



A

ng-book 2

The Complete Book on AngularJS 2



Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

ng-book 2

Felipe Coury, Ari Lerner, Nate Murray, & Carlos Taborda

© 2015 - 2016 Felipe Coury, Ari Lerner, Nate Murray, & Carlos Taborda

Table of Contents

[Book Revision](#)

[Prerelease](#)

[Bug Reports](#)

[Chat With The Community!](#)

[Be notified of updates via Twitter](#)

[Writing your First Angular 2 Web Application](#)

[Simple Reddit Clone](#)

[Getting started](#)

[TypeScript](#)

[Example Project](#)

[Angular's Dependencies](#)

[All Dependencies Loaded](#)

[Adding CSS](#)

[Our First TypeScript](#)

[Making a Component](#)

[Adding a template](#)

[Booting Our Application](#)

[Loading our Application](#)

[Running The App](#)

[Compiling the TypeScript Code to .js](#)

[Using npm](#)

[Serving The App](#)

[Compiling on every change](#)

[Adding Data to the Component](#)

[Working with arrays](#)

[Expanding our Application](#)

[The Application Component](#)

[Adding Interaction](#)

[Adding the Article Component](#)

[Rendering Multiple Rows](#)

[Creating an Article class](#)

[Storing multiple Articles](#)

[Configuring the ArticleComponent with inputs](#)

[Rendering a List of Articles](#)

[Adding New Articles](#)

[Finishing Touches](#)

[Displaying the Article Domain](#)

[Re-sorting Based on Score](#)

[Full Code Listing](#)

[Wrapping Up](#)

[Getting Help](#)

[TypeScript](#)

[Angular 2 is built in TypeScript](#)

[What do we get with TypeScript?](#)

[Types](#)

[Trying it out with a REPL](#)

[Built-in types](#)

[Classes](#)

[Properties](#)

[Methods](#)

[Constructors](#)

[Inheritance](#)

[Utilities](#)

[Fat Arrow Functions](#)

[Template Strings](#)

[Wrapping up](#)

[How Angular Works](#)

[Application](#)

[The Navigation Component](#)

[The Breadcrumbs Component](#)

[The Product List Component](#)

[Product Model](#)

[Components](#)

[Component Decorator](#)

[Component selector](#)

[Component template](#)

[Adding A Product](#)

[Viewing the Product with Template Binding](#)

[Adding More Products](#)

[Selecting a Product](#)

[Listing products using <products-list>](#)

[The ProductsList Component](#)

[Configuring the ProductsList @Component Options](#)

[Component inputs](#)

[Component outputs](#)

[Emitting Custom Events](#)

[Writing the ProductsList Controller Class](#)

[Writing the ProductsList View Template](#)

[The Full ProductsList Component](#)

[The ProductRow Component](#)

[ProductRow Component Configuration](#)

[ProductRow Component Definition Class](#)

[ProductRow template](#)

[ProductRow Full Listing](#)

[The ProductImage Component](#)

[The PriceDisplay Component](#)

[The ProductDepartment Component](#)

[The Completed Project](#)

[A Word on Data Architecture](#)

[Built-in Components](#)

[Introduction](#)

[NgIf](#)

[NgSwitch](#)

[NgStyle](#)

[NgClass](#)

[NgFor](#)

[Getting an index](#)

[NgNonBindable](#)

[Conclusion](#)

[Forms in Angular 2](#)

[Forms are Crucial, Forms are Complex](#)

[FormControls and FormGroups](#)

[FormControl](#)

[FormGroup](#)

[Our First Form](#)

[Simple SKU Form: @Component Annotation](#)

[Simple SKU Form: template](#)

[Simple SKU Form: Component Definition Class](#)

[Try it out!](#)

[Using FormBuilder](#)

[Injecting REACTIVE FORM DIRECTIVES](#)

[Using FormBuilder](#)

[Using myForm in the view](#)

[Try it out!](#)

[Adding Validations](#)

[Explicitly setting the sku FormControl as an instance variable](#)

[Custom Validations](#)

[Watching For Changes](#)

[ngModel](#)

[Wrapping Up](#)

[Data Architecture in Angular 2](#)

[An Overview of Data Architecture](#)

[Data Architecture in Angular 2](#)

[Data Architecture with Observables - Part 1: Services](#)

[Observables and RxJS](#)

[Note: Some RxJS Knowledge Required](#)

[Learning Reactive Programming and RxJS](#)

[Chat App Overview](#)

[Components](#)

[Models](#)

[Services](#)

[Summary](#)

[Implementing the Models](#)

[User](#)

[Thread](#)

[Message](#)

[Implementing UserService](#)

[currentUser stream](#)

[Setting a new user](#)

[UserService.ts](#)

[The MessagesService](#)

[the newMessages stream](#)

[the messages stream](#)

[The Operation Stream Pattern](#)

[Sharing the Stream](#)

[Adding Messages to the messages Stream](#)

[Our completed MessagesService](#)

[Trying out MessagesService](#)

[The ThreadsService](#)

[A map of the current set of Threads \(in threads\)](#)

[A chronological list of Threads, newest-first \(in orderedthreads\)](#)

[The currently selected Thread \(in currentThread\)](#)

[The list of Messages for the currently selected Thread \(in currentThreadMessages\)](#)

[Our Completed ThreadsService](#)

[Data Model Summary](#)

[Data Architecture with Observables - Part 2: View Components](#)

[Building Our Views: The ChatApp Top-Level Component](#)

[The chatThreads Component](#)

[chatThreads Controller](#)

[chatThreads template](#)

[The Single chatThread Component](#)

[chatThread Controller and ngOnInit](#)

[chatThread template](#)

[chatThread Complete Code](#)

[The ChatWindow Component](#)

[The ChatMessage Component](#)

[Setting incoming](#)

[The ChatMessage template](#)

[The Complete ChatMessage Code Listing](#)

[The ChatNavBar Component](#)

[The ChatNavBar @Component](#)

[The ChatNavBar Controller](#)

[The ChatNavBar template](#)

[The Completed ChatNavBar](#)

[Summary](#)

[Next Steps](#)

[HTTP](#)

[Introduction](#)

[Using @angular/http](#)

[import from @angular/http](#)

[A Basic Request](#)

[Building the SimpleHTTPComponent @Component](#)

[Building the SimpleHTTPComponent template](#)

[Building the SimpleHTTPComponent Controller](#)

[Full SimpleHTTPComponent](#)

[Writing a YouTubeSearchComponent](#)

[Writing a SearchResult](#)

[Writing the YouTubeService](#)

[Writing the SearchBox](#)

[Writing SearchResultComponent](#)

[Writing YouTubeSearchComponent](#)

[@angular/http API](#)

[Making a POST request](#)

[PUT / PATCH / DELETE / HEAD](#)

[RequestOptions](#)

[Summary](#)

[Routing](#)

[Why Do We Need Routing?](#)

[How client-side routing works](#)

[The beginning: using anchor tags](#)

[The evolution: HTML5 client-side routing](#)

[Writing our first routes](#)

[Components of Angular 2 routing](#)

[RouterConfig](#)

[RouterOutlet using <router-outlet>](#)

[RouterLink using \[routerLink\]](#)

[Putting it all together](#)

[Creating the Components](#)

[HomeComponent](#)

[AboutComponent](#)

[ContactComponent](#)

[Application component](#)

[Configuring the Routes](#)

[Routing Strategies](#)

[Path location strategy](#)

[Running the application](#)

[Route Parameters](