one for every character the user typed into the search field.

Typing in the same characters to the search control we get fewer search terms printed to the screen:



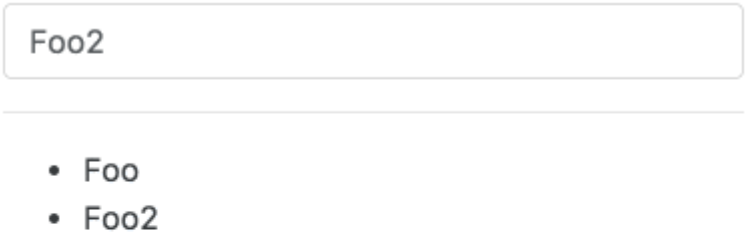

```
Foo
Foo2
Foo2
```

But looking at the above we get `Foo2` printed twice in a row, this would trigger a second unnecessary API request. That's because the user typed `Foo2`, then deleted `2` and added `2` again very quickly to get back to `Foo2`.

Ideally we only want to make the API call if the search term has *changed*. Like before there is an operator with RxJS we can use called `distinctUntilChanged` which only publishes to its output stream if the value being published is *different* from before. We can use it like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
      .debounceTime(400)
      .distinctUntilChanged()
      .subscribe(term => {
        this.searches.push(term);
      });
}
```

Typing in the same characters to the search control we get even fewer search terms printed to the screen:




```
Foo
Foo2
```

Specifically we only get `Foo2` printed once saving another API call.

Summary

We can process a model driven form the traditional way by calling a function on submission of the form and then processing the form action there.

But with Angular we also have another option of processing the form by means of an observable chain which we then subscribe to.

By using reactive forms and some RxJS operators we can implement powerful functionality in a few lines of code.

One solution is not better than the other, reactive forms are better when there needs to be some real-time processing of the form as the user types in content. Handling model driven forms with submit handlers is better when there needs to be a discrete action applied when the user presses a button.

They are not mutually exclusive, you can perform some form processing on the submit function and some processing by subscribing to to the observable.

This brings to a conclusion the model driven approach of creating and handling forms in Angular. In the next lecture we will refactor our model driven form into a template driven form.

Listing

http://plnkr.co/edit/WgNjxnsXicLHtjCumEC9?p=preview

script.ts

```
import {
    NgModule,
```

```
    Component,
    OnInit
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormControl
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import 'rxjs/Rx';

@Component({
  selector: 'reactive-model-form',
  template: `<input type="search"
       class="form-control"
       placeholder="Please enter search term"
       [formControl]="searchField">
<hr/>
<ul>
  <li *ngFor="let search of searches">{{ search }}</li>
</ul>
`
})
class ReactiveModelFormComponent implements OnInit {

  searchField: FormControl;
  searches: string[] = [];

  ngOnInit() {
    this.searchField = new FormControl();
    this.searchField.valueChanges
        .debounceTime(400)
        .distinctUntilChanged()
        .subscribe(term => {
          this.searches.push(term);
        });
  }
}

@Component({
  selector: 'app',
  template: `<reactive-model-form></reactive-model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule
```

```
  ],
  declarations: [
    AppComponent,
    ReactiveModelFormComponent
  ],
  bootstrap: [
    AppComponent
  ],
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Template Driven Forms

In this lecture we'll be converting the model driven form we've been building so far in this section into a *template driven* form.

Learning Objectives

- Know the differences and similarities between template driven forms and model driven forms.
- How to use the `ngModel` directive to link template input controls to properties on the component.
- How to implement form validation in the template driven approach.
- How to submit and reset a form in the template driven approach.

Overview

The key in understanding the *template driven* approach is that it *still* uses the same models as the *model driven* approach. In the template driven approach Angular creates the models, the `FormGroups` and `FormControls`, for us via directives we add to the template.

That's why in this course we teach the model driven approach first. So you'll have a good knowledge of the underlying model structure that is still present in template driven forms.

Template Drive Forms are just Model Driven Form but *driven* by directives in the the template versus code in the component.
In template driven we use directives to create the model.
In model driven we create a model on the component and then use directives to map elements in the template to our form model.

Form setup

We create a basic form component, exactly the same as the model form component we started with in this section, with a basic form template, a dynamic select box and a simple component like so:

```
@Component({
    selector: 'template-form',
    templateUrl: `
<form novalidate>
  <fieldset>
    <div class="form-group">
      <label>First Name</label>
      <input type="text"
             class="form-control">
    </div>

    <div class="form-group">
      <label>Last Name</label>
```

```
      <input type="text"
             class="form-control">
    </div>
  </fieldset>

  <div class="form-group">
    <label>Email</label>
    <input type="email"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Password</label>
    <input type="password"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Language</label>
    <select class="form-control">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">
              {{lang}}
      </option>
    </select>
  </div>
</form>
`
})
class TemplateFormComponent implements OnInit {

    langs:string[] = [
        'English',
        'French',
        'German',
    ];

    ngOnInit() {
    }
}
```

Remove all the `formGroup`, `formGroupName`, `formControl` and `formControlName` directives from our template, these are from the `ReactiveFormsModule` and used for model driven forms.

Directives

The directives we need to build template driven forms are in the `FormsModule` not the

ReactiveFormsModule, so lets import that and add it to our NgModule as an import and remove the ReactiveFormsModule.

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

One of the directives pulled in via the FormsModule is called NgForm.

This directive has a selector which matches the <form> tag.

So just by adding FormsModule to our NgModule imports our template form is already associated wth an instance of the NgForm directive.

This instance of ngForm is *hidden* but we can expose it with a local template reference variable attached to the form element, like so:

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

Now we can use the variable f in our template and it will point to our instance of the ngForm directive.

One of the things the ngForm directive does is create a top level FormGroup and lets us call functions as if it was an instance of a FormGroup.

If you remember one of the properties on a FormGroup was value, this is an object representation of all the forms controls.

So just like model driven forms we can output that to screen simply with a pre tag and the json pipe, like so:

```
<pre>{{f.value | json}}</pre>
```

If you ran this code now all that would get printed out is {}. The forms value is an empty object, even if you started typing into the input fields the value would not update.

This is because the ngForm directive doesn't *automatically* detect all the controls that exist inside the <form> tag it's linked to. So although it's created a top level FormGroup, it's empty.

We need go through and *explicitly* register each template control with the ngForm directive. ngForm will then create a FormControl instance and map that to the FormGroup.

In model driven forms we added formControlName directives to map template form controls to existing model form controls on the component.

In template driven forms we need Angular to create the model form controls for us for each template form control. To do that we need to do two things to each template form control:

1. Add the NgModel directive

2. Add the `name` attribute.

The `NgModel` directive creates the `FormControl` instance to manage the template form control and the `name` attribute tells the `NgModel` directive what *key* to store that `FormControl` under in the parent `FormGroup`, like so:

```
<input name="foo" ngModel>
```

is equivalent to:

```
let model = new FormGroup({
  "foo": new FormControl()
});
```

After adding `ngModel` to our template email input control, like so:

```
<div class="form-group">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         ngModel>
</div>
```

We can also see that `f.value` now shows the value of the email field, like so:

```
{
  "email": "asim@codecraft.tv"
}
```

If we now added the `name` attribute and ngModel directive to all of our template form controls we would see `f.value` print out:

```
{
  "firstName": "",
  "lastName": "",
  "email": "",
  "password": "",
  "language": ""
}
```

This isn't exactly the same as before, in our model driven form we wanted to group `firstName` and `lastName` into a nested `FormGroup` called `name`.

We can do the same in template driven forms with the `ngModelGroup` directive, lets add that to the

parent `fieldset` element of our `firstName` and `lastName` template form controls, like so:

```
<fieldset ngModelGroup="name"> ... </fieldset>
```

We can now see that the output of `f.value` matches what we have before

```
{
  "name": {
    "firstName": "",
    "lastName": ""
  },
  "email": "",
  "password": "",
  "language": ""
}
```

We have now mirrored the model we created in the model driven form using template driven forms, with `FormControls` and a nested `FormGroup`.

Two way data binding

Another feature of the `ngModel` directive is that it lets us setup *two way* data binding between a template form control and a variable on our component.

So when the user changes the value in the template form control the value of the variable on the component automatically updates and when we change the variable on the component the template form control automatically updates.

The syntax for using the `ngModel` directive in this way is a little bit different, let's set this up for our email field. First we add a string property called `email` on our component so we have somewhere to store the email, like so:

```
class TemplateFormComponent implements OnInit {
    email: string; ①
    langs:string[] = [
        'English',
        'French',
        'German',
    ];

    ngOnInit() {
    }
}
```

① We add an `email` property so we can store the email the user enters on the component.

Then we setup *two way* data binding by changing our email `ngModel` directive to:

```
<input ... [(ngModel)]="email" >
```

The `[( )]` syntax is a combination of the syntax for input property binding `[]` and output event binding `()`

The long form of writing the above would be:

```
<input ... [ngModel]="email" (ngModelChange)="email = $event" >
```

But the `[()]` syntax is shorter and clearly shows we are implementing *two way* data binding on this input control.

Domain model

In Angular we typically won't data bind to a simple string or object on our component but a *domain model* we've defined via a class, lets create one for our Signup form called Signup.

```
class Signup {
  constructor(public firstName:string = '',
              public lastName:string = '',
              public email:string = '',
              public password:string = '',
              public language:string = '') {
  }
}
```

Then on our component we replace our `email` property with:

```
model: Signup = new Signup();
```

Now lets bind all our input controls to our model directly, like so:

```
<input ... [(ngModel)]="model.email" >
```

Validation

In the model driven approach we defined the validators via code in the component.

In the template driven approach we define the validators via `directives` and HTML5 attributes in our template itself, lets add validators to our form template.

All the fields apart from the language were *required*, so we'll just add the `required` attribute to those input fields, like so:

```
<input type="email"
           class="form-control"
           name="email"
           [(ngModel)]="model.email"
           required>
```

The email field also had a *pattern* validator, we can add that via an attribute as well, like so:

```
<input type="email"
     class="form-control"
     name="email"
     [(ngModel)]="model.email"
     required
     pattern="[^ @]*@[^ @]*">
```

The password field also had a *min length* validator, we can add that via an attribute also, like so:

```
<input type="password"
       class="form-control"
       name="password"
       [(ngModel)]="model.password"
       required
       minlength="8">
```

The `attributes` we are adding to add validation to our control are parts of the standard HTML5 specification. They are built-in to HTML5 and not part of Angular.

Validation styling

Similar to model driven forms we can access each model form controls state by going through the top level form group.

The `ngForm` directive makes the top level `FormGroup` available to us via the `.form` property, so we can show the *valid*, *dirty* or *touched* state of our email field like so:

```
<pre>Valid? {{f.form.controls.email?.valid}}</pre>
<pre>Dirty? {{f.form.controls.email?.dirty}}</pre>
<pre>Touched? {{f.form.controls.email?.touched}}</pre>
```

The ? is called the *elvis* operator, it means:

*"Only try to call the property on the right of ? if the property on the left of ? is **not** null"*

So if `form.controls.email` was `null` or `undefined` it would not try to call `form.controls.email.valid` (which would throw an error).

In template driven forms the controls can sometimes be `null` when Angular is building the page, so to be safe we use the *elvis* operator. We don't need to use this in model driven forms since the models are created already in our component by the time the HTML form is shown on the page.

So again similar to model driven forms we can use this in conjunction with the `ngClass` directive and the validation classes from twitter bootstrap to style our form to give visual feedback to the user when it's invalid.

Lets add validation styling to our email field, like so:

```
<div class="form-group"
     [ngClass]="{
      'has-danger': f.form.controls.email?.invalid && (f.form.controls.email?.dirty ||
f.form.controls.email?.touched),
      'has-success': f.form.controls.email?.valid && (f.form.controls.email?.dirty ||
f.form.controls.email?.touched)
 }">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         [(ngModel)]="model.email"
         required
         pattern="[^ @]*@[^ @]*">
</div>
```

The above code displays a red border round the input control when it's invalid and a green border when it's valid.

Writing shorter validation expressions

The `NgForm` directive does provide us with a shortcut to the `controls` property so we can write `f.controls.email?.valid` instead of `f.form.controls.email?.valid`.

But both are still pretty *wordy*, and if we wanted to get access to a nested form control like `firstName` it can become even more cumbersome, `f.controls.name.firstName?.valid`.

Using the `ngModel` directive however provides us with a *much* shorter alternative.

We can get access to the instance of our `ngModel` directive by using a *local template reference*

variable, like so:

```
<input ... [(ngModel)]="model.email" #email="ngModel"> </input>
```

Then in our template we can use our local variable `email`.

Since `NgModel` created the `FormControl` instance to manage the template form control in the first place, it stored a *reference* to that `FormControl` in its `control` property which we can now access in the template like so `email.control.touched`. This is such a common use case that the `ngModel` directive provides us a shortcut to the `control` property, so we can just type `email.touched` instead.

We can then shorten our validation class expression and re-write the template for our email control like so:

```
<div class="form-group"
     [ngClass]="{
      'has-danger': email.invalid && (email.dirty || email.touched), ①
      'has-success': email.valid && (email.dirty || email.touched)
 }">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         [(ngModel)]="model.email"
         required
         pattern="[^ @]*@[^ @]*"
         #email="ngModel"> ②
</div>
```

① We can now access the form control directly through the template local variable called `email`.

② We create a template local variable pointing to the instance of the ngModel directive on this input control.

So now our template is a lot less *verbose*.

As long as we named our local reference variables the same name we named our form controls in the *model driven version of this form* we can just re-use the same `ngClass` syntax, like so:

```
<div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
```

Validation messages

As for form validation messages, we can use *exactly the same method* that we used in model driven forms. As long as we named the local reference variables the same as the form controls in the model driven approach we can use exactly the same HTML in our template driven forms, like so:

```
<div class="form-control-feedback"
     *ngIf="email.errors && (email.dirty || email.touched)">
  <p *ngIf="email.errors.required">Email is required</p>
  <p *ngIf="email.errors.minlength">Email must contain at least the @ character</p>
</div>
```

Submitting the form

Submitting a form is exactly the same in model driven forms as it is in template driven forms.

We need a submit button, this is just button with a `type="submit"` somewhere between the opening and closing `form` tags.

```
<form>
  .
  .
  .
  <button type="submit" class="btn btn-primary" >Submit</button>
</form>
```

By default this would just try to post the form to the current URL in the address bar, to hijack this process and call a function on our component instead we use the `ngSubmit` directive (which comes from the `FormsModule`).

```
<form (ngSubmit)="onSubmit()">...</form>
```

This is an output event binding which calls a function on our component called `onSubmit` when the user clicks the submit button.

However, we don't want the form submitted when the form is invalid. We can easily disable the submit button when the form is invalid, like so:

```
<button type="submit" class="btn btn-primary" [disabled]="f.invalid">Submit</button>
```

Resetting the form

In the model driven approach we reset the form by calling the function `reset()` on our `myform` model.

We need to do the same in our template driven approach but we don't have access to the underlying form model in our component. We only have access to it in our template via our local reference variable `f.form`

However, we can get a reference to the `ngForm` instance in our component code by using a `ViewChild` decorator which we covered in the section on components earlier on in this course.

This decorator gives us a reference in our component to something in our template.

First we create a property on our component to hold an instance of `NgForm`, like so:

```
form: any;
```

Then we import the `ViewChild` decorator from `@angular/core`, like so:

```
import { ViewChild } from '@angular/core';
```

Finally we decorate our property with the `ViewChild` decorator. We pass to `ViewChild` the *name* of the local reference variable we want to link to, like so:

```
@ViewChild('f') form: any;
```

And then in our `onSubmit()` function we can just call `form.reset()` like we did in the model driven approach.

The full listing for our component is now:

```
class TemplateFormComponent {

  model: Signup = new Signup();
  @ViewChild('f') form: any;

  langs: string[] = [
    'English',
    'French',
    'German',
  ];

  onSubmit() {
    if (this.form.valid) {
      console.log("Form Submitted!");
      this.form.reset();
    }
  }
}
```

Now when we submit the form it *blanks out* all the fields and also resets the states of the form controls so any validation styling and errors reset also to the original pristine condition.

Summary

In this lecture we converted our model driven form into a template driven form.

We learnt that the template driven form still uses the same classes as the model driven form but in the template drive approach the models are created by directives in the template instead of explicitly created on the component.

The `ngForm` directive automatically attaches to `<form>` and creates a top level `FormGroup`.

We learnt about the `?` elvis operator and how we can use it when some properties in a dot chain can be undefined or null.

We learnt how to use the ngModel directive, how it creates the FormControl instance for us and how we can use it to implement two way data binding to a domain model on our component.

We learnt how to use a domain model in Angular.

We learnt how to implement validation in template driven forms as well as how to submit and reset a form.

Listing

http://plnkr.co/edit/f4Dj1ZPVJHe6kcuF3Bk2?p=preview

script.ts

```
import {
    NgModule,
    Component,
    OnInit,
    ViewChild
} from '@angular/core';
import {
    FormsModule,
    FormGroup,
    FormControl
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

class Signup {
  constructor(public firstName: string = '',
              public lastName: string = '',
              public email: string = '',
              public password: string = '',
              public language: string = '') {
```

```
  }
}


@Component({
  selector: 'template-form',
  template: `<!--suppress ALL -->
<form novalidate
      (ngSubmit)="onSubmit()"
      #f="ngForm">

    <fieldset ngModelGroup="name">
        <div class="form-group"
             [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
   }">
            <label>First Name</label>
            <input type="text"
                   class="form-control"
                   name="firstName"
                   [(ngModel)]="model.firstName"
                   required
                   #firstName="ngModel">
            <div class="form-control-feedback"
                 *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
                <p *ngIf="firstName.errors.required">First name is required</p>
            </div>
        </div>

        <div class="form-group"
             [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
   }">
            <label>Last Name</label>
            <input type="text"
                   class="form-control"
                   name="lastName"
                   [(ngModel)]="model.lastName"
                   required
                   #lastName="ngModel">
            <div class="form-control-feedback"
                 *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
                <p *ngIf="lastName.errors.required">Last name is required</p>
            </div>
        </div>
    </fieldset>


    <div class="form-group"
```

```
         [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
        <label>Email</label>
        <input type="email"
               class="form-control"
               name="email"
               [(ngModel)]="model.email"
               required
               pattern="[^ @]*@[^ @]*"
               #email="ngModel">
        <div class="form-control-feedback"
             *ngIf="email.errors && (email.dirty || email.touched)">
            <p *ngIf="email.errors.required">Email is required</p>
            <p *ngIf="email.errors.pattern">Email must contain at least the @
character</p>
        </div>
    </div>


    <div class="form-group"
         [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
  }">
        <label>Password</label>
        <input type="password"
               class="form-control"
               name="password"
               [(ngModel)]="model.password"
               required
               minlength="8"
               #password="ngModel">
        <div class="form-control-feedback"
             *ngIf="password.errors && (password.dirty || password.touched)">
            <p *ngIf="password.errors.required">Password is required</p>
            <p *ngIf="password.errors.minlength">Password must be at least 8
characters long</p>
        </div>
    </div>

    <div class="form-group">
        <label>Language</label>
        <select class="form-control"
                name="language"
                [(ngModel)]="model.language">
            <option value="">Please select a language</option>
            <option *ngFor="let lang of langs"
                    [value]="lang">{{lang}}
            </option>
```

```
        </select>
    </div>

    <button type="submit"
            class="btn btn-primary"
            [disabled]="f.invalid">Submit
    </button>

    <pre>{{f.value | json}}</pre>
</form>
`
})
class TemplateFormComponent {

  model: Signup = new Signup();
  @ViewChild('f') form: any;

  langs: string[] = [
    'English',
    'French',
    'German',
  ];

  onSubmit() {
    if (this.form.valid) {
      console.log("Form Submitted!");
      this.form.reset();
    }
  }
}

@Component({
  selector: 'app',
  template: `<template-form></template-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    TemplateFormComponent
  ],
  bootstrap: [
    AppComponent
  ],
```

```
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

In this section we covered the two main methods of implementing forms in Angular, the template driven and model driven approach.

The template driven approach is easier to setup and use. It uses two way data binding between the template input controls and your domain model with the `ngModel` directive which saves a lot of development time.

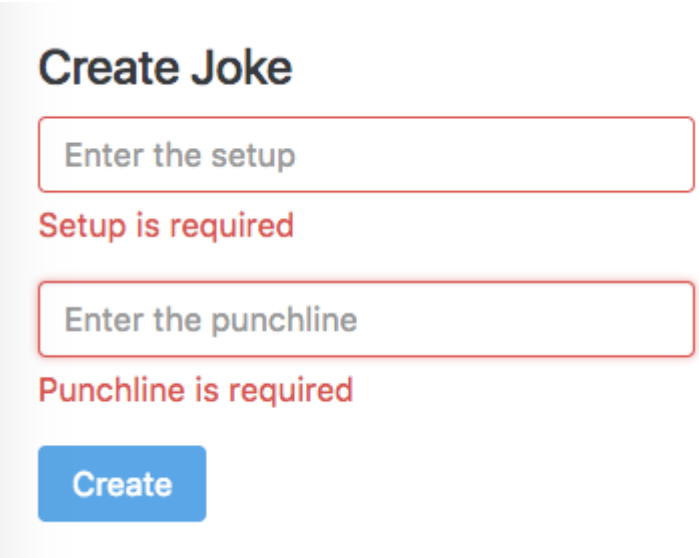
However the logic of the form is described in the HTML template so *testing* requires a more complex setup with a full end to end testing environment in a browser simulating user interactions.

In the model driven approach the definition of the form and most of the logic exists on the component as javascript. It's harder to setup but can be unit tested in isolation without needing to interact with the form template making it much easier to test. It also provides convenient features to validate and post-process input in real-time via the `valuesChanged` observable.

In future advanced chapters we'll cover topics such as custom validation and generalised error handling.

Activity

Change the simple form we had in our Joke application into a *Model Driven* form with both form validation styling and validation error messages, like so:



Steps

Fork this plunker:

Change the `JokeFormComponent` so the form is implemented using model driven forms.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/zVVqkWuNwnu6onBOqPLe?p=preview

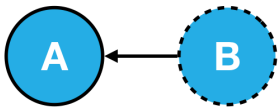
Dependency Injection & Providers

Overview

As our application grows beyond one module then we need to deal with the issue of dependencies.

What is a dependency?

When module `A` in an application needs module `B` to run, then module `B` is a *dependency* of module `A`.



Realistically when writing applications we can't get away from building numerous dependencies between parts of our code.

For example lets imagine we have a class of `EmailSender`, like so:

```
class MailChimpService extends EmailService { }

class EmailSender {
  emailService: EmailService;

  constructor() {
    this.emailService = new MailChimpService("APIKEY12345678910");
  }

  sendEmail(mail: Mail) {
    this.emailService.sendEmail(mail);
  }
}
emailSender = new EmailSender();
emailSender.sendEmail(mail);
```

Whats wrong with the above code?

Inflexible

Hard to re-use in other configurations.

It hardcodes `MailChimpService` as the email service that *actually* sends the email.

How would you use this class if you wanted to use another email provider?

Hard to test

How can you test the above code?

Calling `sendEmail(mail)` sends a real email to a real email address using an external service we

have no control over.

How do we test that calling `sendEmail` really sends an email?

Brittle

Hard to maintain.

If we changed our API key we need to make sure it's changed in *every* instance we've use the `MailChimpService`.

Even if we put the API key in a global config variable what if the `MailChimpService` changed *how* authentication happens and now it uses a *username/password* combination.

The above is described as *tight coupling*. The `EmailSender` class is said to be *tightly coupled* with the `MailChimpService` class. This makes the code *inflexible*, *hard to test* and *brittle*.

Now we can't get away from the fact that the `EmailSender` class needs the `MailChimpService` class to function. `MailChimpService` is a *dependency* of `EmailSender`.

But we can change the above code so that it's *easy to reuse*, *easy to test* and *easier to maintain*.

```
class MailChimpService extends EmailService { }

class EmailSender {
  emailService: EmailService;

  constructor(emailService: EmailService) { ①
    this.emailService = emailService;
  }

  sendEmail(mail: Mail) {
    this.emailService.sendEmail(mail);
  }
}
emailSender = new EmailSender(new MailChimpService());
emailSender.sendEmail(mail);
```

① The `emailService` is now passed *into* our class via the constructor.

Previously the `EmailSender` constructor was responsible for creating an *instance* of its dependency the `MailChimpService`.

Now *something else* is responsible for creating the *instance* of `MailChimpService` and then passing it into the `EmailSender` via it's constructor.

So if we wanted to create an instance of the `EmailSender` class we now need to pass in all the required dependencies in the constructor.

The dependencies are now said to be *decoupled* from our `EmailSender` class, how does this help us with our 3 points:

Flexible/Easier to re-use

We can *re-use* the `EmailSender` class but with a different email service.

For example if we wanted to use `SendGridService` instead of `MailChimpService`. As long as `SendGridService` still has a function with the signature `sendEmai(mail)` we can pass into the `EmailSender` constructor an instance of `SendGridService` instead of `MailChimpService`, like so:

```
emailSender = new EmailSender(new SendGridService());
```

Easier to test

Following on from the above we can now *test* our `EmailSender` class much easier.

We can pass in a *dummy* class which doesn't actually send emails however *does* let us check to see if the `sendEmail` function was called, like so:

```
MockedEmailService extends EmailService {
  mailSent: boolean = false;

  sendEmail(mail: Mail) {
    this.mailSent = true;
  }
}
let mockService = MockedEmailService()
emailSender = new EmailSender(mockService);
if (mockService.mailSent === true) { ... }
```

Easier to maintain

Since the `EmailSender` class is not responsible for creating concrete instances of the email service if, for instance, the `MailChimpService` required some new configuration then the `EmailSender` class isn't affected.

As long as the `MailChimpService` implements the `sendEmail` function, *how* it's constructed and functions internally is of no concern to the `EmailSender` class.

This idea of moving the responsibility of creating concrete instances of dependency's to something else is called *Inversion of Control*, or *IoC*.

The specific design pattern for implementing IoC above is called *Dependency Injection*, we injected the dependencies of `EmailSender` in the constructor.

Dependency injection is an important application design pattern it's used not only in Angular but throughout software development as a whole.

Angular has its own dependency injection framework, and we really can't build an Angular application without it. It's used so widely that almost everyone just calls it *DI*.

Components

The DI framework in Angular consists of 4 concepts working together:

Token

This uniquely identifies something that we want injected. A *dependancy* of our code.

Dependancy

The actual code we want injected.

Provider

This is a map between a *token* and a list of *dependancies*.

Injector

This is a function which when passed a *token* returns a *dependancy* (or a list of dependencies) === Summary

In this section you will learn:

- How the Angular DI framework works under the covers.
- What are injectors & child injectors.
- What function do the `@Inject` and `@Injectable` decorators play in the DI framework.
- What are the different types of dependencies we can inject in Angular.
- How to configure DI in Angular with Angular module providers, component providers and component view providers.

Injectors

Learning Objectives

- Know how an injector resolves a *token* into a *dependency*.
- Know how child injectors work.

Creating Injectors

At the core of the DI framework is an *injector*.

An injector is passed a *token* and returns a *dependency* (or list of).

We say that an injector *resolves* a token into a dependency.

Normally we never need to implement an *injector*. Angular handles low level injectable implementation details for us and typically we just configure it to give us to behaviour we want.

However to explain how injectors work we will implement some injectable code, like so:

```
import { ReflectiveInjector } from '@angular/core'; ①

class MandrillService {}; ②
class SendGridService {};

let injector = ReflectiveInjector.resolveAndCreate([  ③
  MandrillService,
  SendGridService
]);

let emailService = injector.get(MandrillService); ④
console.log(emailService);
```

① We import our injector class.

② We create two service classes, a `MandrillService` which sends email via the Mandrill platform and the `SendGridService` which sends email via the SendGrid platform.

③ We configure our injector by providing an array of classes.

④ We pass in a token, the class name, into our injector and ask it to *resolve* to a dependency. In this case it simply returns an instance of `MandrillService`.

The injector doesn't return the class, but an *instance* of the class instantiated with `new`, like so:

```
emailService = new MandrillService()
```

Dependency caching

The dependencies returned from injectors are cached. So multiple calls to the *same* injector for the *same* token will return the *same* instance, like so:

```
let emailService1 = injector.get(MandrillService);
let emailService2 = injector.get(MandrillService);
console.log(emailService1 === emailService2); // true
```

A *different* injector for the *same* token might return a *different* instance of a dependency but the *same* injector will always return the *same* instance.

`emailService1` and `emailService2` point to exactly the same thing, therefore we can share *state* between two different parts of our application by injecting the same dependency, like so:

```
let emailService1 = injector.get(MandrillService);
emailService1.foo = "moo";

let emailService2 = injector.get(MandrillService);
console.log(emailService2.foo); // moo ①
```

① Since `emailService1` and `emailService2` are the same instance, setting a property on one will mean it can be read from the other and vice versa.

Child Injectors

Injectors can have one or more *child* injectors. These behave just like the parent injector with a few additions.

A. Each injector creates it's own *instance* of a dependency

```
import { ReflectiveInjector } from '@angular/core';

class EmailService {}

let injector = ReflectiveInjector.resolveAndCreate([EmailService]); ①
let childInjector = injector.resolveAndCreateChild([EmailService]);

console.log(injector.get(EmailService) === childInjector.get(EmailService)); // false
②
```

① The `childInjector` and parent `injector` are both configured with the same providers.

② The `childInjector` resolves to a different *instance* of the dependency compared to the parent `injector`.

Both injectors are configured with the *same* `EmailService`. They each resolved the dependency and

returned different instances.

I've mentioned previously that different injectors return different instances of dependencies, this is also true even if the injector is a *child* injector, it will still resolve to a different instance to the parent.

B. Child injectors forward requests to their parent injector if they can't resolve the token locally.

```
import { ReflectiveInjector } from '@angular/core';

class EmailService {}

let injector = ReflectiveInjector.resolveAndCreate([EmailService]); ①
let childInjector = injector.resolveAndCreateChild([]); ②

console.log(injector.get(EmailService) === childInjector.get(EmailService)); // true
③
```

① We configure a parent injector with `EmailService`.

② We create a child injector from the parent injector, this child injector is not configured with any providers.

③ The parent and child injectors resolve the same token and both return the *same* instance of the dependency.

We request the token `EmailService` from the `childInjector`, it can't find that token locally so it asks it's parent injector which returns the instance it had cached from a previous direct reqeust.

Therefore the dependency returned by the child and the parent is exactly the same instance .

Don't worry if the significance of this isn't clear yet, it'll become clear in the lecture on configuring DI in Angular.

Summary

We configure injectors with providers.

We pass to injectors a token and then resolve this into a dependancy.

Injectors cache dependancies, so multiple calls result in the same instance being returned.

Different injectors hold different caches, so resolving the same token from a different injector will return a different instance.

We create child injectors from parent injectors.

A child injector will forward a request to their parent if it can't resolve the token itself.

So far we've only covered providers that provide *classes* providers can provide other types of dependencies which is the topic of the next lecture.

Listing

http://plnkr.co/edit/0Sp1eLq1bZysbyF3N86p?p=preview

script.ts

```
import {ReflectiveInjector} from '@angular/core';
import {OpaqueToken} from '@angular/core';


// Simple Injector Example
{
  console.log("Simple Injector Example");
  class MandrillService {
  }
  class SendGridService {
  }

  let injector = ReflectiveInjector.resolveAndCreate([
    MandrillService,
    SendGridService
  ]);
  let emailService = injector.get(MandrillService);
  console.log(emailService);

  // Injector Caching Example
  {
    console.log("Injector Caching Example");
    let emailService1 = injector.get(MandrillService);
    let emailService2 = injector.get(MandrillService);
    console.log(emailService1 === emailService2); // true
  }

  // Injector Caching Caching State Sharing Example
  {
    console.log("Injector Caching Caching State Sharing Example");
    let emailService1 = injector.get(MandrillService);
    emailService1.foo = "moo";

    let emailService2 = injector.get(MandrillService);
    console.log(emailService2.foo); // moo
  }

}

//  Child Injector Forwards Request to Parent
{
  console.log("Child Injector Forwards Request to Parent");
  class EmailService {
  }
```

```
  let injector = ReflectiveInjector.resolveAndCreate([EmailService]);
  let childInjector = injector.resolveAndCreateChild([]);

  console.log(injector.get(EmailService) === childInjector.get(EmailService)); // true
}

//  Child Injector Returns Different Instance
{
  console.log("Child Injector Returns Different Instance");
  class EmailService {
  }
  class PhoneService {
  }

  let injector = ReflectiveInjector.resolveAndCreate([EmailService]);
  let childInjector = injector.resolveAndCreateChild([EmailService]);

  console.log(injector.get(EmailService) === childInjector.get(EmailService));
}
```

Provider

Learning Objectives

- Know how we can configure injectors with providers.
- Know the 4 different types of dependencies we can configure provider to provide.

Providers

As mentioned in the previous lecture we can *configure* injectors with *providers* and a provider links a *token* to a *dependency*.

But so far we seem to be configuring our injector with just a list of classes, like so:

```
let injector = ReflectiveInjector.resolveAndCreate([
  MandrillService,
  SendGridService
]);
```

The above code is in fact a shortcut for:

```
let injector = ReflectiveInjector.resolveAndCreate([
  { provide: MandrillService, useClass: MandrillService },
  { provide: SendGridService, useClass: SendGridService },
]);
```

The real configuration for a provider is an object which describes a *token* and configuration for how to create the associated *dependency*.

The `provide` property is the *token* and can either be a *type*, a *string* or an instance of something called an `InjectionToken`.

The other properties of the provider configuration object depend on the *kind* of dependency we are configuring, since we are configuring classes in this instance we the `useClass` property.

If all we want to do is providing classes, it's such a common pattern that Angular lets you define a list of class names as the providers.

Switching dependencies

The above is an excellent example of how we can use the DI framework to architect our application for *ease of re-use*, *ease of testing* and less *brittle* code.

If we wanted to *re-use* our application and move from `Mandrill` to `SendGrid` without using DI we would have to search through all the code for where we have requested `MandrillService` to be

injected and replace with SendGridService.

A better solution is to configure the DI framework to return either MandrilService or SendGridService depending on the context, like so:

```
import { ReflectiveInjector } from '@angular/core';

class MandrillService {};
class SendGridService {};

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "EmailService", useClass: MandrillService }  ①
]);

let emailService = injector.get("EmailService");
console.log(emailService); // new MandrillService()
```

① The token is "EmailService" and the dependency is the class MandrillService

The above is configured so when code requests the token "EmailService" it returns an instance of the class MandrillService.

To switch to using the SendGridService throughout our application we can just configure our injector with a different provider, like so:

```
let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "EmailService", useClass: SendGridService } ①
]);
```

① The token is "EmailService" and the dependency is the class SendGridService

Now the DI framework just returns instances of SendGridService whenever the token "EmailService" is requested.

Provider configurations

So far we've only seen how we can configure a provider to provide classes however there are 4 types of dependencies providers can provide in Angular.

useClass

We can have a provider which maps a *token* to a *class*, like so:

```
let injector = ReflectiveInjector.resolveAndCreate([
  { provide: Car, useClass: Car },
]);
```

The above is so common that there is a shortcut, if all we want to provide is a class we can simply

pass in the class name, like so:

```
let injector = ReflectiveInjector.resolveAndCreate([Car]);
```

useExisting

We can make two tokens map to the same *thing* via aliases, like so:

```
import { ReflectiveInjector } from '@angular/core';

class MandrillService {};
class SendGridService {};
class GenericEmailService {};

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: GenericEmailService, useClass: GenericEmailService }, ①
  { provide: MandrillService, useExisting: GenericEmailService }, ②
  { provide: SendGridService, useExisting: GenericEmailService } ③
]);

let emailService1 = injector.get(SendGridService); ④
console.log(emailService1); // GenericEmailService {}
let emailService2 = injector.get(MandrillService); ⑤
console.log(emailService2); // GenericEmailService {}
let emailService3 = injector.get(GenericEmailService); ⑥
console.log(emailService3); // GenericEmailService {}
console.log(emailService1 === emailService2 === emailService3); // false ⑦
```

① The token `GenericEmailService` resolves to an instance of `GenericEmailService`.

② This provider maps the token `MandrillService` to whatever the existing `GenericEmailService` provider points to.

③ This provider maps the token `SendGridService` to whatever the existing `GenericEmailService` provider points to.

④ Requesting a resolve of `SendGridService`, `MandrillService` or `GenericEmailService` return an instance of `GenericEmailService`.

⑤ All three instances of `GenericEmailService` returned are *the same instance*.

Whenever anyone requests `MandrillService` or `SendGridService` we return an instance of `GenericEmailService` instead.

useValue

We can also provide a simple *value*, like so:

```
import { ReflectiveInjector } from '@angular/core';

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "APIKey", useValue: 'XYZ1234ABC' }
]);

let apiKey = injector.get("APIKey");
console.log(apiKey); // "XYZ1234ABC"
```

Or if we wanted to pass in an object we can, like so:

```
import { ReflectiveInjector } from '@angular/core';

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "Config",
    useValue: {
      'APIKey': 'XYZ1234ABC',
      'APISecret': '555-123-111'
    }
  }
]);

let config = injector.get("Config");
console.log(config); // Object {APIKey: "XYZ1234ABC", APISecret: "555-123-111"}
```

If the intention however is to pass around *read-only* constant values then passing an object is a problem since any code in your application will be able to *change* properties on that object. What `Config` points to *can't* be changed but the properties of `Config` *can* be changed.

So when passing in an object that you intend to be *immutable* (unchanging over time) then use the `Object.freeze` method to stop client code from being able to *change* the config object, like so:

```
let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "Config",
    useValue: Object.freeze({ ①
      'APIKey': 'XYZ1234ABC',
      'APISecret': '555-123-111'
    })
  }
]);
```

① By using `Object.freeze` that objects values can't be changed, in effect it's read-only.

useFactory

We can also configure a provider to call a function every-time a token is requested, leaving it to the provider to figure out *what* to return, like so:

```
import { ReflectiveInjector } from '@angular/core';

class MandrillService {};
class SendGridService {};
const isProd = true;

let injector = ReflectiveInjector.resolveAndCreate([
  {
    provide: "EmailService",
    useFactory: () => { ①
      if (isProd) {
        return new MandrillService();
      } else {
        return new SendGridService();
      }
    }
  },
]);

let emailService1 = injector.get("EmailService");
console.log(emailService1); // MandrillService {}
```

① When the injector resolves to this provider, it calls the `useFactory` function and returns whatever is returned by this function as the dependency.

Just like other providers the result of the call is *cached*. So even though we are using a factory and creating an instance with `new` ourselves calling `injector.get(EmailService)` again will still return the *same* instance of `MandrillService` that was created with the first call.

```
let emailService1 = injector.get(EmailService);
let emailService2 = injector.get(EmailService);
console.log(emailService2 === emailService2); // true ①
```

① Returns the same instance

Summary

We can configure providers to return 4 different kinds of dependencies: classes, values, aliases and factories.

In the next lecture we will look at the different ways we can define *tokens*.

Listing

script.ts

```
import {ReflectiveInjector} from '@angular/core';
import {OpaqueToken} from '@angular/core';
```

```
// Switching Dependencies Example
{
  console.log("Switching Dependencies Example");
  class MandrillService {
  }
  class SendGridService {
  }

  let injector = ReflectiveInjector.resolveAndCreate([
    {provide: "EmailService", useClass: MandrillService}
  ]);

  let emailService = injector.get("EmailService");
  console.log(emailService); // new MandrillService()

  {
    let injector = ReflectiveInjector.resolveAndCreate([
      {provide: "EmailService", useClass: SendGridService}
    ]);

    let emailService = injector.get("EmailService");
    console.log(emailService); // new SendGridService()
  }
}

// useClass Provider
{
  console.log("useClass");
  class EmailService {
  }
  class MandrillService extends EmailService {
  }
  class SendGridService extends EmailService {
  }

  let injector = ReflectiveInjector.resolveAndCreate([
    {provide: EmailService, useClass: SendGridService}
  ]);

  let emailService = injector.get(EmailService);
  console.log(emailService);
}

// useExisting
{
  console.log("useExisting");
  class MandrillService {
  }
  class SendGridService {
```

```
  }
  class GenericEmailService {
  }

  let injector = ReflectiveInjector.resolveAndCreate([
    {provide: GenericEmailService, useClass: GenericEmailService},
    {provide: MandrillService, useExisting: GenericEmailService},
    {provide: SendGridService, useExisting: GenericEmailService}
  ]);

  let emailService1 = injector.get(SendGridService);
  console.log(emailService1); // GenericEmailService {}
  let emailService2 = injector.get(MandrillService);
  console.log(emailService2); // GenericEmailService {}
  let emailService3 = injector.get(GenericEmailService);
  console.log(emailService3); // GenericEmailService {}
  console.log(emailService1 === emailService2 &&  emailService2 === emailService3); //
true
}

// useValue
{
  console.log("useValue");
  let injector = ReflectiveInjector.resolveAndCreate([
    {
      provide: "Config",
      useValue: Object.freeze({
        'APIKey': 'XYZ1234ABC',
        'APISecret': '555-123-111'
      })
    }
  ]);

  let config = injector.get("Config");
  console.log(config); // Object {APIKey: "XYZ1234ABC", APISecret: "555-123-111"}
}

// useFactory
{
  console.log("useFactory");
  class MandrillService {
  }
  class SendGridService {
  }

  const isProd = true;

  let injector = ReflectiveInjector.resolveAndCreate([
    {
      provide: "EmailService",
      useFactory: () => {
```

```
        if (isProd) {
          return new MandrillService();
        } else {
          return new SendGridService();
        }
      }
    },
  ]);

  let emailService1 = injector.get("EmailService");
  console.log(emailService1); // MandrillService {}
}
```

Tokens

There are a number of different types of tokens we can use when configuring providers.

Learning Objectives

- Know the 3 different ways of defining tokens.

String tokens

We can use strings as tokens, like so:

```
import { ReflectiveInjector } from '@angular/core';

class MandrillService {};
class SendGridService {};

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "EmailService", useClass: MandrillService }  ①
]);

let emailService = injector.get("EmailService");
console.log(emailService); // new MandrillService()
```

① We use the token `"EmailService"`.

In the above example we use the string `"EmailService"` as the token in our class provider configuration.

Although it's possible to use strings as tokens their use isn't recommended and instead it's preferable to use either type tokens or an instance of an `InjectionToken`.

Type tokens

We can implement the above example but instead of using strings as the token we can use a type, specifically we can use a class name as the *type*.

Let's implement the above with `EmailService` as a base class which `MandrillService` and `SendGridService` extend, like so:

```
class EmailService {};
class MandrillService extends EmailService {};
class SendGridService extends EmailService {};

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: EmailService, useClass: SendGridService }
]);

let emailService = injector.get(EmailService);
console.log(emailService);
```

We now use the base class `EmailService` as the token and not the string `"EmailService"`

InjectionToken

The third way of defining a token is via an instance of an `InjectionToken`, like so:

```
import { ReflectiveInjector } from '@angular/core';
import { InjectionToken } from '@angular/core';

class MandrillService {};
class SendGridService {};
let EmailService = new InjectionToken<string>("EmailService"); ①

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: EmailService, useClass: SendGridService } ②
]);

let emailService = injector.get(EmailService);
console.log(emailService);
```

① We create an instance of `InjectionToken` and store it in a variable.

② We use the instance of `InjectionToken` as the token in our provider.

The *string* `"EmailService"` that we pass to `InjectionToken` is only used to print a meaningful message to the developer when there is an error. It doesn't need to be unique.

If using a *base class* as the token is not an option then using an `InjectionToken` is the *preferred* method over using strings because it prevents name clashes that can occur, like so:

```
import { ReflectiveInjector } from '@angular/core';

class MandrillService {};
class SendGridService {};

let MandrillServiceToken = "EmailService";
let SendGridServiceToken = "EmailService";

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: SendGridServiceToken, useClass: SendGridService },
  { provide: MandrillServiceToken, useClass: MandrillService },
]);

let emailService1 = injector.get(SendGridServiceToken);
let emailService2 = injector.get(MandrillServiceToken);
console.log(emailService1 === emailService2);
```

In the above code we tried to use the *string* variables `MandrillServiceToken` and `SendGridServiceToken` as tokens, however they happened to use the same string `"EmailService"`.

So when configuring the injector the above is the same as doing:

```
let injector = ReflectiveInjector.resolveAndCreate([
  { provide: "EmailService", useClass: SendGridService },
  { provide: "EmailService", useClass: MandrillService },
]);
```

When configuring an injector with the *same* token multiple times, the last provider just *overwrites* the previous providers.

So in the above code the injector returns an instance of `MandrilService` as the dependency for both the `SendGridServiceToken` and the `MandrillServiceToken`.

The example above might feel quite forced but the issue is quite real. It's common for different third party libraries built by different people or teams to use the *same* configuration strings and therefore cause confusing clashes in you application.

`InjectionToken` solves this shortcoming of using string as tokens since each *instance* of an `InjectionToken` is unique even if the same descriptive string was passed in, like so:

```
import { InjectionToken } from '@angular/core';

export const EmailService1 = new InjectionToken<string>("EmailService");
export const EmailService2 = new InjectionToken<string>("EmailService");
console.log(EmailService1 === EmailService2); // false
```

Even though the two `InjectionToken` above have the same description, they are *not* equal.

The string we pass into the token is just used to give better error messages.

So now we can re-write the code like so:

```
import { ReflectiveInjector } from '@angular/core';
import { InjectionToken } from '@angular/core';

class MandrillService {};
class SendGridService {};

const MandrillServiceToken = new InjectionToken<string>("EmailService");
const SendGridServiceToken = new InjectionToken<string>("EmailService");

let injector = ReflectiveInjector.resolveAndCreate([
  { provide: SendGridServiceToken, useClass: SendGridService },
  { provide: MandrillServiceToken, useClass: MandrillService },
]);

let emailService1 = injector.get(SendGridServiceToken);
let emailService2 = injector.get(MandrillServiceToken);
console.log(emailService1 === emailService2);  // false
```

The injector returns the correct dependency for the requested token.

Summary

A *token* can be either a string, a class or an instance of `InjectionToken`.

String tokens can cause name clashes so we prefer to use *InjectionTokens* instead.

In the next lecture we will look at how we actually configure DI in Angular.

Listing

http://plnkr.co/edit/Rxs8GQOWQ4OZ5qZQWcO2?p=preview

script.ts

```
import {ReflectiveInjector} from '@angular/core';
import {InjectionToken} from '@angular/core';

// String Token (Fail Case) Example
{
  console.log("String Token (Fail Case) Example");
  class MandrillService {
  }
  class SendGridService {
  }

  let MandrillServiceToken = "EmailService";
  let SendGridServiceToken = "EmailService";

  let injector = ReflectiveInjector.resolveAndCreate([
    {provide: SendGridServiceToken, useClass: SendGridService},
    {provide: MandrillServiceToken, useClass: MandrillService},
  ]);

  let emailService1 = injector.get(SendGridServiceToken);
  let emailService2 = injector.get(MandrillServiceToken);
  console.log(emailService1 === emailService2);
}

// OpaqueToken
{
  console.log("InjectionToken");
  class MandrillService {
  }
  class SendGridService {
  }

  const MandrillServiceToken = new InjectionToken<string>("EmailService");
  const SendGridServiceToken = new InjectionToken<string>("EmailService");

  let injector = ReflectiveInjector.resolveAndCreate([
    {provide: SendGridServiceToken, useClass: SendGridService},
    {provide: MandrillServiceToken, useClass: MandrillService},
  ]);

  let emailService1 = injector.get(SendGridServiceToken);
  let emailService2 = injector.get(MandrillServiceToken);
  console.log(emailService1 === emailService2);  // false
}
```

Configuring Dependency Injection in Angular

So far in this section we've only covered how to use the low level Dependency Injection API.

With Angular however we'll never need to create injectors ourselves, Angular does this for us automatically when our application is bootstrapped.

All we need to do is to *configure* Angular with our providers and also tell it when we want something *injected* into a class constructor.

Learning Objectives

- What is the injector tree and how it maps to the component tree.
- How an where to configure injectors.
- Know when to use the `@Inject` and `@Injectable` decorators.
- Know when we *don't* need to use the `@Inject` and `@Injectable` decorators.

The Injector Tree

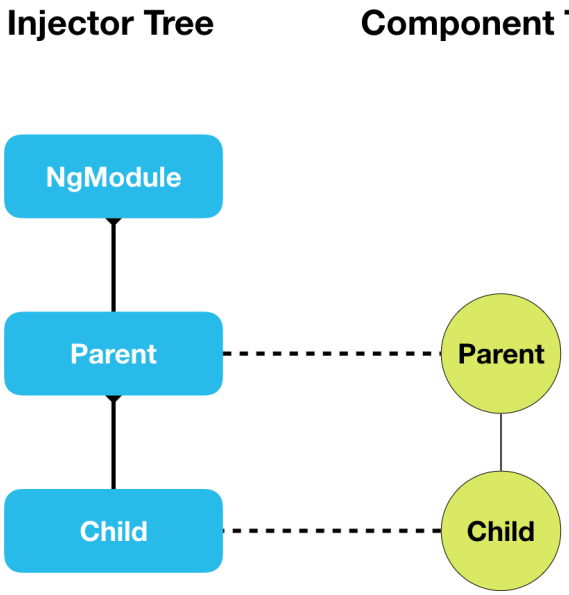
An Angular application will have a tree of injectors mirroring the component tree.

We have a top level parent injector which is attached to our `NgModule`.

Then we have *child* injectors descending in a hierarchy matching the component tree.

So a *parent* component will have a child injector stemming from `NgModule`.

A *child* component of parent component will have a child injector stemming from *Parent*.



Configuring Injectors

The `NgModule` decorator has a property called `providers` which accepts a list of providers exactly the same as we would pass to the `ReflectiveInjector` via the `resolveAndCreate` function we looked at previously, like so:

```
@NgModule({
  providers: [EmailService, JobService]
})
class AppModule { }
```

This creates a top level *parent* injector and configures it with two class providers, `EmailService` and `JobService`.

We can also configure our *Components* and *Directives* the same way using a property called `providers` on the `Component` and `Directive` decorators, like so:

```
@Component({
  selector: 'my-comp',
  template: `...`,
  providers: [EmailService]
})
```

This creates a *child injector* who's parent injector is the injector on the parent component. If there is no parent component then the parent injector is the top level `NgModule` injector.

With components we have another property called `viewProviders` which creates a special injector that resolves dependencies only for *this* components view children and doesn't act as a parent

injector for any content children, like so:

```
@Component({
  selector: 'my-comp',
  template: `...`,
  viewProviders: [EmailService]
})
```

We'll be going through a specific examples of each in the next lecture.

Using Dependency Injection in Angular

The above is how we *configure* DI in Angular so it creates injectors and configures them to resolve dependencies.

When Angular creates a component it uses the DI framework to figure out *what* to pass to the component class constructor as parameters.

However we may need to give Angular some hints by using either the `@Inject` or `@Injectable` decorators.

Let's explain with an example, we have a class called `SimpleService` and another class called `OtherService`.

```
class OtherService {
    constructor() { };
}

class SimpleService {
    constructor() { };
}
```

We configure NgModule with these classes as providers.

```
@NgModule({
    ...
    providers: [OtherService, SimpleService]
})
export class AppModule {
}
```

Then to make Angular resolve and create a `SimpleService` we add a component which requests an instance of `SimpleService` via it's constructor, like so:

```
@Component({
    selector: 'simple',
    template: `<p>Simple is as simple does</p>`,
})
class SimpleComponent {
    constructor(private simpleService:SimpleService) { }
}
```

@Inject decorator

The above works fine but, lets try and inject into `SimpleService` an instance of `OtherService`, like so:

```
class SimpleService {
  otherService: OtherService;

  constructor(otherService: OtherService) {
    this.otherService = otherService;
  };
}
```

This doesn't work and in fact gives us the error:

```
Can't resolve all parameters for SimpleService: (?).
```

We need to *explicitly* tell Angular *what* we want injected for the `otherService` parameter so we use the `@Inject` decorator like so:

```
import { Inject } from '@angular/core';
.
.
.
class SimpleService {
  otherService: OtherService;

  constructor(@Inject(OtherService) otherService: OtherService) {
      this.otherService = otherService;
  };
}
```

The first param to `@Inject` is the token we want to resolve this dependency with.

The above now works, when Angular tries to construct the class it gets the instance of `OtherService` passed in from the DI framework.

@Injectable decorator

Decorating all our constructor arguments with `@Inject` can be tiresome however so instead we can decorate our *entire* class wth `@Injectable`, like so:

```
@Injectable()
class SimpleService {
  otherService: OtherService;

  constructor(otherService: OtherService) {
      this.otherService = otherService;
  };
}
```

I find the term `@Injectable` confusing, it implies that you need to decorate a class with `@Injectable` to *inject* it into other classes.

`@Injectable` is actually a shortcut for having to decorate every parameter in your constructor with `@Inject`.

It does this for you behind the scenes by looking at the *types of each parameter*, so if we *don't* supply a type `@Injectable` *doesn't* work, like so:

```
@Injectable()
class SimpleService {
  otherSimple: OtherSimpleService;

  constructor(otherSimple: any) { ①
      this.otherSimple = otherSimple;
  };
}
```

① We don't provide a type for `otherService` so the DI framework doesn't know what to inject and therefore `Can't resolve all parameters for SimpleService: (?).` is printed to the console.

Personally I believe `@Injectable` should be renamed `@AutoInject` to reduce confusion.

@Injectable versus @Component versus @Directive

You might ask then why did we not use the `@Injectable` decorator on the component we inject `SimpleService` into, like so:

```
@Component({
  selector: 'simple',
  templateUrl: `<p>Simple is as simple does</p>`,
})
class SimpleComponent {
  constructor(private simpleService:SimpleService) { }
}
```

That's because the other decorators in Angular, such as `@Component` and `@Directive`, already perform the same function as `@Injectable`.

We *only* need to use `@Injectable` on classes which don't already use one of the other Angular decorators.

Summary

There is one top level injector created for each `NgModule` and then for each component in our app, from the root component down, there is a tree of injectors created which map to the component tree.

We configure these injectors with providers by adding the configuration to either the `providers` property on the `NgModule`, `Component` and `Directive` decorators or to the `viewProviders` property on the `Component` decorator.

We use the `@Inject` parameter decorator to instruct Angular we want to resolve a token and inject a dependency into a constructor.

We use the `@Injectable` class decorators to automatically resolve and inject all the parameters of class constructor.

This only works if each parameter has a TypeScript type associated with it, which the DI framework uses as the token.

We don't need to use the `@Injectable` class decorator on classes which are already decorated with one of the other Angular decorators, such as `@Component`.

Now we know *where* we can configure providers in the DI framework in Angular.

In the next lecture we will cover the differences between configuring providers on `NgModule`, `Component.providers` and `Component.viewProviders`.

Listing

script.ts

```
import {NgModule, Component, Injectable, Inject, TypeDecorator} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
```

```
class OtherService {
  constructor() {
  };
}

// This version doesn't work as Angular doesn't know it should be injecting
otherService
// class SimpleService {
//   otherService: OtherService;
//   constructor(otherService: OtherService) {
//     this.otherService = otherService;
//   };
// }

// This version works but we have to decorate every parameter to our constructor with
@Inject
// class SimpleService {
//   otherService: OtherService;
//
//   constructor(@Inject(OtherService) otherService: OtherService) {
//     this.otherService = otherService;
//   };
//
// }

// This works because @Injectable automatically injects every parameter to the
constructor as long as that parameter has a type
@Injectable()
class SimpleService {
  otherService: OtherService;

  constructor(otherService: OtherService) {
    this.otherService = otherService;
  };
}

// This DOESN'T work because the otherService parameter doesn't have a type
// @Injectable
// class SimpleService {
//   otherService: OtherService;
//
//   constructor(otherService: any) {
//     this.otherService = otherService;
//   };
// }

@Component({
  selector: 'simple',
  template: `<p>Simple is as simple does</p>`,
})
```

```
class SimpleComponent {
  constructor(private simpleService: SimpleService) {
  }
}


@Component({
  selector: 'app',
  template: '<simple></simple>'
})
class AppComponent {
}


@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, SimpleComponent],
  bootstrap: [AppComponent],
  providers: [OtherService, SimpleService]
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

NgModule.providers vs Component.providers vs Component.viewProviders

We can configure injectors in Angular by:

1. `providers` on *NgModule*.
2. `providers` on *Components* and *Directives*.
3. `viewProviders` on *Components*.

So the question is where do you configure your provider?

Understanding *where* to configure your provider is a key piece of understanding *how* to architect your application, so we are going to explain this via a real practical example.

Learning Objectives

- Know the difference between configuring a provider on an `NgModule`, a component or directives `providers` property and a components `viewProviders` property.

Setup

We create a class called `SimpleService` which has one property called `value` which holds a `string`.

```
class SimpleService {
  value: string;
}
```

We also have a component called `ParentComponent` which has a child component called `ChildComponent`.

```
@Component({
 selector: 'child',
 template: `
<div class="child">
   <p>Child</p>
   {{ service.value }} ①
</div>
`
})
class ChildComponent {
  constructor(private service: SimpleService) { } ②
}
```

① We use *string interpolation* to bind to the `value` property of `SimpleService`.

② We *inject* an instance of `SimpleService` into the constructor.

```
@Component({
 selector: 'parent',
 template: `
<div class="parent">
   <p>Parent</p>
   <form novalidate>
        <div class="form-group">
        <input type="text"
               class="form-control"
               name="value"
               [(ngModel)]="service.value"> ①
      </div>
  </form>
  <child></child> ②
</div>
`
})
class ParentComponent {
  constructor(private service: SimpleService) { } ③
}
```

① We use *two way data binding* to bind to the `value` property of `SimpleService`.

② We render the `ChildComponent` inside this `ParentComponent`.

③ We *inject* an instance of `SimpleService` into the constructor.

The `ParentComponent` has just one input box which reads and writes to the `SimpleService value` property using two way `ngModel` binding, the `ChildComponent` just renders the `value` to the screen with `{{ }}`.

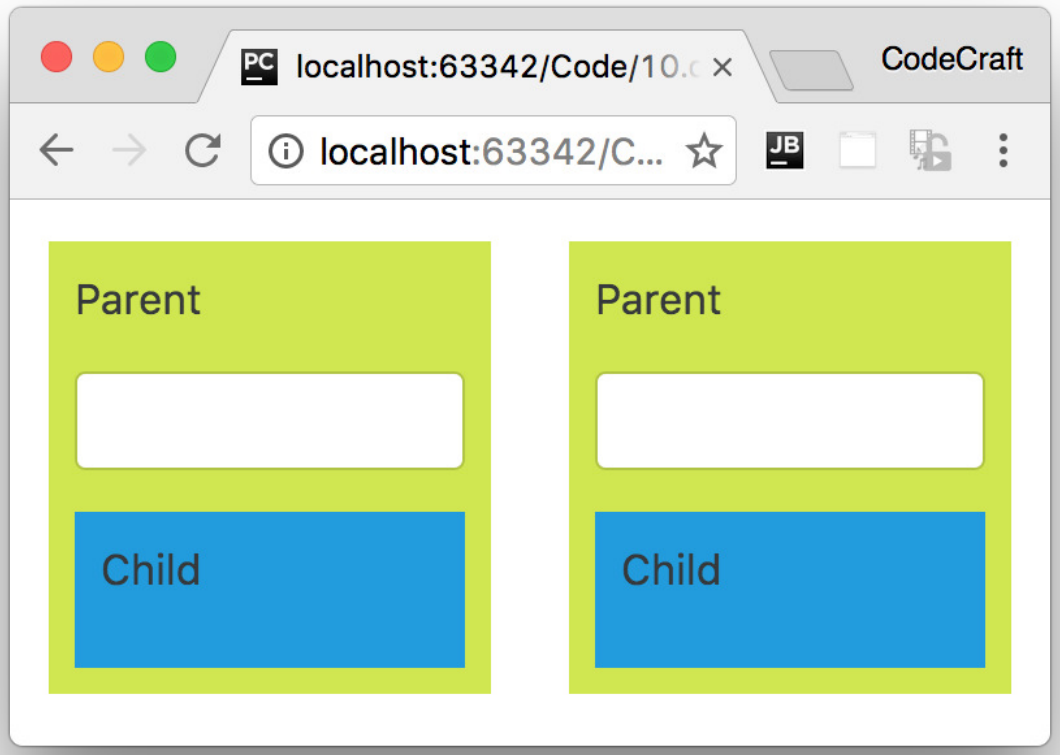
We render two *side by side* `<parent>` tags in our root `AppComponent` module, like so:

```
@Component({
 selector: 'app',
 template: `
 <div class="row">
  <div class="col-xs-6">
    <parent></parent>
  </div>
  <div class="col-xs-6">
    <parent></parent>
  </div>
</div>
 `
})
class AppComponent {
}
```

We set up our `NgModule` and bootstrap it, like so:

```
@NgModule({
  imports:  [ BrowserModule, FormsModule ],
  declarations:  [ AppComponent, ParentComponent, ChildComponent ],
  bootstrap:  [ AppComponent ]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

In the end when we run our application we should end up with something that looks like this:



We have also added some css styles on our component which has been removed from the above code, the full code can be found in the listing at the end of this lecture.

NgModule.providers

We'll first configure our `SimpleService` on the root `NgModule`, like so:

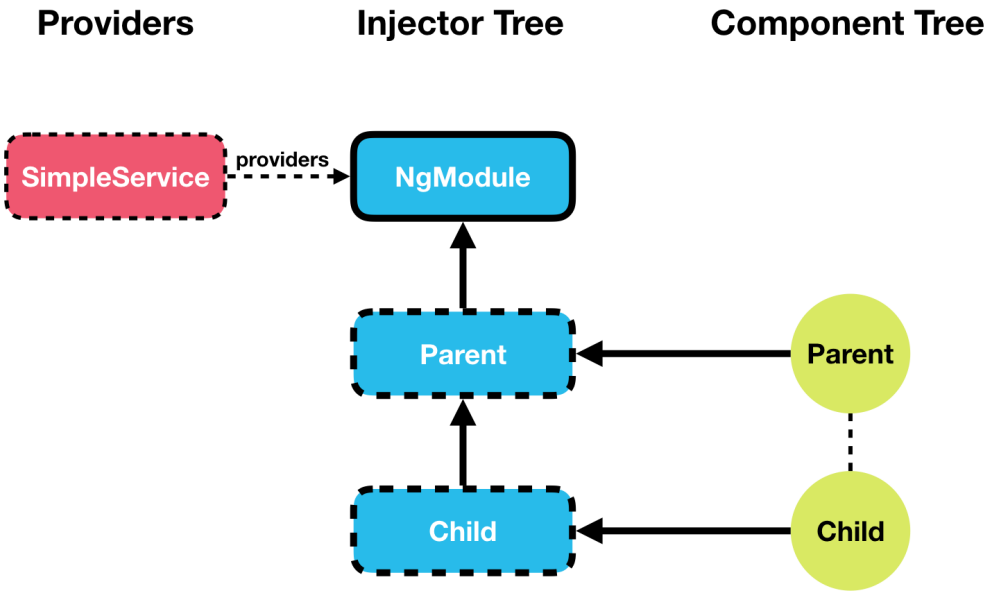
```
@NgModule({
  imports:  [ BrowserModule, FormsModule ],
  declarations:  [ AppComponent, ParentComponent, ChildComponent ],
  bootstrap:  [ AppComponent ],
  providers: [ SimpleService ] ①
})
class AppModule { }
```

① We've configured our NgModule with a class provider of `SimpleService`.

In this configuration the service has been injected onto our applications root `NgModule` and therefore is in our root `injector`.

So every request to resolve and inject the token `SimpleService` is going to be forwarded to our single

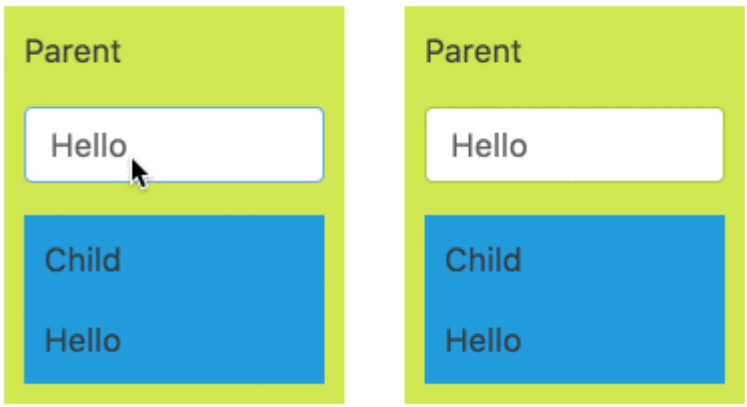
root injector.



Therefore since we only have one injector which is resolving the dependency, every-time we request an instance of `SimpleService` to be injected into one of our components it's *always* going to inject the *same* instance.

Remember if we request the *same* token from the *same* injector we get the *same* instance.

Since we've bound the input field directly to the simple service `value` field **and** it's the same instance of simple service used everywhere, then when we type into one input control it automatically updates the other input control and also the child components.



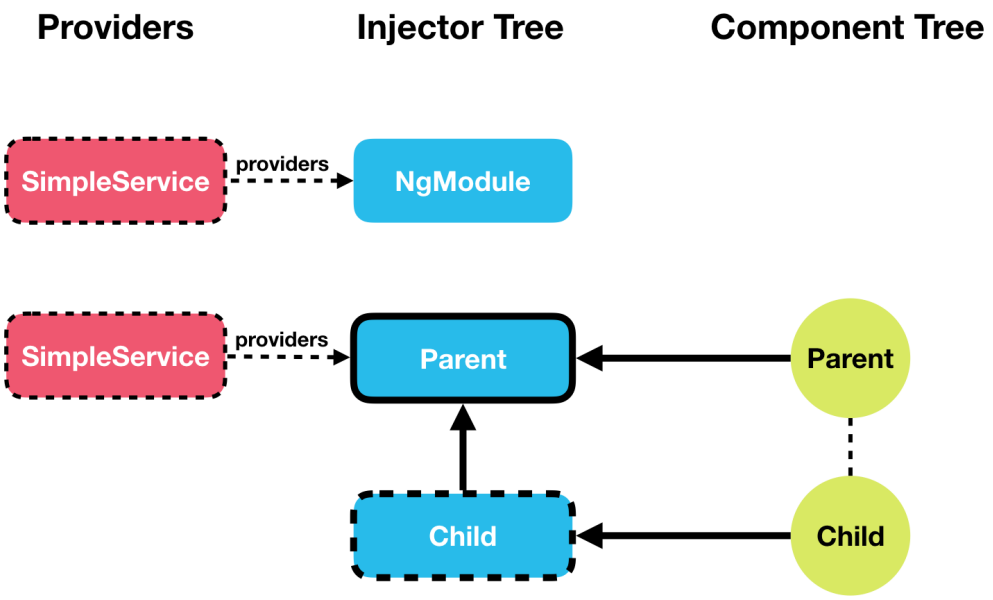
If we want to share *one* instance of a service across the *entirety* of our application we configure it on our `NgModule`.

Component.providers

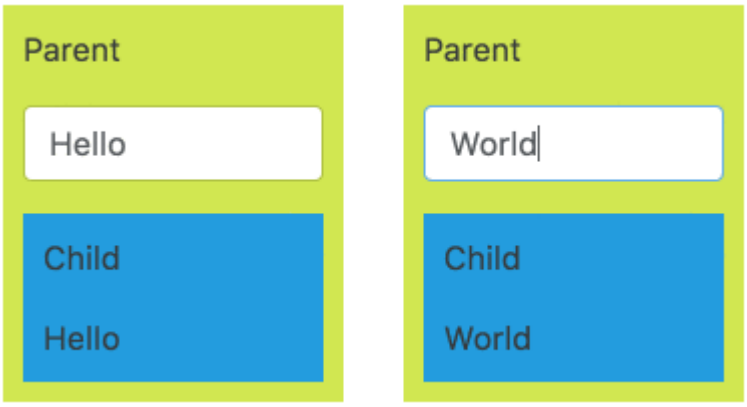
Let's now see what happens when we configure our `SimpleService` *additionally* on the `ParentComponent` via the `providers` property.

```
@Component({
 selector: 'parent',
 template: `...`,
 providers: [ SimpleService ]
})
class ParentComponent {
  constructor(private service: SimpleService) { }
}
```

Now *each* `ParentComponent` has it's *own* child injector with `SimpleService` configured, like so:



We can see from the running the code above that if we type into one parent component only *that* parent component and it's child component automatically updates, like so:



Each instance of `ParentComponent` now has it's *own* instance of `SimpleService`, so state is not shared globally but only between a `ParentComponent` and it's child components.

That's because each instance of `ParentComponent` has it's own child injector with `SimpleService` configured as a provider.

Remember when we request the *same* token from *different* injectors we get the *different* instances.

When we configured the `SimpleService` on the parent component it created a child injector, and when we tried to inject `SimpleService` into the parent component constructor it resolved and created an instance of `SimpleService` from it's own injector.

If we want to have *one* instance of a service *per* component, and shared with all the components children, we configure it on the `providers` property on our component decorator.

Component.viewProviders

If we now configure the `SimpleService` provider on the `viewProviders` property on the `ParentComponent` nothing changes, we still get the functionality we had before.

But lets use content projection and the `ng-content` component to change the child component from being a *view child* of parent to to being a *content child* of parent. i.e. lets pass in `<child></child>` to the parent component like so:

```
<parent><child></child></parent>
```

So we change the `AppComponent` template to pass in child to the parent component, like so:

```
 <div class="row">
  <div class="col-xs-6">
    <parent><child></child></parent>
  </div>
  <div class="col-xs-6">
    <parent><child></child></parent>
  </div>
</div>
```

Change the `ParentComponent` template to *project* the passed in content to the same place the child component used to be, like so:

```
<div class="parent">
   <p>Parent</p>
   <form novalidate>
        <div class="form-group">
        <input type="text"
               class="form-control"
               name="value"
               [(ngModel)]="service.value">
      </div>
  </form>
  <ng-content></ng-content> ①
</div>
```

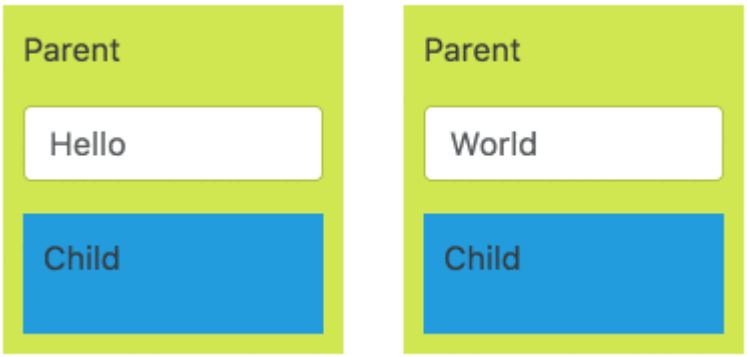
① We use content projection to *insert* the `ChildComponent` where it used to be hard coded.

Now even though child is still rendered under parent, it's considered a *content child* and not a *view child*.

Lets now change the configuration of `ParentComponent` to use `viewProviders` instead.

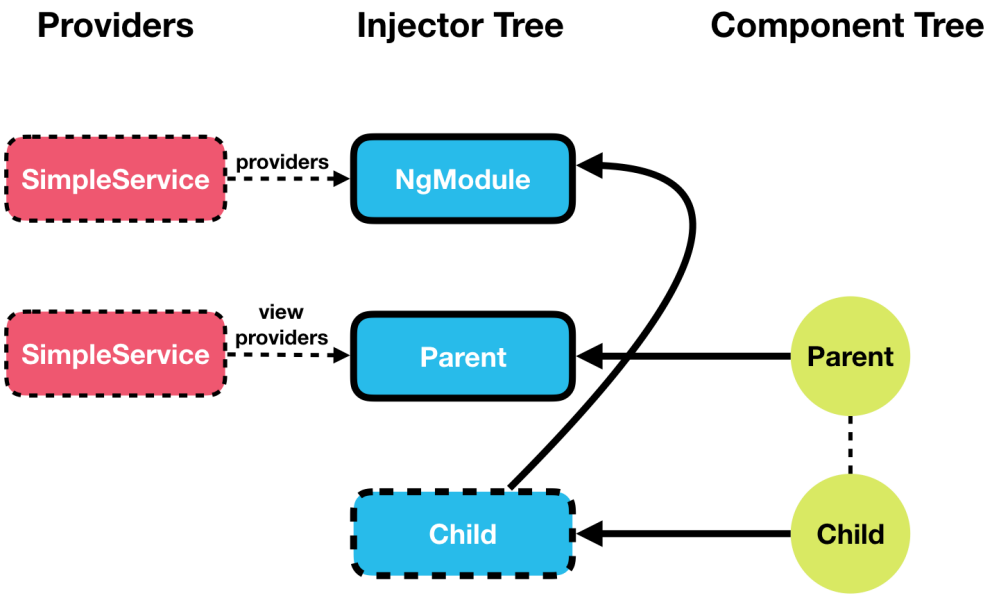
```
@Component({
 selector: 'parent',
 template: `...`,
 viewProviders: [SimpleService ]
})
class ParentComponent {
  constructor(private service: SimpleService) { }
}
```

Now when we type into the `ParentComponent` the child component doesn't update automatically.



That's because when using `viewProviders` the component creates an injector which is **only** used by the *current component* and any *view children*.

If you are a *content child*, as our child component now is, then it uses the injector in `NgModule` to resolve the dependency.



If we want to have *one* instance of a service *per* component, and shared with *only* the components view children and *not* the components content children, we configure it on the `viewProviders` property on our component decorator.

Summary

We can configure the DI framework in Angular in three main ways.

We can configure a provider on the NgModule, on a component or directives `providers` property and on a components `viewProviders` property.

Deciding where to configure your provider and understanding the different is key in understanding how to architect an Angular application.

If we want an instance of a dependency to be shared globally and share *state* across the application we configure it on the `NgModule`.

If we want a separate instance of a dependency to be shared across each instance of a component and it's children we configure it on the components `providers` property.

If we want a separate instance of a dependency to be shared across each instance of a component and only it's view children we configure it on the components `viewProviders` property.

Listing

http://plnkr.co/edit/PTyIJYIrPWqjMSNi9krS?p=preview

script.ts

```
import { NgModule, Component, Injectable } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';


class SimpleService {
  value: string;
}

@Component({
 selector: 'child',
 template: `
 <div class="child">
   <p>Child</p>
   {{ service.value }}
</div>
 `,
 styles: [`
  .child {
    background-color: #239CDE;
    padding: 10px;
  }
 `],
 // providers: [SimpleService]
})
class ChildComponent {
  constructor(private service: SimpleService) { }
}

@Component({
 selector: 'parent',
 template: `
 <div class="parent">
   <p>Parent</p>
```

```
   <form novalidate>
            <div class="form-group">
            <input type="text"
                   class="form-control"
                   name="value"
                   [(ngModel)]="service.value">
        </div>
  </form>
  <ng-content></ng-content>
</div>
 `,
 styles: [`
  .parent {
    background-color: #D1E751;
    padding: 10px;
  }
 `],
 viewProviders: [SimpleService ]
  // providers: [SimpleService]
})
class ParentComponent {
  constructor(private service: SimpleService) { }
}



@Component({
 selector: 'app',
 template: `
 <div class="row">
    <div class="col-xs-6">
        <parent><child></child></parent>
    </div>
    <div class="col-xs-6">
        <parent><child></child></parent>
    </div>
</div>
 `
})
class AppComponent {
}


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

```
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

Dependency Injection is a way of architecting an application so code is easier to re-use, easier to test and easier to maintain.

It a method of decoupling a piece of code from the dependencies it needs in order to run.

It uses *Inversion of Control* so the responsibility of creating dependencies and and passing them in to dependant pieces of code is handled by something else.

Angular comes with a Dependency Injection (DI) framework of it's own and it's used throughout Angulars code.

To understand how to use Angular and architect your application you need to have a very good understanding of the DI framework.

The DI framework in Angular consists of 4 concepts working together:

Token

This uniquely identifies something that we want injected. A *dependancy* of our code.

Dependancy

The actual code we want injected.

Provider

This is a map between a *token* and a list of *dependancies*.

Injector

This is a function which when passed a *token* returns a *dependancy* (or a list of dependencies)

We configure injectors with providers, Angular then uses these injectors to resolve dependencies using tokens and injecting them into constructors as arguments.

There are 4 types of providers, a class provider, a value provider, a factory function provider and an alias provider.

If we want a dependency to be shared across our entire application we would configure it on our `NgModule`.

If we want a separate instance of a dependency to be shared across each instance of a component and it's children we configure it on the components `providers` property.

If we want a separate instance of a dependency to be shared across each instance of a component and only it's view children we configure it on the components `viewProviders` property.

Activity

Convert the Joke application into an application that uses a service to store the jokes.

Also support a max jokes provider which we inject into this service and we can configure with different values.

Make sure the number of jokes doesn't exceed this max value.

Steps

Fork this plunker:

http://plnkr.co/edit/ByyuZPDMgXhalRuAggoE?p=preview

Complete the `JokeService` class with the required functionality.

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/cEkzoPuUrACQ6SN5ieyK?p=preview

HTTP

Overview

To build a web application these days we need to make a number of calls to external HTTP APIs.

Angular comes with its own HTTP client library which we can use to make those calls.

In JavaScript making HTTP requests is an *asynchronous* operation. It just sends the HTTP request to the API and *doesn't* wait for a response before continuing with the next line of code.

When the API responds milliseconds or seconds or minutes later then we get notified and we can start processing the response.

In Angular there are two ways of handling these asynchronous operations. We can use *Promises* which we covered in the lecture on TypeScript & ES6, or we can use *Observables* which we covered in the section on RxJS and Observables.

In this chapter you are going to learn:

- How to setup your application to make HTTP Calls.
- How to use the core HTTP APIs.
- How to architect your application to use HTTP via Promises.
- How to architect your application to use HTTP via Observables.
- What is JSONP and how to architect your application to use JSONP via Observables.

Core HTTP API

Learning Objectives

- How to setup our application so we can inject and use the Http client library.
- How to send the various types of http requests.
- How to send custom headers with our requests.
- How to handle errors.

Providing the Http dependencies

The *http client* is a service that you can *inject* into your classes in Angular, like so:

```
class MyClass {
  constructor(private http: Http) {
  }
}
```

It lives in the `@angular/http` module along side a number of other classes which are helpful when working with http.

```
import { Http, Response, RequestOptions, Headers } from '@angular/http';
```

Importing these classes isn't enough though. Like all things you can *inject*, these are dependencies and therefore must be configured via a provider on the DI framework.

We want it configured into our root `NgModule` since we want to use the http client potentially everywhere.

The immediate thought might be to add it to the providers param on our NgModule like so:

```
@NgModule({
    providers: [Http]
})
```

However the provider configuration for Http isn't so straight forward, so the Angular team have given us a handy way to configure our `NgModule` with all the required providers for Http.

We simply import something called the `HttpModule` and add it to the `imports` list in our `NgModule`, like so:

```
import { HttpModule } from '@angular/http';
.
.
.
@NgModule({
    imports: [
        HttpModule
    ]
})
```

Importing `HttpModule` into our `NgModule` configures our `NgModules` injector with all the providers needed to use Http in our app.

Demo API & sample app

To demonstrate the http client lib we will need a test API we can use, one such API is http://httpbin.org this has a number of API end points where we can test different HTTP actions.

For this lecture we'll be fleshing out this simple component. It is a series of buttons which call functions on the component with each function performing a different action using the http client lib.

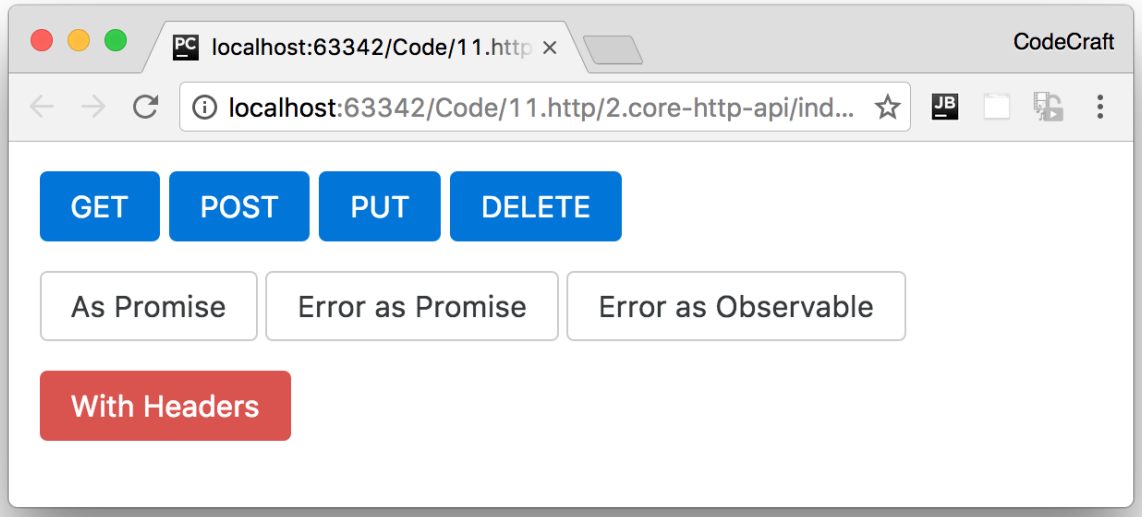
```
@Component({
 selector: 'app',
 template: `
 <div class="row">
  <div class="m-t-1">
    <button class="btn btn-primary" (click)="doGET()">GET</button>
    <button class="btn btn-primary" (click)="doPOST()">POST</button>
    <button class="btn btn-primary" (click)="doPUT()">PUT</button>
    <button class="btn btn-primary" (click)="doDELETE()">DELETE</button>
    </div>
</div>

<div class="row">
  <div class="m-t-1">
    <button class="btn btn-secondary" (click)="doGETAsPromise()">As Promise</button>
    <button class="btn btn-secondary" (click)="doGETAsPromiseError()">Error as
Promise</button>
    <button class="btn btn-secondary" (click)="doGETAsObservableError()">Error as
Observable</button>
    </div>
</div>

<div class="row">
  <div class="m-t-1">
    <button class="btn btn-danger" (click)="doGETWithHeaders()">With Headers</button>
    </div>
```

```
</div>
 `
})
class AppComponent {
  apiRoot: string = "http://httpbin.org"; ①

  constructor(private http: Http) { }  ②

  doGET() {
    console.log("GET");
  }

  doPOST() {
    console.log("POST");
  }

  doPUT() {
    console.log("PUT");
  }

  doDELETE() {
    console.log("DELETE");
  }

  doGETAsPromise() {
    console.log("GET AS PROMISE");
  }

  doGETAsPromiseError() {
    console.log("GET AS PROMISE ERROR");
  }

  doGETAsObservableError() {
    console.log("GET AS OBSERVABLE ERROR");
  }

  doGETWithHeaders() {
    console.log("GET WITH HEADERS");
  }
}
```

① We store the root of our API in a variable so we can easily refer to it in our functions.

② We've injected the `Http` service onto our component and stored it as a private property.

Running the above we wil see an application like so:



HTTP Verbs

Now we have http setup we can inject it into a class and start making HTTP calls.

In the HTTP protocol describes a set of *verbs* for each URL. For a given URL we can perform a `GET`, `PUT`, `POST`, `DELETE`, `HEAD` and `OPTIONS` request.

Lets flesh out the functions in our component to demonstrate the various features.

GET

To perform a `GET` request we simply call the `get` function on our http client. This returns an observable which for now we are just going to subscribe to and print the response to the console, like so:

```
doGET() {
  console.log("GET");
  let url = `${this.apiRoot}/get`;
  this.http.get(url).subscribe(res => console.log(res.text())); ①
}
```

① `res.text()` is whatever was returned in the body of the HTTP response from the server.

If we now press the `GET` button we see the below printed to the console

```
GET
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate, sdch",
    "Accept-Language": "en-US,en;q=0.8",
    "Cache-Control": "no-cache",
    "Host": "httpbin.org",
    "Origin": "http://evil.com/",
    "Pragma": "no-cache",
    "Referer": "http://run.plnkr.co/ODORtE0cj708R2eO/",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/53.0.2785.80 Safari/537.36"
  },
  "origin": "2.139.81.90",
  "url": "http://httpbin.org/get"
}
```

Our server returns *JSON* and although it might look like we are printing out an object we are in fact printing out a JSON formatted string.

To convert the JSON formatted string into an object we just call `res.json()` instead of `res.text()`, like so:

```
this.http.get(url).subscribe(res => console.log(res.json()));
```

This parses the JSON string into an object and so we print and an object out to the console instead, like so:

```
GET
Object {args: Object, headers: Object, origin: "2.139.81.90", url:
"http://httpbin.org/get"}
```

GET with query params

We can pass to the `get` function a set of optional query parameters as the second argument.

This is in the format of an object like structure but to help in creating our query params object we can use the helper `URLSearchParams` class like so:

```
import {URLSearchParams} from '@angular/http';
.
.
.
doGET() {
  console.log("GET");
  let url = `${this.apiRoot}/get`;
  let search = new URLSearchParams();
  search.set('foo', 'moo');
  search.set('limit', 25);
  this.http.get(url, {search: search}).subscribe(res => console.log(res.json())); ①
}
```

① We pass the query params as a second argument to the `get` function.

We can take advantage of *destructuring* in ES6. If you remember if the key and value of an object are the *same* name then you can shorten `{search: search}` into just `{ search }`, like so:

```
doGET() {
  console.log("GET");
  let url = `${this.apiRoot}/get`;
  let search = new URLSearchParams();
  search.set('foo', 'moo');
  search.set('limit', 25);
  this.http.get(url, {search}).subscribe(res => console.log(res.json()));
}
```

DELETE

To perform a *DELETE* request we just call the `delete` function. The format of the function is exactly the same as the `get` function above, we can even pass in query params like so:

```
doDELETE() {
  console.log("DELETE");
  let url = `${this.apiRoot}/delete`;
  let search = new URLSearchParams();
  search.set('foo', 'moo');
  search.set('limit', 25);
  this.http.delete(url, {search}).subscribe(res => console.log(res.json()));
}
```

POST with data

To perform a *POST* request we call the `post` function.

The format of the `post` function is similar to `get` but when we do a *POST* request we usually want to pass in data as well.

So the second parameter to `post` is not a set of query parameters but instead an object which will be passed as the *payload* for the request, like so:

```
doPOST() {
  console.log("POST");
  let url = `${this.apiRoot}/post`;
  this.http.post(url, {moo:"foo",goo:"loo"}).subscribe(res =>
console.log(res.json()));
}
```

this prints out:

```
POST
Object {args: Object, data: "{    "moo": "foo",    "goo": "loo"  }", files: Object,
form: Object, headers: Object⋯}
```

If we wanted to post to the url but add query params we can do the same as we did with `get` but we pass the query params in the third param instead, like so:

```
doPOST() {
  console.log("POST");
  let url = `${this.apiRoot}/post`;
  let search = new URLSearchParams();
  search.set('foo', 'moo');
  search.set('limit', 25);
  this.http.post(url, {moo:"foo",goo:"loo"}, {search}).subscribe(res =>
console.log(res.json()));
}
```

PUT with data

To perform a PUT request we just call the `put` function. It works in exactly the same was as the `post` function above:

```
doPUT() {
  console.log("PUT");
  let url = `${this.apiRoot}/put`;
  let search = new URLSearchParams();
  search.set('foo', 'moo');
  search.set('limit', 25);
  this.http.put(url, {moo:"foo",goo:"loo"}, {search}).subscribe(res =>
console.log(res.json()));
}
```

Promises

If you prefer to work with promises over observables it's easy to convert between the two.

We just call `.toPromise()` on the observable that gets returned and this will convert it into a promise instead, like so:

```
doGETAsPromise() {
  console.log("GET AS PROMISE");
  let url = `${this.apiRoot}/get`;
  this.http.get(url)
    .toPromise()
    .then(res => console.log(res.json()));
}
```

If you just ran the above as is you would get this error:

```
this.http.get(...).toPromise is not a function
```

That's because the `toPromise()` function is an observable operator and to reduce the size of your application the Angular team opted not to include all the observable operators by default, instead we need to explicitly import each operator ourselves, like so:

```
import 'rxjs/add/operator/toPromise';
```

If we wanted to import *all* RxJS operators we can import:

```
import 'rxjs/RX';
```

```
 +
 +
But the above will _unnecessarily_ bloat the application so isn't
recommended for production.
```

Running the above promise based application results in the same information being printed to the console as the previous observable based function.

Handling errors

We can simulate errors from our test server just by performing a `GET` on a `POST` endpoint, like so:

```
doGETAsPromiseError() {
  console.log("GET AS PROMISE ERROR");
  let url = `${this.apiRoot}/post`; ①
  this.http.get(url)
    .toPromise()
    .then(res => console.log(res.json()));
}
```

① The API is for a POST request but we are performing a GET.

Whether we are handling the response as an Observable or as a Promise we can add error handling function as the second param, so in the case of a promise it looks like so:

```
doGETAsPromiseError() {
  console.log("GET AS PROMISE ERROR");
  let url = `${this.apiRoot}/post`;
  this.http.get(url)
    .toPromise()
    .then(
      res => console.log(res.json()),
      msg => console.error(`Error: ${msg.status} ${msg.statusText}`) ①
    );
}
```

① Error handler is the second argument to `then`

With *Promises* we can also use the `.catch()` handler to handle errors, see the section on ES6 & TypeScript for more information.

The above prints out

```
Error: 405 METHOD NOT ALLOWED
```

And in the case of an observable it looks like so:

```
doGETAsObservableError() {
  console.log("GET AS OBSERVABLE ERROR");
  let url = `${this.apiRoot}/post`;
  this.http.get(url).subscribe(
    res => console.log(res.json()),
    msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
  );
}
```

Status codes

If you are looking for a refresher on what each HTTP status codes mean this site is a good reference: https://httpstatuses.com/

Headers

HTTP headers are bits of meta-data which the browser attaches to your HTTP requests when it sends them to the server. Things like your IP address, the type of browser you are using and so on are added to your headers.

In fact the test API we are using echos the headers our browser sends in the response it sends back - so we can see what was sent.

Some APIs require that we send some *custom* headers with our requests and we can do that easily with the Angular http client.

To send headers with our requests we first need to import two helper classes from the http module.

```
import { Headers, RequestOptions } from '@angular/http';
```

For our sample lets send a basic *Authorization* header. This is just a header called `Authorization` with a value that is a username and password converted to base64, no need to worry too much about the specifics just understand that the built in javascript function `btoa` converts a `string` to `base64`.

This is a really poor way to authenticate with your server, we are just using it as an example. We covert `username:password` to a base64 string and pass that to the server via a header called `Authorization`

The full listing for sending a http request with header is like so:

```
doGETWithHeaders() {
  console.log("GET WITH HEADERS");
  let headers = new Headers();
  headers.append('Authorization', btoa('username:password'));
  let opts = new RequestOptions();
  opts.headers = headers;
  let url = `${this.apiRoot}/get`;
  this.http.get(url, opts).subscribe(
    res => console.log(res.json()),
    msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
  );
}
```

We first create an instance of headers and add our specific header to the mix, like so:

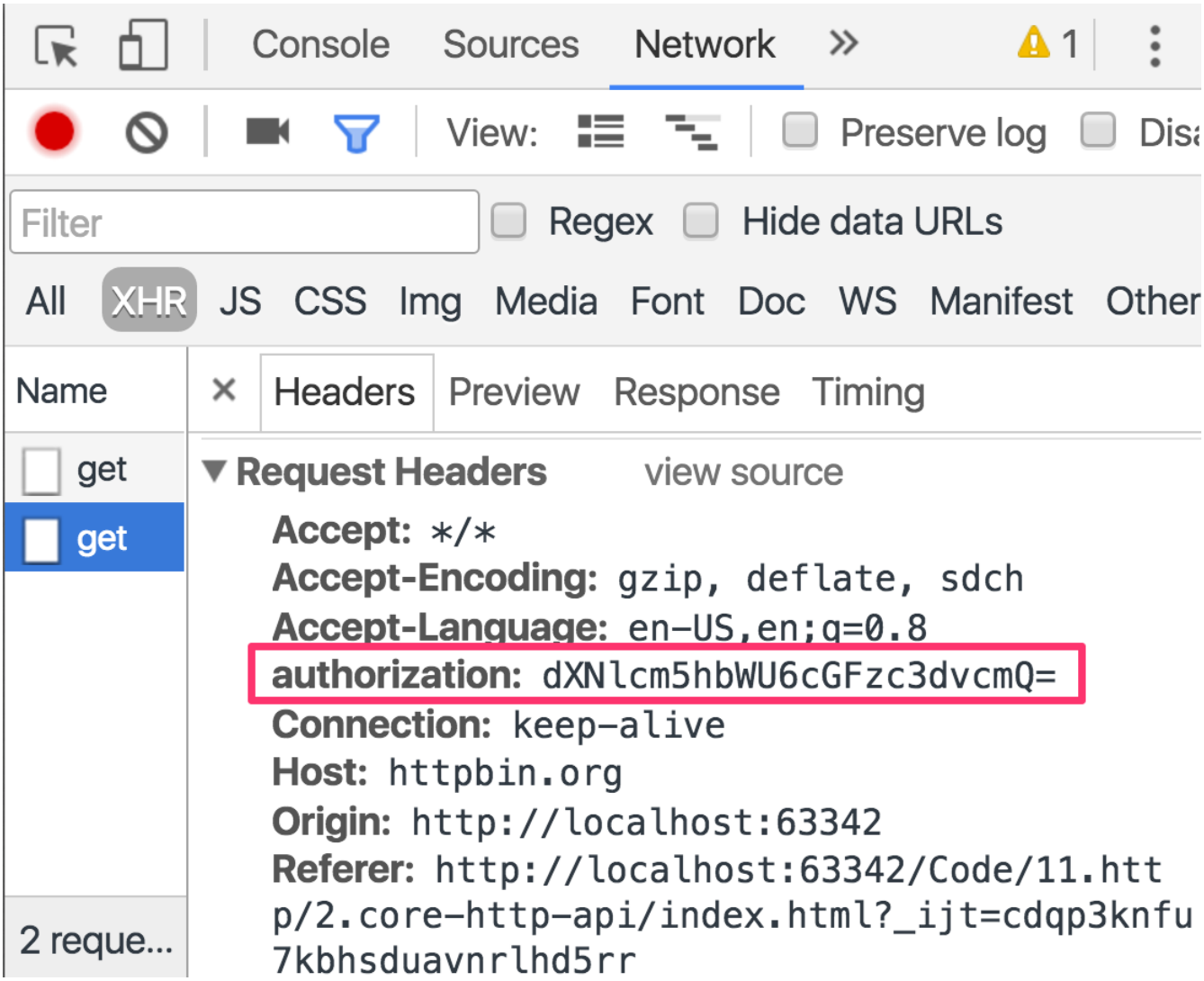
```
let headers = new Headers();
headers.append('Authorization', btoa('username:password'));
```

We then create a request options and add our headers as a property of those options.

```
let opts = new RequestOptions();
opts.headers = headers;
```

And finally we pass the options to the appropriate http function, in this example we are using `get` and for `get` the options are passed as the *second* parameter, for `put` and `post` however it's the *third* parameter

And now when we press the `GET WITH HEADERS` button and look at the network panel we can see that the authorisation header was sent along with our request.



Summary

We inject the Http client library into our classes, it's a dependency that needs to be configured in Angulars DI framework.

Rather than separately setup each provider for all the different parts of the Http client library we can instead import the `HttpModule` and add to our `NgModule` imports list. This sets up the injector on our `NgModule` with all the necessary providers.

We make requests for all the HTTP verbs by calling matching functions on our Http client library.

By default these requests return observables which we can subscribe to, we can also convert these observables into promises and handle the asynchronous responses that way.

We handle errors using the typical observable or promise error handlers.

We send custom headers alongside our requests by passing in an appropriately configured `RequestOptions` object to our http functions.

This lecture covered the basics of using the http client library, in the next lecture we'll go through a specific example using promises.

Listing

script.ts

```
import {NgModule, Component} from '@angular/core';
import {HttpModule, Http, URLSearchParams, Headers, RequestOptions} from
'@angular/http';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
// import 'rxjs/add/operator/toPromise';
import 'rxjs/Rx';


@Component({
  selector: 'app',
  template: `
 <div class="row">
  <div class="m-t-1">
    <button class="btn btn-primary" (click)="doGET()">GET</button>
    <button class="btn btn-primary" (click)="doPOST()">POST</button>
    <button class="btn btn-primary" (click)="doPUT()">PUT</button>
    <button class="btn btn-primary" (click)="doDELETE()">DELETE</button>
    </div>
</div>

<div class="row">
  <div class="m-t-1">
    <button class="btn btn-secondary" (click)="doGETAsPromise()">As Promise</button>
    <button class="btn btn-secondary" (click)="doGETAsPromiseError()">Error as
Promise</button>
    <button class="btn btn-secondary" (click)="doGETAsObservableError()">Error as
Observable</button>
    </div>
</div>
```

```
<div class="row">
  <div class="m-t-1">
    <button class="btn btn-danger" (click)="doGETWithHeaders()">With Headers</button>
    </div>
</div>
 `
})
class AppComponent {
  apiRoot: string = "http://httpbin.org";

  constructor(private http: Http) {
  }

  doGET() {
    console.log("GET");
    let url = `${this.apiRoot}/get`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.get(url, {search}).subscribe(res => console.log(res.json()));
  }

  doPOST() {
    console.log("POST");
    let url = `${this.apiRoot}/post`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.post(url, {moo: "foo", goo: "loo"}, {search}).subscribe(res =>
console.log(res.json()));
  }

  doPUT() {
    console.log("PUT");
    let url = `${this.apiRoot}/put`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.put(url, {moo: "foo", goo: "loo"}, {search}).subscribe(res =>
console.log(res.json()));
  }

  doDELETE() {
    console.log("DELETE");
    let url = `${this.apiRoot}/delete`;
    let search = new URLSearchParams();
    search.set('foo', 'moo');
    search.set('limit', 25);
    this.http.delete(url, {search}).subscribe(res => console.log(res.json()));
  }
```

```
  doGETAsPromise() {
    console.log("GET AS PROMISE");
    let url = `${this.apiRoot}/get`;
    this.http.get(url)
        .toPromise()
        .then(res => console.log(res.json()));
  }

  doGETAsPromiseError() {
    console.log("GET AS PROMISE ERROR");
    let url = `${this.apiRoot}/post`;
    this.http.get(url)
        .toPromise()
        .then(
            res => console.log(res.json()),
            msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
        );
  }

  doGETAsObservableError() {
    console.log("GET AS OBSERVABLE ERROR");
    let url = `${this.apiRoot}/post`;
    this.http.get(url).subscribe(
        res => console.log(res.json()),
        msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
    );
  }

  doGETWithHeaders() {
    console.log("GET WITH HEADERS");
    let headers: Headers = new Headers();
    headers.append('Authorization', btoa('username:password'));
    let opts: RequestOptions = new RequestOptions();
    opts.headers = headers;
    let url = `${this.apiRoot}/get`;
    this.http.get(url, opts).subscribe(
        res => console.log(res.json()),
        msg => console.error(`Error: ${msg.status} ${msg.statusText}`)
    );
  }
}


@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
class AppModule {
```

```
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

HTTP Example with Promises

In this lecture we are going to create a mini application which lets us search iTunes for music.

We'll be using the freely available iTunes API to perform the search: https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/

Learning Objectives

- What is CORS and how to temporarily bypass this security feature in your browser.
- How to create an intermediate service to handle our Http requests to an API endpoint.
- How to provide the service and consume it in components.
- How to convert the responses into instances of a domain model.
- How to handle asynchronous work by using promises.

Cross Origin Resource Sharing

To use this API we need to deal with a thorny issue called *CORS*, Cross Origin Resource Sharing.

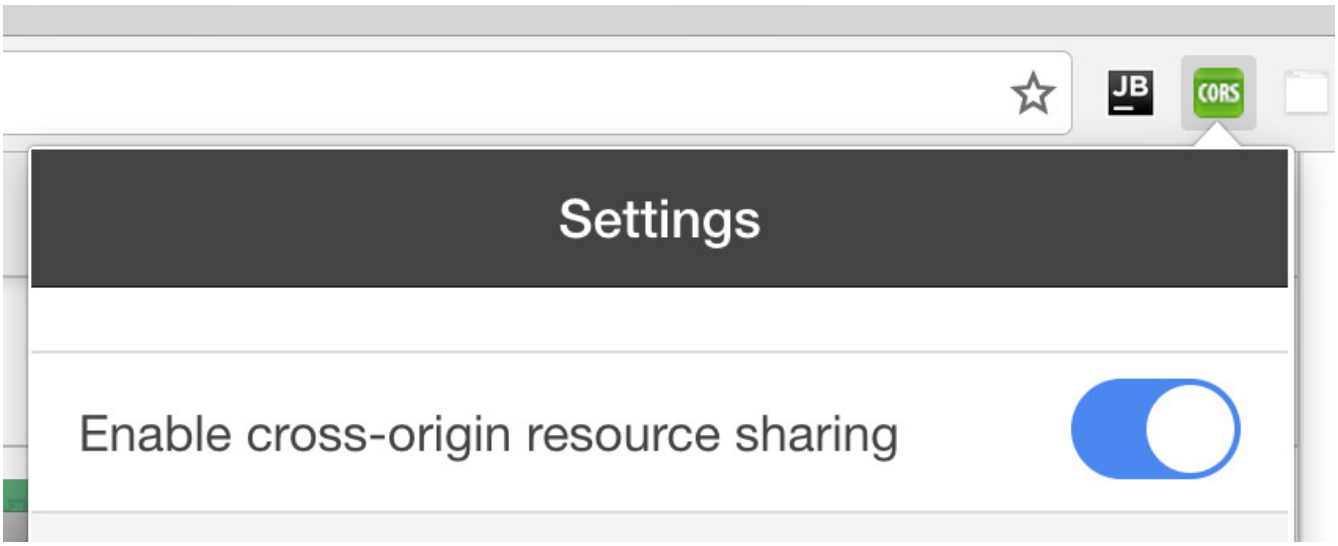
This is a security measure implemented in all browsers that stops you from using an API in a potentially unsolicited way and most APIs, including the iTunes API, are protected by it.

Because of CORS if we tried to make a request to the iTunes API url with the http client library the browser would issue a *CORS* error.

Chrome Workaround

For chrome I prefer to use a chrome plugin which lets us switch CORS on and off with the click of a button, install the plugin from this link: https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi

This adds a button to your chrome toolbar, make sure it is enabled and green like so:

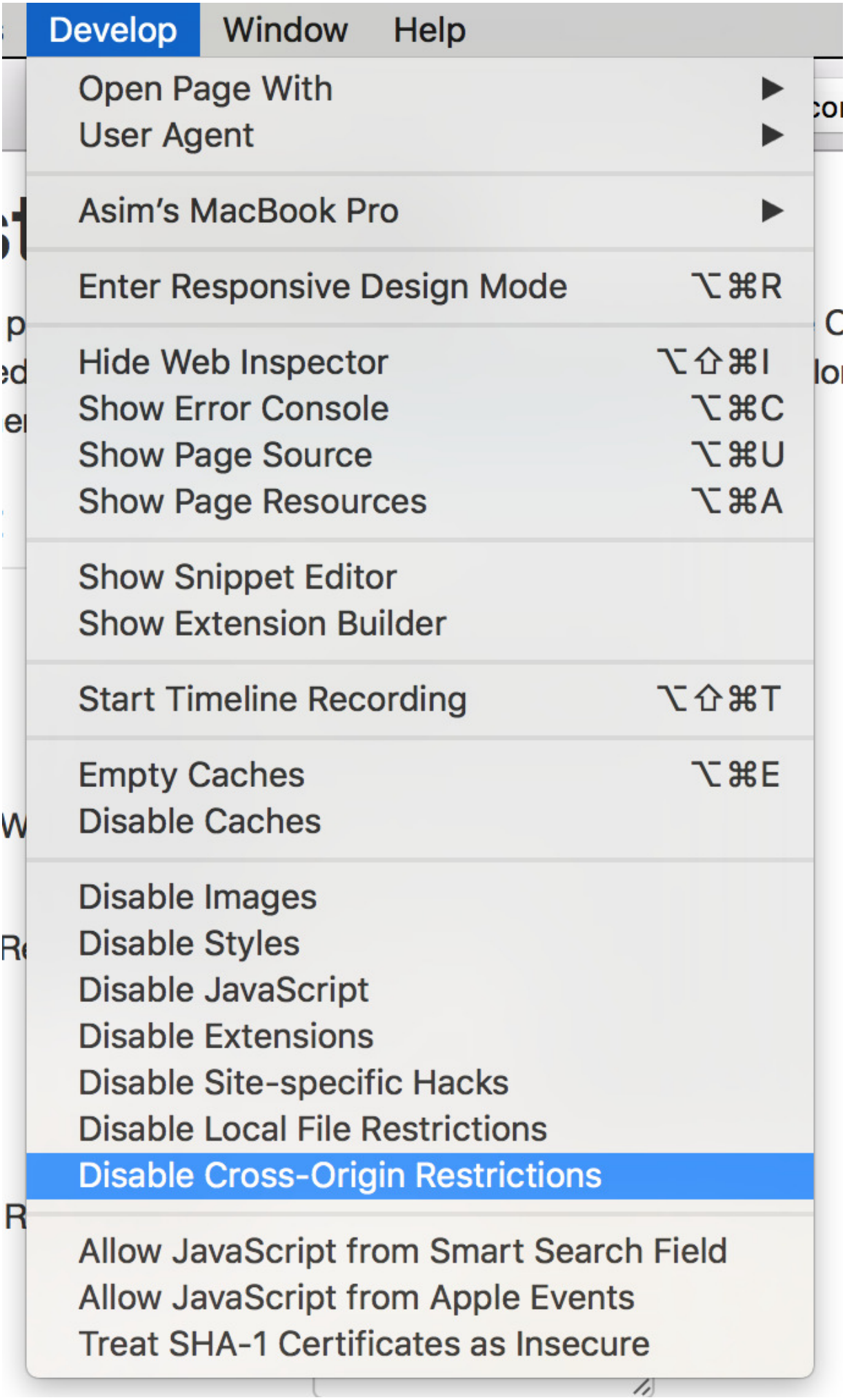


Firefox Workaround

If you are using FireFox you can try this plugin: https://addons.mozilla.org/en-Gb/firefox/addon/cors-everywhere/

Safari Workaround

If you are using Safari start by enabling the *Develop* menu from Preferences → Advanced. When this is done you may need to restart Safari. In the *Develop* menu make sure that *Disable Cross-Origin Restrictions* is checked, you will need to refresh the page for the changes to take effect.


Develop
Window
Help
Open Page With
User Agent
Asim's MacBook Pro
Enter Responsive Design Mode ⌥⌘R
Hide Web Inspector ⌥⇧⌘I
Show Error Console ⌥⌘C
Show Page Source ⌥⌘U
Show Page Resources ⌥⌘A
Show Snippet Editor
Show Extension Builder
Start Timeline Recording ⌥⇧⌘T
Empty Caches ⌥⌘E
Disable Caches
Disable Images
Disable Styles
Disable JavaScript
Disable Extensions
Disable Site-specific Hacks
Disable Local File Restrictions
Disable Cross-Origin Restrictions
Allow JavaScript from Smart Search Field
Allow JavaScript from Apple Events
Treat SHA-1 Certificates as Insecure

Creating the app component

First lets create a simple app component with our search field:

```
@Component({
  selector: 'app',
  template: `
<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search> ①
  </div>
  <button class="btn btn-primary"
          (click)="doSearch(search.value)"> ②
          Search
  </button>
</form>
 `
})
class AppComponent {
  constructor() { }

  doSearch(term:string) {
  }
}
```

① This input control has a template reference variable called `search`.

② Pressing the *Search* button calls the doSearch() function and passes it the search term.

We have a form input control with a search button. When someone presses the search button it calls the `doSearch` function with the *search term* we want to search the iTunes database with.

Creating a search service

So far in this section we've used the http client client library directly from a component by injecting it into the components constructor.

However the recommended method is to create an *intermediate* service which uses the Http client library to make the requests. Instead of returning *raw* json data the service converts the response into one or more concrete instances of classes, into something called a domain model.

This makes it easier to test our component later on. We don't have to *mock* raw HTTP calls but instead, we can define test data as instances of our domain model, more on that later in our lecture on testing.

So we are going to create our own *intermediate* service which I am going to call `SearchService`.

```
@Injectable() ①
class SearchService {
  apiRoot:string = 'https://itunes.apple.com/search';
  results:Object[];
  loading:boolean;

  constructor(private http:Http) { ②
    this.results = [];
    this.loading = false;
  }

  search(term:string) {
  }
}
```

① We decorate the class with `@Injectable` so Angular knows it should inject the Http service into the constructor.

② The http client is injected into our `SearchService` in the constructor.

We also *provide* our `SearchService` and make it available globally via the `NgModule` providers list.

Using promises

The `search` function is going to make an *asynchronous* call using the http client lib to the iTunes API.

There are two ways we like to handle asynchronous functions in Angular one is via *Promises* and the other via *Observables*.

In this lecture we will use *Promises* and in the next we'll solve the same problem using *Observables*.

Since we are using promises we need the `search` function to return a `promise`, like so:

```
search(term:string) {
  let promise = new Promise((resolve, reject) => {
    //TODO
  });
  return promise;
}
```

If you remember from the section on ES6 and TypeScript we can use promises by returning an instance of a `Promise` class.

We put the code for our search function where the `TODO` comment lives.

When the http response arrives from the iTunes API, we finish our processing and call the `resolve()` function. This lets any interested parties know the asynchronous task is complete and let

them perform any follow on tasks.

If the http response returned an error we call the `reject()` function which again lets any interested parties know there was an error.

So now lets flesh out this function with our http `get` request, like so:

```
search(term:string) {
  let promise = new Promise((resolve, reject) => {
    let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
    this.http.get(apiURL)
      .toPromise()
      .then(
        res => { // Success
          console.log(res.json());
          resolve();
        }
      );
  });
  return promise;
}
```

First we construct a url which will instruct iTunes to perform a search for us, like so:

```
let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
```

Then we make a *GET* request to that url with:

```
this.http.get(apiURL)
```

It return an *Observable*, we are working with *Promises* in this example so lets convert it to a *Promise* with:

```
.toPromise()
```

Remember to add this import to the top of the file `import 'rxjs/add/operator/toPromise';`

When the API responds it calls the function we pass to `then` with a `Response` object and we are just printing out the json for now.

```
.then(
  res => { // Success
  console.log(res.json());
  }
);
```

Consuming our search service

To use the `SearchService` we need to configure it as a provider on our `NgModule`

```
@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule { }
```

We also need to inject it into our `AppComponent` and call the `itunes.search()` function when the user presses the Search button

```
class AppComponent {
  constructor(private itunes:SearchService) { }

  doSearch(term:string) {
    this.itunes.search(term)
  }
}
```

Now when we type in *Moo* into the input control and press *Search* this calls the API with https://itunes.apple.com/search?term=moo&media=music&limit=20 and prints the results to the console, like so:

```
Object {resultCount: 20, results: Array[20]}
```

We are interested only in the `results` property, this is the list of search results.

Rather than return the search results to the component, lets instead *store* the results on the `SearchService` to make it easier to share the results between components and at the same time lets add an error handler, like so:

```
search(term:string) {
  let promise = new Promise((resolve, reject) => {
    let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
    this.http.get(apiURL)
      .toPromise()
      .then(
        res => { // Success
        this.results = res.json().results;
        resolve();
        },
        msg => { // Error
        reject(msg);
        }
      );
  });
  return promise;
}
}
```

We also call the `resolve()` and `reject()` functions above after we have finished processing the callback from iTunes so the calling function gets notified and can perform any post processing.

Lets now render these results in our app component.

We are keeping the data on the iTunes service and to loop over them we use an `NgFor` directive like so:

```
<ul>
  <li *ngFor="let track of itunes.results">
    {{track.trackName}}
  </li>
</ul>
```

Now when we run the app and type in *Moo* we get a set of track returned and for each one we print out the track name.

That's great but we can do better, lets use the twitter bootstrap list ui style and also show the track image and link to the track on iTunes.

The HTML below might look complex but it's just standard twitter bootstrap layout.

```
<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let track of itunes.results">
    <img src="{{track.artworkUrl30}}">
    <a target="_blank"
       href="{{track.trackViewUrl}}">{{ track.trackName }}
    </a>
  </li>
</ul>
```

That's it now when we run the application and search for *Moo* we get something that looks like this:

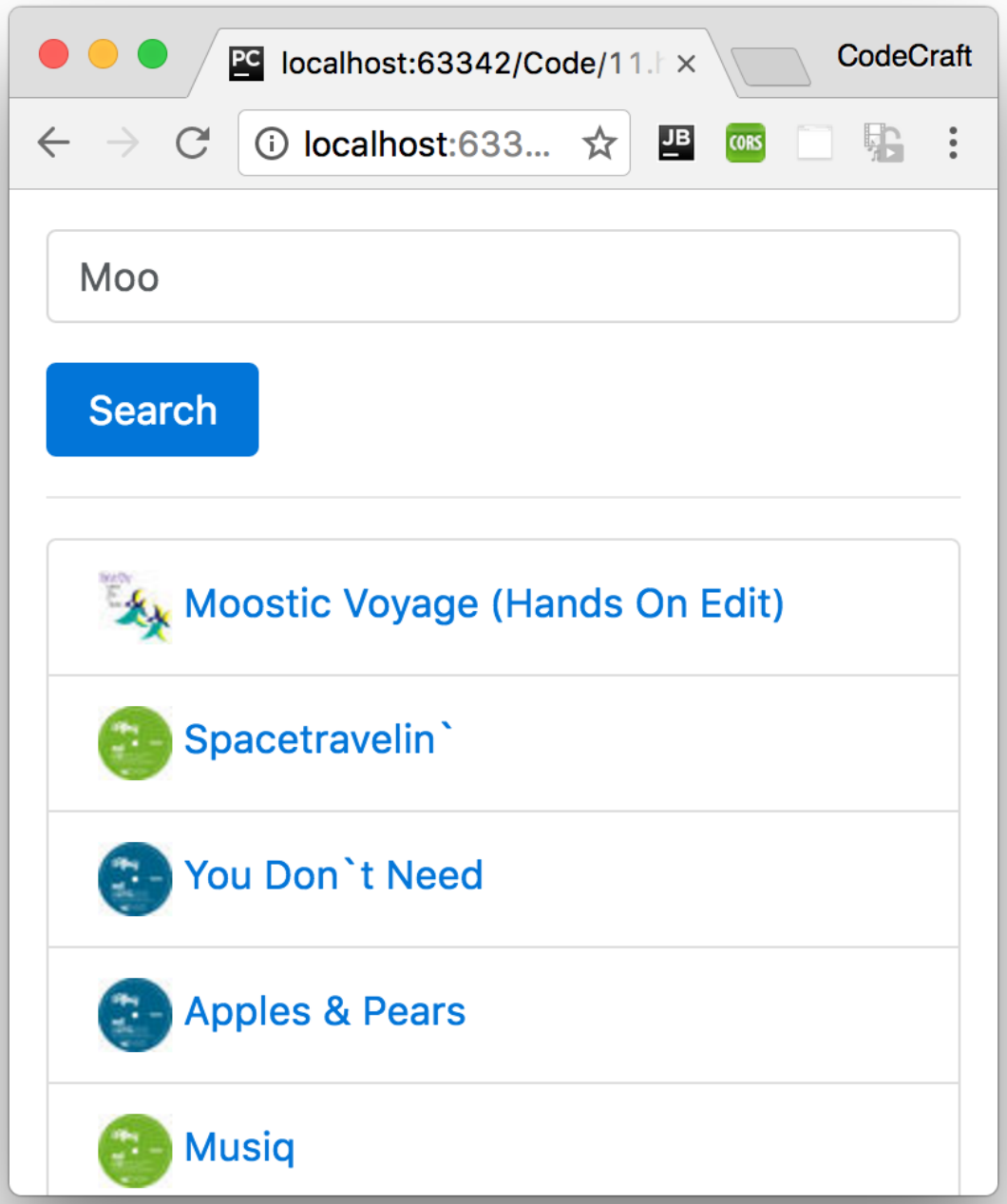


Figure 4. iTunes Track Search App

Adding a loading indicator

So now it's all working but there is one feature which is important when doing any asynchronous work and that's to show a loading indicator to the end user, so they know the application is working and not *broken*.

Lets show the text `"Loading···"` when we are still waiting for iTunes to return a response.

We create a boolean called `loading` on our `AppComponent` and then conditionally show a `p` tag if

`loading` is `true`, like so:

```
<div class="text-center">
  <p class="lead" *ngIf="loading">Loading...</p>
</div>
```

We initialise loading to `false` so it doesn't show when we first load the page, then we set it to `true` when we call the `doSearch()` function so it does show.

```
class AppComponent {
  private loading: boolean = false;

  constructor(private itunes:SearchService) { }

  doSearch(term:string) {
    this.loading = true;
    this.itunes.search(term)
  }
}
```

The above logic would show the `"Loading···"` message but we have *no* way of setting the `loading` boolean to `false` again, we have no way to *remove* the `"Loading···"` message.

We need to call some code when all the data has been returned by the iTunes API.

The `SearchService` already returns a promise and calls resolve on that promise when this happens. To hook into that we add a `then` handler onto the returned promise and set the `loading` boolean to `false` there, like so:

```
doSearch(term:string) {
  this.loading = true;
  this.itunes.search(term).then( () => this.loading = false)
}
```

Now when the `itunes.search()` function completes we set `loading` to `false` and the `"Loading···"` message disappears.

Using a domain model

We now have a functioning app.

However the format we are storing data in is just a plain json object, its far better to parse our raw json data into instances of classes, a domain model.

First lets create a class called `SearchItem` which can encapsulate the fields we are interested in.

```
class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}
```

And then in our `SearchService` lets parse the JSON and store instances of `SearchItems` instead.

A good method of achieving this is to use the `map` array function, this transforms each element in the array by passing it through a function like so:

```
this.results = res.json().results.map(item => { ①
  return new SearchItem(
      item.trackName,
      item.artistName,
      item.trackViewUrl,
      item.artworkUrl30,
      item.artistId
  );
});
```

① We loop through the `results` array, calling the function passed into map on each item. This function converts the raw JSON item into an instance of a `SearchItem` instead.

At the same time lets be explicit and set the type of the results property on our `SearchService` to `SearchItem[]`.

The full listing for our `SearchService` now looks like:

```
@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];
  loading: boolean;

  constructor(private http: Http) {
    this.results = [];
    this.loading = false;
  }

  search(term: string) {
    let promise = new Promise((resolve, reject) => {
      let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
      this.http.get(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                // this.results = res.json().results;
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
    return promise;
  }
}
```

The results property is now an array of `SearchItems`.

The names of the properties on our model have also changed, for instance `trackName` in the raw JSON returned from iTunes is now `track` on our `SearchItem` so we need to change our template bindings to match, like so:

```
<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    <a target="_blank"
       href="{{track.link}}">{{ track.name }}
    </a>
  </li>
</ul>
```

Summary

The recommended method to interact via a Http service is by creating an intermediate service which has the responsibly of communicating with the API and converting the raw data into one or more domain models.

In this lecture we handled asynchronous code by using promises. By converting the observable returned from he http client lib into a promise and also by returning and resolving our *own* promise from `SearchService`.

In the next lecture we'll look at how we can implement the same solution by using observables.

Listing

http://plnkr.co/edit/QiQ06ttauD5Iil6C9ZsZ?p=preview

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {HttpModule, Http, Response} from '@angular/http';
import 'rxjs/Rx';

class SearchItem {
  constructor(public track: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}


@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];
```

```
  loading: boolean;

  constructor(private http: Http) {
    this.results = [];
    this.loading = false;
  }

  search(term: string) {
    let promise = new Promise((resolve, reject) => {
      let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
      this.http.get(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                // this.results = res.json().results;
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
    return promise;
  }
}


@Component({
  selector: 'app',
  template: `
<form class="form-inline">
    <div class="form-group">
        <input type="search"
               class="form-control"
               placeholder="Enter search string"
               #search>
    </div>
    <button type="button" class="btn btn-primary"
(click)="doSearch(search.value)">Search</button>
</form>

<hr/>
```

```
<div class="text-center">
  <p class="lead" *ngIf="loading">Loading...</p>
</div>

<ul class="list-group">
    <li class="list-group-item"
        *ngFor="let track of itunes.results">
        <img src="{{track.thumbnail}}">
        <a target="_blank"
           href="{{track.link}}">{{ track.track }}
        </a>
    </li>
</ul>
 `
})
class AppComponent {
  private loading: boolean = false;

  constructor(private itunes: SearchService) {
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
  }
}

@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

HTTP Example with Observables

In the previous lecture we architected an application which made HTTP calls and handled all asynchronous work by using *Promises*.

In this lecture we are going to implement exactly the same application but using *Observables* instead.

Please note we still need to use the CORS plugin, or one of the other work arrounds, that we discussed in the previous lecture.

Learning Objectives

Our goal in this lecture is not to just replace *Promises* with *Observables* but instead to go a deeper and implement most of our functionality with an *Observable Chain*.

Working with Observables does require a different way of thinking about programming, It's my hope that after this lecture you will have a much better idea of how to solve similar problems with RxJS and observables.

Returning an Observable from the service

In the promise example we stored the returned results on the service itself in the `results` propery.

In this observable example we are instead going to make the `search` function *return* an observable which the AppComponent is going to subscribe to, like so:

```
import {Observable} from 'rxjs';
.
.
.
search(term:string): Observable<SearchItem[]> { .. }
```

The return type is `Observable<SearchItem[]>`, it's going to return an observable where each item in the observable is `SearchItem[]`, each item in the observable is going to be an *array* of `SearchItems`.

That's our intention at least but we need to adjust our search function to make it a reality, a first step would look like so:

```
search(term:string): Obervable<SearchItem[]> {
  let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
  return this.http.get(apiURL)
}
```

We've removed the Promise code and since `http.get(⋯)` returns an `Observable`, specifically an `Observable` of `Response` types so the type would be `Observable<Response>`, we just return that instead.

However the above has two problems:

1. The return type of `http.get(…)` is `Observable<Response>` and not `Observable<SearchItem[]>`
2. It's missing the code to convert the raw json to our `SearchItem` domain model.

Basically we still need to convert the `Response` to an array of `SearchItems`.

We can do that with our Observable by running a `map` operation where we convert each `Response` to an array of `SearchItems`, like so:

```
search(term: string): Observable<SearchItem[]> {
  let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
  return this.http.get(apiURL) ①
      .map(res => { ②
        return res.json().results.map(item => { ③
          return new SearchItem( ④
              item.trackName,
              item.artistName,
              item.trackViewUrl,
              item.artworkUrl30,
              item.artistId
          );
        });
      });
}
```

① `this.http.get(apiURL)` returns an `Observable`.

② `map` is an observable operator which calls a function for each item on its input stream and pushes the result of the function to it's output stream. In this case each input item is a `Response` object.

③ We loop over each item in the `results` property of the `Response` object and transform the item via a function.

④ We convert the raw data returned from the iTunes API into an instance of `SearchItem`

`map` in this instance is an operator from RxJS so to use it we need to explicitly import it via:

```
import 'rxjs/add/operator/map';
```

The end result is that the code above converts the `Observable<Response>` that gets returned from `this.http.get(…)` to an `Observable<SearchItem[]>` which it then returns to the caller.

The caller in our case is the `AppComponent` so we need to change that to work with `Observables` as well.

One way to use this `Observable` in our `AppComponent` would be just to subscribe to it and store the

results locally, like so:

```
class AppComponent {
  private loading: boolean = false;
  private results: SearchItem[];

  constructor(private itunes:SearchService) { }

  doSearch(term:string) {
    this.loading = true;
    this.itunes.search(term).subscribe( data => {
      this.loading = false;
      this.results = data ①
    });
  }
}
```

① We subscribe to the `Observable<SearchItem[]>` returned from the service and store each `SearchItem[]` to our local `results` property.

We also need to change our template to loop over the local `results` property and not the old `itunes.results` property.

```
<div  *ngFor="let item of results"> ... </div>
```

Running the above application works, we can congratulate ourselves for using Observables with HTTP request, but we can go deeper.

Using the `async` pipe

The above is a good start but we are not really using `Observables` to their full extent, for one thing we are subscribing to the observable and storing the results locally on the component, we can skip that by just using the `async` pipe in our template like so:

```
class AppComponent {
  private loading: boolean = false;
  private results: Observable<SearchItem[]>;

  constructor(private itunes:SearchService) { }

  doSearch(term:string) {
    this.loading = true;
    this.results = this.itunes.search(term); ①
  }
}
```

① `results` now stores the `Observable` itself and not an array of `SearchItems`. We don't need to

subscribe but simply store the result of calling `itunes.search`.

To make the above work we need to use the `async` pipe in our template, like so:

```
<div  *ngFor="let item of results | async"> ... </div>
```

Observables all the way down

Pressing a *Search* button every-time we want to make a search is so *2012*, lets change our application so it uses a *reactive* form and performs a search as we type. We've covered reactive forms before so we just need to change our `AppComponent` to add a search field, like so:

```
<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           [formControl]="searchField">
  </div>
</form>
```

We also change our component to setup a form model, like so:

```
class AppComponent {
  private loading: boolean = false;
  private results: Observable<SearchItem[]>;
  private searchField: FormControl;

  constructor(private itunes:SearchService) { }

  ngOnInit() {
    this.searchField = new FormControl();
    this.searchField.valueChanges
      .debounceTime(400)
      .distinctUntilChanged()
      .subscribe(); // Need to call subscribe to make it hot!
  }

  doSearch(term:string) {
    this.itunes.search(term)
  }
}
```

To summarise the changes:

1. We've added a `searchField FormControl` to our `AppComponent` properties

```
private searchField: FormControl;
```

2. We've linked our `searchField` to our template form element with a `[formControl]` directive

```
[formControl]="searchField"
```

3. To support the above we've imported code from the form module and added them to our `NgModules` imports.

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

@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    HttpModule,
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
```

4. We've implemented the `ngOnInit` function so we can work with our search control once it's been linked to the form element.

```
ngOnInit() { ... }
```

5. We now `subscribe` to the observable our `searchField` exposes via the `valuesChanged` property, we use `debounceTime` and `distintUntilChanged` so we only get notified when the user *really* wants us to make a query.

```
this.searchField.valueChanges
  .debounceTime(400)
  .distinctUntilChanged()
  .subscribe(); // Need to call subscribe to make it hot!
```

6. To enable the use of these operators we've explicitly included them from RxJS

```
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
```

Linking two observables together

We now have an `Observable` on the `AppComponent` which emits a search term every time we want to perform a search.

We also have the `SearchService` returning an `Observable` via the search function with the results of performing a search.

How do we link these two together?

The `valueChanges` observable on our `searchField` is of type `Observable<string>`

Through a chain of operators we want to convert that `Observable<string>` into an `Observable<SearchItem[]>`.

To start with lets use the `map` operator, like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .map( term => this.itunes.search(term)) ①
    // Need to call subscribe to make it hot!
    .subscribe( value => console.log(value));
}
```

① We call the search service for every emit of a search term on the input stream.

If we ran the above and looked at the logs, instead of seeing see an array of `SearchItems` printing out we see something that looks like an `Observable`, like so:

```
Observable {_isScalar: false, source: Observable, operator: MapOperator}
```

What's happening is that `itunes.search(term)` isn't returning `SearchItem[]` it's returning `Observable<SearchItem[]>`.

So the map operator *isn't* converting a `string` to `SearchItem[]` it's converting a `string` to `Observable<SearchItem[]>`.

So the subscribe function is receiving `Observable<SearchItem[]>` and not `SearchItem[]` as we want.

One workaround would then be to just try doing two subscribes, like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .map( term => this.itunes.search(term))
    .subscribe( value => { ①
      value.subscribe( other => console.log(other) ) ②
    });
}
```

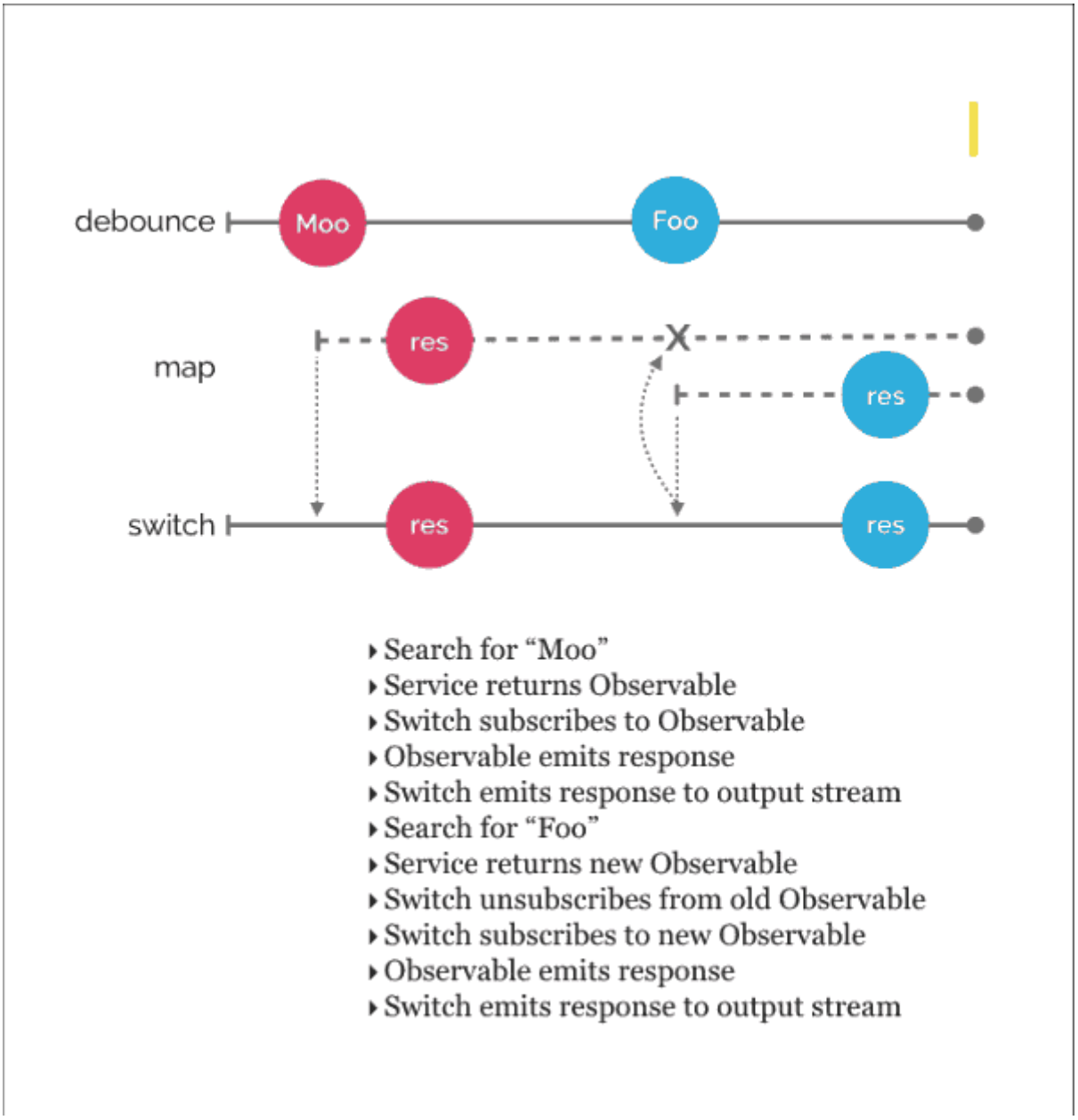
① First `subscribe` receives `Observable<SearchItem[]>`.

② We call subscribe again on each of these observables to get the `SearchItem[]`.

This is a common problem with `Observables` so there is a better way, we can use another operator called `switch`.

`switch` expects a stream of `Observables`, when it get an `Observable` pushed onto it's input stream it *unsubscribes* from any previous `Observables` and *subscribes* to the new one and then emits any values from that `Observable` onto it's output stream.

To put it another way it coverts `Observable<Observable<SearchItem[]>>` into `Observable<SearchItem[]>` which is exactly the problem we are solving with the two `subscribes` above.



Using `switch` with `map` is *such* a common occurrence that there is a combined operator called `switchMap`, which we'll use

Our code now looks like this:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .switchMap( term => this.itunes.search(term))
    .subscribe( value => console.log(value));
}
```

To use `switchMap` we need to import it from the RxJS library, like so:

```
import 'rxjs/add/operator/switchMap';
```

Rendering results

Now that our observable chain returns `Observable<SearchItem[]>` we can just assign it to our local results property, like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.results = this.searchField.valueChanges ①
    .debounceTime(400)
    .distinctUntilChanged()
    .switchMap( term => this.itunes.search(term));
}
```

① Store `Observable<SearchItem[]>` on `results`.

Remember in our template we are using the `async` pipe. This does a `subscribe` on our behalf so we also removed the `subscribe` from our chain.

Loading indicator

The final thing we need to do is add our `loading` boolean to the chain, set it to `true` and `false` at the right times so our loading message shows correctly, we can do that by using the `do` operator.

We use `do` to create *side effects* in our application, it lets us do things *outside* of just manipulating items via streams. It should be used *infrequently* but this is a perfect example, setting *state* on our component depending on where we are in the processing chain

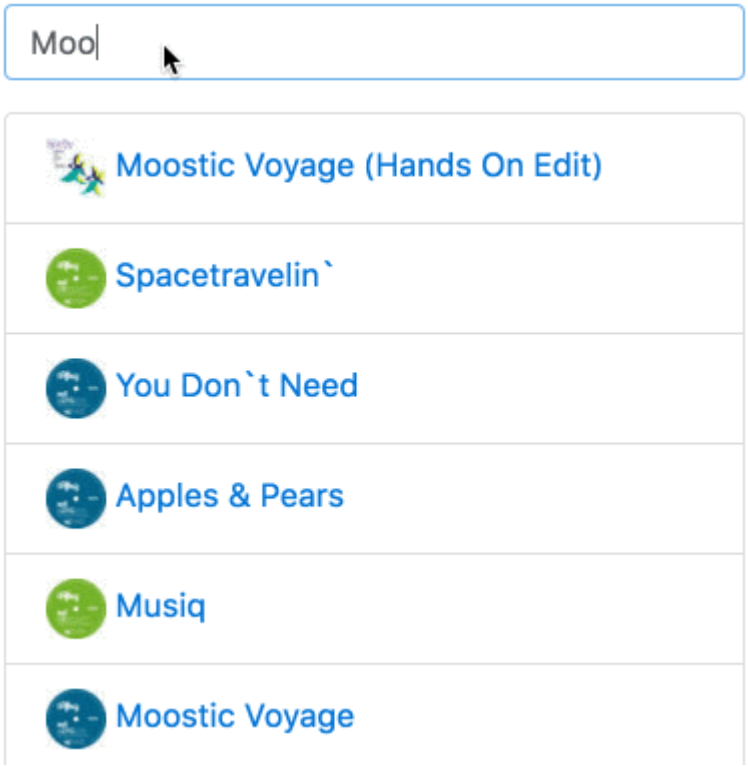
So now our function looks like so:

```
ngOnInit() {
  this.searchField = new FormControl();
  this.results = this.searchField.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .do( () => this.loading = true)
    .switchMap( term => this.itunes.search(term))
    .do( () => this.loading = false )
}
```

We need to import the `do` operator from RxJS

```
import 'rxjs/add/operator/do';
```

Finally if we run our application we should see something like this:



Summary

In this lecture we've covered, in depth, how to use observables when making HTTP requests.

The goal of this lecture was to show you how you can *evolve* your application from one that uses just a little bit of observables to one that uses a lot more.

Hopefully you now have a much better idea of *how* to architect your application using observables.

We could go even further, for instance the `loading` property could be it's own observable, how far and how deep to go is up to you. Depending on how comfortable you feel with Observables and how well they match to your use case.

They can be really powerful but at other times they can be a big hindrance with little benefit. In Angular you can use as much or as little reactive programming as you want, it doesn't prescribe one way or the other.

So far we've *not* dealt with the issue of *CORS* and we cheated in creating these apps by installing a plugin to chrome which circumnavigates *CORS* security issues.

In the next lecture we'll cover how to use *JSONP* to solve *CORS* issues, if the API supports JSONP, and how to implement JSONP requests in Angular.

Listing

http://plnkr.co/edit/Bpa4RiTKu8fd0oYKVskT?p=preview

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {HttpModule, Http, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Observable} from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/do';

class SearchItem {
  constructor(public track: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';

  constructor(private http: Http) {
  }

  search(term: string): Observable<SearchItem[]> {
    let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20`;
    return this.http.get(apiURL)
        .map(res => {
          return res.json().results.map(item => {
            return new SearchItem(
                item.trackName,
                item.artistName,
                item.trackViewUrl,
                item.artworkUrl30,
                item.artistId
            );
          });
        });
  }
}
```

```
@Component({
  selector: 'app',
  template: `
<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           [formControl]="searchField">
  </div>
</form>

<div class="text-center">
  <p class="lead" *ngIf="loading">Loading...</p>
</div>

<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let track of results | async">
    <img src="{{track.thumbnail}}">
    <a target="_blank"
       href="{{track.link}}">{{ track.track }}
    </a>
  </li>
</ul>
 `
})
class AppComponent {
  private loading: boolean = false;
  private results: Observable<SearchItem[]>;
  private searchField: FormControl;

  constructor(private itunes: SearchService) {
  }

  ngOnInit() {
    this.searchField = new FormControl();
    this.results = this.searchField.valueChanges
        .debounceTime(400)
        .distinctUntilChanged()
        .do(_ => this.loading = true)
        .switchMap(term => this.itunes.search(term))
        .do(_ => this.loading = false)
  }

  doSearch(term: string) {
    this.itunes.search(term)
  }
}

@NgModule({
```

```
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    HttpModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

JSONP Example with Observables

Learning Objectives

- Know what JSONP is and how it overcomes CORS.
- Know how to make JSONP requests in Angular.

What is JSONP?

JSONP is a method of performing API requests which go around the issue of *CORS*.

This is a security measure implemented in all browsers that stops you from using an API in a potentially unsolicited way and most APIs, including the iTunes API, are protected by it.

Because of CORS if we tried to make a request to the iTunes API url with the http client library the browser would issue a *CORS* error.

The explanation is deep and complex but a quick summary is that unless an API sets certain headers in the response a browser will reject the response, the iTune API we use *doesn't* set those headers so by default any response gets rejected.

So far in this section we've solved this by installing a chrome plugin which intercepts the response and adds the needed headers *tricking* the browser into thinking everything is ok but it's not a solution you can use if you want to release you app to the public.

If we switch *off* the plugin and try our application again we get this error in the console

```
XMLHttpRequest cannot load
https://itunes.apple.com/search?term=love&media=music&limit=20. No 'Access-Control-
Allow-Origin' header is present on the requested resource. Origin
'http://run.plnkr.co' is therefore not allowed access.
```

`JSONP` is a solution to this problem, it treats the API as if it was a *javascript file*. It dynamically adds `https://itunes.apple.com/search?term=love&media=music&limit=20` as if it were a `script` tag in our our HTML like so:

```
<script src="https://itunes.apple.com/search?term=love&media=music&limit=20"></script>
```

The browser then just downloads the javascript file and since browsers don't check for CORS when downloading javascript files it a works around the issue of CORS.

Imagine the response from the server was `{hello:'world'}`, we don't just want to download the file because:

1. The browser would download the json and try to *execute* it, since json isn't javascript *nothing* would actually happen.

2. We want to do something with the json once it's downloaded, ideally call a function passing it the json data.

So APIs that support JSONP return something that looks like javascript, for example it might return something like so:

```
process_response({hello:'world'});
```

When the file is downloaded it's executed as if it was javascript and calls a function called `process_response` passing in the json data from the API.

Of course we need a function called `process_response` ready and waiting in our application so it can be called but if we are using a framework that supports JSONP these details are handled for us.

That's is JSONP in a nutshell.

1. We treat the API as a javascript file.
2. The API wraps the JSON response in a function who's name we define.
3. When the browser downloads the *fake* API script it runs it, it calls the function passing it the JSON data.

We can only use JSONP when:

1. The API itself supports JSONP. It needs to return the JSON response wrapped in a function and it usually lets us pass in the function name we want it to use as one of the query params.
2. We can only use it for GET requests, it doesn't work for PUT/POST/DELETE and so on.

Refactor to JSONP

Now we know a little bit about JSONP and since the iTunes API supports JSONP lets refactor our observable application to one that uses JSONP instead.

For the most part the functionality is the same as our `Http` example but we use another client lib called `Jsonp` and the providers for our `Jsonp` solution is in the `JsonpModule`.

So firstly lets import and the new `JsonpModule` and replace all instances of `Http` with `Jsonp`.

```
import {JsonpModule, Jsonp, Response} from '@angular/http';
.
.
.
@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule ①
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule { }
```

① Add `JsonpModule` to the module imports.

Then lets change our `SearchService` to use the `Jsonp` library instead of `Http`, like so:

```
@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';

  constructor(private jsonp: Jsonp) { ①
  }

  search(term: string) {
    let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`; ②
    return this.jsonp.request(apiURL)  ③
        .map(res => {
          return res.json().results.map(item => {
            return new SearchItem(
                item.trackName,
                item.artistName,
                item.trackViewUrl,
                item.artworkUrl30,
                item.artistId
            );
          });
        });
  }
}
```

① We injected `Jsonp` into the constructor and stored it on a property called `jsonp`.

② We changed our apiURL by adding in `callback=JSONP_CALLBACK`.

③ We call `jsonp.request(⋯)` instead of `http.get(⋯)`

The iTunes API supports JSONP, we just need to tell it what name to use via the callback query parameter.

We passed it the special string `JSONP_CALLBACK`.

Angular will replace `JSONP_CALLBACK` with an automatically generated function name every time we make a request

That's it, the rest of the code is exactly the same and the application works like before.

Remember if you installed the CORS plugin or added some other work around then you can switch them off, you don't need them any more.

Summary

JSONP is a common solution in the web world to get around the issue of CORS.

We can only use JSONP with APIs that support JSONP.

To use JSONP in Angular we use the `Jsonp` client lib which is configured by importing the `JsonpModule` and adding it to our `NgModule` imports.

We make a JSONP request by calling the `request` function, it still returns an `Observable` so for the rest of your application that fact you are using `Jsonp` instead of `Http` should be invisible.

Listing

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Observable} from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/do';

class SearchItem {
  constructor(public track: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
```

```
  }
}

@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';

  constructor(private jsonp: Jsonp) {
  }

  search(term: string) {
    let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
    return this.jsonp.request(apiURL)
        .map(res => {
          return res.json().results.map(item => {
            return new SearchItem(
                item.trackName,
                item.artistName,
                item.trackViewUrl,
                item.artworkUrl30,
                item.artistId
            );
          });
        });
  }
}

@Component({
  selector: 'app',
  template: `
<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           [formControl]="searchField">
  </div>
</form>

<div class="text-center">
  <p class="lead" *ngIf="loading">Loading...</p>
</div>

<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let track of results | async">
    <img src="{{track.thumbnail}}">
    <a target="_blank"
       href="{{track.link}}">{{ track.track }}
    </a>
```

```
  </li>
</ul>
 `
})
class AppComponent {
  private loading: boolean = false;
  private results: Observable<SearchItem[]>;
  private searchField: FormControl;

  constructor(private itunes: SearchService) {
  }

  ngOnInit() {
    this.searchField = new FormControl();
    this.results = this.searchField.valueChanges
        .debounceTime(400)
        .distinctUntilChanged()
        .do(_ => this.loading = true)
        .switchMap(term => this.itunes.search(term))
        .do(_ => this.loading = false)
  }
}

@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

We use the `Http` client library in Angular to make HTTP requests.

We use the library by injecting it into classes and as such it needs to be configured via Anuglars DI framework.

To setup all the necessary providers we just need to import the `HttpModule` and add it to the imports property of our `NgModule`.

When architecting Angular application we don't call the `Http` library directly from components but instead create intermediate services which call the APIs on our behalf.

We also also prefer to convert the responses into domain models and return those instead of the raw response from the API.

The request functions return `Observables`, we can implement any required processing by using an *observable chain* or we can convert the observable into promises and handle the processing in a more classically procedural way.

We can use JSONP to bypass CORS with supported APIs, to do we we just need to replace instances of `Http` with `Jsonp` and `HttpModule` with `JsonpModule`.

Activity

Change the Http example app we created using Promises into one that uses Jsonp instead.

Steps

Fork this plunker:

http://plnkr.co/edit/3vy1nWs8h0f5D9Go3tyR?p=preview

Change the code to use `Jsonp` instead of `Http`.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/yZevXsUg0YMSzhMUdA5p?p=preview

Routing

Overview

The goal of this section is to build on the music search application we created in the previous section and also add pages for Artist, Track and Album listings.

We *could* implement this app by hiding and showing components on the page either via the `NgIf` directive or binding to the `[hidden]` property. but then the URL in the address bar doesn't change and that has a few drawbacks:

1. Refreshing the page would reset the application back to the start, i.e. if we had performed a search refreshing the page would mean we lose the results.
2. We cannot bookmark our position in the application and come back to it later.
3. We cannot send links to other people and have exactly the view show up for them as it shows for us.

Another way to think about the URL in your address bar is that it defines the current **state** of your application.

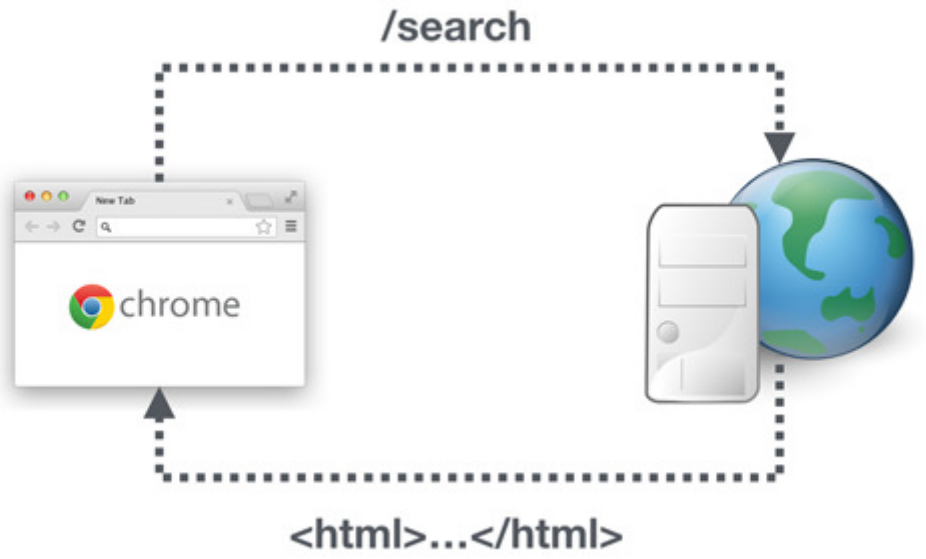
State is a computer science terms and means *"all the stored information, at a given instant in time, to which program has access"*.

So the state of an application is the current value of all the variables in the application.

An address in a URL can't store that much information but it can store enough so that perhaps we can re-fetch some and re-calculate the rest to get to the same *state* as before.

Giving the a URL to someone else should enable them to bring up the same *state* in their browser.

In traditional applications built with *Server Side Routing* when you change the URL in your browser, the browser makes a request to the server to return some HTML which it will display.



However we want to implement something called *Client Side Routing*. When the URL changes in the browser we want our local application that's running in the browser (the *client*) to handle the change, we don't want the request sent to the server.

When we first navigate to a new site the server returns the html, javascript and css needed to render that page. All further changes to the URL are handled locally by the client application. Typically the client application will make one or more API requests to get the information it needs to show the new page.

There is only ever a *single page* returned from the server, all further modifications of the page are handled by the *client* and that's why it's called a *Single Page Application*.

The advantages of an SPA are:

1. Can be faster. Instead of making a time-consuming request to a far away server every time the

URL changes the client app updates the page much faster.

2. Less bandwidth required. We don't send over a big html page for every URL change, instead we might just call a smaller API which returns *just enough data* to render the change in the page.
3. Convenience. Now a single developer can build most of the functionality of a site instead of splitting the effort between a front end and server side developer.

Angular has a couple of modules which let us implement our application as an SPA, the concept as a whole in Angular is called the *Component Router* and in this section you will learn how to build an SPA in Angular using the component router.

In this section you will learn:

- How to define and configure the different URLs (routes) in your application.
- How to navigate between those routes on the client side so the browser doesn't send a request to the server.
- What path location strategies we can use in Angular and the advantages/disadvantages of each.
- How to use parametrised routes where part of the URL is a variable.
- How to implement nested routes.
- How to implement router guards to prevent certain people from accessing certain URLs.

Route Configuration

Learning Objectives

- How to configure and setup the Angular router.
- How to handle *redirects* and *catch all* routes in Angular.

Components

To explain routing in Angular we start with an app very similar to the one we created in the section on Http.

We have a `SearchComponent` which lets us perform searches using the iTunes API, we are using JSONP to bypass CORS and *Promises* instead of *Observables* to keep things simple.

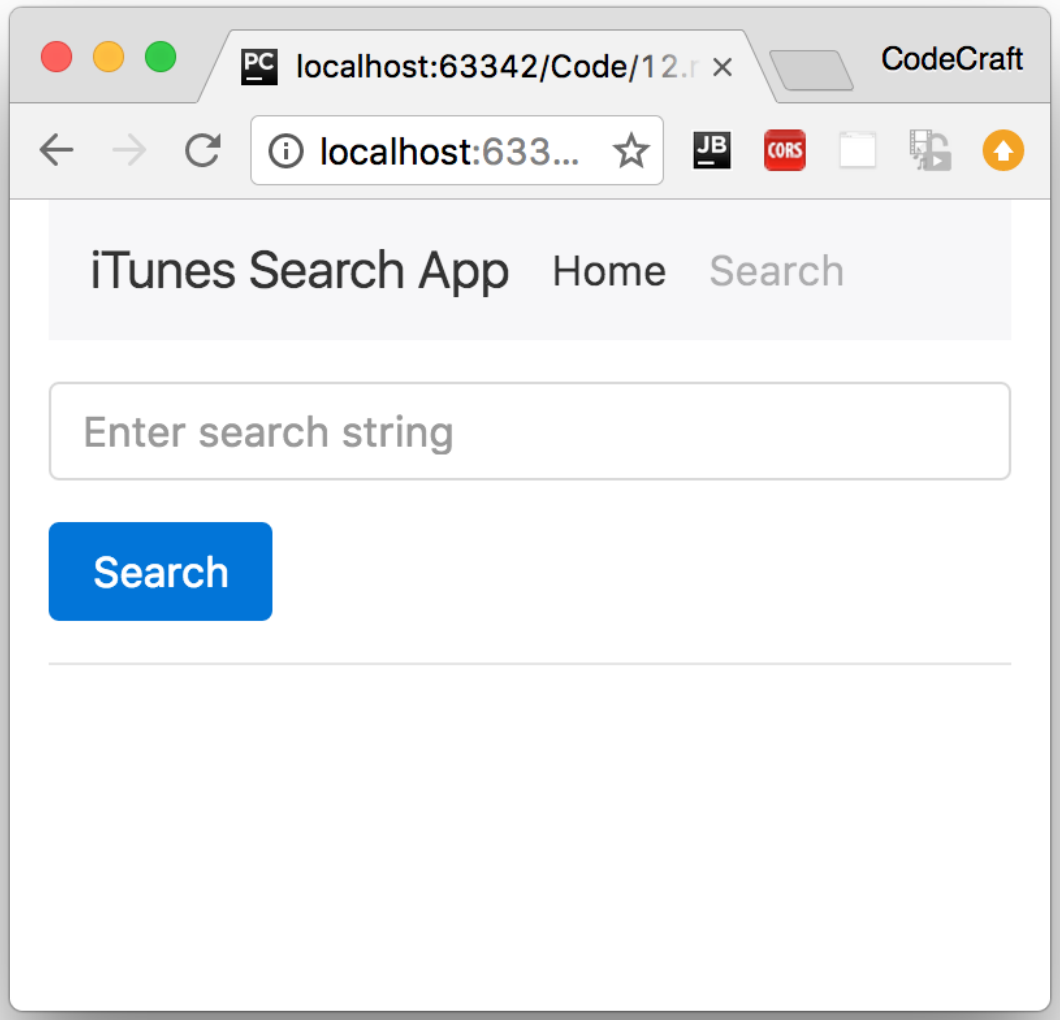
In addition we've also created a `HeaderComponent` with a selector of `app-header` and two menu items, *Home* and *Search*. We also have a `HomeComponent` with selector `app-home` which just shows a simple welcome message.

In one configuration, the main `AppComponent` renders these two components like so:

```
<app-header></app-header>
<div class="m-t-1">
  <app-search></app-search>
</div>
```

`m-t-1` above is a class from twitter bootstrap which adds a top margin to the element so we can clearly distinguish the different elements.

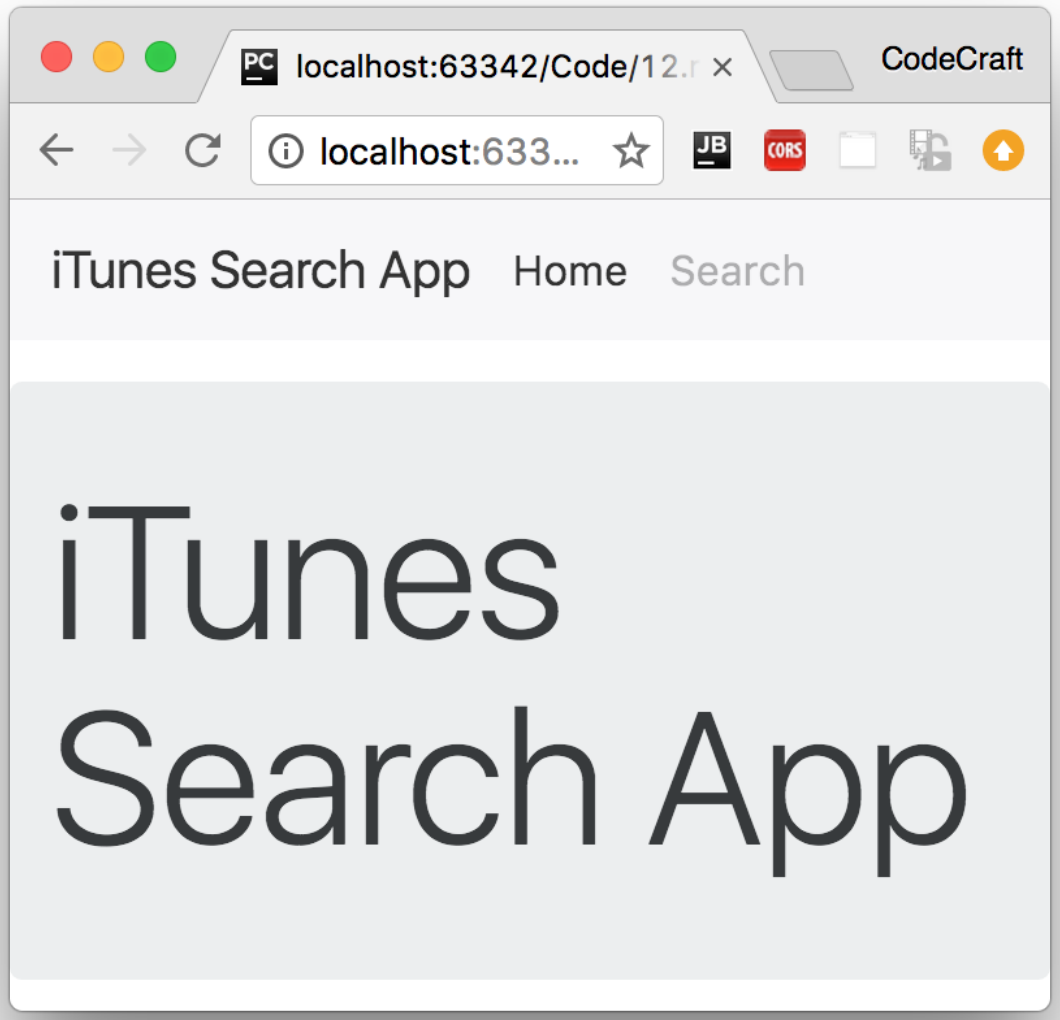
The above is the structure we want when the user navigates to `/search`, this renders the page with the header and the search component.



If we changed the template to be:

```
<app-header></app-header>
<div class="m-t-1">
  <app-home></app-home>
</div>
```

This renders the page with the header and the home component, we want this structure when the user navigates to the root / URL.



Routes & RouterModule

Our goal with routing is to have the `HomeComponent` rendered when the url is `/` and the `SearchComponent` shown when the url is `/search`

First we need to setup some imports, like so:

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

The mapping of URLs to *Components* we want displayed on the page is done via something called a *Route Configuration*, at it's core it's just an array which we can define like so:

```
const routes: Routes = [
 { path: '', component: HomeComponent },
 { path: 'search', component: SearchComponent }
];
```

- The `path` property describes the URL this route will handle.
- The `component` property is the name of the component we want to display when the URL in the browser matches this path.

`Routes` is a typescript type of `Route[]`, an array of individual `Route` instances.

We then *install* these routes into our application by importing `RouterModule.forRoot(routes)` into our `NgModule`, like so:

```
@NgModule({
    imports: [
      .
      .
        RouterModule.forRoot(routes, {useHash: true})
    ]
    .
    .
    .
})
class AppModule { }
```

We'll go through the meaning of the `{useHash: true}` argument when we cover *Path Location Strategies* later on in this section. For now just know that this prepends `/#` to all of our urls, so our root url would be `/#/` and our search url would be `/#/search`

RouterOutlet Directive

We've configured our application so if the user navigates to `/#/search` we want the `SearchComponent` shown or if they navigate to the root url `/#/` then we want the `HomeComponent` shown.

But where exactly do we want the component shown?

We need to add a directive called `router-outlet` somewhere in our template HTML. This directive tells Angular *where* it should insert each of those components in the route, we'll add ours to the `AppComponent`, like so:

```
<app-header></app-header>
<div class="m-t-1">
  <router-outlet></router-outlet> ①
</div>
```

① We place `<router-outlet>` where we want the component inserted.

Now if we run the application and visit the root URL we are shown the `HomeComponent` and if we visit the `/search` URL we are shown the `SearchComponent`.

Redirects

There are a few more ways to configure our routes, for example we might like to change our routes to add some redirects like so:

```
const routes:Routes = [
    {path: '', redirectTo: 'home', pathMatch: 'full'}, ①
    {path: 'find', redirectTo: 'search'}, ①
    {path: 'home', component: HomeComponent},
    {path: 'search', component: SearchComponent}
];
```

① The `redirectTo` property describes the path we want to redirect this user to if they navigate to this URL.

Now if the user visits the root (empty) URL they are redirected to `/home` instead.

For the special case of an *empty* URL we also need to add the `pathMatch: 'full'` property so Angular knows it should be matching exactly the empty string and not *partially* the empty string.

We've also added a redirect from `/find` to `/search`, since this isn't empty we don't need to add the `pathMatch` property.

Catch all route

We can also add a *catch all* route by using the path `**`, if the URL doesn't match *any* of the other routes it will match this route.

```
const routes:Routes = [
    {path: '', redirectTo: 'home', pathMatch: 'full'},
    {path: 'find', redirectTo: 'search'},
    {path: 'home', component: HomeComponent},
    {path: 'search', component: SearchComponent},
    {path: '**', component: HomeComponent} ①
];
```

① If nothing matches we show the `HomeComponent`

Now if we navigate to `/foo` it will show us the welcome message.

In our example above we are just showing the `HomeComponent` but normally we might show an error 404 page.

Summary

A route in our application is defined by a mapping of a URL to a component or a redirect to another URL.

We can create an array of `Routes` and then install them in our application by importing them into our `NgModule` using `RouterModule.forRoot(routes)`.

In this lecture we've shown how we can configure routes and manually type in the different urls in the address bar to make the application render different components depending on which URL the user visits.

Next we'll show how you can navigate between these different routes in Angular without having to manually type the URL into the browser address bar.

Listing

http://plnkr.co/edit/oGV8bOKHJWQofFiv9SjY?p=preview

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Routes, RouterModule} from "@angular/router";
import {Observable} from 'rxjs';
import 'rxjs/add/operator/toPromise';

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];
```

```
  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}


@Component({
  selector: 'app-search',
  template: `<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="doSearch(search.value)">
    Search
  </button>
</form>

<hr />
```

```
<div class="text-center">
  <p class="lead"
     *ngIf="loading">Loading...</p>
</div>

<div class="list-group">
  <a href="#"
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
 `
})
class SearchComponent {
  private loading: boolean = false;

  constructor(private itunes: SearchService) {
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
  }
}

@Component({
  selector: 'app-home',
  template: `
<div class="jumbotron">
  <h1 class="display-3">iTunes Search App</h1>
</div>
 `
})
class HomeComponent {
}

@Component({
  selector: 'app-header',
  template: `
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="#">iTunes Search App</a>
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link" href="#">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Search</a>
    </li>
```

```
  </ul>
</nav>
 `
})
class HeaderComponent {
}

@Component({
  selector: 'app',
  template: `
    <app-header></app-header>
    <div class="m-t-1">
    <router-outlet></router-outlet>
  </div>
 `
})
class AppComponent {
}

const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {path: '**', component: HomeComponent}
];


@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule,
    RouterModule.forRoot(routes, {useHash: true})
  ],
  declarations: [
    AppComponent,
    SearchComponent,
    HomeComponent,
    HeaderComponent
  ],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Navigation

Learning Objectives

How to navigate between the different routes in an Angular application.

Navigating by hardcoded URLS

We could simply *hardcode* the URLs in the `href` anchor attributes on our navigation header, like so:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="/#/">iTunes Search App</a>
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link" href="/#/">Home</a> ①
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/#/search">Search</a>
    </li>
  </ul>
</nav>
```

① We simply add a standard `href` with a value of `/#/`

This works for our example because we are using something called a `HashLocationStrategy` (more on that later) but hardcoding like this doesn't work with the other location strategy available in Angular, `PathLocationStrategy`.

If we were using that strategy clicking one of those links would result in the browser trying to request the whole page again from the server which defeats the purpose of trying to create an SPA.

Navigating programmatically via the router

In Angular we can also programmatically navigate via a `Router` service we inject into our component, like so:

```
import {Router} from "@angular/router";
.
.
.
@Component({
  selector: 'app-header',
  template: `
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" (click)="goHome()">iTunes Search App</a> ①
  <ul class="nav navbar-nav">
    <li class="nav-item">
      <a class="nav-link" (click)="goHome()">Home</a> ①
    </li>
    <li class="nav-item">
      <a class="nav-link" (click)="goSearch()">Search</a> ①
    </li>
  </ul>
</nav>
 `
})
class HeaderComponent {
  constructor(private router: Router) {} ②

  goHome() {
    this.router.navigate(['']); ③
  }

  goSearch() {
    this.router.navigate(['search']); ③
  }
}
```

① We added click handlers to each anchor tag to call functions on our `HeaderComponent`.

② We inject and store a reference to the `Router` into our `HeaderComponent`.

③ We then call the `navigate` function on the `router` to navigate between the different URLs.

Link params array

The value we pass into the `navigate` function might look a bit strange, we call it a *link params array* and it equivalent to the URL split by the `/` character into an array.

We don't have to pass in the `#` character in the parameters to the `navigate` function, it automatically adds them in if we are using the `HashLocationStrategy`.

We can demonstrate by changing our search route from

```
{path: 'search', component: SearchComponent},
```

to

```
{path: 'search/foo/moo', component: SearchComponent},
```

Then to navigate to our search page we pass into the navigate function an link params array like so:

```
this.router.navigate(['search', 'foo', 'moo']);
```

Navigating via a link params array has one big advantage in that parts of the URL can be *variables*, like so:

```
let part = "foo";
this.router.navigate(['search', part, 'moo']);
```

This becomes a *lot* more useful when we start dealing with parametrised routes later on in this section.

Navigating via a routerLink directive

We can also control navigation by using the `routerLink` directive in the template itself, like so:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" [routerLink]="['home']">iTunes Search App</a>
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link"  [routerLink]="['home']">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link"  [routerLink]="['search']">Search</a>
    </li>
  </ul>
</nav>
```

The `routerLink` directive takes as input the same link params array format that the `router.navigate(⋯)` function takes.

routerLinkActive

An important feature of any navigation component is giving the user feedback about *which* menu item they are currently viewing. Another way to describe this is giving the user feedback about which *route* is currently *active*.

With the twitter bootstrap navigation styles we give this feedback by adding a class of `active` to the *parent* element to the anchor tag, like so:

```
<li class="nav-item active"> ①
  <a class="nav-link" [routerLink]="['home']">Home</a>
</li>
```

① Adding `active` to the parent element highlights the anchor tag.

To help in adding and removing classes depending on the currently active route Angular provides another directive called `routerLinkActive`.

A `routerLinkActive` directive is associated with a *route* through a `routerLink` directive.

It takes as input an array of classes which it will add to the element it's attached to *if it's route is currently active*, like so:

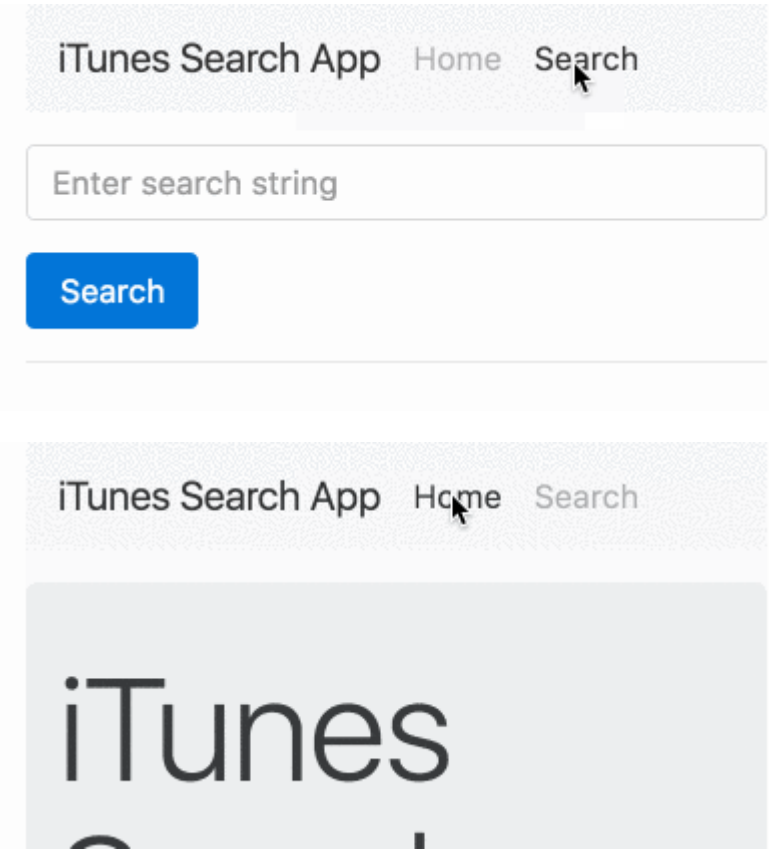
```
<a class="nav-link"
   [routerLink]="['home']"
   [routerLinkActive]="['active']">
   Home
</a>
```

The above will add a class of `active` to the anchor tag if we are currently viewing the home route.

However this isn't so useful for us in twitter bootstrap since we need the `active` class set on the *parent* `li` element.

But that's fine, the `routerLinkActive` directive can be set on a *parent* element of the `routerLink` directive and it will still associate itself with the route, like so:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand"
     [routerLink]="['home']">iTunes Search App
  </a>
  <ul class="nav navbar-nav">
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['home']">Home
      </a>
    </li>
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['search']">Search
      </a>
    </li>
  </ul>
</nav>
```



Summary

In this lecture we've shown how we can navigate between routes in Angular programmatically via the `router` and via the template by using the `routerLink` directive.

We've also explained that both these methods require a _link params array+ to be passed in order to function.

And finally we've shown how to add some user feedback as to the currently active route by using the `routerLinkActive` directive.

In the next lecture we'll explain how to add *variable* parameters to routes via parametrised routes.

Listing

http://plnkr.co/edit/6OXkV8NFB91LFoTEVGKk?p=preview

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Routes, RouterModule, Router} from "@angular/router";
import {Observable} from 'rxjs';
import 'rxjs/add/operator/toPromise';
```

```
class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}


@Component({
  selector: 'app-search',
  template: `<form class="form-inline">
  <div class="form-group">
```

```
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="doSearch(search.value)">
    Search
  </button>
</form>

<hr />

<div class="text-center">
  <p class="lead"
     *ngIf="loading">Loading...</p>
</div>

<div class="list-group">
  <a href="#"
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
 `
})
class SearchComponent {
  private loading: boolean = false;

  constructor(private itunes: SearchService) {
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
  }
}

@Component({
  selector: 'app-home',
  template: `
<div class="jumbotron">
  <h1 class="display-3">iTunes Search App</h1>
</div>
 `
})
class HomeComponent {
}
```

```
@Component({
  selector: 'app-header',
  template: `<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand"
     [routerLink]="['home']">iTunes Search App
  </a>
  <ul class="nav navbar-nav">
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['home']">Home
      </a>
    </li>
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['search']">Search
      </a>
    </li>
  </ul>
</nav>
 `
})
class HeaderComponent {
  constructor(private router: Router) {
  }

  goHome() {
    this.router.navigate(['']);
  }

  goSearch() {
    this.router.navigate(['search']);
  }
}

@Component({
  selector: 'app',
  template: `
    <app-header></app-header>
    <div class="m-t-1">
    <router-outlet></router-outlet>
  </div>
 `
})
class AppComponent {
}

const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
```

```
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {path: '**', component: HomeComponent}
];


@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule,
    RouterModule.forRoot(routes, {useHash: true})
  ],
  declarations: [
    AppComponent,
    SearchComponent,
    HomeComponent,
    HeaderComponent
  ],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Parameterised Routes

Learning Objectives

- How to configure parameterised routes in our route definition object.
- How components can be notified with what the parameter values are when the URL gets visited.
- How to have optional parameters in routes.

Configuration

Sometimes we need part of the path in one or more of our routes (the URLs) to be a *variable*, a common example of this is an *ID*.

Lets say we have a blog and each article in our blog has an ID, the URLs for each blog article might look like

```
/blog/1
/blog/2
/blog/3
/blog/4
and so on...
```

Now we could write a route for each article like so:

```
const routes: Routes = [
 { path: 'blog/1', component: Blog1Component },
 { path: 'blog/2', component: Blog2Component },
 { path: 'blog/3', component: Blog3Component },
 { path: 'blog/4', component: Blog4Component },
];
```

But a better solution is to have *one* route with *one* component called `BlogComponent` and pass to the `BlogComponent` the *number* part of the URL.

That's called a *parameterised route* and we would implement it like so:

```
const routes: Routes = [
 { path: 'blog/:id', component: BlogComponent } ①
];
```

① The path has a variable called `id`, we know it's a variable since it begins with a colon `:`

A path can have any number of variables as long as they all start with `:` and have different names.

Non-parameterised routes *always* take priority over parameterised routes, so in the below config:

```
const routes: Routes = [
 { path: 'blog/:id', component: BlogComponent },
 { path: 'blog/moo', component: MooComponent },
];
```

If we visited `/blog/moo` we would show `MooComponent` even though it matches the path for the first `blog/:id` route as well.

Non-parameterised routes take precedence over parameterised routes.

Activated route

So how do we pass into `BlogComponent` the value of the *id* variable? If we visited `/blog/1` how does `BlogComponent` know the `id` is `1` and therefore to show the appropriate article.

To do that we use something called an `ActivatedRoute`.

We import it first and then inject it into the constructor of `BlogComponent`. It exposes an Observable which we can subscribe to, like so:

```
import {ActivatedRoute} from "@angular/router";
.
.
.
constructor(private route: ActivatedRoute) {
    this.route.params.subscribe( params => console.log(params) );
}
```

Now if we navigated to `/blog/1` the number `1` would get emitted on the observable and this would get printed to the console as:

```
{ id: 1 }
```

Example

For the rest of this lecture we are going to continue building the iTunes search app we've been working on in the other lectures.

As we perform searches with the app the URL doesn't change. Therefore if I refresh the page *after* I perform a search then I lose all my search results - the *state* of my app is lost.

Lets turn the search route into a *parameterised* route where the search term is in the URL, so if I refresh the page it will perform the same search and get us back to where we were.

Parameterised Route Configuration

We begin with just adding a variable called `term` to our route configuration, like so:

```
{path: 'search/:term', component: SearchComponent},
```

But if we now load the app and try to go to the search page nothing happens, the URL changes to `/search` but we are still shown the `HomeComponent`.

The reason for that is that now our url `/search` doesn't match a route so we are falling back to the catch all route which just shows the `HomeComponent`.

To match our new parameterised route we would have to navigate to something like `/search/U2`. We would actually need to pass a parameter to match that route.

So to support both `/search` and `/search/U2` we need *two* routes in our configuration, like so:

```
{path: 'search', component: SearchComponent},
{path: 'search/:term', component: SearchComponent},
```

Now our app supports both `/search` and `/search/U2`.

Activated route

Next lets import `ActivatedRoute` and inject it into the constructor of our `SearchComponent`, like so:

```
constructor(private itunes:SearchService,
            private route: ActivatedRoute) {
    this.route.params.subscribe( params => console.log(params));
}
```

We subscribe for updates to the params of the currently active route and just print them out to the console.

Now if we navigate to `/search/U2` we get `{ term: 'U2' }` printed in our console as expected but we are not actually performing a search for `U2`. To do that all we need to do is call `doSearch(···)` from the activated route subscribe callback, like so:

```
constructor(private itunes:SearchService,
            private route: ActivatedRoute) {
  this.route.params.subscribe( params => this.doSearch(params['term'])); ①
}
```

① We call `doSearch` and pass in the `term` parameter from the URL.

Now when we visit `/search/U2` the `SearchComponent` is notified via the `ActivatedRoute` and we perform the query and show the user the search results for `U2`.

But now if we now search for another term, for example `Foo`, we get the results we expect but the URL *doesn't change*, it's still `/search/U2`.

When using routing if some part of the state of your application is in the URL then you need to update your application by navigating to the URL.

That way the URL matches the state of your app and if you bookmarked or shared the URL then visiting it again would get you back to the same state.

In our case what this means is that when the user submits a search, instead of calling `doSearch(…)` we instead navigate to the appropriate search URL and then let the `ActivatedRoute` service notify the `SearchComponent` the route changed and let that perform the search.

This way the URL changes *every-time* we do a search.

First we make the click handler on the *Search* button point to another function called `onSearch(…)` instead of `doSearch(…)`.

```
<button type="button"
        class="btn btn-primary"
        (click)="onSearch(search.value)">
        Search
</button>
```

And in our `onSearch` function we just *navigate* to the correct search URL.

```
onSearch(term:string) {
  this.router.navigate(['search', term]); ①
}
```

① The second parameter to the link params array is a variable, it's the search `term` the user typed in.

Now when we search `Foo` the url changes to `/search/Foo` the `SearchComponent` gets notified through the `ActivatedRoute` service and performs the search, therefore the URL and the *state* of our application are now in sync.

Optional params

Going back to the route config and the solution of having *two* routes configured, one for when there is a search term and another for when there isn't a search term.

```
{path: 'search', component: SearchComponent},
{path: 'search/:term', component: SearchComponent}
```

Another way to think about the above is that the variable `term` is *optional*. It might be present and it might not and we want the app to function correctly in either case.

Angular has a mechanism to support *optional* params with only one route defined.

Firstly lets get rid of the second route with the fixed term variable, leaving us with just one route to support our search like so:

```
const routes:Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {path: '**', component: HomeComponent}
];
```

Next in the `onSearch(⋯)` function instead of navigating to a route with the fixed term param, we can instead pass in an object containing whatever params we want, like so:

```
onSearch(term:string) {
  this.router.navigate(['search', {term: term}]);
}
```

Now when we perform a search the URL changes, *but* it doesn't quite looks like something we are used to, instead of `/search/U2` we see `/search;term=U2`

This is called Matrix URL notation, a series of *optional* key=value pairs separated with the semicolon ; character.

We can see that the param `term` still gets passed along via the `ActivatedRoute` as `{term: "U2"}`.

In-fact if we added some other params to the mix via the URL they also get printed to the console, like so:

```
/search;term=U2;foo=moo
```

Will print out

```
Object {term: "U2", foo: "moo"}
```

And also if we pass nothing we get an empty object printed out to the console

```
/search
```

Will print out

```
Object {}
```

So now the `term` param is optional, but since it's optional it sometimes can be blank. Lets add a little bit of defensive coding in our subscribe function so we don't call `doSearch` if no term has been provided, like so:

```
  constructor(private itunes: SearchService,
              private route: ActivatedRoute,
              private router: Router) {
    this.route.params.subscribe(params => {
      console.log(params);
      if (params['term']) { ①
        this.doSearch(params['term'])
      }
    });
  }
```

① Only call `doSearch(···)` if `term` has a value.

Now if we visit `/search` we are shown a blank screen ready for a search.

- If we perform a search for e.g. `U2` the url changes to `/search;term=U2`
- If we *refresh* the page then the application performs a search for `U2` again.

Summary

With parameterised routes we can support variable paths in our routes.

Angular also supports *optional* routes via passing in an object to the navigate function and the matrix url notation.

So far we've only shown how we can output *one* component on the page depending on the route. In the next lecture we are going to show how we can have nested routes and output multiple different components on the page depending on the URL.

Listing

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Routes, RouterModule} from "@angular/router";
import {Observable} from 'rxjs';
import 'rxjs/add/operator/toPromise';
```

```
import {Routes, RouterModule, Router, ActivatedRoute} from "@angular/router";

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}


@Component({
  selector: 'app-search',
```

```
  template: `<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="onSearch(search.value)">
    Search
  </button>
</form>

<hr />

<div class="text-center">
  <p class="lead"
     *ngIf="loading">Loading...</p>
</div>

<div class="list-group">
  <a href="#"
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
 `
})
class SearchComponent {
  private loading: boolean = false;

  constructor(private itunes: SearchService,
              private route: ActivatedRoute,
              private router: Router) {
    this.route.params.subscribe(params => {
      console.log(params);
      if (params['term']) {
        this.doSearch(params['term'])
      }
    });
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
  }

  onSearch(term: string) {
```

```
    this.router.navigate(['search', {term: term}]);
  }
}

@Component({
  selector: 'app-home',
  template: `
<div class="jumbotron">
  <h1 class="display-3">iTunes Search App</h1>
</div>
 `
})
class HomeComponent {
}

@Component({
  selector: 'app-header',
  template: `<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand"
     [routerLink]="['home']">iTunes Search App
  </a>
  <ul class="nav navbar-nav">
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['home']">Home
      </a>
    </li>
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['search']">Search
      </a>
    </li>
  </ul>
</nav>
 `
})
class HeaderComponent {
  constructor(private router: Router) {
  }

  goHome() {
    this.router.navigate(['']);
  }

  goSearch() {
    this.router.navigate(['search']);
  }
}
```

```
@Component({
  selector: 'app',
  template: `
    <app-header></app-header>
    <div class="m-t-1">
    <router-outlet></router-outlet>
  </div>
 `
})
class AppComponent {
}

const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {path: '**', component: HomeComponent}
];


@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule,
    RouterModule.forRoot(routes, {useHash: true})
  ],
  declarations: [
    AppComponent,
    SearchComponent,
    HomeComponent,
    HeaderComponent
  ],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Nested Routes

Learning Objectives

- How to configure child routes.
- How to define paths relative to the current path or the root of the application.
- How to get access to parent parameterised route params.

Goal

The goal for this lecture is to change our iTunes search app so that when we click on a search result we are taken to an *Artist* page.

This *Artist* page has two more menu items, *Tracks* and *Albums*. If we click on *Tracks* we want to show the list of tracks for this artist, if we click on *Albums* we want to show the list of albums for this artist.

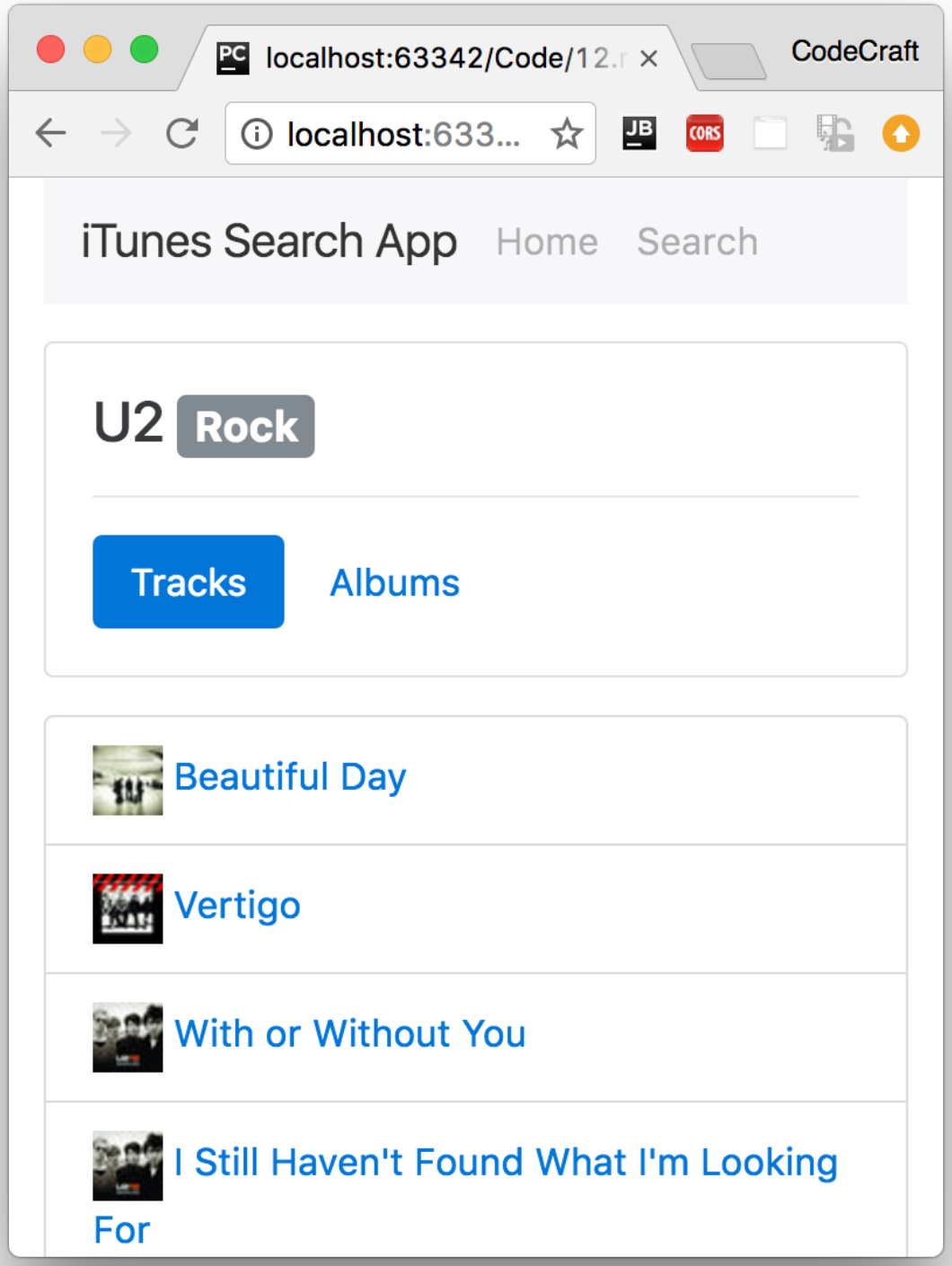


Figure 5. Artist Track Listing Page

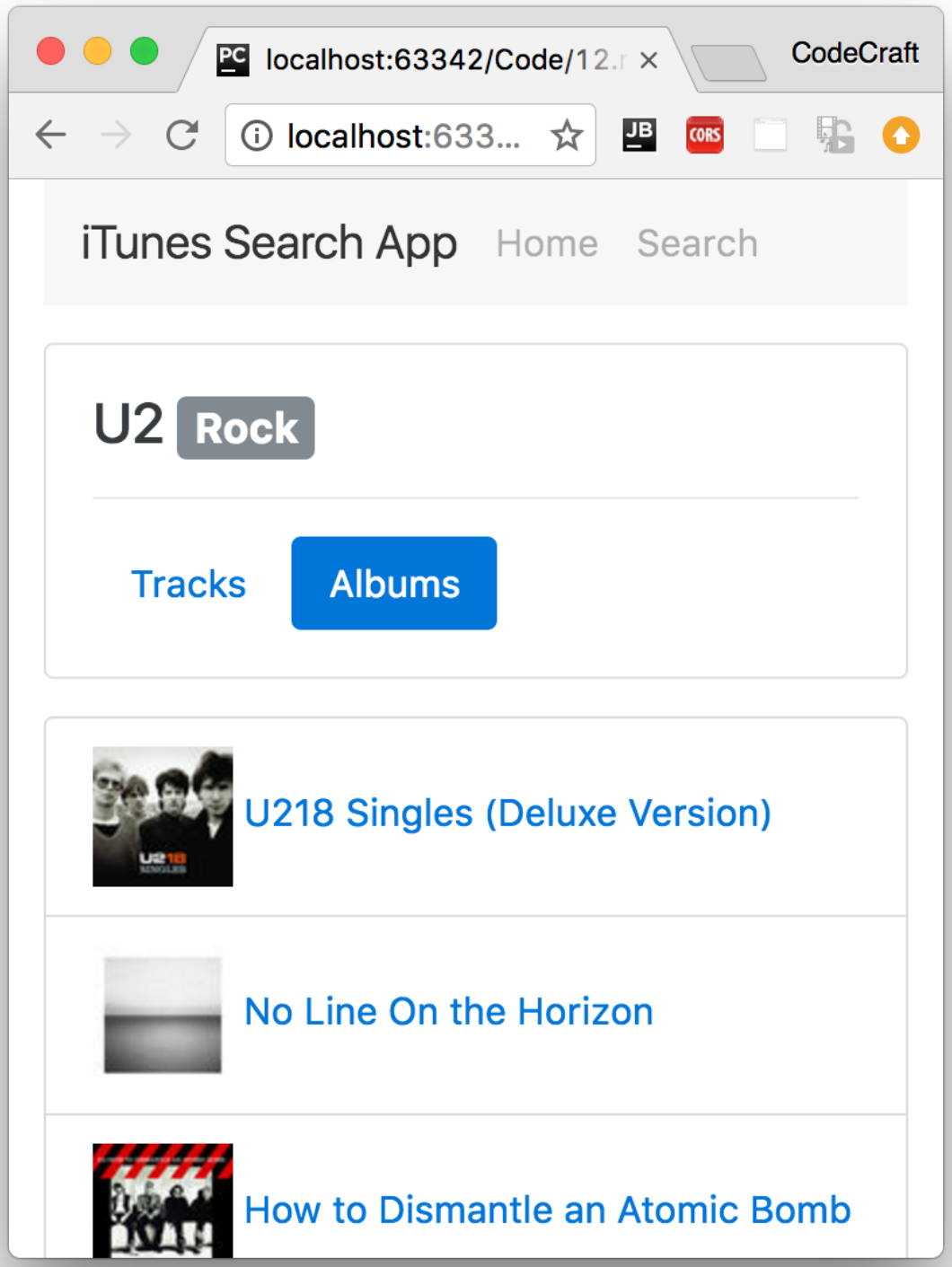


Figure 6. Artist Album Listing Page

We can get information about an Artist using the iTunes API by passing in the id of an artist

`https://itunes.apple.com/lookup?id=16586443`

A list of tracks for that artist by additionally passing in `entity=song`:

`https://itunes.apple.com/lookup?id=16586443&entity=song`

A list of albums for that artist by additionally passing in `entity=album`:

https://itunes.apple.com/lookup?id=16586443&entity=album

The concept of having *two* sets of menu items. A top level between *Home*, *Search* and *Artist* and a second level under *Artist* between *Tracks* and *Albums* is called *Nested Routing* and it's the topic of this lecture.

Setup

We've added 3 more components to our sample project and included them in our `NgModule` declarations:

1. `ArtistComponent` - This shows some details about an *Artist* and contains either the `ArtistTrackListComponent` or `ArtistAlbumListComponent`.
2. `ArtistTrackListComponent` - This will show a track listing for the current *Artist*.
3. `ArtistAlbumListComponent` - This will show an Album listing for the current *Artist*.

For now each component's template just has a heading with the name of the component, like so:

```
@Component({
  selector: 'app-artist',
  template: `
<h1>Artist</h1>
 `
})
class ArtistComponent {
}
```

```
@Component({
  selector: 'app-artist-track-list',
  template: `
<h1>Artist Track Listing</h1>
 `
})
class ArtistTrackListComponent {
}
```

```
@Component({
  selector: 'app-artist-album-list',
  template: `
<h1>Artist Album Listing</h1>
 `
})
class ArtistAlbumListComponent {
}
```

Route Configuration

We need another route for our `ArtistComponent`, we want to pass to this component an `artistId` so it needs to be a *parameterised route*, the `aritstId` is mandatory so we are *not* using optional params.

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {path: 'artist/:artistId', component: ArtistComponent}, ①
  {path: '**', component: HomeComponent}
];
```

① Add a path of `'artist/:artistId'` for `ArtistComponent`

In our search results listing we add a `routerLink` to the result item so it navigates to our `Artist` page.

```
<div class="list-group">
  <a [routerLink]="['artist', track.artistId]" ①
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
```

① Add `routerLink` directive.

Relative routes

If we ran the above code you'll notice that searching for U2 transforms the URL to:

```
/#/search;term=U2
```

and then if we click on a *U2* song the URL *incorrectly* becomes

```
/#/search;term=U2/artist/78500
```

And we don't get shown the Artist page.

The URL looks incorrect, it's a concatenation of the search URL with the artist URL. We should be expecting a URL like so:

```
/#/artist/78500
```

That's because when we navigated to

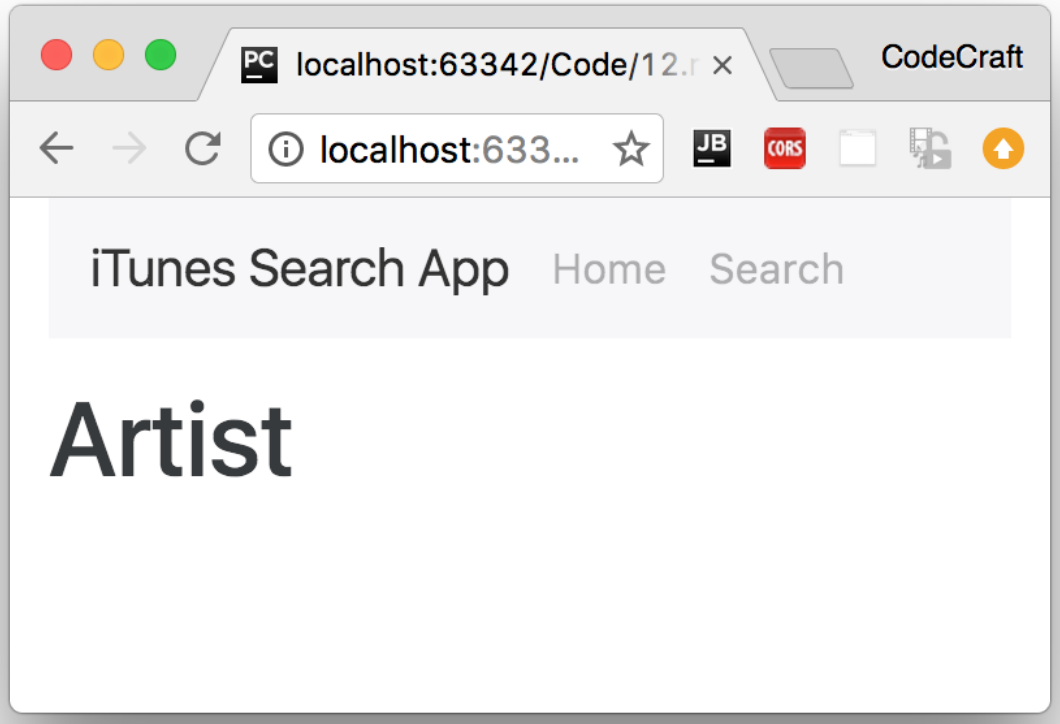
```
[routerLink]="['artist', item.artistId]"
```

we did so *relative* to the current URL and the current URL at the time was the search URL.

What we want to do is to navigate relative to the _root _of our application, relative to `/` - we can do that simply by pre-pending `/` to our path, like so:

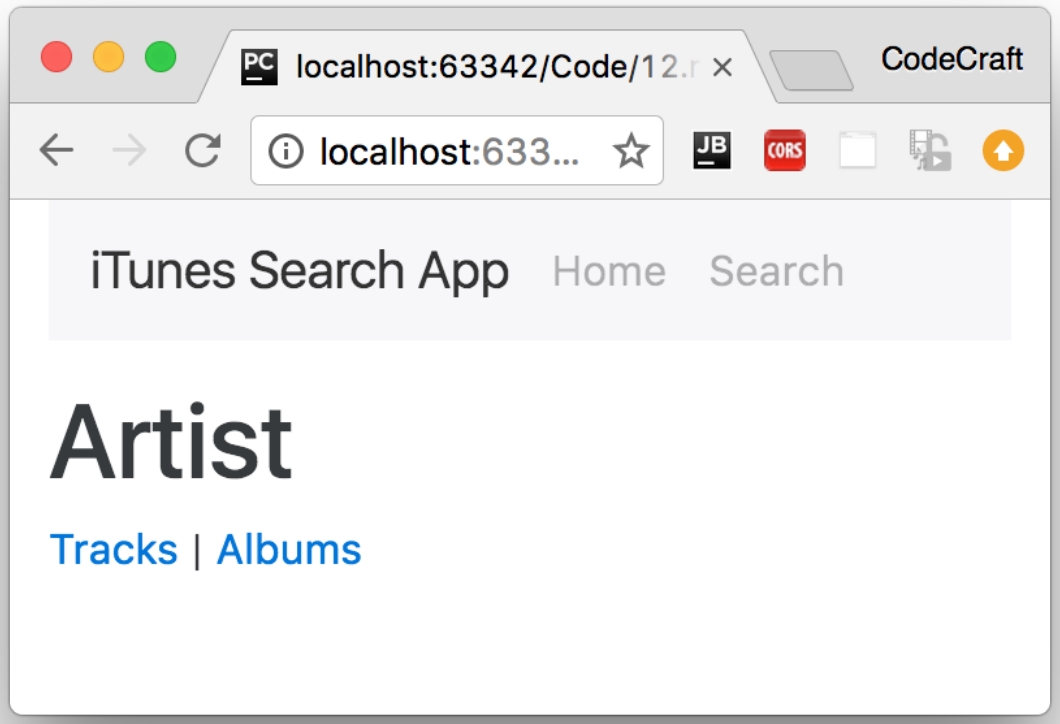
```
[routerLink]="['/artist', item.artistId]"
```

Now when we navigate we do so *relative* to the *root* URL and when we click the first U2 song the URL changes to `/#/artist/78500` and we are shown the artist page.



Child routes

When the Artist page is shown we want there to be two menu options, one called *Tracks* and the other called *Albums*, like so:



When a user clicks on *Tracks* underneath I would like to show a list of all the tracks this artist has released, if the user clicks on *Albums* then I want to show a list of all the albums they have released.

Explained in terms that the *Router* would understand:

> *"Inside the* `ArtistComponent` *I want to* ***conditionally*** *show either the* `AritstAlbumsListComponent` *or the* `ArtistTrackListComponent` *depending on which menu item the user has selected".*

Let me first define the route configuration, each route can have a property called `children` where you can define the child routes of this route. I'm going to add my routes for `AritstAlbumsListComponent` and `ArtistTrackListComponent`, like so:

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    children: [
      {path: '', redirectTo: 'tracks'}, ①
      {path: 'tracks', component: ArtistTrackListComponent}, ②
      {path: 'albums', component: ArtistAlbumListComponent}, ③
    ]
  },
  {path: '**', component: HomeComponent}
];
```

① If a user navigates to say `/artist/1234` it will redirect to `/artist/1234/tracks` (since we would *at least* want one of either the track or album components to be shown).

② This route matches a URL like `/artist/1234/tracks`.

③ This route matches a URL like `/artist/1234/albums`.

Now I can add the menu items to my artist template, like so:

```
<h1>Artist</h1>
<p>
  <a [routerLink]="['tracks']">Tracks</a> |
  <a [routerLink]="['albums']">Albums</a>
</p>
```

Notice that the paths in the `routerLink` directives above *don't* start with `/` so they are *relative* to the *current URL* which is the `/artist/:artistId` route.

A more explicit way of saying you want to route relative to the current URL is to prepend it with `./` like so:

```
<h1>Artist</h1>
<p>
  <a [routerLink]="['./tracks']">Tracks</a> |
  <a [routerLink]="['./albums']">Albums</a>
</p>
```

Pre-pending with `./` clearly expresses our intent that the path is relative so lets use this syntax instead.

The final thing we need to do is to tell Angular where we want the `ArtistTrackListComponent` and

`ArtistAlbumListComponent` components to be injected into the page, we do that by adding in another `router-outlet` directive in our Artist template like so:

```
<h1>Artist</h1>
<p>
  <a [routerLink]="['./tracks']">Tracks</a> |
  <a [routerLink]="['./albums']">Albums</a>
</p>

<router-outlet></router-outlet>
```

Now we have two `router-outlets` one nested inside another Angular figures out *which* outlet to insert the component in by the *nesting level* of the route and the router outlet.

Parent route params

To query the list of tracks for an Artist from the iTunes API the `ArtistTrackListComponent` needs the `artistId`.

As a reminder the route configuration for our `ArtistTrackListComponent` is

```
{path: 'tracks', component: ArtistTrackListComponent}
```

and the route configuration for it's parent is

```
{
    path: 'artist/:artistId',
    component: ArtistComponent,
    children: [
      {path: '', redirectTo: 'tracks'},
      {path: 'tracks', component: ArtistTrackListComponent},
      {path: 'albums', component: ArtistAlbumListComponent},
    ]
}
```

The *parent route* has the `:artistId` as a route param, however if we injected `ActivatedRoute` into our child `ArtistTrackListComponent` and tried to print out the params surprisingly we just get an empty object printed out.

```
class ArtistTrackListComponent {
  constructor(private route: ActivatedRoute) {
    this.route.params.subscribe(params => console.log(params)); // Object {}
  }
}
```

The reason for this is that `ActivatedRoute` only passes you the parameters for the *current* components route and since the route for `ArtistTrackListComponent` doesn't have any route parameters it gets passed nothing, we want to get the params for the *parent* route.

We can do this by calling `parent` on our `ActivatedRoute` like so:

```
class ArtistTrackListComponent {
  constructor(private route: ActivatedRoute) {
    this.route.parent.params.subscribe(params => console.log(params)); // Object
{artistId: 12345}
  }
}
```

This returns the params for the parent route.

Summary

We can nest routes, this means that for a given URL we can render a tree of components.

We do this by using multiple `router-outlet` directives and configuring child routes on our route configuration object.

Next up we will learn about protecting access to different routes via the use of *Router Guards*.

The application above is complete in terms of child routing but still needs rounding out in terms of making the other API requests for tracks and albums and prettying up the template HTML.

You won't learn anything else about routing from going through the process of finishing off the app, but for your interest i've provided the full listing below.

As with all our examples they are for illustrative purposes only, to follow the official style guides we should be putting each component in a separate file and our http request should all be wrapped in a separate service.

I will however leave that as an exercise for the reader if they so wish.

Listing

http://plnkr.co/edit/y1MZX3bEd1LJMN5KGKli?p=preview

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
```

```
import {Routes, RouterModule} from "@angular/router";
import {Observable} from 'rxjs';
import 'rxjs/add/operator/toPromise';
import {Routes, RouterModule, Router, ActivatedRoute} from "@angular/router";

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}
```

```
@Component({
  selector: 'app-search',
  template: `<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="onSearch(search.value)">
    Search
  </button>
</form>

<hr />

<div class="text-center">
  <p class="lead"
     *ngIf="loading">Loading...</p>
</div>

<div class="list-group">
  <a [routerLink]="['/artist', track.artistId]"
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
 `
})
class SearchComponent {
  private loading: boolean = false;

  constructor(private itunes: SearchService,
              private route: ActivatedRoute,
              private router: Router) {
    this.route.params.subscribe(params => {
      console.log(params);
      if (params['term']) {
        this.doSearch(params['term'])
      }
    });
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
```

```
  }

  onSearch(term: string) {
    this.router.navigate(['search', {term: term}]);
  }
}

@Component({
  selector: 'app-home',
  template: `
<div class="jumbotron">
  <h1 class="display-3">iTunes Search App</h1>
</div>
 `
})
class HomeComponent {
}

@Component({
  selector: 'app-header',
  template: `<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand"
     [routerLink]="['home']">iTunes Search App
  </a>
  <ul class="nav navbar-nav">
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['home']">Home
      </a>
    </li>
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['search']">Search
      </a>
    </li>
  </ul>
</nav>
 `
})
class HeaderComponent {
  constructor(private router: Router) {
  }

  goHome() {
    this.router.navigate(['']);
  }

  goSearch() {
    this.router.navigate(['search']);
```

```
  }
}

@Component({
  selector: 'app-artist-track-list',
  template: `
<ul class="list-group">
    <li class="list-group-item"
        *ngFor="let track of tracks">
        <img src="{{track.artworkUrl30}}">
        <a target="_blank"
           href="{{track.trackViewUrl}}">{{ track.trackName }}
        </a>
    </li>
</ul>
 `
})
class ArtistTrackListComponent {
  private tracks: any[];

  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
    this.route.parent.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&entity=so
ng&callback=JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.tracks = res.json().results.slice(1);
          });
    });
  }
}

@Component({
  selector: 'app-artist-album-list',
  template: `<ul class="list-group">
    <li class="list-group-item"
        *ngFor="let album of albums">
        <img src="{{album.artworkUrl60}}">
        <a target="_blank"
           href="{{album.collectionViewUrl}}">{{ album.collectionName }}
        </a>
    </li>
</ul>
 `
})
class ArtistAlbumListComponent {
  private albums: any[];
```

```
  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
    this.route.parent.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&entity=al
bum&callback=JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.albums = res.json().results.slice(1);
          });
    });
  }
}

@Component({
  selector: 'app-artist',
  template: `<div class="card">
  <div class="card-block">
    <h4>{{artist?.artistName}} <span class="tag tag-
default">{{artist?.primaryGenreName}}</span></h4>
    <hr />
    <footer>
      <ul class="nav nav-pills">
        <li class="nav-item">
          <a class="nav-link"
             [routerLinkActive]="['active']"
             [routerLink]="['./tracks']">Tracks
          </a>
        </li>
        <li class="nav-item">
          <a class="nav-link"
             [routerLinkActive]="['active']"
             [routerLink]="['./albums']">Albums
          </a>
        </li>
      </ul>
    </footer>
  </div>
</div>

<div class="m-t-1">
  <router-outlet></router-outlet>
</div>
 `
})
class ArtistComponent {
  private artist: any;

  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
```

```
    this.route.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&callback=
JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.artist = res.json().results[0];
            console.log(this.artist);
          });
    });
  }
}

@Component({
  selector: 'app',
  template: `
    <app-header></app-header>
    <div class="m-t-1">
    <router-outlet></router-outlet>
  </div>
 `
})
class AppComponent {
}

const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    children: [
      {path: '', redirectTo: 'tracks', pathMatch: 'full'},
      {path: 'tracks', component: ArtistTrackListComponent},
      {path: 'albums', component: ArtistAlbumListComponent},
    ]
  },
  {path: '**', component: HomeComponent}
];


@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule,
    RouterModule.forRoot(routes, {useHash: true})
```

```
  ],
  declarations: [
    AppComponent,
    SearchComponent,
    HomeComponent,
    HeaderComponent,
    ArtistAlbumListComponent,
    ArtistTrackListComponent,
    ArtistComponent
  ],
  bootstrap: [AppComponent],
  providers: [SearchService]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Router Guards

In traditional server side applications the application would check permissions on the server and return a `403` error page if the user didn't have permissions, or perhaps redirect them to a login/register page if they were not signed up.

`403` is a HTTP error code specifically this one means *Permission Denied*

We want to have the same functionality in our client side SPA, and with *Router Guards* we can.

With *Router Guards* we can prevent users from accessing areas that they're not allowed to access, or, we can ask them for confirmation when leaving a certain area.

Learning Objectives

In this lecture you will learn:

- The 4 different types of Router Guards
- Implementing Router Guards as classes
- How to implement a login guard for our iTunes app

Guard Types

In this chapter, we take a look at a few of the different types of guards and how to implement them for specific use cases.

- Maybe the user must login (authenticate) first.
- Perhaps the user has logged in but is not authorized to navigate to the target component.
- We might ask the user if it's OK to discard pending changes rather than save them.

There are four different types of Guards:

CanActivate
: Checks to see if a user can visit a route.

CanActivateChild
: Checks to see if a user can visit a routes children.

CanDeactivate
: Checks to see if a user can exit a route.

Resolve
: Performs route data retrieval before route activation.

CanLoad
: Checks to see if a user can route to a module that lazy loaded.

For a given route we can implement zero or any number of *Guards*.

We'll go through the first three as the last two are very advanced use cases and need lazy loading modules which we we haven't covered.

CanActivate

Guards are implemented as services that need to be provided so we typically create them as `@Injectable` classes.

Guards return either `true` if the user can access a route or `false` if they can't.

They can also return an `Observable` or `Promise` that later on resolves to a boolean in case the guard can't answer the question straight away, for example it might need to call an API. Angular will keep the user waiting until the guard returns `true` or `false`.

Lets create a simple `CanActivate` guard.

First we need to import the `CanActivate` interface, like so:

```
import {CanActivate} from "@angular/router";
```

Then lets create an Injectable class called `AlwaysAuthGuard` which implements the `canActivate` function, like so:

```
class AlwaysAuthGuard implements CanActivate {
  canActivate() {
    console.log("AlwaysAuthGuard");
    return true;
  }
}
```

This guard returns `true` *all the time*, so doesn't really guard anything. It lets all users through but at the same time our guard logs `"AlwaysAuthGuard"` to the console so we can at least see when it's being used.

We need to provide this guard, for this example lets configure it via our `NgModule`, like so:

```
@NgModule({
  .
  .
  providers: [
    .
    .
    AlwaysAuthGuard
  ]
})
```

Finally we need to add this guard to one or more of our routes, lets add it to our `ArtistComponent` route like so:

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    canActivate: [AlwaysAuthGuard], ①
    children: [
      {path: '', redirectTo: 'tracks'},
      {path: 'tracks', component: ArtistTrackListComponent},
      {path: 'albums', component: ArtistAlbumListComponent},
    ]
  },
  {path: '**', component: HomeComponent}
];
```

① We added our `AlwaysAuthGuard` to the list of `canActivate` guards for this route.

Since it holds an array we could have multiple guards for a single route.

If this was a `canActivateChild` guard we would be adding it to the `canActivateChild` property and so on for the other guard types.

Now every-time we navigate to the `ArtistComponent` route we get `"AlwaysAuthGuard"` printed to the console so we know that the `AlwaysAuthGuard` is working.

OnlyLoggedInUsersGuard

The most typical use case for the `CanActivate` guard is some form of checking to see if the user has permissions to view a page.

Normally in an Angular application we would have a service which held whether or not the current user is logged in or what permissions they have.

We will simulate this via a mock `UserService` like so:

```
class UserService {
  isLoggedIn(): boolean {
    return false;
  }
}
```

This service has one function `isLoggedIn()` which always returns `false`.

Lets create another guard called `OnlyLoggedInUsersGuard` which only allows logged in users to view a route.

```
@Injectable()
class OnlyLoggedInUsersGuard implements CanActivate { ①
  constructor(private userService: UserService) {}; ②

  canActivate() {
    console.log("OnlyLoggedInUsers");
    if (this.userService.isLoggedIn()) { ③
      return true;
    } else {
      window.alert("You don't have permission to view this page"); ④
      return false;
    }
  }
}
```

① We create a new `CanActivate` guard called `OnlyLoggedInUsersGuard`

② We inject and store `UserService` into the constructor for our class.

③ If the user is logged in the guard passes and lets the user through.

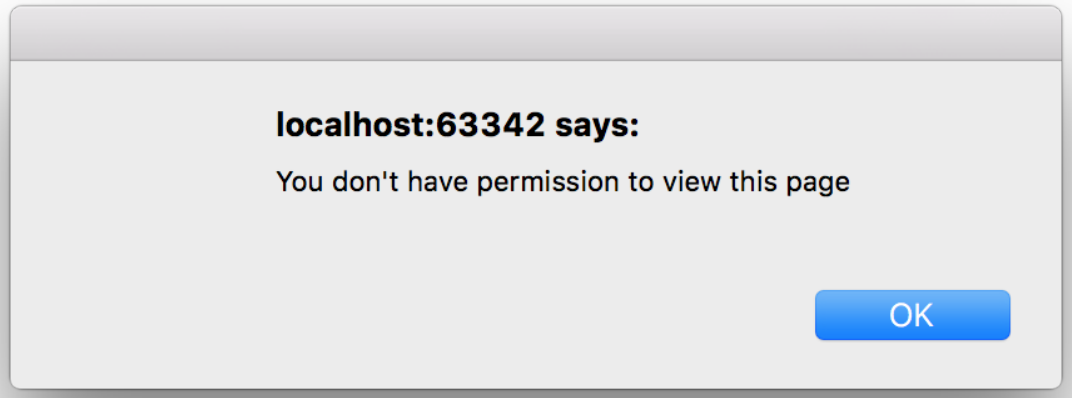
④ If the user is *not* logged in the guard fails, we show the user an alert and the page doesn't navigate to the new URL.

Finally we need to add this guard to the list of guards for our search route, like so:

```
{
  path: 'artist/:artistId',
  component: ArtistComponent,
  canActivate: [OnlyLoggedInUsersGuard, AlwaysAuthGuard], ①
  children: [
    {path: '', redirectTo: 'tracks'},
    {path: 'tracks', component: ArtistTrackListComponent},
    {path: 'albums', component: ArtistAlbumListComponent},
  ]
}
```

① We add `OnlyLoggedInUsersGuard` to the list of guards for our route.

Now when we try to navigate to the search view we are blocked from doing so and shown a window alert, like so:



If we want to redirect users to a login page we may inject `Router` into the constructor and then use the `navigate` function to redirect them to the appropriate login page.

So the rest of the samples in this chapter work we will change the `isLoggedIn` function on our `UserService` to return `true` instead. If you want to play around with this functionality in the plunker remember to switch that back to returning `false`.

CanActivateChild

As well as `CanActivate` we also have `CanActivateChild` which we implement in similar way.

Lets do the same as the `CanActivate` example and create a guard called `AlwaysAuthChildrenGuard`.

```
import {CanActivateChild} from "@angular/router";

class AlwaysAuthChildrenGuard implements CanActivateChild {
  canActivateChild() {
    console.log("AlwaysAuthChildrenGuard");
    return true;
  }
}
```

Remember to provide it on our `NgModule`

We add the guard to the `canActivateChild` child property on our `ArtistComponent` route

```
{
  path: 'artist/:artistId',
  component: ArtistComponent,
  canActivate: [OnlyLoggedInUsersGuard, AlwaysAuthGuard],
  canActivateChild: [AlwaysAuthChildrenGuard],
  children: [
    {path: '', redirectTo: 'tracks'},
    {path: 'tracks', component: ArtistTrackListComponent},
    {path: 'albums', component: ArtistAlbumListComponent},
  ]
}
```

Now every-time we try to activate either the `ArtistTrackListComponent` or `ArtistAlbumListComponent` child routes it checks the `AlwaysAuthChildrenGuard` to see if the user has permission.

Guard function parameters

To help in determining whether or not a guard should accept or deny access the guard function can be passed certain arguments:

1. `component: Component` this is the component itself.
2. `route: ActivatedRouteSnapshot` - this is the future route that will be activated if the guard passes, we can use it's params property to extract the route params.
3. `state: RouterStateSnapshot` - this is the future `RouterState` if the guard passes, we can find the URL we are trying to navigate to from the url property.

CanDeactivate

A third type of guard we can add to our application is a `CanDeactivate` guard which is usually used to warn people if they are navigating away from a page where they have some unsaved changes.

Lets create a simple `CanDeactivate` guard which checks to see if the user navigates *away* from the search page without actually performing a search.

Firstly lets create a function called `canDeactivate` on our `SearchComponent`, it should be the component that decides whether or not it has unsaved changes.

```
canDeactivate() {
  return this.itunes.results.length > 0;
}
```

As a proxy for unsaved changes we are just seeing if the user has performed a search, if so then the results array should be > 0.

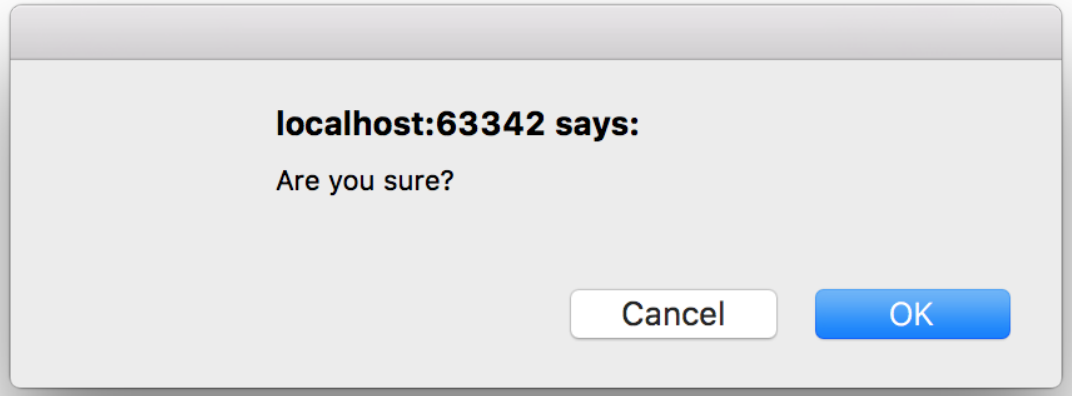
Next lets create a `CanDeactivate` guard.

```
class UnsearchedTermGuard implements CanDeactivate<SearchComponent> { ①
  canDeactivate(component: SearchComponent, ①
                route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): boolean {
    console.log("UnsearchedTermGuard");
    console.log(route.params);
    console.log(state.url);
    return component.canDeactivate() || window.confirm("Are you sure?");
  }
}
```

① We implement from the `CanDeactivate` interface but this is a *generic* interface so needs for us to *additionally* provide it the type of our the component in-between some angle brackets `CanDeactivate<SearchComponent>`. This is so it knows the type of the component to pass into the `canDeactivate` function itself.

1. We setup our guard function to accept the guard function parameters, the component, future activated route and future router state.
2. If `component.canDeactivate()` returns false this will then show a confirmation alert window asking the user if they are sure.

Now if we navigate to the search page, we don't perform a search and then we try to navigate away we get a popup confirmation box. If we say yes we navigate away if we say no we stay on the page, like so:



As of 18/10/2016 there is a bug in Angular version 2.1.0. The `canDeactivate` function is not passed the *future* `ActivatedRouteSnapshot` or the future `RouterStateSnapshot`.

This is why our example use case is so woefully inadequate.

If you want to encourage the Angular team to fix this issue then please give a thumbs up to this issue on github https://github.com/angular/angular/issues/9853

Summary

With guards we can add checks to restrict access to a user to certain pages on our site.

Depending on the type of guard the guard function also has some arguments passed to it which we can take advantage of if we want, namely the future `ActivatedRoute` and the future `RouterState` and for `CanDeactivate` guards we also have the ability to get the component itself.

Guards themselves are just classes and as such can have any other dependencies injected into their constructor so can work in conjunction with other services to figure out if the guard passes or fails.

Guard functions can return either a `boolean` or an `Observable<boolean>` or `Promise<boolean>` which resolves to a `boolean` at some point in the future.

A route can be configured with multiple guards and the guards are checked in the order they were added to the route.

We are almost complete with this section on *Routing*, in the next lecture we will cover the concept of *Path Strategies*.

Listing

script.ts

```
import {NgModule, Component, Injectable} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {JsonpModule, Jsonp, Response} from '@angular/http';
import {ReactiveFormsModule, FormControl, FormsModule} from '@angular/forms';
import {Routes, RouterModule} from "@angular/router";
import {Observable} from 'rxjs';
import 'rxjs/add/operator/toPromise';
import {
    Routes,
    RouterModule,
    Router,
    ActivatedRoute,
    CanActivate,
    CanActivateChild,
    CanDeactivate,
    ActivatedRouteSnapshot,
```

```
    RouterStateSnapshot
} from "@angular/router";


// Domain models

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public link: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

// Services

@Injectable()
class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.trackViewUrl,
                      item.artworkUrl30,
                      item.artistId
                  );
                });
                resolve();
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
```

```
  }
}

@Injectable()
class UserService {
  isLoggedIn(): boolean {
    return true; // Switch to `false` to make OnlyLoggedInUsersGuard work
  }
}

// Components

@Component({
  selector: 'app-search',
  template: `<form class="form-inline">
  <div class="form-group">
    <input type="search"
           class="form-control"
           placeholder="Enter search string"
           #search>
  </div>
  <button type="button"
          class="btn btn-primary"
          (click)="onSearch(search.value)">
    Search
  </button>
</form>

<hr />

<div class="text-center">
  <p class="lead"
     *ngIf="loading">Loading...</p>
</div>

<div class="list-group">
  <a [routerLink]="['/artist', track.artistId]"
     class="list-group-item list-group-item-action"
     *ngFor="let track of itunes.results">
    <img src="{{track.thumbnail}}">
    {{ track.name }} <span class="text-muted">by</span> {{ track.artist }}
  </a>
</div>
 `
})
class SearchComponent {
  private loading: boolean = false;

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

```
    this.route.params.subscribe(params => {
      console.log(params);
      if (params['term']) {
        this.doSearch(params['term'])
      }
    });
  }

  doSearch(term: string) {
    this.loading = true;
    this.itunes.search(term).then(_ => this.loading = false)
  }

  onSearch(term: string) {
    this.router.navigate(['search', {term: term}]);
  }

  canDeactivate() {
    return this.itunes.results.length > 0;
  }
}

@Component({
  selector: 'app-home',
  template: `
<div class="jumbotron">
  <h1 class="display-3">iTunes Search App</h1>
</div>
 `
})
class HomeComponent {
}

@Component({
  selector: 'app-header',
  template: `<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand"
     [routerLink]="['home']">iTunes Search App
  </a>
  <ul class="nav navbar-nav">
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['home']">Home
      </a>
    </li>
    <li class="nav-item"
        [routerLinkActive]="['active']">
      <a class="nav-link"
         [routerLink]="['search']">Search
      </a>
```

```
    </li>
  </ul>
</nav>
 `
})
class HeaderComponent {
  constructor(private router: Router) {
  }

  goHome() {
    this.router.navigate(['']);
  }

  goSearch() {
    this.router.navigate(['search']);
  }
}

@Component({
  selector: 'app-artist-track-list',
  template: `
<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let track of tracks">
    <img src="{{track.artworkUrl30}}">
    <a target="_blank"
       href="{{track.trackViewUrl}}">{{ track.trackName }}
    </a>
  </li>
</ul>
 `
})
class ArtistTrackListComponent {
  private tracks: any[];

  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
    this.route.parent.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&entity=so
ng&callback=JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.tracks = res.json().results.slice(1);
          });
    });
  }
}

@Component({
```

```
  selector: 'app-artist-album-list',
  template: `<ul class="list-group">
  <li class="list-group-item"
      *ngFor="let album of albums">
    <img src="{{album.artworkUrl60}}">
    <a target="_blank"
       href="{{album.collectionViewUrl}}">{{ album.collectionName }}
    </a>
  </li>
</ul>
 `
})
class ArtistAlbumListComponent {
  private albums: any[];

  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
    this.route.parent.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&entity=al
bum&callback=JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.albums = res.json().results.slice(1);
          });
    });
  }
}

@Component({
  selector: 'app-artist',
  template: `<div class="card">
  <div class="card-block">
    <h4>{{artist?.artistName}} <span class="tag tag-
default">{{artist?.primaryGenreName}}</span></h4>
    <hr />
    <footer>
      <ul class="nav nav-pills">
        <li class="nav-item">
          <a class="nav-link"
             [routerLinkActive]="['active']"
             [routerLink]="['./tracks']">Tracks
          </a>
        </li>
        <li class="nav-item">
          <a class="nav-link"
             [routerLinkActive]="['active']"
             [routerLink]="['./albums']">Albums
          </a>
        </li>
```

```
      </ul>
    </footer>
  </div>
</div>

<div class="m-t-1">
  <router-outlet></router-outlet>
</div>
 `
})
class ArtistComponent {
  private artist: any;

  constructor(private jsonp: Jsonp,
              private route: ActivatedRoute) {
    this.route.params.subscribe(params => {

this.jsonp.request(`https://itunes.apple.com/lookup?id=${params['artistId']}&callback=
JSONP_CALLBACK`)
          .toPromise()
          .then(res => {
            console.log(res.json());
            this.artist = res.json().results[0];
            console.log(this.artist);
          });
    });
  }
}

@Component({
  selector: 'app',
  template: `
  <app-header></app-header>
  <div class="m-t-1">
    <router-outlet></router-outlet>
  </div>
 `
})
class AppComponent {
}

// Guards

class AlwaysAuthGuard implements CanActivate {
  canActivate() {
    console.log("AlwaysAuthGuard");
    return true;
  }
}

@Injectable()
```

```
class OnlyLoggedInUsersGuard implements CanActivate {
  constructor(private userService: UserService) {
  };

  canActivate() {
    console.log("OnlyLoggedInUsers");
    if (this.userService.isLoggedIn()) {
      return true;
    } else {
      window.alert("You don't have permission to view this page");
      return false;
    }
  }
}


class AlwaysAuthChildrenGuard implements CanActivateChild {
  canActivateChild() {
    console.log("AlwaysAuthChildrenGuard");
    return true;
  }
}

class UnsearchedTermGuard implements CanDeactivate<SearchComponent> {
  canDeactivate(component: SearchComponent,
                route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): boolean {
    console.log("UnsearchedTermGuard");
    console.log(state.url);
    return component.canDeactivate() || window.confirm("Are you sure?");
  }
}

// Routes

const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent, canDeactivate: [UnsearchedTermGuard]},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    canActivate: [OnlyLoggedInUsersGuard, AlwaysAuthGuard],
    canActivateChild: [AlwaysAuthChildrenGuard],
    children: [
      {path: '', redirectTo: 'tracks', pathMatch: 'full'},
      {path: 'tracks', component: ArtistTrackListComponent},
      {path: 'albums', component: ArtistAlbumListComponent},
    ]
  },
```

```
  {path: '**', component: HomeComponent}
];

// Bootstrap

@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    FormsModule,
    JsonpModule,
    RouterModule.forRoot(routes, {useHash: true})
  ],
  declarations: [
    AppComponent,
    SearchComponent,
    HomeComponent,
    HeaderComponent,
    ArtistAlbumListComponent,
    ArtistTrackListComponent,
    ArtistComponent
  ],
  bootstrap: [AppComponent],
  providers: [
    SearchService,
    UserService,
    OnlyLoggedInUsersGuard,
    AlwaysAuthGuard,
    AlwaysAuthChildrenGuard,
    UnsearchedTermGuard
  ]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Routing Strategies

With client side SPA's we have two strategies we can use to implement client side routing, one is called the `HashLocationStrategy` and the other is called the PathLocationStrategy.

The default in Angular is the `PathLocationStrategy`, if we do nothing that is the strategy Angular will employ.

Learning Objectives

- Know the difference between the `HashLocationStrategy` and `PathLocationStrategy`.
- Know the pros and cons of each and be able to make a judgement call for when to use each.

HashLocationStrategy

To enable `HashLocationStrategy` in an Angular application we pass `{useHash: true}` when we are providing our routes with `RouterModule`, like so:

```
RouterModule.forRoot(routes, {useHash: true})
```

URL can contain some data prepended with a `#` character.

The `#` part of the url is called the *hash fragment*.

It's normally used so that people can link to a particular section in a HTML page, specifically anchor tags, for example if there is an anchor tag with an name attribute of `routing-strategies` like so:

```
<a name="routing-strategies"></a>
```

Then if you created a bookmark of

```
http://somedomain.com/page#routing-strategies
```

Then the browser would open `somedomain.com/page` and then *scroll down* so that the `<a name="routing-strategies"></a>` tag is at the top of the page.

However it has another very important characteristic in that anything past the `#` in a URL *never gets sent to the server*.

So if your URL was `https://codecraft.tv/contact/#/foo/moo/loo` then the browser makes a GET request to `https://codecraft.tv/contact/` **only**.

The `#/foo/moo/loo` part of the URL is **never** sent.

If you were to look at your logs on the server you would never see any reference to `#/foo/moo/loo`

Another way to think about the *hash fragment*, since it's never sent to the server, is that its for storing the state of your *client* application.

It's therefore an ideal solution for implementing *client* side routing:-

- It's part of the URL so can be bookmarked and sent to other people.
- It won't confuse the server side since the hash fragment is never sent to the server.
- It can be programmatically changed via JavaScript.

And that's exactly why, for a number of years, the primary way of implementing client side routing was via hash fragments.

Taking a look at the app we've built so far, if running locally the URLs look something like:

```
localhost:4040/#/search
localhost:4040/#/artist/1234/tracks
```

According to the the server there is only ever one *URL* `localhost:4040`, the other *hash fragment* stuff is ignored by the server.

This is why we call what we are building a *Single Page Application*, there is only ever one *page* requested from the server. In the above example it's `localhost:4040` - the other pages are just changes to the hash fragment which the client application deals with, from the perspective of the server the whole site is just a single page.

PathLocationStrategy

This is the *default* strategy in Angular so we need to do nothing to enable it.

It takes advantage of a relatively new HTML5 API called `pushstate` (from the HTML5 history API).

By using `pushstate` we can change the URL and **not** have the browser request the page from the server and *without* needing to use a hash fragment.

So if we were at

```
localhost:4040/search
```

By using the `pushstate` API we can change the URL to

```
localhost:4040/artist/1234/tracks
```

And the browser **won't** make a GET request to the server for `/artist/1234/tracks`

That sounds perfect for client side routing right?

- We get a URL that looks just like any other URL so can be bookmarked, shared and so on.
- The browser doesn't send the request to the server so the routing is handled on the client side.

Unfortunately it has one big downside, if we then reloaded the page or bookmarked and opened it later the browser **would** make a request to the server for e.g. `localhost:4040/artist/1234/tracks`

By using a hash fragment the server *never* needs to know about any application URL, it will only ever get asked for the root page and it will only ever return the root page.

But by using a `PathLocationStrategy` the server needs to be able to return the main application code for every URL, not just the root URL.

So with `PathLocationStrategy` we need to co-operate with a server side that supports this functionality, it's possible and quite easy to implement a server side like this but it does require some effort and cooperation.

The local development server started by the Angular CLI *does* support this functionality so if you wanted to try it out you can.

base href

When using the `PathLocationStrategy` we need to tell the browser what will be prefixed to the requested path to generate the URL.

We do that by specifying a base href, either in the head section of our HTML like so:

```
<base href='/my/app'/>
```

Or you can *provide* it to the DI framework it via the symbol `APP_BASE_HREF`.

The value of the base href gets prepended to every navigation request we make, so if we ask to navigate to `['moo','foo']` with the above href the URL would change to `/my/app/moo/foo`

Angular Universal

What if we could build an app that rendered the page on the server side and returned that to the client, and from that point on let the client handle the routing?

What if we reloaded the page at `localhost:4040/artist/1234/tracks` and the server at `localhost:4040` rendered the page.

The server called the iTunes APIs and generated the HTML for the tracks page, it returned it to the browser and the browser just displayed it. Then if the user clicked on search the client application takes over and handles the routing on the client side.

That is something called *Angular Universal*, or *Isomorphic Rendering* but essentially it's the ability to run Angular in both the *browser* and the *server side*.

The big benefit of Angular Universal is that pages can be cached on the server side and applications

will then load much faster.

For Angular Universal to work URLs need to be passed to the server side which is why it can only work with a `PathLocationStrategy` and not a `HashLocationStrategy`.

Summary

The default client side routing strategy used in Angular is the `PathLocationStrategy`.

This changes the URL programmatically using the HTML5 History API in such a way that the browser doesn't make a request to the server for the new URL.

For this to work we do need to serve our Angular application from a server that supports requests on multiple different URLs, at a minimum all this server side needs to do is return the same page for all the different URLs that's requested from it.

It's not a lot of work but does need some co-operation from the server side.

`PathLocationStrategy` also sets us up for a future architecture where we can speed up loading time by pre-rendering the pages with Angular running on the server side and then once it's downloaded to the browser the client can take over routing. This is called *Angular Universal* and it's currently in development.

`HashLocationStrategy` uses the hash fragment part of the URL to store state for the client, it easier to setup and doesn't require any co-operation from the server side but has the downside that it won't work with *Angular Universal* once that's released.

Wrapping Up

In this section we learnt how to build a *Single Page Application* (SPA) in Angular.

An SPA is one were the client application in the browser handles changes to the URL instead og the browser requesting the URL from a server.

With Angular we map the URL in the browser to *components* we display on the page, this is called the *Component Router*.

We use directives called `router-outlet` to tell Angular where we want those components inserted into the page.

The mapping above is what we call a *Route* and we configure Angular with a series of routes by *providing* it to the main `NgModule` using the `RouterModule.forRoot(routes)` function.

We can have part of the URL be a variable which we can pass into the component when it gets inserted into the page via the `ActivatedRoute` service, this is called a *Parameterised Route*.

We can *nest* routes, i.e. have a tree of Components nested one inside another by having multiple `router-outlet` directives and also configuring child routes in the router configuration.

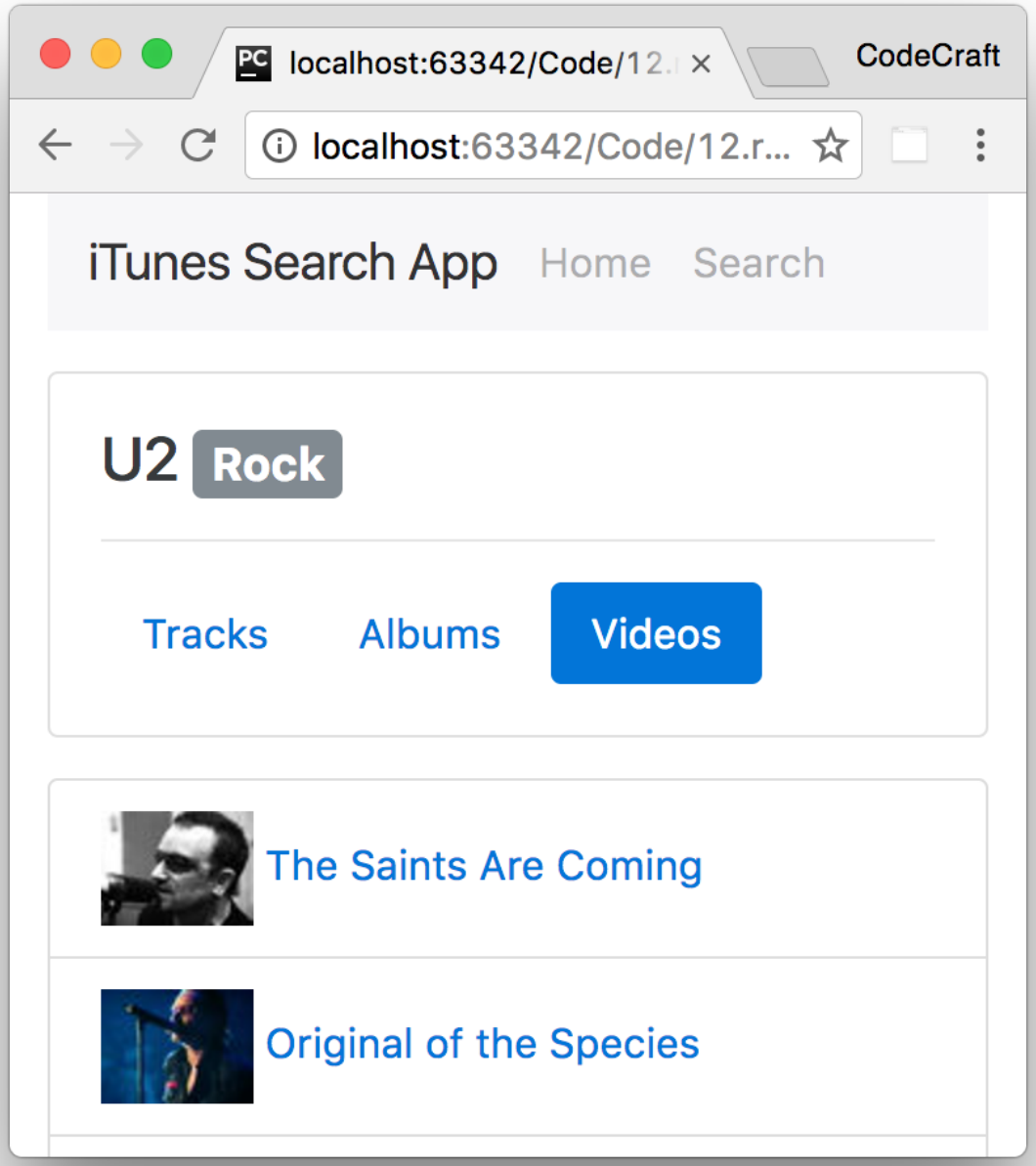
We can block access to certain routes by using *Router Guards*. Using *Router Guards* we can even warn the user if they are about to leave a page with unsaved changes.

There are two strategies we can use for performing client side routing in Angular, the HashLocationStrategies which makes use of the *hash fragment* part of a URL or we can use the `PathLocationStrategy` which takes advantages of the `pushstate` HTML5 API.

Activity

Extend the application we created in this section.

Add another child route which shows a list of music videos for a given artist.



Use `entity=musicVideo` in the iTunes API to return a set of music videos.

Steps

Fork this plunker:

http://plnkr.co/edit/4ra2p8NlVg9uepSaIHph?p=preview

Complete the `ArtistMusicVideoListComponent` component.

Remember you also need to make changes to the route configuration and `NgModule`!

Read any **TODO** comments in the plunker for hints.

Solution

When you are ready compare your answer to the solution in this plunker:

http://plnkr.co/edit/isEQpsCw59ElUZZy4LIu?p=preview

Unit Testing

Overview

Just like it's predecessor Angular 1, Angular has been designed with testability as a primary goal.

When we talking about testing in Angular we are usually talking about two different types of testing:

Unit Testing

This is sometimes also called *Isolated* testing. It's the practice of testing small isolated pieces of code. If your test uses some external resource, like the network or a database, it's *not* a unit test.

Functional Testing

This is defined as the testing of the complete functionality of an application. In practice with web apps, this means interacting with your application as it's running in a browser just like a user would interact with it in real life, i.e. via clicks on a page.

This is also called ***End To End*** or ***E2E*** testing.

In this section we are only going to cover how to perform *unit testing* with Angular.

You will learn:

- How to write and run tests in Angular using Jasmine & Karma.
- How to write tests for classes, components, directives and pipes.
- How to use the Angular Test Bed to help test behaviours that depend on the Angular framework.
- How to write tests using mocks and spys.
- How to write tests that depend on change detection.
- How to write tests that involve an asynchronous action.
- How to write tests that require the dependency injection framework.
- How to write tests that involve the Http service.
- How to write tests that involve model driven forms (template driven forms require E2E testing).
- How to write tests to test the Router.

Jasmine & Karma

We can test our Angular applications from scratch by writing and executing pure javascript functions. Creating instances of the relevant classes, calling functions and checking the actual versus expected result.

But since testing is such a common activity with javascript there are a number of testing libraries and frameworks we can use which reduce the amount of time it takes to write tests.

Two such tools and frameworks that are used when testing Angular is *Jasmine* and *Karma* and a discussion of those is the topic of this lecture.

Learning Objectives

- What is the Jasmine test framework?
- How to write tests in Jasmine?
- What is the Karma test runner?
- How to create and run tests using the Angular CLI?
- How to create and run tests in Plunker?

Jasmine

Jasmine is a javascript testing framework that supports a software development practice called Behaviour Driven Development, or BDD for short. It's a specific flavour of Test Driven Development (TDD).

Jasmine, and BDD in general, attempts to describe tests in a human readable format so that non-technical people can understand what is being tested. However even if you *are* technical reading tests in BDD format makes it a lot easier to understand what's going on.

For example if we wanted to test this function:

```
function helloWorld() {
  return 'Hello world!';
}
```

We would write a jasmine test *spec* like so:

```
describe('Hello world', () => { ①
  it('says hello', () => { ②
    expect(helloWorld()) ③
        .toEqual('Hello world!'); ④
  });
});
```

① The `describe(string, function)` function defines what we call a *Test Suite*, a collection of individual *Test Specs*.

② The `it(string, function)` function defines an individual *Test Spec*, this contains one or more *Test Expectations*.

③ The `expect(actual)` expression is what we call an *Expectation*. In conjunction with a *Matcher* it describes an *expected* piece of behaviour in the application.

④ The `matcher(expected)` expression is what we call a *Matcher*. It does a boolean comparison with the `expected` value passed in vs. the `actual` value passed to the `expect` function, if they are false the *spec* fails.

Built-in matchers

Jasmine comes with a few pre-built matchers like so:

```
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);
```

You can see concrete examples of how these matchers are used by looking at the Jasmine docs here: http://jasmine.github.io/edge/introduction.html#section-Included_Matchers

Setup and teardown

Sometimes in order to test a feature we need to perform some setup, perhaps it's creating some test objects. Also we may need to perform some cleanup activities after we have finished testing, perhaps we need to delete some files from the hard drive.

These activities are called *setup* and *teardown* (for cleaning up) and Jasmine has a few functions we can use to make this easier:

beforeAll

This function is called **once**, *before* all the specs in `describe` test suite are run.

afterAll

This function is called **once** *after* all the specs in a test suite are finished.

beforeEach

This function is called *before* **each** test specification, `it` function, has been run.

afterEach

This function is called *after* **each** test specification has been run.

We might use these functions like so:

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello World";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
        .toEqual(expected);
  });
});
```

Running Jasmine tests

To manually run Jasmine tests we would create a HTML file and include the required jasmine javascript and css files like so:

```
<link rel="stylesheet" href="jasmine.css">
<script src="jasmine.js"></script>
<script src="jasmine-html.js"></script>
<script src="boot.js"></script>
```

We then load in the parts of our application code that we want to test, for example if our *hello world* function was in `main.js`:

```
<link rel="stylesheet" href="jasmine.css">
<script src="jasmine.js"></script>
<script src="jasmine-html.js"></script>
<script src="boot.js"></script>

<script src="main.js"></script>
```

The order of script tags is important.

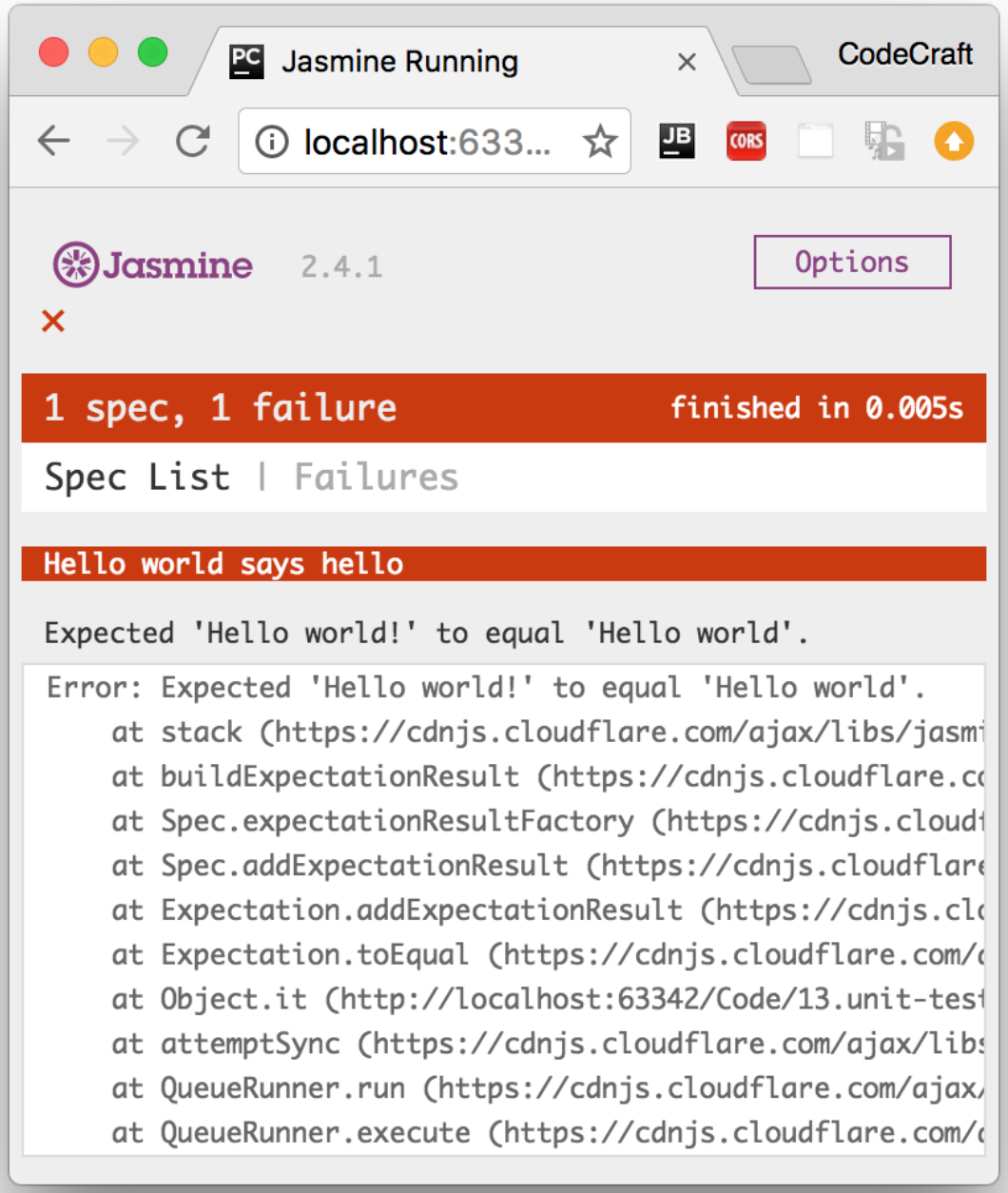
We then would load each individual test suite file, for example if we placed our test suite code above in a file called `test.js` we would load it in like so:

```
<link rel="stylesheet" href="jasmine.css">
<script src="jasmine.js"></script>
<script src="jasmine-html.js"></script>
<script src="boot.js"></script>

<script src="main.js"></script>

<script src="test.js"></script>
```

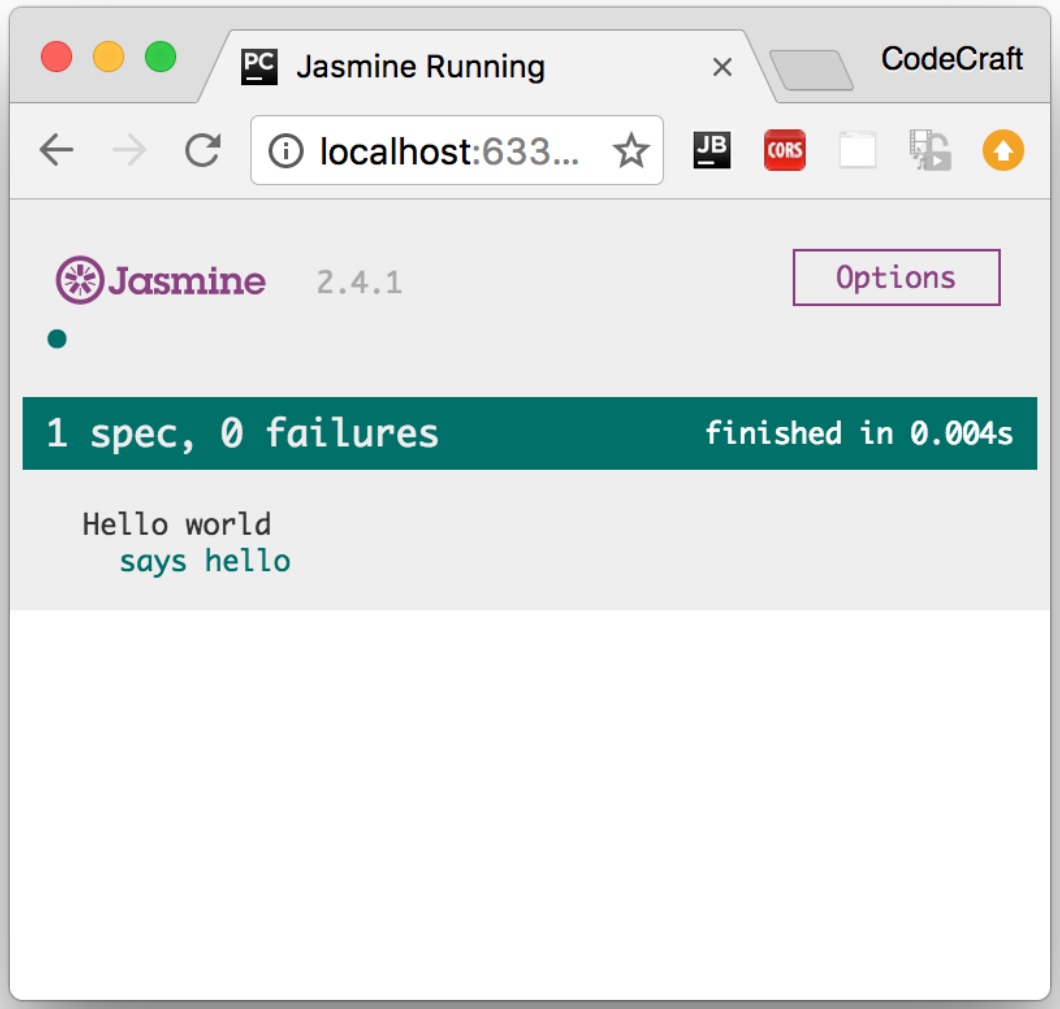
To run the tests we simply open up the HTML file in a browser.

Once all the files requested via `script` and `link` are loaded by a browser the function `window.onload` is called, this is when Jasmine actually runs the tests.

The results are displayed in the browser window, a failing run looks like:



A passing run looks like:



If we wanted to test our code in different browsers we simply load up the HTML file in the browser we want to test in.

If we wanted to debug the code we would use the developer tools available to us in that browser.

Karma

Manually running Jasmine tests by refreshing a browser tab repeatedly in different browsers every-time we edit some code can become tiresome.

Karma is a tool which lets us spawn browsers and run jasmine tests inside of them all from the command line. The results of the tests are also displayed on the command line.

Karma can also watch your development files for changes and re-run the tests automatically.

Karma lets us run jasmine tests as part of a development tool chain which requires tests to be runnable and results inspectable via the command line.

It's not necessary to know the internals of how Karma works. When using the Angular CLI it handles the configuration for us and for the rest of this section we are going to run the tests using only Jasmine.

Angular CLI

When creating Angular projects using using the Angular CLI it defaults to creating and running unit tests using Jasmine and Karma.

Whenever we create files using the CLI as well as creating the main code file it also creates simple jasmine spec file named the same as the main code file but ending in .spec.ts, like so:

If we create a Pipe using the CLI like so:

```
ng generate pipe My
```

This would create two files:

- `my-pipe.ts` - This is the main code file where we put the code for the pipe.
- `my-pipe.spec.ts` - This is the jasmine test suite for the pipe.

The spec file will have some code already bootstrapped, like so:

```
/* tslint:disable:no-unused-variable */

import { TestBed, async } from '@angular/core/testing';
import { MyPipe } from './my.pipe';

describe('Pipe: My', () => {
  it('create an instance', () => {
    let pipe = new MyPipe();
    expect(pipe).toBeTruthy();
  });
});
```

The code that gets bootstrapped depends on the item that we are creating.

To run all the tests in our application we simply type `ng test` in our project root.

This runs all the tests in our project in Jasmine via Karma.

It watches for changes to our development files, bundles all the developer files together and re-runs the tests automatically.

Angular Plunker

When building real Angular applications I recommend sticking with the file and folder structure

defined by the Angular CLI as well as using the built-in test runner.

However for this section, to give you an easy way to view and play with the code we are going to use *only* Jasmine and execute tests by refreshing a browser.

This so we can easily share code via a Plunker like we have done for all the other sections in this course.

An Angular Jasmine Plunker looks very similar to a normal Jasmine Plunker appart from a few key differences:

1. We also include the required Angular libraries and some patches for Jasmine so it works better with Angular.

```
<script src="https://unpkg.com/systemjs@0.19.27/dist/system.src.js"></script>
<script src="https://unpkg.com/reflect-metadata@0.1.3"></script>
<script src="https://unpkg.com/zone.js@0.6.25?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/long-stack-trace-
zone.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/proxy.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/sync-test.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/jasmine-patch.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/async-test.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/fake-async-test.js?main=browser"></script>
```

2. We then add the spec files we want to test into a special array called `spec_files`.

```
var __spec_files__ = [
  'app/auth.service.spec'
];
```

3. We then load a *shim* javascript file which triggers running the test specs once we have finished transpiling and loading the transpiled files in the browser.

```
<script src="browser-test-shim.js"></script>
```

Disabled and focused tests

You can disable tests without commenting them our by just pre-pending `x` to the `describe` or `it` functions, like so:

```
xdescribe('Hello world', () => { ①
  xit('says hello', () => { ①
    expect(helloWorld())
        .toEqual('Hello world!');
  });
});
```

① These tests will not be run.

Conversely you can also focus on specific tests by pre-pending with `f`, like so:

```
fdescribe('Hello world', () => { ①
  fit('says hello', () => { ①
    expect(helloWorld())
        .toEqual('Hello world!');
  });
});
```

① Out of all the tests in all the tests suites and tests specs, these are the only ones that will be run.

Summary

Jasmine is a testing framework that supports *Behavior Driven Development*. We write tests in *Test Suites* which are composed of one or more *Test Specs* which themselves are composed of one or more *Test Expectations*.

We can run Jasmine tests in a browser ourselves by setting up and loading a HTML file, but more commonly we use a command line tool called Karma. Karma handles the process of creating HTML files, opening browsers and running tests and returning the results of those tests to the command line.

If you use the Angular CLI to manage projects it automatically creates stub jasmine spec files for you when generating code. It also handles the Karama configuration, transpilation and bundling of your files so all you need to do in order to run your tests is type the command `ng test`.

For the purposes of this section we will be using a simple browser based Jasmine test runner so we can share the code easily via plunker.

Listing

http://plnkr.co/edit/8ApdkvletoEc4Q0maXSN?p=preview

index.html

```
<!-- Run application specs in a browser -->
<!DOCTYPE html>
<html>
<head>
  <title>Jasmine Running</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.css">
</head>
<body>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine-
html.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/boot.js"></script>
  <script src="main.js"></script>
  <script src="test.js"></script>
</body>

</html>
```

main.js

```
function helloWorld() {
  return 'Hello world!';
}
```

test.js

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello world!";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
        .toEqual(expected);
  });
});
```

Testing Classes & Pipes

Learning Objectives

- How to unit test an instance of a class.

Sample class & test suite

We'll start our unit testing journey with all you will ever need to know, how to test a `class`.

Everything in Angular is an instance of a class, be it a Component, Directive, Pipe and so on. So once you know how to test a basic class you can test everything.

Let's imagine we have a simple class called `AuthService` it's something we want to provide to Angulars DI framework but that doesn't play a part in how we want to test it.

app/auth.service.ts

```
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

It has one function called `isAuthenticated` which returns `true` if there is a token stored in the browsers `localStorage`.

To test this class we create a test file called `auth.service.spec.ts` that sits next to our `auth.service.ts` file, like so:

app/auth.service.spec.ts

```
import {AuthService} from './auth.service'; ①

describe('Service: Auth', () => { ②
});
```

① We first import the `AuthService` class we want to run our tests against.

② We add a `describe` test suite function to hold all our individual test specs.

Setup & teardown

We want to run our test specs against *fresh* instances of `AuthService` so we use the `beforeEach` and `afterEach` functions to setup and clean instances like so:

app/auth.service.spec.ts

```
describe('Service: Auth', () => {
  let service: AuthService;

  beforeEach(() => { ①
    service = new AuthService();
  });

  afterEach(() => { ②
    service = null;
    localStorage.removeItem('token');
  });
});
```

① Before each test spec is run we create a new instance of `AuthService` and store on the `service` variable.

② After each test spec is finished we null out our `service` and also remove any tokens we stored in localStorage.

Creating test specs

Now we create some test specs, the first spec I want to create should check if the `isAuthenticated` function returns `true` when there is a token.

```
it('should return true from isAuthenticated when there is a token', () => { ①
  localStorage.setItem('token', '1234'); ②
  expect(service.isAuthenticated()).toBeTruthy(); ③
});
```

① We pass to the `it` function a human readable description of what we are testing. This is shown in the test report and makes it easy to understand what feature isn't working.

② We setup some *spec only* data in local storage which should trigger the effect we want.

③ We test an expectation that the `service.isAuthenticated()` function returns something that resolves to `true`.

We also want to test the reverse case, when there is no token the function should return `false`:

```
it('should return false from isAuthenticated when there is no token', () => {
  expect(service.isAuthenticated()).toBeFalsy();
});
```

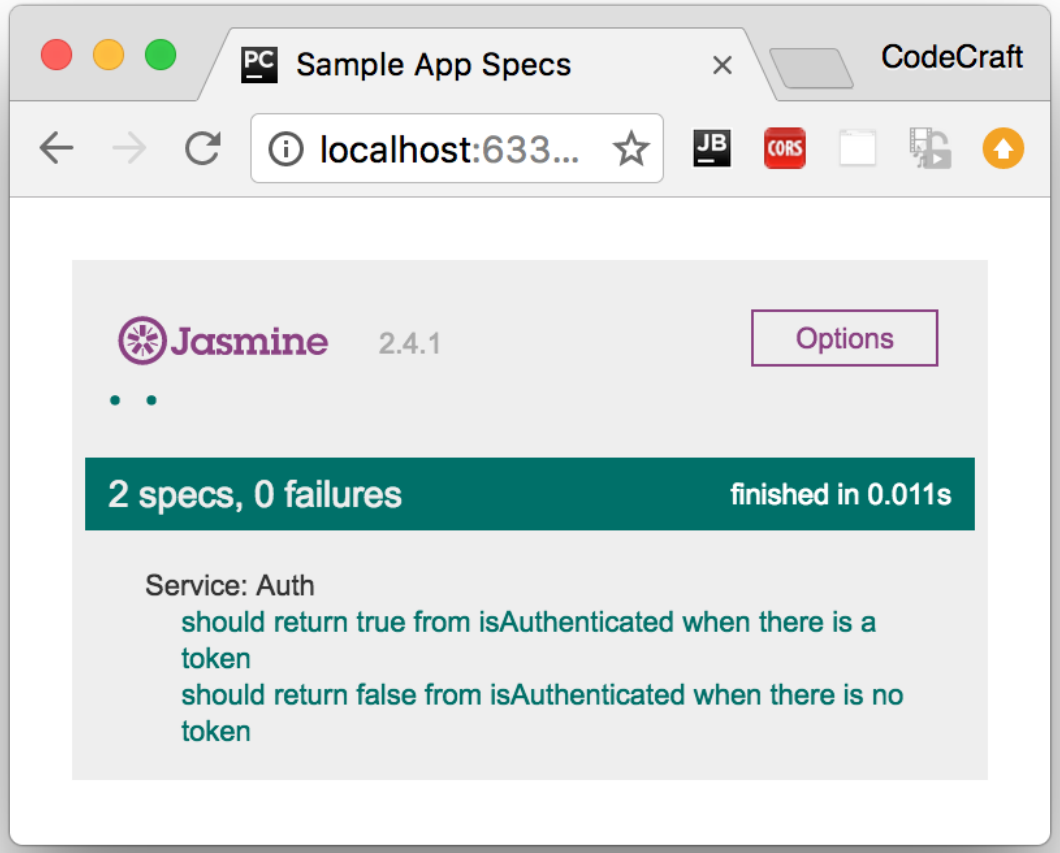
We *know* that in this function the `token` isn't set since we make sure to clear out the token in the `afterEach` function.

We now test an expectation that the `service.isAuthenticated()` function returns something that

resolves to `false`.

Running the tests

To run our tests we simply open up the HTML file in the browser, you can just click the plunker link and make sure to press run in the toolbar.



Pipes

Pipes are by far the simplest part of Angular, they can be implemented as a class with one function and therefore can be tested with *just* Jasmine and the knowledge we've gained so far.

In the section on pipes we built one called `DefaultPipe`, this pipe lets us provide default values for variables in templates like so:

```
{{ image | default:"http://example.com/default-image.png" }}
```

The code for this pipe looked like so:

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

@Pipe({
  name: 'default'
})
export class DefaultPipe implements PipeTransform {

  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }
    return image;
  }
}
```

Our starting test suite file looks like so:

```
describe('Pipe: Default', () => {
  let pipe: DefaultPipe;

  beforeEach(() => {
    pipe = new DefaultPipe();
  });
});
```

In our setup function we create an instance of our pipe class.

Pipe classes have one function called `transform` so in order to test pipes we just need to test this one function, passing inputs and expecting outputs.

Our first test spec checks to see that if the pipe doesn't recieve an input it returns the default value, like so:

```
it('providing no value returns fallback', () => {
  expect(pipe.transform('', 'http://place-hold.it/300')).toBe('http://place-
hold.it/300');
});
```

We pass in empty string as the input to the transform function and therefore it returns the second

argument back to us.

For testing pipes there isn't much else to it, we simply check the various inputs and expected ouputs of our transform function.

In order to run this test spec file in our test Plunker remember to add it to the list of test spec files in the `spec_files` array.

If your *Pipe* requires dependencies to be injected into the constructor it might be better to use the *Angular Test Bed* which we cover later on in this section.

Summary

That's it really, we can test any *isolated* class that doesn't require anything else with a simple jasmine spec file, nothing more complex required.

Since everything in Angular is represented as classes, we could stop here - you have most of the tools already to write tests for directives, components, pipes and so on.

However our code often requires *other* code to work, it has dependencies. So how we write *isolated* tests for pieces of code which by nature are not isolated and need dependencies is the topic of the next lecture.

Listing

http://plnkr.co/edit/5uULMJkAgug0e4iqnbi2?p=preview

auth.service.ts

```
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

auth.service.spec.ts

```
import {AuthService} from './auth.service';

describe('Service: Auth', () => {

  let service: AuthService;

  beforeEach(() => {
    service = new AuthService();
  });

  afterEach(() => {
    service = null;
    localStorage.removeItem('token');
  });

  it('should return true from isAuthenticated when there is a token', () => {
    localStorage.setItem('token', '1234');
    expect(service.isAuthenticated()).toBeTruthy();
  });

  it('should return false from isAuthenticated when there is no token', () => {
    expect(service.isAuthenticated()).toBeFalsy();
  });

});
```

default.pipe.ts

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

@Pipe({
  name: 'default'
})
export class DefaultPipe implements PipeTransform {

  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }
    return image;
  }
}
```

default.pipe.spec.ts

```
/* tslint:disable:no-unused-variable */

import {DefaultPipe} from './default.pipe';

describe('Pipe: Default', () => {
  let pipe: DefaultPipe;

  beforeEach(() => {
    pipe = new DefaultPipe();
  });

  it('providing no value returns fallback', () => {
    expect(pipe.transform('', 'http://place-hold.it/300')).toBe('http://place-
hold.it/300');
  });

  it('providing a value returns value', () => {
    expect(pipe.transform('http://place-hold.it/300', 'fallback')).toBe('http://place-
hold.it/300');
  });

  it('asking for https returns https', () => {
    expect(pipe.transform('', 'http://place-hold.it/300', true)).toBe('https://place-
hold.it/300');
  });
});
```

Testing with Mocks & Spies

In this lecture we are going to discuss how to test classes which have dependencies in isolation by using *Mocks*.

Learning Objectives

- How to *Mock* with fake classes.
- How to *Mock* by extending classes and overriding functions.
- How to *Mock* by using a real instance and a *Spy*.

Sample code

Let's imagine we have a `LoginComponent` which works *with* the `AuthService` we tested in the previous lecture, like so:

login.component.ts

```
import {Component} from '@angular/core';
import {AuthService} from "./auth.service";

@Component({
  selector: 'app-login',
  template: `<a [hidden]="needsLogin()">Login</a>`
})
export class LoginComponent {

  constructor(private auth: AuthService) {
  }

  needsLogin() {
    return !this.auth.isAuthenticated();
  }
}
```

We inject the `AuthService` into the `LoginComponent` and the component shows a *Login* button if the `AuthService` says the user isn't *authenticated*.

The `AuthService` is the same as the previous lecture:

auth.service.ts

```
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

Testing with the real `AuthService`

We could test the `LoginComponent` by using a real instance of `AuthService` but if you remember to *trick* `AuthService` into returning `true` for the `isAuthenticated` function we needed to setup some data via `localStorage`.

```
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: AuthService;

  beforeEach(() => { ①
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => { ②
    localStorage.removeItem('token');
    service = null;
    component = null;
  });


  it('canLogin returns false when the user is not authenticated', () => {
    expect(component.needsLogin()).toBeTruthy();
  });

  it('canLogin returns false when the user is not authenticated', () => {
    localStorage.setItem('token', '12345'); ③
    expect(component.needsLogin()).toBeFalsy();
  });
});
```

① We create an instance of `AuthService` and inject it into out `LoginComponent` when we create it.

② We clean up data and localStorage after each test spec has been run.

③ We setup some data in `localStorage` in order to get the behaviour we want from `AuthService`.

So in order to test `LoginComponent` we would need to know the *inner workings* of `AuthService`.

That's not very *isolated* but also not *too* much to ask for in this scenario. However imagine if `LoginComponent` required a number of *other* dependencies in order to run, we would need to know the inner workings of a number of other classes just to test our `LoginComponent`.

This results in *Tight Coupling* and our tests being very *Brittle*, i.e. likely to break easily. For example if the `AuthService` changed *how* it stored the token, from `localStorage` to `cookies` then the `LoginComponent` test would break since *it* would still be setting the token via `localStorage`.

This is why we need to test classes in *isolation*, we just want to worry about `LoginComponent` and not about the myriad of other things `LoginComponent` depends on.

We achieve this by *Mocking* our dependencies. Mocking is the act of creating something that looks like the dependency but is something *we* control in our test. There are a few methods we can use to create mocks.

Mocking with fake classes

We can create a fake `AuthService` called `MockedAuthService` which just returns whatever we want for our test.

We can even remove the `AuthService` import if we want, there really is no dependency on anything else. The `LoginComponent` is tested in isolation:

```
import {LoginComponent} from './login.component';

class MockAuthService { ①
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
  }
}

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: MockAuthService;

  beforeEach(() => { ②
    service = new MockAuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    service = null;
    component = null;
  });


  it('canLogin returns false when the user is not authenticated', () => {
    service.authenticated = false; ③
    expect(component.needsLogin()).toBeTruthy();
  });

  it('canLogin returns false when the user is not authenticated', () => {
    service.authenticated = true; ③
    expect(component.needsLogin()).toBeFalsy();
  });
});
```

① We create a class called `MockAuthService` which has the same `isAuthenticated` function as the real `AuthService` class. The one difference is that we can control what `isAuthenticated` returns by setting the value of the `authenticated` property.

② We inject into our `LoginComponent` an instance of the `MockAuthService` instead of the real `AuthService`.

③ In our tests we trigger the behaviour we want from the service by setting the `authenticated` property.

By using a fake `MockAuthService` we:

- Don't depend on the real `AuthService`, in fact we don't even need to import it into our specs.

- Make our code less brittle, if the inner workings of the real `AuthService` ever changes our tests will still be valid and still work.

Mocking by overriding functions

Sometimes creating a complete fake copy of a real class can be complicated, time consuming and unnecessary.

We can instead simply extend the class and override one or more specific function in order to get them to return the test responses we need, like so:

```
class MockAuthService extends AuthService {
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
  }
}
```

In the above class `MockAuthService` *extends* `AuthService`. It would have access to all the other functions and properties that exist on `AuthService` but only override the `isAuthenticated` function so we can easily control it's behaviour and isolate our `LoginComponent` test.

The rest of the test suite using mocking via overriding functions is the same as the previous version with fake classes.

Mock by using a real instance with Spy

A *Spy* is a feature of Jasmine which lets you take an existing class, function, object and *mock* it in such a way that you can control what gets returned from functions.

Let's re-write our test to use a Spy on a real instance of `AuthService` instead, like so:

```
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: AuthService;
  let spy: any;

  beforeEach(() => { ①
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => { ②
    service = null;
    component = null;
  });


  it('canLogin returns false when the user is not authenticated', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(false); ③
    expect(component.needsLogin()).toBeTruthy();
    expect(service.isAuthenticated).toHaveBeenCalled(); ④

  });

  it('canLogin returns false when the user is not authenticated', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
    expect(component.needsLogin()).toBeFalsy();
    expect(service.isAuthenticated).toHaveBeenCalled();
  });
});
```

① We create a real instance of `AuthService` and inject it into the `LoginComponent`.

② In our teardown function there is no need to delete the token from `localStorage`.

③ We create a *spy* on our `service` so that if the `isAuthenticated` function is called it returns `false`.

④ We can even check to see if the `isAuthenticated` function was called.

By using the spy feature of jasmine we can make any function return anything we want:

```
spyOn(service, 'isAuthenticated').and.returnValue(false);
```

In our example above we make the `isAuthenticated` function return `false` or true in each test spec according to our needs.

Summary

Testing with real instances of dependencies causes our test code to know about the inner workings of other classes resulting in tight coupling and brittle code.

The goal is to test pieces of code in isolation without needing to know about the inner workings of their dependencies.

We do this by creating Mocks; we can create Mocks using fake classes, extending existing classes or by using real instances of classes but taking control of them with Spys.

Listing

http://plnkr.co/edit/08ppx8olCnTMpkPdW3eC?p=preview

login.component.ts

```
import {Component} from '@angular/core';
import {AuthService} from "./auth.service";

@Component({
  selector: 'app-login',
  template: `<a [hidden]="needsLogin()">Login</a>`
})
export class LoginComponent {

  constructor(private auth: AuthService) {
  }

  needsLogin() {
    return !this.auth.isAuthenticated();
  }
}
```

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: AuthService;
  let spy: any;

  beforeEach(() => {
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    service = null;
    component = null;
  });


  it('canLogin returns false when the user is not authenticated', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(false);
    expect(component.needsLogin()).toBeTruthy();
    expect(service.isAuthenticated).toHaveBeenCalled();

  });

  it('canLogin returns false when the user is not authenticated', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
    expect(component.needsLogin()).toBeFalsy();
    expect(service.isAuthenticated).toHaveBeenCalled();
  });
});
```

Angular Test Bed

The Angular Test Bed (ATB) is a higher level *Angular Only* testing framework that allows us to easily test behaviours that depend on the Angular Framework.

We still write our tests in Jasmine and run using Karma but we now have a slightly easier way to create components, handle injection, test asynchronous behaviour and interact with our application.

This lecture will be an introduction to the ATB and we will continue to use it for the rest of this section.

Learning Objectives

- What is the ATB and how to use it.
- When to use ATB vs. plain vanilla Jasmine tests.

Configuring

Lets demonstrate how to use the *ATB* by converting the component we tested with plain vanilla Jasmine to one that uses the *ATB*.

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });
  });
});
```

In the `beforeEach` function for our test suite we *configure* a testing module using the `TestBed` class.

This creates a test *Angular Module* which we can use to instantiate components, perform dependency injection and so on.

We configure it in exactly the same way as we would configure a normal `NgModule`. On this case we pass in the `LoginComponent` in the declarations and the `AuthService` in the providers.

Fixtures and DI

Once the ATB is setup we can then use it to instantiate components and resolve dependencies, like so:

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>; ①
  let authService: AuthService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent); ②

    // get test component from the fixture
    component = fixture.componentInstance; ③

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService); ④

  });
});
```

① A `fixture` is a wrapper for a component *and* it's template.

② We create an instance of a component fixture through the `TestBed`, this injects the `AuthService` into the component constructor.

③ We can find the actual *component* from the `componentInstance` on the `fixture`.

④ We can get resolve dependencies using the TestBed injector by using the `get` function.

Since the `LoginComponent` doesn't have it's own child injector the AuthService that gets injected in is the same one as we get from the `TestBed` above.

Test specs

Now we've configured the `TestBed` and extracted the component and service we can run through the same test specs as before:

```
it('canLogin returns false when the user is not authenticated', () => {
  spyOn(authService, 'isAuthenticated').and.returnValue(false);
  expect(component.needsLogin()).toBeTruthy();
  expect(authService.isAuthenticated).toHaveBeenCalled();
});

it('canLogin returns false when the user is not authenticated', () => {
  spyOn(authService, 'isAuthenticated').and.returnValue(true);
  expect(component.needsLogin()).toBeFalsy();
  expect(authService.isAuthenticated).toHaveBeenCalled();
});
```

When to use ATB

We will continue to use ATB for the rest of this section because:

- It allows us to test the interaction of a directive or component with it's template.
- It allows us to easily test change detection.
- It allows us to test and use Angulars DI framework,
- It allows us to test using the `NgModule` configuration we use in our application.
- It allows us to test user interaction via clicks & input fields

Summary

The ATB lets us test parts of our code as if it is being run in the context of a real Angular app.

It's usefulness will become more apparent in future lectures, the next one being how to use the ATB to test change detection and property binding.

Listing

http://plnkr.co/edit/jqw3ddMQU7zPQg9KBXJE?p=preview

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: AuthService;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService);

  });

  it('canLogin returns false when the user is not authenticated', () => {
    spyOn(authService, 'isAuthenticated').and.returnValue(false);
    expect(component.needsLogin()).toBeTruthy();
    expect(authService.isAuthenticated).toHaveBeenCalled();
  });

  it('canLogin returns false when the user is not authenticated', () => {
    spyOn(authService, 'isAuthenticated').and.returnValue(true);
    expect(component.needsLogin()).toBeFalsy();
    expect(authService.isAuthenticated).toHaveBeenCalled();
  });
});
```

Testing Change Detection

Trying to test whether changes in the state of our application trigger changes in the view without the Angular Test Bed is complicated. However with the ATB it's much simpler.

In this lecture start interacting with our components template. We add a test to make sure that the bindings in the view updates as we expect when variables in our component change.

Learning Objectives

- How to inspect a components view.
- How to trigger change detection so a components view updates based on state changes in our application.

Setup

We'll continue testing our `LoginComponent` from previous lectures but this time we'll update the template so we have both a *Login* and *Logout* button like so:

```
@Component({
  selector: 'app-login',
  template: `
  <a>
    <span *ngIf="needsLogin()">Login</span>
    <span *ngIf="!needsLogin()">Logout</span>
  </a>
`
})
export class LoginComponent {

  constructor(private auth: AuthService) {
  }

  needsLogin() {
    return !this.auth.isAuthenticated();
  }
}
```

Our test spec file starts close to the version we had in the last lecture like so:

```
/* tslint:disable:no-unused-variable */
import {TestBed, async, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";
import {DebugElement} from "@angular/core"; ①
import {By} from "@angular/platform-browser"; ①

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: AuthService;
  let el: DebugElement; ②

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService);

    //  get the "a" element by CSS selector (e.g., by class name)
    el = fixture.debugElement.query(By.css('a')); ③
  });
});
```

① We've imported a few more classes that are needed when interacting with a components view, `DebugElement` and `By`.

② We have another variable called `el` which holds something called a `DebugElement`.

③ We store a reference to a DOM element in our `el` variable.

The `fixture` as well as holding an instance of the component also holds a reference to something called a `DebugElement`, this is a wrapper to the low level DOM element that represents the components view, via the `debugElement` property.

We can get references to other child nodes by querying this `debugElement` with a `By` class. The `By` class lets us query using a a number of methods, one is via a css class like we have in our example another way is to request by a type of directive like `By.directive(MyDirective)`.

We request a reference to the `a` tag that exists in the components view, this is the button which either says *Login* or *Logout* depending on whether the `AuthService` says the user is authenticated or not.

We can find out the text content of the tag by calling `el.nativeElement.textContent.trim()`, we'll be using that snippet in the test specs later on.

Lets now add a basic test spec like so:

```
it('login button hidden when the user is authenticated', () => {
  // TODO
});
```

Detect Changes

The first expectation we place in our test spec might look a bit strange

```
it('login button hidden when the user is authenticated', () => {
  expect(el.nativeElement.textContent.trim()).toBe('');
});
```

We initially *expect* the text inside the `a` tag to be *blank*.

That's because when Angular first loads no change detection has been triggered and therefore the view doesn't show either the `Login` or `Logout` text.

`fixture` is a wrapper for our components environment so we can control things like change detection.

To trigger change detection we call the function `fixture.detectChanges()`, now we can update our test spec to:

```
it('login button hidden when the user is authenticated', () => {
  expect(el.nativeElement.textContent.trim()).toBe('');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
});
```

Once we trigger a change detection run Angular checks property bindings and since the `AuthService` defaults to not authenticated we show the text `Login`.

Now lets change the `AuthService` so it now returns authenticated, like so:

```
it('login button hidden when the user is authenticated', () => {
  expect(el.nativeElement.textContent.trim()).toBe('');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  spyOn(authService, 'isAuthenticated').and.returnValue(true);
  expect(el.nativeElement.textContent.trim()).toBe('Login');
});
```

But at this point the button content still *isn't* `Logout`, we need to trigger another change detection run like so:

```
it('login button hidden when the user is authenticated', () => {
  expect(el.nativeElement.textContent.trim()).toBe('');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  spyOn(authService, 'isAuthenticated').and.returnValue(true);
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Logout');
});
```

Now we've triggered a second change detection run Angular detected that the `AuthService` returns true and the button text updated to *Logout* accordingly.

Summary

By using the ATB and `fixtures` we can *inspect* the components view through `fixture.debugElement` and also trigger a change detection run by calling `fixture.detectChanges()`.

Next up we'll look at how to can test asynchronous functions in Angular.

Listing

login.component.ts

```
import {Component} from '@angular/core';
import {AuthService} from "./auth.service";

@Component({
  selector: 'app-login',
  template: `
  <a>
    <span *ngIf="needsLogin()">Login</span>
    <span *ngIf="!needsLogin()">Logout</span>
  </a>
`
})
export class LoginComponent {

  constructor(private auth: AuthService) {
  }

  needsLogin() {
    return !this.auth.isAuthenticated();
  }
}
```

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, async, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";
import {DebugElement} from "@angular/core";
import {By} from "@angular/platform-browser";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: AuthService;
  let el: DebugElement;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);
```

```
    // get test component from the fixture
    component = fixture.componentInstance;

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService);

    //  get the "a" element by CSS selector (e.g., by class name)
    el = fixture.debugElement.query(By.css('a'));
  });

  it('login button hidden when the user is authenticated', () => {
    // To being with Angular has not done any change detection so the content is
blank.
    expect(el.nativeElement.textContent.trim()).toBe('');

    // Trigger change detection and this lets the template update to the initial value
which is Login since by
    // default we are not authenticated
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Login');

    // Change the authetication state to true
    spyOn(authService, 'isAuthenticated').and.returnValue(true);

    // The label is still Login! We need changeDetection to run and for angular to
update the template.
    expect(el.nativeElement.textContent.trim()).toBe('Login');
    // Which we can trigger via fixture.detectChange()
    fixture.detectChanges();

    // Now the label is Logout
    expect(el.nativeElement.textContent.trim()).toBe('Logout');
  });
});
```

Testing Asynchronous Code

Learning Objectives

- Understand the issues faced when testing async code in Jasmine.
- Know how to use the Jasmine `done` function to handle async code.
- Know how to use the alternative Angular only solutions for testing async code.

Test setup

We want to see how we can test *asynchronous* functions.

So we change our `AuthService.isAuthenticated()` function to an *asynchronous* one that return a promise which resolves into a boolean at a later time.

```
export class AuthService {
  isAuthenticated(): Promise<boolean> {
    return Promise.resolve(!!localStorage.getItem('token'));
  }
}
```

We also then change our `LoginComponent`:

```
export class LoginComponent implements  OnInit {

  needsLogin: boolean = true;

  constructor(private auth: AuthService) {
  }

  ngOnInit()  {
    this.auth.isAuthenticated().then((authenticated) => {
      this.needsLogin = !authenticated;
    })
  }
}
```

We've changed `needsLogin` from a function into a property and we set the value of this property in the `then` callback from the promise returned from `AuthService`.

Importantly we've done the above in the `ngOnInit()` lifecycle function. Probably not the *best* place to put this functionality given that the value might change over time but good for demonstration purposes.

No asynchronous handling

Our first attempt might be to try to test our application without taking into account the asynchronous nature of our app, like so:

```
it('Button label via jasmine.done', () => {
  fixture.detectChanges(); ①
  expect(el.nativeElement.textContent.trim()).toBe('Login'); ②
  spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true)); ③
  component.ngOnInit(); ④
  fixture.detectChanges(); ⑤
  expect(el.nativeElement.textContent.trim()).toBe('Logout'); ⑥
});
```

① We issue our first change detection run so the view does it's initial update.

② We expect the button text to display *Login*

③ We change our `AuthService` so it returns a promise resolved to `true`.

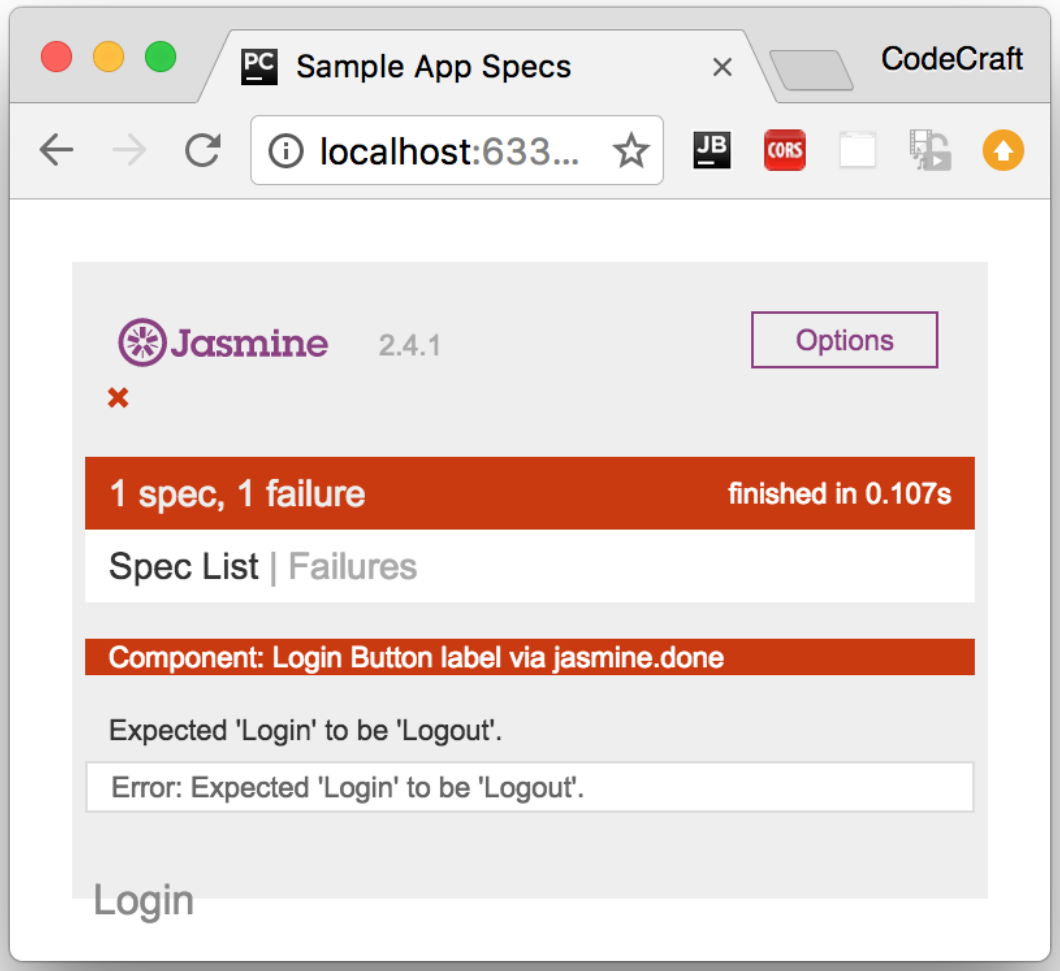
④ We call `component.ngOnInit()`.

⑤ We issue our second change detection run.

⑥ We now expect the button text to read *Logout*.

When performing testing *we* need to call component lifecycle hooks ourselves, like `ngOnInit()`. Angular won't do this for us in the test environment.

If we ran the above code we would see it doesn't pass:



It's failing on the last expectation. By the time we run the last expectation the `AuthService.isAuthenticated()` function hasn't *yet* resolved to a value. Therefore the `needsLogin` property on the `LoginComponent` hasn't been updated.

There are a few ways we can handle asynchronous code in our tests, one is the Jasmine way and two are Angular specific, lets start with the Jasmine way.

Jasmines `done` function

Jasmine has a built-in way to handle async code and that's by the passed in `done` function in the test specs.

So far we've been defining our test specs without any parameters but it can take a parameter, a done function which we call when all the async processing is complete, like so:

```
it('Button label via jasmine.done', (done) => { ①
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  let spy = spyOn(authService,
'isAuthenticated').and.returnValue(Promise.resolve(true));
  component.ngOnInit();
  spy.calls.mostRecent().returnValue.then(() => { ②
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Logout');
    done(); ③
  });
});
```

① The jasmine test spec function is passed a function as the first param, we usually call this parameter `done`.

② We can add a callback function (using the spy) which is called when the promise returned from `isAuthenticated` function resolved. In this function we know that the component has the new value of `needsLogin` and we can add our additional expectation here.

③ When we are done with our asynchronous tasks we tell Jasmine via the `done` function.

Jasmine lets us create asynchronous tests by giving us an explict `done` function which we call when the test is complete.

Although it works trying to understand the code can be difficult as it jumps about and is not executed in the order it's written in.

`async` and `whenStable`

Angular has another method for us to test asynchronous code via the `async` and `whenStable` functions.

Let's rewrite the above test to use these and then we will explain the differences.

```
it('Button label via async() and whenStable()', async(() => { ①
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));
  fixture.whenStable().then(() => { ②
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Logout');
  });
  component.ngOnInit();
}));
```

① We wrap our test spec function in another function called `async`.

② We place the tests we need to run after the `isAuthenticated` promise resolves inside this function.

This `async` function executes the code inside it's body in a special *async test zone*. This intercepts and *keeps track* of all promises created in it's body.

Only when all of those pending promises have been resolved does it then resolves the promise returned from `whenStable`.

So by using the `async` and `whenStable` functions we now *don't* need to use the Jasmine spy mechanism of detecting when the `isAuthenticated` promise has been resolved, like the previous example.

This mechanism is slightly better than using the plain Jasmine solution but there is another version which gives us fine grained control and also allows us to lay out our test code as if it were synchronous.

`fakeAsync` and `tick`

```
it('Button label via fakeAsync() and tick()', fakeAsync(() => { ①
  expect(el.nativeElement.textContent.trim()).toBe('');
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Login');
  spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));
  component.ngOnInit();

  tick(); ②
  fixture.detectChanges();
  expect(el.nativeElement.textContent.trim()).toBe('Logout');
}));
```

① Like `async` we wrap the test spec function in a function called `fakeAsync`.

② We call `tick()` when there are pending asynchronous activities we want to complete.

Like the `async` function the `fakeAsync` function executes the code inside it's body in a special *fake async test zone*. This intercepts and *keeps track* of all promises created in it's body.

The `tick()` function *blocks execution* and simulates the passage of time until all pending asynchronous activities complete.

So when we call `tick()` the application sits and waits for the promise returned from `isAuthenticated` to be resolved and then lets execution move to the next line.

The code above is now layed our *linearly*, as if we were executing synchronous code, there are no callbacks to confuse the mind and everything is simpler to understand.

`fakeAsync` does have some drawbacks, it doesn't track XHR requests for instance.

Summary

If the code we are testing is asynchronous then we need to take this into account when writing our

tests.

There are three mechanisms we can use.

The jasmine `done` function and spy callbacks. We attach specific callbacks to spies so we know when promises are resolves, we add our test code to thos callbacks and then we call the `done` function. This works but means we need to know about all the promises in our application and be able to hook into them.

We can use the Angular `async` and `whenStable` functions, we don't need to track the promises ourselves but we still need to lay our code out via callback functions which can be hard to read.

We can use the Angular `fakeAsync` and `tick` functions, this additionally lets us lay out our async test code as if it were synchronous.

Listing

http://plnkr.co/edit/83TAHD1hE3s7XBhz5sL3?p=preview

auth.service.ts

```
export class AuthService {
  isAuthenticated(): Promise<boolean> {
    return Promise.resolve(!!localStorage.getItem('token'));
  }
}
```

login.component.ts

```
import {Component} from '@angular/core';
import {AuthService} from "./auth.service";

@Component({
  selector: 'app-login',
  template: `
  <a>
    <span *ngIf="needsLogin">Login</span>
    <span *ngIf="!needsLogin">Logout</span>
  </a>
`
})
export class LoginComponent implements  OnInit {

  needsLogin: boolean = true;

  constructor(private auth: AuthService) {
  }

  ngOnInit()  {
    this.auth.isAuthenticated().then((authenticated) => {
      this.needsLogin = !authenticated;
    })
  }
}
```

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, async, whenStable, fakeAsync, tick, ComponentFixture} from
'@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";
import {DebugElement} from "@angular/core";
import {By} from "@angular/platform-browser";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: AuthService;
  let el: DebugElement;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
```

```
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService);

    //  get the "a" element by CSS selector (e.g., by class name)
    el = fixture.debugElement.query(By.css('a'));
  });

  it('Button label via fakeAsync() and tick()', fakeAsync(() => {
    expect(el.nativeElement.textContent.trim()).toBe('');
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Login');

    spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));

    component.ngOnInit();
    // Simulates the passage of time until all pending asynchronous activities
complete
    tick();
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Logout');
  }));

  it('Button label via async() and whenStable()', async(() => {
    // async() knows about all the pending promises defined in it's function body.
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Login');
    spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));

    fixture.whenStable().then(() => {
      // This is called when ALL pending promises have been resolved
      fixture.detectChanges();
      expect(el.nativeElement.textContent.trim()).toBe('Logout');
    });

    component.ngOnInit();

  }));

  it('Button label via jasmine.done', (done) => {
    fixture.detectChanges();
    expect(el.nativeElement.textContent.trim()).toBe('Login');
```

```
    // Make the authService return a promise that resolves to true
    let spy = spyOn(authService,
'isAuthenticated').and.returnValue(Promise.resolve(true));
    // We trigger the component to check the authService again
    component.ngOnInit();

    // We now want to call a function when the Promise returned from
authService.isAuthenticated() is resolved
    spy.calls.mostRecent().returnValue.then(() => {
      // The needsChanged boolean has been updated on the Component so to update the
template we trigger change detection
      fixture.detectChanges();
      // Now the label is Logout
      expect(el.nativeElement.textContent.trim()).toBe('Logout');
      // We tell jasmine we are done with this test spec
      done();
    });
  });
});
```

Testing Dependency Injection

Learning Objectives

- Know how we can configure the injectors for testing in our Angular application.
- Know the various methods we can use to resolve tokens for testing.

Resolving via `TestBed`

This is how we've injected dependencies so far in this section.

The `TestBed` acts as a dummy Angular Module and we can configure it like one including with a set of providers like so:

```
TestBed.configureTestingModule({
  providers: [AuthService]
});
```

We can then ask the `TestBed` to resolve a token into a dependency using it's internal injector, like so:

```
testBedService = TestBed.get(AuthService);
```

If most of our test specs need the same dependency mocked the same way we can resolve it once in the `beforeEach` function and mock it it there.

Resolving via the `inject` function

```
it('Service injected via inject(...) and TestBed.get(...) should be the same
instance',
    inject([AuthService], (injectService: AuthService) => {
      expect(injectService).toBe(testBedService);
    })
);
```

The `inject` function wraps the test spec function but lets us also *inject* dependencies using the parent injector in the `TestBed`.

We use it lik so:

```
inject(
  [token1, token2, token2],
  (dep1, dep2, dep3) => { }
)
```

The first param is an array of tokens we want to resolve dependencies for, the second parameter is a function whose arguments are the resolved dependencies.

Using the `inject` function:

- Makes it clear what dependencies each spec function uses.
- If each test spec requires different mocks and spys this is a better solution that resolving it once per test suite.

This will eventually move to becoming an function decorator like so:

```
@Inject (dep1: Token1, dep2: Token2) => { ... }
```

Overriding the components providers

Before we create a component via the `TestBed` we can *override* it's providers. Lets imagine we have a mock `AuthService` like so:

```
class MockAuthService extends AuthService {
  isAuthenticated() {
    return 'Mocked';
  }
}
```

We can override the components providers to use this mocked `AuthService` like so.

```
TestBed.overrideComponent(
    LoginComponent,
    {set: {providers: [{provide: AuthService, useClass: MockAuthService}]}}
);
```

The syntax is pretty specific, it's called a `MetaDataOverride` and it can have the properties `set`, `add` and `remove`. We use `set` to completely replace the providers array with the values we've set.

Resolving via the component injector

Now our component has been configured with it's own providers it will therefore have a child injector.

When the component is created since it has it's own injector it will resolve the `AuthService` itself and not forward the request to it's parent `TestBed` injector.

If we wanted to get the *same* instance of dependency that was passed to the component constructor we need to resolve using the component injector, we can do that through the component fixture

like so:

```
componentService = fixture.debugElement.injector.get(AuthService);
```

The above code resolves the token using the components child injector.

Summary

We can resolve dependencies in our tests using a number of methods.

We can resolve using the the test bed itself, usually in the `beforeEach` function and store the resolved dependencies for use in our test specs.

We can resolve using the `inject` function at the start of each test spec.

We can also override the default providers for our components using the `TestBed`.

We can then also use the components *child injector* to resolve tokens.

Listing

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture, inject} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

class MockAuthService extends AuthService {
  isAuthenticated() {
    return 'Mocked';
  }
}


describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let testBedService: AuthService;
  let componentService: AuthService;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });
```

```
    // Configure the component with another set of Providers
    TestBed.overrideComponent(
        LoginComponent,
        {set: {providers: [{provide: AuthService, useClass: MockAuthService}]}}
    );

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    // AuthService provided to the TestBed
    testBedService = TestBed.get(AuthService);

    // AuthService provided by Component, (should return MockAuthService)
    componentService = fixture.debugElement.injector.get(AuthService);
  });

  it('Service injected via inject(...) and TestBed.get(...) should be the same
instance',
      inject([AuthService], (injectService: AuthService) => {
        expect(injectService).toBe(testBedService);
      })
  );

  it('Service injected via component should be and instance of MockAuthService', () =>
{
    expect(componentService instanceof MockAuthService).toBeTruthy();
  });
});
```

Testing Components

Learning Objectives

- How to test a components inputs as well as it's outputs.
- How to interact with a components view.

Test setup

We'll continue with our example of testing a `LoginComponent`. We are going to change our component into a more complex version with inputs, outputs, a domain model and a form, like so:

```
import {Component, EventEmitter, Input, Output} from '@angular/core';

export class User { ①
  constructor(public email: string, public password: string) {
  }
}

@Component({
  selector: 'app-login',
  template: `
<form>
  <label>Email</label>
  <input type="email"
         #email>
  <label>Password</label>
  <input type="password"
         #password>
  <button type="button" ②
          (click)="login(email.value, password.value)"
          [disabled]="!enabled">Login
  </button>
</form>
`
})
export class LoginComponent {
  @Output() loggedIn = new EventEmitter<User>(); ③
  @Input() enabled = true; ④

  login(email, password) { ⑤
    console.log(`Login ${email} ${password}`);
    if (email && password) {
      console.log(`Emitting`);
      this.loggedIn.emit(new User(email, password));
    }
  }
}
```

① We create a `User` class which holds the model of a logged in user.

② The button is sometimes disabled depending on the `enabled` input property value and on clicking the button we call the `login` function.

③ The component has an output event called `loggedIn`.

④ The component has an input property called `enabled`.

⑤ In the `login` function we emit a new `user` model on the `loggedIn` event.

The component is more complex and uses inputs, outputs and emits a domain model on the output event.

We are not using the `AuthService` any more.

We also bootstrap our test suite file like so:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let submitEl: DebugElement;
  let loginEl: DebugElement;
  let passwordEl: DebugElement;

  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    submitEl = fixture.debugElement.query(By.css('button'));
    loginEl = fixture.debugElement.query(By.css('input[type=email]'));
    passwordEl = fixture.debugElement.query(By.css('input[type=password]'));
  });
});
```

We just have a few more debug elements stored on our test suite which we'll inspect or interact with in our test specs.

Testing `@Inputs`

To test inputs we need to do things:

1. We need to be able to change the input property `enabled` on our component.
2. We need to check that the button is enabled or disabled depending on the value of our input property.

Solving the first is actually very easy.

Just because it's an `@Input` doesn't change the fact it's a still just a simple property which we can change like any other property, like so:

```
it('Setting enabled to false disables the submit button', () => {
  component.enabled = false;
});
```

For the second we need to check the disabled property value of the buttons DOM element like so:

```
it('Setting enabled to false disables the submit button', () => {
    component.enabled = false;
    fixture.detectChanges();
    expect(submitEl.nativeElement.disabled).toBeTruthy();
});
```

We also need to call `fixture.detectChanges()` to trigger change detection and update the view.

Testing `@Outputs`

Testing outputs it somewhat trickier, especially if we want to test from the view.

Firstly lets see how we can track what gets emmitted by the output event and add some expectations for it.

```
it('Entering email and password emits loggedIn event', () => {
  let user: User;

  component.loggedIn.subscribe((value) => user = value);

  expect(user.email).toBe("test@example.com");
  expect(user.password).toBe("123456");
});
```

The key line above is

```
component.loggedIn.subscribe((value) => user = value);
```

Since the output event is actually an *Observable* we can subscribe to it and get a callback for every item emitted.

We store the emitted value to a user object and then add some *expectations* on the user object.

How do we actually trigger an event to be fired? We *could* call the `component.login(⋯)` function ourselves but for the purposes of this lecture we want to trigger the function from the view.

Firstly lets set some values to our email and password input controls in the view. We've already got references to both those fields in our setup function so we just set the values like so:

```
loginEl.nativeElement.value = "test@example.com";
passwordEl.nativeElement.value = "123456";
```

Next we trigger a `click` on the submit button, but we want to do that *after* we've subscribed to our observable like so:

```
it('Entering email and password emits loggedIn event', () => {
  let user: User;
  loginEl.nativeElement.value = "test@example.com"; ①
  passwordEl.nativeElement.value = "123456";

  component.loggedIn.subscribe((value) => user = value);

  submitEl.triggerEventHandler('click', null); ②

  expect(user.email).toBe("test@example.com");
  expect(user.password).toBe("123456");
});
```

① Setup data in our input controls.

② Trigger a `click` on our submit button, this synchronously emits the user object in the subscribe callback!

Summary

We can test *inputs* by just setting values on a components input properties.

We can test *outputs* by subscribing to an *EventEmitters* observable and storing the emitted values on local variables.

In combination with the previous lectures and the ability to test inputs and outputs we should now have all the information we need to test components in Angular.

In the next lecture we will look at how to test Directives.

Listing

login.component.ts

```
import {Component, EventEmitter, Input, Output} from '@angular/core';

export class User {
  constructor(public email: string, public password: string) {
  }
}

@Component({
  selector: 'app-login',
  template: `
<form>
  <label>Email</label>
  <input type="email"
         #email>
  <label>Password</label>
  <input type="password"
         #password>
  <button type="button"
          (click)="login(email.value, password.value)"
          [disabled]="!enabled">Login
  </button>
</form>
`
})
export class LoginComponent {
  @Output() loggedIn = new EventEmitter<User>();
  @Input() enabled = true;

  login(email, password) {
    console.log(`Login ${email} ${password}`);
    if (email && password) {
      console.log(`Emitting`);
      this.loggedIn.emit(new User(email, password));
    }
  }
}
```

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture, inject, async} from '@angular/core/testing';
import {LoginComponent, User} from './login.component';
import {Component, DebugElement} from "@angular/core";
import {By} from "@angular/platform-browser";


describe('Component: Login', () => {
```

```
  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let submitEl: DebugElement;
  let loginEl: DebugElement;
  let passwordEl: DebugElement;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    });


    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    submitEl = fixture.debugElement.query(By.css('button'));
    loginEl = fixture.debugElement.query(By.css('input[type=email]'));
    passwordEl = fixture.debugElement.query(By.css('input[type=password]'));
  });

  it('Setting enabled to false disabled the submit button', () => {
    component.enabled = false;
    fixture.detectChanges();
    expect(submitEl.nativeElement.disabled).toBeTruthy();
  });

  it('Setting enabled to true enables the submit button', () => {
    component.enabled = true;
    fixture.detectChanges();
    expect(submitEl.nativeElement.disabled).toBeFalsy();
  });

  it('Entering email and password emits loggedIn event', () => {
    let user: User;
    loginEl.nativeElement.value = "test@example.com";
    passwordEl.nativeElement.value = "123456";

    // Subscribe to the Observable and store the user in a local variable.
    component.loggedIn.subscribe((value) => user = value);

    // This sync emits the event and the subscribe callback gets executed above
    submitEl.triggerEventHandler('click', null);

    // Now we can check to make sure the emitted value is correct
    expect(user.email).toBe("test@example.com");
    expect(user.password).toBe("123456");
```

```
  });
})
;
```

Testing Directives

Learning Objectives

- Understand how to test directives using wrapper components.

Test setup

We are going to test a directive called the `HoverFocusDirective`. It has an attribute selector of `hoverfocus` and if its attached to an element hovering over that element sets the background color to `blue`.

```
import {
    Directive,
    HostListener,
    HostBinding
} from '@angular/core';

@Directive({
  selector: '[hoverfocus]'
})
export class HoverFocusDirective {

  @HostBinding("style.background-color") backgroundColor: string;

  @HostListener('mouseover') onHover() {
    this.backgroundColor = 'blue';
  }

  @HostListener('mouseout') onLeave() {
    this.backgroundColor = 'inherit';
  }
}
```

It uses `@HostListener` to listen to `mouseover` and `mouseout` events on it's host element and it also uses `@HostBinding` to set the style property of it's host element.

The starting point for our test suite is very similar to our previous examples:

```
import {TestBed} from '@angular/core/testing';
import {HoverFocusDirective} from './hoverfocus.directive';

describe('Directive: HoverFocus', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [HoverFocusDirective]
    });
  });
});
```

Test wrapper component

To test a directive we typically create a *dummy* testing component so we can interact with the directive and test it's effect on the components view, like so:

```
@Component({
  template: `<input type="text" hoverfocus>` ①
})
class TestHoverFocusComponent {
}
```

① The directive is associated with an `input` control in the components view.

Now we have a component to work with we can configure the test bed and get the required references for the tests, like so:

```
describe('Directive: HoverFocus', () => {

  let component: TestHoverFocusComponent;
  let fixture: ComponentFixture<TestHoverFocusComponent>;
  let inputEl: DebugElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestHoverFocusComponent, HoverFocusDirective] ①
    });
    fixture = TestBed.createComponent(TestHoverFocusComponent); ②
    component = fixture.componentInstance;
    inputEl = fixture.debugElement.query(By.css('input'));
  });
});
```

① We declare *both* the directive we want to test and the dummy test component.

② We grab a reference to the component fixture as well as the component and a the input `DebugElement` from the component view.

Interacting and inspecting the the view

We now have all the pieces we need to create a test spec for the directive itself:

```
it('hovering over input', () => {
  inputEl.triggerEventHandler('mouseover', null); ①
  fixture.detectChanges();
  expect(inputEl.nativeElement.style.backgroundColor).toBe('blue'); ②

  inputEl.triggerEventHandler('mouseout', null);
  fixture.detectChanges();
  console.log(inputEl.nativeElement.style.backgroundColor);
  expect(inputEl.nativeElement.style.backgroundColor).toBe('inherit');
});
```

① We use `triggerEventHandler` to simulate events.

② The `style` property on the `nativeElement` is what we can inspect to see the current style applied to an element.

Summary

To test directives we use dummy test components which we can create using the Angular Test Bed and which we can interact with by using a component fixture.

We can trigger events on `DebugElements` by using the `triggerEventHandler` function and if we want to see what styles are applied to it we can find it via the `nativeElement.style` property.

Listing

http://plnkr.co/edit/snp12WgkVCemGf8s5XKW?p=preview

hoverfocus.directive.ts

```
import {
    Directive,
    HostListener,
    HostBinding
} from '@angular/core';

@Directive({
  selector: '[hoverfocus]'
})
export class HoverFocusDirective {

  @HostBinding("style.background-color") backgroundColor: string;

  @HostListener('mouseover') onHover() {
    this.backgroundColor = 'blue';
  }

  @HostListener('mouseout') onLeave() {
    this.backgroundColor = 'inherit';
  }
}
```

hoverfocus.directive.spec.ts

```
/* tslint:disable:no-unused-variable */
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {Component, DebugElement} from "@angular/core";
import {By} from "@angular/platform-browser";
import {TestBed} from '@angular/core/testing';
import {HoverFocusDirective} from './hoverfocus.directive';

@Component({
  template: `<input type="text" hoverfocus>`
})
class TestHoverFocusComponent {
}


describe('Directive: HoverFocus', () => {

  let component: TestHoverFocusComponent;
  let fixture: ComponentFixture<TestHoverFocusComponent>;
  let inputEl: DebugElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestHoverFocusComponent, HoverFocusDirective]
    });
    fixture = TestBed.createComponent(TestHoverFocusComponent);
    component = fixture.componentInstance;
    inputEl = fixture.debugElement.query(By.css('input'));
  });

  it('hovering over input', () => {
    inputEl.triggerEventHandler('mouseover', null);
    fixture.detectChanges();
    expect(inputEl.nativeElement.style.backgroundColor).toBe('blue');

    inputEl.triggerEventHandler('mouseout', null);
    fixture.detectChanges();
    expect(inputEl.nativeElement.style.backgroundColor).toBe('inherit');
  });
});
```

Testing Model Driven Forms

Learning Objectives

- How to setup the TestBed in order to work with forms.
- How to create test specs that test form validity.
- How to test form submission.

Test setup

To test forms we'll extend the `LoginComponent` we've been working with so far in this section. However we'll convert the simple email, password field form into a model driven form and we'll drop the input `enabled` property.

```
@Component({
  selector: 'app-login',
  template: `
<form (ngSubmit)="login()" ①
      [formGroup]="form"> ②
  <label>Email</label>
  <input type="email"
         formControlName="email"> ③
  <label>Password</label>
  <input type="password"
         formControlName="password"> ③
  <button type="submit">Login</button>
</form>
`
})
export class LoginComponent {
  @Output() loggedIn = new EventEmitter<User>();
  form: FormGroup;

  constructor(private fb: FormBuilder) {
  }

  ngOnInit() { ④
    this.form = this.fb.group({
      email: ['', [
        Validators.required,
        Validators.pattern("[^ @]*@[^ @]*")]],
      password: ['', [
        Validators.required,
        Validators.minLength(8)]],
    });
  }

  login() {
    console.log(`Login ${this.form.value}`);
    if (this.form.valid) {
      this.loggedIn.emit(
          new User(
              this.form.value.email,
              this.form.value.password
          )
      );
    }
  }
}
```

① When the user submits the form we call the `login()` function.

② We associate this template form element with the model `form` on our component.

③ We link specific template form controls to `FormControls` on our `form` model.

④ We initialise our `form` model in the `ngOnInit` lifecycle hook.

The rest of the `LoginComponent` looks the same as before.

Our test suite looks mostly the same but has a few key differences:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ReactiveFormsModule, FormsModule], ①
      declarations: [LoginComponent]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;
    component.ngOnInit(); ②
  });
});
```

① We add the required `ReactiveFormsModule` and `FormsModule` to our test beds imports list.

② We manually trigger the `ngOnInit` lifecycle function on our component, Angular won't call this for us.

Now lets look at how we can test our forms validity.

Form validity

The first test spec we may want to check is that a blank form is invalid. Since we are using model driven forms we can just check the `valid` property on the form model itself, like so:

```
it('form invalid when empty', () => {
  expect(component.form.valid).toBeFalsy();
});
```

We can easily check to see if the form is *valid* by checking the value of `component.form.valid`.

This one of the reasons model driven forms are easier to test than template driven forms, we already have an object on the component we can inspect from our test spec for *correctness*.

With template driven forms the state is in the *view* and unless the component has a reference to the template form with a `ViewChild` decorator there is no way to test the form using a unit test. We would have to perform a full E2E test simulating button clicks and typing in values into forms.

Field validity

We can also check to see if individual fields are valid, for example the email field should initially be invalid.

```
it('email field validity', () => {
  let email = component.form.controls['email']; ①
  expect(email.valid).toBeFalsy(); ②
});
```

① We grab a reference to the actual field itself from the `form.controls` property.

② Just like the form we can check if the field is valid through `email.valid`.

Field errors

As well as checking to see if the field is valid we can also see what specific validators are failing through the `email.errors` property.

Since it's required and the email field hasn't been set I would expect the requrired validator to be failing, we can test for this like so:

```
  it('email field validity', () => {
    let errors = {};
    let email = component.form.controls['email'];
    errors = email.errors || {};
    expect(errors['required']).toBeTruthy(); ①
  });
```

① Because `errors` contains a key of `required` and this has a *value* this means the required validator is *failing* as we expect.

We can set some data on our input control by calling `setValue(⋯)` like so:

```
email.setValue("test");
```

If we did set the email field to be `test` this should fail the pattern validator, since that expects the email to contain a `@`. We can then check to see if the pattern validator is failing like so:

```
email.setValue("test");
errors = email.errors || {};
expect(errors['pattern']).toBeTruthy();
```

Submitting a form

We can submit a form by clicking on the submit button, but we've already covered this in a previous lecture.

Since the `ngSubmit` directive has it's own set of tests it's safe to *assume* that the `(ngSubmit)=login()` expression is working as expected.

So to test form submission with model driven forms we can just call the `login()` function on our controller, like so:

```
component.login();
```

Since our form emits an event from the `loggedIn` output event property we can use the same method we covered in the section on testing components to test this form submission. Namely subscribe to the observable and store a reference to the emitted event for later comparison, like so:

```
  it('submitting a form emits a user', () => {
    expect(component.form.valid).toBeFalsy();
    component.form.controls['email'].setValue("test@test.com");
    component.form.controls['password'].setValue("123456789");
    expect(component.form.valid).toBeTruthy();

    let user: User;
    // Subscribe to the Observable and store the user in a local variable.
    component.loggedIn.subscribe((value) => user = value);

    // Trigger the login function
    component.login();

    // Now we can check to make sure the emitted value is correct
    expect(user.email).toBe("test@test.com");
    expect(user.password).toBe("123456789");
  });
```

Summary

We can easily unit test model driven forms in Angular by just testing the form model itself.

To test template driven forms in Angular we need to launch a full end to end testing environment and interact with a browser to test the form.

Next we will take a look at how to test an applications that makes http requests.

Listing

http://plnkr.co/edit/FrVMMaLc0NQkArGUC8yb?p=preview

login.component.ts

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

export class User {
  constructor(public email: string,
              public password: string) {
  }
}

@Component({
  selector: 'app-login',
  template: `
<form (ngSubmit)="login()"
      [formGroup]="form">
  <label>Email</label>
  <input type="email"
         formControlName="email">
  <label>Password</label>
  <input type="password"
         formControlName="password">
  <button type="submit">Login</button>
</form>
`
})
export class LoginComponent {
  @Output() loggedIn = new EventEmitter<User>();
  form: FormGroup;

  constructor(private fb: FormBuilder) {
  }

  ngOnInit() {
    this.form = this.fb.group({
      email: ['', [
        Validators.required,
```

```
        Validators.pattern("[^ @]*@[^ @]*")]],
      password: ['', [
        Validators.required,
        Validators.minLength(8)]],
    });
  }

  login() {
    console.log(`Login ${this.form.value}`);
    if (this.form.valid) {
      this.loggedIn.emit(
          new User(
              this.form.value.email,
              this.form.value.password
          )
      );
    }
  }
}
```

login.component.spec.ts

```
/* tslint:disable:no-unused-variable */

import {TestBed, ComponentFixture} from '@angular/core/testing';
import {ReactiveFormsModule, FormsModule} from "@angular/forms";
import {LoginComponent, User} from "./login.component";


describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      imports: [ReactiveFormsModule, FormsModule],
      declarations: [LoginComponent]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;
    component.ngOnInit();
  });

  it('form invalid when empty', () => {
```

```
    expect(component.form.valid).toBeFalsy();
  });

  it('email field validity', () => {
    let errors = {};
    let email = component.form.controls['email'];
    expect(email.valid).toBeFalsy();

    // Email field is required
    errors = email.errors || {};
    expect(errors['required']).toBeTruthy();

    // Set email to something
    email.setValue("test");
    errors = email.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['pattern']).toBeTruthy();

    // Set email to something correct
    email.setValue("test@example.com");
    errors = email.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['pattern']).toBeFalsy();
  });

  it('password field validity', () => {
    let errors = {};
    let password = component.form.controls['password'];

    // Email field is required
    errors = password.errors || {};
    expect(errors['required']).toBeTruthy();

    // Set email to something
    password.setValue("123456");
    errors = password.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['minlength']).toBeTruthy();

    // Set email to something correct
    password.setValue("123456789");
    errors = password.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['minlength']).toBeFalsy();
  });

  it('submitting a form emits a user', () => {
    expect(component.form.valid).toBeFalsy();
    component.form.controls['email'].setValue("test@test.com");
    component.form.controls['password'].setValue("123456789");
    expect(component.form.valid).toBeTruthy();
```

```
    let user: User;
    // Subscribe to the Observable and store the user in a local variable.
    component.loggedIn.subscribe((value) => user = value);

    // Trigger the login function
    component.login();

    // Now we can check to make sure the emitted value is correct
    expect(user.email).toBe("test@test.com");
    expect(user.password).toBe("123456789");
  });
})
;
```

Testing Http

Learning Objectives

- How to configure a test suite so we can mock the Http service to send back fake responses.

Test setup

To demonstrate how to test http requests we will add a test for our iTunes `SearchService` which we created in the section on Http.

We will use the promise version of the search service that uses JSONP to get around the issue of CORS.

```
import {Injectable} from '@angular/core';
import {Jsonp} from '@angular/http';
import 'rxjs/add/operator/toPromise';

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  console.log(item);
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.artworkUrl60,
                      item.artistId
                  );
                });
                resolve(this.results);
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}
```

Although we are using JSONP here, testing Http and Jsonp is exactly the same. We just replace instances of `Jsonp` with `Http`.

Configuring the test suite

We want the `Jsonp` and `Http` services to use the `MockBackend` instead of the *real* `Backend`, this is the underling code that actually sends and handles http.

By using the `MockBackend` we can intercept real requests and simulate responses with test data.

The configuration is slightly more complex since we are using a *factory provider*:

```
{
  provide: Http, ①
  useFactory: (backend, options) => new Http(backend, options), ②
  deps: [MockBackend, BaseRequestOptions] ③
}
```

① We are configuring a dependency for the token `Http`.

② The injector calls this function in order to return a new instance of the `Http` class. The arguments to the `useFactory` function are themselves *injected* in, see (3).

③ We define the dependencies to our `useFactory` function via the `deps` property.

For our API however we are using `Jsonp`, we can just replace all mention of `Http` with `Jsonp` like so:

```
{
  provide: Jsonp,
  useFactory: (backend, options) => new Jsonp(backend, options),
  deps: [MockBackend, BaseRequestOptions]
}
```

The above configuration ensures that the `Jsonp` service is constructed using the `MockBackend` so we can control it later on in testing.

Together with the other providers and modules we need our initial test suite file looks like so:

```
describe('Service: Search', () => {

  let service: SearchService;
  let backend: MockBackend;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [JsonpModule],
      providers: [
        SearchService,
        MockBackend,
        BaseRequestOptions,
        {
          provide: Jsonp,
          useFactory: (backend, options) => new Jsonp(backend, options),
          deps: [MockBackend, BaseRequestOptions]
        }
      ]
    });

    backend = TestBed.get(MockBackend); ①

    service = TestBed.get(SearchService); ②
  });
});
```

① We grab a reference to the *mock backend* so we can control the http responses from our test specs.

② We grab a reference to the `SearchService`, this has been created using the `MockBackend` above.

Using the `MockBackend` to simulate a response

Just by using the `MockBackend` instead of the real `Backend` we have stopped the tests from triggering real http requests from being sent out.

Now we need to configure the `MockBackend` to return dummy test data instead, like so:

```
it('search should return SearchItems', fakeAsync(() => {
  let response = { ①
    "resultCount": 1,
    "results": [
      {
        "artistId": 78500,
        "artistName": "U2",
        "trackName": "Beautiful Day",
        "artworkUrl60": "image.jpg",
      }]
  };

  backend.connections.subscribe(connection => { ②
    connection.mockRespond(new Response(<ResponseOptions>{ ③
      body: JSON.stringify(response)
    }));
  });
}));
```

① We create some fake data we want the API to response with.

② The mock backend `connections` property is an observable that emits an `connection` every time an API request is made.

③ For every connection that is requested we tell it to `mockRespond` with our dummy data.

The above code returns the same dummy data for *every* API request, regardless of the URL.

Testing the response

Using HTTP is asynchronous so in order to test we need to use one of the asynchronous testing methods, we'll use the `fakeAsync` method.

```
it('search should return SearchItems', fakeAsync(() => { ①
  let response = {
    "resultCount": 1,
    "results": [
      {
        "artistId": 78500,
        "artistName": "U2",
        "trackName": "Beautiful Day",
        "artworkUrl60": "image.jpg",
      }]
  };

  // When the request subscribes for results on a connection, return a fake response
  backend.connections.subscribe(connection => {
    connection.mockRespond(new Response(<ResponseOptions>{
      body: JSON.stringify(response)
    }));
  });

  // Perform a request and make sure we get the response we expect
  service.search("U2"); ②
  tick(); ③

  expect(service.results.length).toBe(1); ④
  expect(service.results[0].artist).toBe("U2");
  expect(service.results[0].name).toBe("Beautiful Day");
  expect(service.results[0].thumbnail).toBe("image.jpg");
  expect(service.results[0].artistId).toBe(78500);
}));
```

① We use the `fakeAsync` method to execute in the special *fake async zone* and track pending promises.

② We make the *asynchronous* call to `service.search(⋯)`

③ We issue a `tick()` which blocks execution and waits for all the pending promises to be resolved.

④ We now *know* that the service has received and parsed the response so we can write some expectations.

Summary

We can test code that makes Http requests by using a `MockBackend`.

This requires that we configure our `TestBed` so that the `Jsonp` or `Http` services are created using the `MockBackend`.

We grab a reference to the instance of `MockBackend` that was injected and use it to simulate responses.

Since *Http* is asynchronous we use of one of the *async* testing mechanisms so we can write tests

specs for our code.

Listing

search.service.ts

```
import {Injectable} from '@angular/core';
import {Jsonp} from '@angular/http';
import 'rxjs/add/operator/toPromise';

class SearchItem {
  constructor(public name: string,
              public artist: string,
              public thumbnail: string,
              public artistId: string) {
  }
}

@Injectable()
export class SearchService {
  apiRoot: string = 'https://itunes.apple.com/search';
  results: SearchItem[];

  constructor(private jsonp: Jsonp) {
    this.results = [];
  }

  search(term: string) {
    return new Promise((resolve, reject) => {
      this.results = [];
      let apiURL =
`${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP_CALLBACK`;
      this.jsonp.request(apiURL)
          .toPromise()
          .then(
              res => { // Success
                this.results = res.json().results.map(item => {
                  console.log(item);
                  return new SearchItem(
                      item.trackName,
                      item.artistName,
                      item.artworkUrl60,
                      item.artistId
                  );
                });
                resolve(this.results);
              },
              msg => { // Error
                reject(msg);
              }
          );
    });
  }
}
```

script.ts

```
/* tslint:disable:no-unused-variable */
import {
    JsonpModule,
    Jsonp,
    BaseRequestOptions,
    Response,
    ResponseOptions,
    Http
} from "@angular/http";
import {TestBed, fakeAsync, tick} from '@angular/core/testing';
import {MockBackend} from "@angular/http/testing";
import {SearchService} from './search.service';

describe('Service: Search', () => {

  let service: SearchService;
  let backend: MockBackend;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [JsonpModule],
      providers: [
        SearchService,
        MockBackend,
        BaseRequestOptions,
        {
          provide: Jsonp,
          useFactory: (backend, options) => new Jsonp(backend, options),
          deps: [MockBackend, BaseRequestOptions]
        }
      ]
    });

    // Get the MockBackend
    backend = TestBed.get(MockBackend);

    // Returns a service with the MockBackend so we can test with dummy responses
    service = TestBed.get(SearchService);

  });

  it('search should return SearchItems', fakeAsync(() => {
    let response = {
      "resultCount": 1,
      "results": [
        {
          "artistId": 78500,
          "artistName": "U2",
          "trackName": "Beautiful Day",
```

```
          "artworkUrl60": "image.jpg",
        }]
    };

    // When the request subscribes for results on a connection, return a fake response
    backend.connections.subscribe(connection => {
      connection.mockRespond(new Response(<ResponseOptions>{
        body: JSON.stringify(response)
      }));
    });

    // Perform a request and make sure we get the response we expect
    service.search("U2");
    tick();

    expect(service.results.length).toBe(1);
    expect(service.results[0].artist).toBe("U2");
    expect(service.results[0].name).toBe("Beautiful Day");
    expect(service.results[0].thumbnail).toBe("image.jpg");
    expect(service.results[0].artistId).toBe(78500);
  }));
});
```

Testing Routing

Learning Objectives

- Understand how to configure the ATB in order to test routing.
- Understand how to write test specs that involve routing.

Test setup

To test routing we need a few components and a route configuration:

```
import {Component} from "@angular/core";
import {Routes} from "@angular/router";

@Component({
  template: `Search`
})
export class SearchComponent {
}

@Component({
  template: `Home`
})
export class HomeComponent {
}

@Component({
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {
}

export const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent}
];
```

We create three components `HomeComponent`, `SearchComponent` and an `AppComponent` with a `<router-outlet>`.

We also create a route configuration where `''` redirects you to `home` and `home` and `search` show their respective components.

Our basic test suite looks like so:

```
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,
    SearchComponent,
    AppComponent,
    routes
} from "./router"

describe('Router: App', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ ①
        HomeComponent,
        SearchComponent,
        AppComponent
      ]
    });
  });
});
```

① We import and declare our components in the test bed configuration.

Router setup

Normally to setup routing in an Angular application we import the `RouterModule` and *provide* the routes to the `NgModule` with `RouterModule.withRoutes(routes)`.

However when testing routing we use the `RouterTestingModule` instead. This modules sets up the router with a *spy* implementation of the *Location Strategy* that doesn't actually change the URL.

We also need to get the injected `Router` and `Location` so we can use them in the test specs.

Our test suite file now looks like:

```
describe('Router: App', () => {

  let location: Location;
  let router: Router;
  let fixture;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [RouterTestingModule.withRoutes(routes)], ①
      declarations: [
        HomeComponent,
        SearchComponent,
        AppComponent
      ]
    });

    router = TestBed.get(Router); ②
    location = TestBed.get(Location); ③

    fixture = TestBed.createComponent(AppComponent); ④
    router.initialNavigation(); ⑤
  });
});
```

① We import our `RouterTestingModule` with our routes.

② We grab a reference to the injected `Router`.

③ We grab a reference to the injected `Location`.

④ We ask the test bed to create an instance of our root `AppComponent`. We don't need this reference in our test specs but we do need to create the root component with the `router-outlet` so the router has somewhere to insert components.

⑤ This sets up the location change listener and performs the initial navigation.

We are now ready to create our test specs.

Testing routing

In our configuration we've set it so if you land on the root *empty* url you will be redirected to `/home`, lets add a test spec for this:

```
it('navigate to "" redirects you to /home', fakeAsync(() => { ①
  router.navigate(['']); ②
  tick(); ③
  expect(location.path()).toBe('/home'); ④
}));
```

① Routing is an asynchronous activity so we use one of the asynchronous testing methods at our disposal, in this case the `fakeAsync` method.

② We trigger the router to navigate to the empty path.

③ We wait for all pending promises to be resolved.

④ We can then inspect the *path* our application should be at with `location.path()`

Lets also add a test spec for navigating to the search route, like so:

```
it('navigate to "search" takes you to /search', fakeAsync(() => {
  router.navigate(['search']);
  tick();
  expect(location.path()).toBe('/search');
}));
```

The spec is exactly the same as the previous one, our link params array is different since we are triggering a different route and our expectation is again different, but the rest is the same.

Summary

We can test routing in Angular by using `RouterTestingModule` instead of `RouterModule` to provide our routes.

This uses a *spy* implementation of `Location` which doesn't trigger a request for a new URL but does let us know the target URL which we can use in our test specs.

Listing

router.ts

```
import {Component} from "@angular/core";
import {Routes} from "@angular/router";

@Component({
  template: `Search`
})
export class SearchComponent {
}

@Component({
  template: `Home`
})
export class HomeComponent {
}

@Component({
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {
}

export const routes: Routes = [
  // {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent}
];
```

router.spec.ts

```
/* tslint:disable:no-unused-variable */
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,
    SearchComponent,
    AppComponent,
    routes
} from "./router"

describe('Router: App', () => {

  let location: Location;
  let router: Router;
  let fixture;
```

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [    RouterTestingModule.withRoutes(routes)],
      declarations: [
        HomeComponent,
        SearchComponent,
        AppComponent
      ]
    });

    router = TestBed.get(Router);
    location = TestBed.get(Location);

    fixture = TestBed.createComponent(AppComponent);
    router.initialNavigation();
  });

  it('fakeAsync works', fakeAsync(() => {
    let promise = new Promise((resolve) => {
      setTimeout(resolve, 10)
    });
    let done = false;
    promise.then(() => done = true);
    tick(50);
    expect(done).toBeTruthy();
  }));

  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');
  }));
});
```

Wrapping Up

In this section on testing in Angular we covered the basics of unit testing with Jasmine and Karma as well as testing by using the Angular Test Bed.

We initially covered the basics of how to test classes with just Jasmine and in itself that gives us the ability to test 50% of our code.

But to go deeper we need to be able to test behaviours that require the Angular framework and that is where the Angular Test Bed comes in.

It lets us test behaviours that require a component interacting with it's view and gives us fine grained control on change detection and asynchronous processing.

We covered specific recipes for testing:

- Using Mocks and Spys.
- Asynchronous code.
- Dependency injection.
- Components.
- Directives.
- Model Driven Forms
- Http and Jsonp.
- Routing.

Advanced Topics

Custom Form Validators

Learning Objectives

- Know how the built-in validators work in both the model driven and template driven forms.
- Know how to create a basic *hardcoded* custom validator for both model driven and template driven forms.

Built in validators

We have a few *built in* validators in Angular:

- required
- minlength
- maxlength
- pattern

We can use these in two ways:

1. As functions we can pass to the `FormControl` constructor in model driven forms.

```
new FormControl('', Validators.required)
```

The above creates a *form control* with a *required* validator function attached

2. As directives in template driven forms.

```
<input name="fullName" ngModel required>
```

These `required`, `minlength`, `maxlength` and `pattern` attributes are *already* in the official HTML specification.

They are a core part of HTML and we don't actually need Angular in order to use them. If they are present in a form then the browser will perform some default validation itself.

However we do need a way for Angular to recognise their presence and support the same validation logic in our own Angular forms.

If you remember template driven forms are just model driven forms but with the *creation* of the model *driven* by the template, they still have an underlying model.

Therefore just like model driven forms we need to attach a *validator function* to the underlying model form control.

Angular does this by secretly creating special *validator* directives which have selectors matching

`required`, `minlength`, `maxlength` and `pattern`.

So if you have imported `FormsModule` into your `NgModule` then anytime Angular sees a `required` tag in the HTML it will link it to an instance of a directive called RequiredValidator

This directive validator applies the same `Validators.required` function as we use in model driven forms.

That's how the built-in validators work, lets try to create our own custom validators that work with both model and template driven forms.

Custom model form validator

Validators at their core are just functions, they take as input a `FormControl` instance and returns either `null` if it's `valid` or an error object if it's not.

We'll create a custom email validator function which only accepts emails on the domain `codecraft.tv`:

```
function emailDomainValidator(control: FormControl) { ①
  let email = control.value; ②
  if (email && email.indexOf("@") != -1) { ③
    let [_, domain] = email.split("@"); ④
    if (domain !== "codecraft.tv") { ⑤
      return {
        emailDomain: {
          parsedDomain: domain
        }
      }
    }
  }
  return null; ⑥
}
```

① Accepts an instance of a `FormControl` as the first param.

② We get the email value from the form control.

③ Only bother checking if the email contains an "@" character.

④ Extract the domain part from the email.

⑤ If the domain is not `codecraft.tv` then return an error object with some perhaps helpful tips as to why it's failing.

⑥ Return `null` because if we have reached here the validator is passing.

To use this validator in our model driven form we pass it into the `FormControl` on construction, like so:

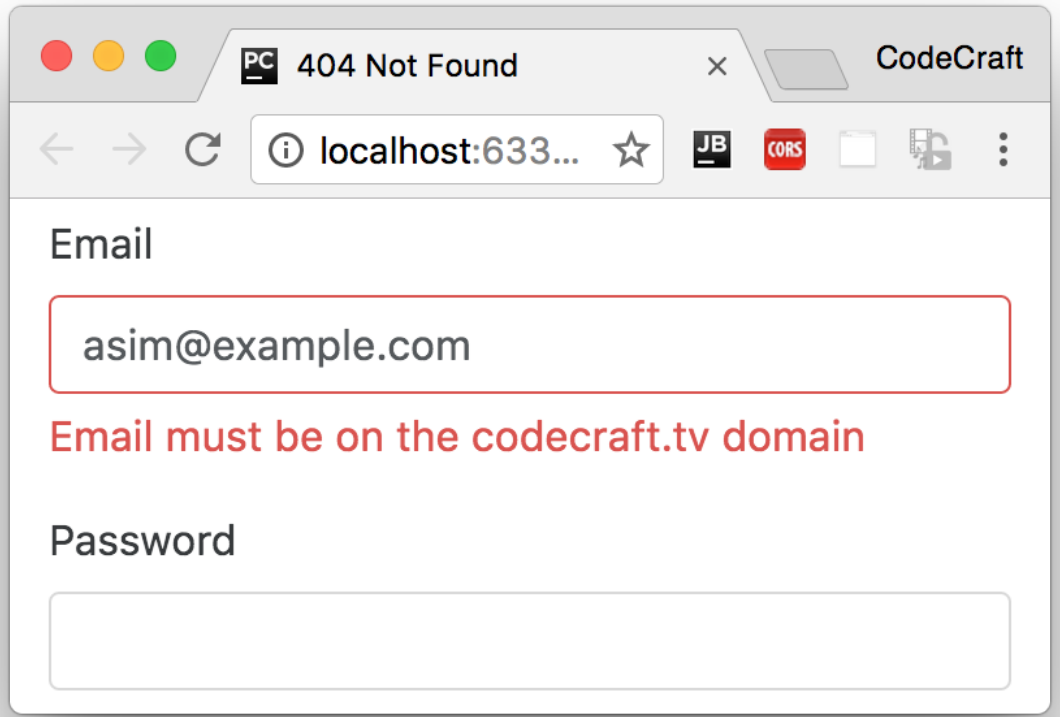
```
this.email = new FormControl('', [
  Validators.required,
  Validators.pattern("[^ @]*@[^ @]*"),
  emailDomainValidator
]);
```

Just like other validators lets add a helpful message to the user if the validator fails so they know how to fix it:

```
<div class="form-control-feedback"
     *ngIf="email.errors && (email.dirty || email.touched)">
  <p *ngIf="email.errors.required">Email is required</p>
  <p *ngIf="password.errors.pattern">The email address must contain at least the @
character</p>
  <p *ngIf="email.errors.emailDomain">Email must be on the codecraft.tv domain</p> ①
</div>
```

① The error object returned from the validator function is merged into to the `email.errors` object so is the key `emailDomain` is present then we know the `emailDomainValidator` is failing.

Now if we try to type in an email address that *doesn't* end in `codecraft.tv` we see this validation message printed on screen:



Next up we'll look at how we can re-package our validator function for use in template driven forms.

Custom template driven form validator

To use our validator function in a template driven form we need to:

1. Create a directive and attach it to the template form control.
2. Provide the directive with the validator function on the token `NG_VALIDATORS`.

```
import {NG_VALIDATORS} from '@angular/forms';
.
.
.
@Directive({
  selector: '[emailDomain][ngModel]', ①
  providers: [
    {
      provide: NG_VALIDATORS, ②
      useValue: emailDomainValidator, ③
      multi: true ④
    }
  ]
})
class EmailDomainValidator {
}
```

① Attached to all input controls which have both the `emailDomain` and `ngModel` attribute.

② We provide on the special token `NG_VALIDATORS`.

③ The `emailDomainValidator` function we created for the model driven form is the dependency.

④ This provider is a special kind of provider called a multi provider.

Multi providers return *multiple* dependencies as a list for a given token. So with our provider above we are just *adding* to the list of dependencies that are returned when we request the `NG_VALIDATORS` token.

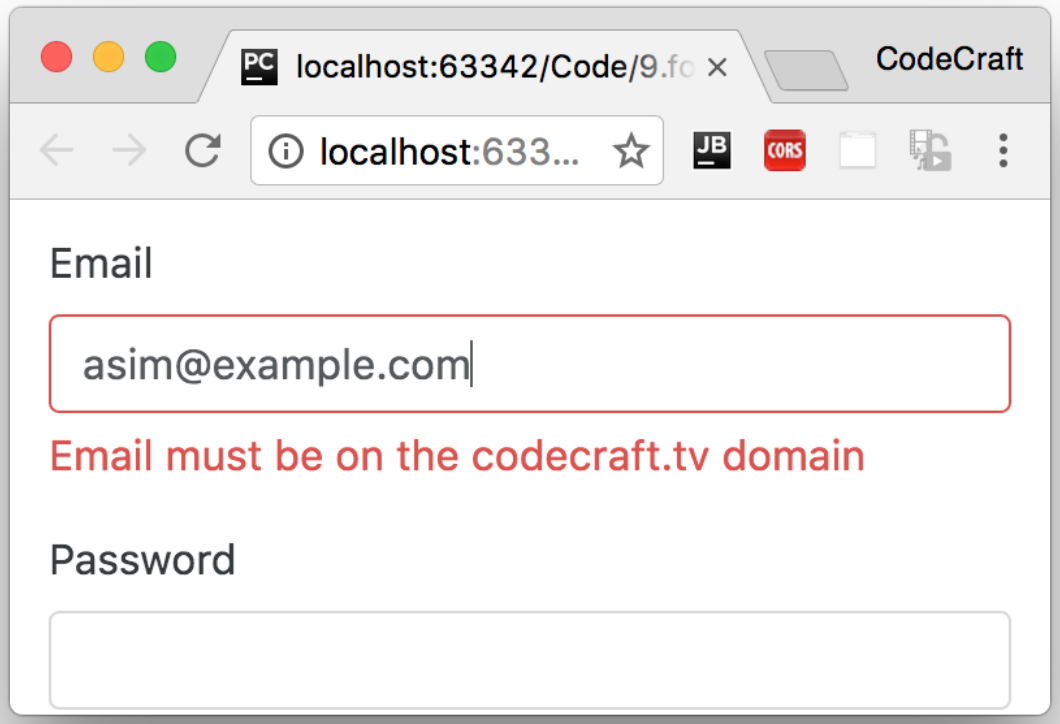
We declare this new directive on our `NgModule`:

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    TemplateFormComponent,
    EmailDomainValidator
  ],
  bootstrap: [
    AppComponent
  ],
})
class AppModule {
}
```

Finally we add this directive to our template form control like so:

```
<input type="email"
       class="form-control"
       name="email"
       [(ngModel)]="model.email"
       required
       pattern="[^ @]*@[^ @]*"
       emailDomain
       #email="ngModel">
```

Now just like the model driven form when we type into the email field an email that doesn't end in codecraft.tv we see the same error:



Summary

A validator in Angular is a function which returns null if a control is valid or an error object if it's invalid.

For model driven forms we create custom validation functions and pass them into the `FormControl` constructor.

For template driven forms we need to create validator directives and provide the validator function to the directive via DI.

Through careful planning we can share the same validation code between the model driven and template driven forms.

The validator we created in this lecture hardcodes the *domain*, in the next lecture we will look at how we can make our validators configurable with different domains.

Model Driven Listing

http://plnkr.co/edit/mLdGJG23cyPkhzncye5t?p=preview

script.ts

```
import {
```

```
    NgModule,
    Component,
    Pipe,
    OnInit
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormsModule,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

function emailDomainValidator(control: FormControl) {
  let email = control.value;
  if (email && email.indexOf("@") != -1) {
    let [_, domain] = email.split("@");
    if (domain !== "codecraft.tv") {
      return {
        emailDomain: {
          parsedDomain: domain
        }
      }
    }
  }
  return null;
}

@Component({
  selector: 'model-form',
  template: `<form novalidate
      [formGroup]="myform">

  <fieldset formGroupName="name">
    <div class="form-group"
         [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
      }">
      <label>First Name</label>
      <input type="text"
             class="form-control"
             formControlName="firstName"
             required>
      <div class="form-control-feedback"
           *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
        <p *ngIf="firstName.errors.required">First Name is required</p>
      </div>
```

```
    </div>

    <div class="form-group"
         [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
      }">
      <label>Last Name</label>
      <input type="text"
             class="form-control"
             formControlName="lastName"
             required>
      <div class="form-control-feedback"
           *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
        <p *ngIf="lastName.errors.required">Last Name is required</p>
      </div>
    </div>
  </fieldset>


  <div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
    <label>Email</label>
    <input type="email"
           class="form-control"
           formControlName="email"
           required>
    <div class="form-control-feedback"
         *ngIf="email.errors && (email.dirty || email.touched)">
      <p *ngIf="email.errors.required">Email is required</p>
      <p *ngIf="password.errors.pattern">The email address must contain at least the @
character</p>
      <p *ngIf="email.errors.emailDomain">Email must be on the codecraft.tv domain</p>
    </div>

  </div>

  <div class="form-group"
       [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
   }">
    <label>Password</label>
    <input type="password"
           class="form-control"
           formControlName="password"
           required>
```

```
    <div class="form-control-feedback"
         *ngIf="password.errors && (password.dirty || password.touched)">
      <p *ngIf="password.errors.required">Password is required</p>
      <p *ngIf="password.errors.minlength">Password must be 8 characters long, we need
another {{password.errors.minlength.requiredLength -
password.errors.minlength.actualLength}} characters </p>
    </div>
  </div>

  <div class="form-group"
       [ngClass]="{
        'has-danger': language.invalid && (language.dirty || language.touched),
        'has-success': language.valid && (language.dirty || language.touched)
      }">
    <label>Language</label>
    <select class="form-control"
            formControlName="language">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">{{lang}}
      </option>
    </select>
  </div>

  <pre>{{myform.value | json}}</pre>
</form>`
})
class ModelFormComponent implements OnInit {
  langs: string[] = [
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;
  firstName: FormControl;
  lastName: FormControl;
  email: FormControl;
  password: FormControl;
  language: FormControl;


  ngOnInit() {
    this.createFormControls();
    this.createForm();
  }

  createFormControls() {
    this.firstName = new FormControl('', Validators.required);
    this.lastName = new FormControl('', Validators.required);
    this.email = new FormControl('', [
      Validators.required,
```

```
      Validators.pattern("[^ @]*@[^ @]*"),
      emailDomainValidator
    ]);
    this.password = new FormControl('', [
      Validators.required,
      Validators.minLength(8)
    ]);
    this.language = new FormControl('');
  }

  createForm() {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName,
      }),
      email: this.email,
      password: this.password,
      language: this.language
    });
  }
}


@Component({
  selector: 'app',
  template: `<model-form></model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule],
  declarations: [
    AppComponent,
    ModelFormComponent
  ],
  bootstrap: [
    AppComponent
  ]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Template Driven Listing

http://plnkr.co/edit/iXI6fJaV02xPN9PcRY3b?p=preview

script.ts

```
import {
    NgModule,
    Component,
    OnInit,
    ViewChild,
    Directive,
    Inject,
    Input,
} from '@angular/core';
import {
    NG_VALIDATORS,
    FormsModule,
    FormGroup,
    FormControl,
    ValidatorFn,
    Validators
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

class Signup {
  constructor(public firstName: string = '',
              public lastName: string = '',
              public email: string = '',
              public password: string = '',
              public language: string = '') {
  }
}

function emailDomainValidator(control: FormControl) {
  let email = control.value;
  if (email && email.indexOf("@") != -1) {
    let [_, domain] = email.split("@");
    if (domain !== "codecraft.tv") {
      return {
        emailDomain: {
          parsedDomain: domain
        }
      }
    }
  }
  return null;
}

@Directive({
```

```
  selector: '[emailDomain][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useValue: emailDomainValidator,
      multi: true
    }
  ]
})
class EmailDomainValidator {
}


@Component({
  selector: 'template-form',
  template: `<!--suppress ALL -->
<form novalidate
      (ngSubmit)="onSubmit()"
      #f="ngForm">

    <fieldset ngModelGroup="name">
        <div class="form-group"
             [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
   }">
            <label>First Name</label>
            <input type="text"
                   class="form-control"
                   name="firstName"
                   [(ngModel)]="model.firstName"
                   required
                   #firstName="ngModel">
            <div class="form-control-feedback"
                 *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
                <p *ngIf="firstName.errors.required">First name is required</p>
            </div>
        </div>

        <div class="form-group"
             [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
   }">
            <label>Last Name</label>
            <input type="text"
                   class="form-control"
                   name="lastName"
                   [(ngModel)]="model.lastName"
                   required
                   #lastName="ngModel">
```

```
            <div class="form-control-feedback"
                 *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
                <p *ngIf="lastName.errors.required">Last name is required</p>
            </div>
        </div>
    </fieldset>


    <div class="form-group"
         [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
        <label>Email</label>
        <input type="email"
               class="form-control"
               name="email"
               [(ngModel)]="model.email"
               required
               pattern="[^ @]*@[^ @]*"
               emailDomain
               #email="ngModel">
        <div class="form-control-feedback"
             *ngIf="email.errors && (email.dirty || email.touched)">
            <p *ngIf="email.errors.required">Email is required</p>
            <p *ngIf="email.errors.pattern">Email must contain at least the @
character</p>
            <!--<p *ngIf="email.errors.emailDomain">Email must be on the codecraft.tv
domain</p>-->
            <p *ngIf="email.errors.emailDomain">Email must be on the {{
email.errors.emailDomain.requiredDomain }} domain</p>
        </div>
    </div>


    <div class="form-group"
         [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
  }">
        <label>Password</label>
        <input type="password"
               class="form-control"
               name="password"
               [(ngModel)]="model.password"
               required
               minlength="8"
               #password="ngModel">
        <div class="form-control-feedback"
             *ngIf="password.errors && (password.dirty || password.touched)">
            <p *ngIf="password.errors.required">Password is required</p>
```

```
            <p *ngIf="password.errors.minlength">Password must be at least 8
characters long</p>
        </div>
    </div>

    <div class="form-group">
        <label>Language</label>
        <select class="form-control"
                name="language"
                [(ngModel)]="model.language">
            <option value="">Please select a language</option>
            <option *ngFor="let lang of langs"
                    [value]="lang">{{lang}}
            </option>
        </select>
    </div>

    <button type="submit"
            class="btn btn-primary"
            [disabled]="f.invalid">Submit
    </button>

    <pre>{{f.value | json}}</pre>
</form>
`
})
class TemplateFormComponent {

  model: Signup = new Signup();
  @ViewChild('f') form: any;

  langs: string[] = [
    'English',
    'French',
    'German',
  ];

  onSubmit() {
    if (this.form.valid) {
      console.log("Form Submitted!");
      this.form.reset();
    }
  }
}

@Component({
  selector: 'app',
  template: `<template-form></template-form>`
})
class AppComponent {
}
```

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    TemplateFormComponent,
    EmailDomainValidator
  ],
  bootstrap: [
    AppComponent
  ]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Configurable Custom Form Validators

Learning Objectives

- Know how to create advanced *configurable* custom validators for both model driven and template driven forms.

Configurable model driven validators

In the previous lecture we created a custom email validator which checked that emails ended in a certain domain, a domain which we hardcoded in the validator.

What if we wanted to make the domain name configurable, so the validator can work for other domains?

We solve this by turning our validator function into a *factory function*. The factory function returns a validator function configured as we want, like so:

```
function emailDomainValidatorFactory(requiredDomain) { ①
  return function (control: FormControl) { ②
    let email = control.value;
    if (email && email.indexOf("@") != -1) {
      let [_, domain] = email.split("@");
      if (domain !== requiredDomain) { ③
        return {
          emailDomain: {
            valid: false,
            parsedDomain: domain
          }
        }
      }
    }
    return null;
  }
}
```

① We pass into our factory function the domain we want to limit emails to.

② We return from our factory function the *actual* validator function.

③ We use the passed in `requiredDomain` to check emails against instead of a hardcoded string.

We can then use this in our FormControl like so:

```
this.email = new FormControl('', [
  Validators.required,
  Validators.pattern("[^ @]*@[^ @]*"),
  emailDomainValidatorFactory('codecraft.tv')
]);
```

To match the built-in validators we can create our own Validators class and have the factory function as a static member function, like so:

```
class CodeCraftValidators {
  static emailDomain(requiredDomain) {
    return function (control: FormControl) {
      let email = control.value;
      if (email && email.indexOf("@") != -1) {
        let [_, domain] = email.split("@");
        if (domain !== requiredDomain) {
          return {
            emailDomain: {
              valid: false,
              parsedDomain: domain
            }
          }
        }
      }
      return null;
    }
  }
}
```

Then we can use the validator in our form control like so:

```
this.email = new FormControl('', [
  Validators.required,
  Validators.pattern("[^ @]*@[^ @]*"),
  CodeCraftValidators.emailDomain('codecraft.tv')
]);
```

Configurable template driven validators

To configure our directive validator we need to:

1. *Provide* the required domain name to the DI framework.
2. *Inject* the required domain name into our directive constructor.
3. Use the factory function to create a configured validator function.
4. Provide this configured validator function to the directives providers.

Numbers 1 & 2 are quite simple. For numbers 3 & 4 we need to take advantage of another fact. A validator as well as being a function can also be a class with a member function called `validate`. So we can actually turn our directives class into a validator class.

Firstly lets provide the configurable domain name in our `NgModule` like so:

```
@NgModule({
  .
  .
  .
  providers: [
    {provide: 'RequiredDomain', useValue: 'codecraft.tv'}
  ]
})
```

Then lets update our directive so that it uses both the provided `RequiredDomain` and the `CodeCraftValidators.emailDomain` factory function.

```
class EmailDomainValidator {

  private valFn = ValidatorFn;

  constructor(@Inject('RequiredDomain') requiredDomain: string) { ①
    this.valFn = CodeCraftValidators.emailDomain(requiredDomain) ②
  }

  validate(control: FormControl) { ③
    return this.valFn(control);
  }
}
```

① We inject into the constructor the `RequiredDomain` token that we configured on the `NgModule`.

② We use the `CodeCraftValidators.emailDomain` factory function to create a configured validator function and store that locally.

③ Now whenever validate is called we simply call our configured validator function.

We also need to change the directives provider so it now points to using the class instead of the validator function we used previously:

```
@Directive({
  selector: '[emailDomain][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useClass: EmailDomainValidator, ①
      multi: true
    }
  ]
})
```

① This provider now provides an instance of the `EmailDomainValidator` class instead.

The validation error messages were also hardcoded to show `codecraft.tv` we need to make them show the required domain instead.

We can simply return the required domain in the error object and then use that in our error message, like so:

```
class CodeCraftValidators {
  static emailDomain(requiredDomain) {
    return function (control: FormControl) {
      console.log("here");
      let email = control.value;
      if (email && email.indexOf("@") != -1) {
        let [_, domain] = email.split("@");
        if (domain !== requiredDomain) {
          return {
            emailDomain: {
              parsedDomain: domain,
              requiredDomain: requiredDomain ①
            }
          }
        }
      }
      return null;
    }
  }
}
```

① Pass the `requiredDomain` back in the error object.

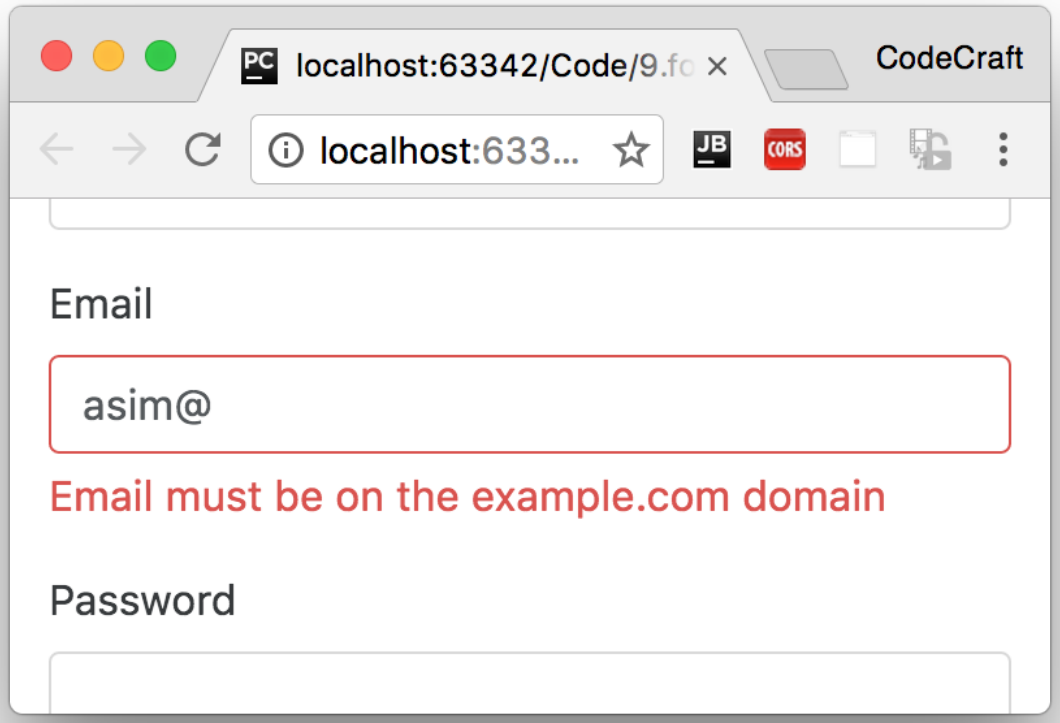
Then we just just the `requiredDomain` variable in our validation message:

```
<p *ngIf="email.errors.emailDomain">Email must be on the {{
email.errors.emailDomain.requiredDomain }} domain</p>
```

Now if we change the domain name in the global provider like so:

```
@NgModule({
  providers: [
    {provide: 'RequiredDomain', useValue: 'example.com'}
  ]
})
```

The email domain validator now check against this new domain instead:



Bindable template driven validators

We can also configure out template validator directive via property binding in the template instead, like so:

```
<input type="email"
       class="form-control"
       name="email"
       [(ngModel)]="model.email"
       required
       pattern="[^ @]*@[^ @]*"
       [emailDomain]='codecraft.tv' ①
       #email="ngModel">
```

① We can configure the validator via template property binding.

Then we update our `EmailDomainValidator` class so it can take the required domain as an input:

```
class EmailDomainValidator {
  @Input('emailDomain') emailDomain: string; ①
  private valFn = Validators.nullValidator;

  ngOnChanges(): void { ②
    if (this.emailDomain) {
      this.valFn = CodeCraftValidators.emailDomain(this.emailDomain)
    } else {
      this.valFn = Validators.nullValidator;
    }
  }

  validate(control: FormControl) {
    return this.valFn(control);
  }
}
```

① First we create an input property on our directive named the same as our selector.

② We know the `emailDomain` input property has been set when the ngOnChanges lifecycle function is called, this is where we now initialise our directive with the configured validator function.

Our directive needs one more change in order to function, we are providing this as a class, we need to provide as an alias so we get exactly the same `instance` provided to the validators.

```
@Directive({
  selector: '[emailDomain][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: EmailDomainValidator,
      multi: true
    }
  ]
})
```

Now we have a template driven form validator which can be configured via input property binding.

Summary

For model driven forms we use a factory function which returns a validator function configured as we want.

For template driven forms we create a validator class, often re-using the same factory function as

was used in model driven forms.

Model Driven Listing

http://plnkr.co/edit/mLdGJG23cyPkhzncye5t?p=preview

script.ts

```
import {
    NgModule,
    Component,
    Pipe,
    OnInit
} from '@angular/core';
import {
    ReactiveFormsModule,
    FormsModule,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

function emailDomainValidator(control: FormControl) {
  let email = control.value;
  if (email && email.indexOf("@") != -1) {
    let [_, domain] = email.split("@");
    if (domain !== "codecraft.tv") {
      return {
        emailDomain: {
          parsedDomain: domain
        }
      }
    }
  }
  return null;
}

@Component({
  selector: 'model-form',
  template: `<form novalidate
      [formGroup]="myform">

  <fieldset formGroupName="name">
    <div class="form-group"
         [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
      }">
```

```
      <label>First Name</label>
      <input type="text"
             class="form-control"
             formControlName="firstName"
             required>
      <div class="form-control-feedback"
           *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
        <p *ngIf="firstName.errors.required">First Name is required</p>
      </div>

    </div>

    <div class="form-group"
         [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
      }">
      <label>Last Name</label>
      <input type="text"
             class="form-control"
             formControlName="lastName"
             required>
      <div class="form-control-feedback"
           *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
        <p *ngIf="lastName.errors.required">Last Name is required</p>
      </div>
    </div>
  </fieldset>


  <div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
   }">
    <label>Email</label>
    <input type="email"
           class="form-control"
           formControlName="email"
           required>
    <div class="form-control-feedback"
         *ngIf="email.errors && (email.dirty || email.touched)">
      <p *ngIf="email.errors.required">Email is required</p>
      <p *ngIf="password.errors.pattern">The email address must contain at least the @
character</p>
      <p *ngIf="email.errors.emailDomain">Email must be on the codecraft.tv domain</p>
    </div>

  </div>

  <div class="form-group"
```

```
       [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
    }">
     <label>Password</label>
     <input type="password"
            class="form-control"
            formControlName="password"
            required>
     <div class="form-control-feedback"
          *ngIf="password.errors && (password.dirty || password.touched)">
       <p *ngIf="password.errors.required">Password is required</p>
       <p *ngIf="password.errors.minlength">Password must be 8 characters long, we need
another {{password.errors.minlength.requiredLength -
password.errors.minlength.actualLength}} characters </p>
     </div>
  </div>

  <div class="form-group"
       [ngClass]="{
        'has-danger': language.invalid && (language.dirty || language.touched),
        'has-success': language.valid && (language.dirty || language.touched)
       }">
     <label>Language</label>
     <select class="form-control"
             formControlName="language">
       <option value="">Please select a language</option>
       <option *ngFor="let lang of langs"
               [value]="lang">{{lang}}
       </option>
     </select>
  </div>

  <pre>{{myform.value | json}}</pre>
</form>`
})
class ModelFormComponent implements OnInit {
  langs: string[] = [
    'English',
    'French',
    'German',
  ];
  myform: FormGroup;
  firstName: FormControl;
  lastName: FormControl;
  email: FormControl;
  password: FormControl;
  language: FormControl;


  ngOnInit() {
```

```
    this.createFormControls();
    this.createForm();
  }

  createFormControls() {
    this.firstName = new FormControl('', Validators.required);
    this.lastName = new FormControl('', Validators.required);
    this.email = new FormControl('', [
      Validators.required,
      Validators.pattern("[^ @]*@[^ @]*"),
      emailDomainValidator
    ]);
    this.password = new FormControl('', [
      Validators.required,
      Validators.minLength(8)
    ]);
    this.language = new FormControl('');
  }

  createForm() {
    this.myform = new FormGroup({
      name: new FormGroup({
        firstName: this.firstName,
        lastName: this.lastName,
      }),
      email: this.email,
      password: this.password,
      language: this.language
    });
  }
}


@Component({
  selector: 'app',
  template: `<model-form></model-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule],
  declarations: [
    AppComponent,
    ModelFormComponent
  ],
  bootstrap: [
```

```
    AppComponent
  ]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Template Driven Listing

http://plnkr.co/edit/iXI6fJaV02xPN9PcRY3b?p=preview

script.ts

```
import {
    NgModule,
    Component,
    OnInit,
    ViewChild,
    Directive,
    Inject,
    Input,
} from '@angular/core';
import {
    NG_VALIDATORS,
    FormsModule,
    FormGroup,
    FormControl,
    ValidatorFn,
    Validators
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

class Signup {
  constructor(public firstName: string = '',
              public lastName: string = '',
              public email: string = '',
              public password: string = '',
              public language: string = '') {
  }
}

function emailDomainValidator(control: FormControl) {
  let email = control.value;
  if (email && email.indexOf("@") != -1) {
    let [_, domain] = email.split("@");
    if (domain !== "codecraft.tv") {
      return {
        emailDomain: {
```

```
          parsedDomain: domain
        }
      }
    }
  }
  return null;
}

@Directive({
  selector: '[emailDomain][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useValue: emailDomainValidator,
      multi: true
    }
  ]
})
class EmailDomainValidator {
}


@Component({
  selector: 'template-form',
  template: `<!--suppress ALL -->
<form novalidate
      (ngSubmit)="onSubmit()"
      #f="ngForm">

    <fieldset ngModelGroup="name">
        <div class="form-group"
             [ngClass]="{
        'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),
        'has-success': firstName.valid && (firstName.dirty || firstName.touched)
   }">
            <label>First Name</label>
            <input type="text"
                   class="form-control"
                   name="firstName"
                   [(ngModel)]="model.firstName"
                   required
                   #firstName="ngModel">
            <div class="form-control-feedback"
                 *ngIf="firstName.errors && (firstName.dirty || firstName.touched)">
                <p *ngIf="firstName.errors.required">First name is required</p>
            </div>
        </div>

        <div class="form-group"
             [ngClass]="{
        'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),
```

```
        'has-success': lastName.valid && (lastName.dirty || lastName.touched)
    }">
            <label>Last Name</label>
            <input type="text"
                   class="form-control"
                   name="lastName"
                   [(ngModel)]="model.lastName"
                   required
                   #lastName="ngModel">
            <div class="form-control-feedback"
                 *ngIf="lastName.errors && (lastName.dirty || lastName.touched)">
                <p *ngIf="lastName.errors.required">Last name is required</p>
            </div>
        </div>
     </fieldset>


     <div class="form-group"
          [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
    }">
        <label>Email</label>
        <input type="email"
               class="form-control"
               name="email"
               [(ngModel)]="model.email"
               required
               pattern="[^ @]*@[^ @]*"
               emailDomain
               #email="ngModel">
        <div class="form-control-feedback"
             *ngIf="email.errors && (email.dirty || email.touched)">
            <p *ngIf="email.errors.required">Email is required</p>
            <p *ngIf="email.errors.pattern">Email must contain at least the @
character</p>
            <!--<p *ngIf="email.errors.emailDomain">Email must be on the codecraft.tv
domain</p>-->
            <p *ngIf="email.errors.emailDomain">Email must be on the {{
email.errors.emailDomain.requiredDomain }} domain</p>
        </div>
     </div>


     <div class="form-group"
          [ngClass]="{
        'has-danger': password.invalid && (password.dirty || password.touched),
        'has-success': password.valid && (password.dirty || password.touched)
   }">
        <label>Password</label>
        <input type="password"
```

```
               class="form-control"
               name="password"
               [(ngModel)]="model.password"
               required
               minlength="8"
               #password="ngModel">
        <div class="form-control-feedback"
             *ngIf="password.errors && (password.dirty || password.touched)">
            <p *ngIf="password.errors.required">Password is required</p>
            <p *ngIf="password.errors.minlength">Password must be at least 8
characters long</p>
        </div>
    </div>

    <div class="form-group">
        <label>Language</label>
        <select class="form-control"
                name="language"
                [(ngModel)]="model.language">
            <option value="">Please select a language</option>
            <option *ngFor="let lang of langs"
                    [value]="lang">{{lang}}
            </option>
        </select>
    </div>

    <button type="submit"
            class="btn btn-primary"
            [disabled]="f.invalid">Submit
    </button>

    <pre>{{f.value | json}}</pre>
</form>
`
})
class TemplateFormComponent {

  model: Signup = new Signup();
  @ViewChild('f') form: any;

  langs: string[] = [
    'English',
    'French',
    'German',
  ];

  onSubmit() {
    if (this.form.valid) {
      console.log("Form Submitted!");
      this.form.reset();
    }
```

```
  }
}

@Component({
  selector: 'app',
  template: `<template-form></template-form>`
})
class AppComponent {
}


@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    TemplateFormComponent,
    EmailDomainValidator
  ],
  bootstrap: [
    AppComponent
  ]
})
class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);
```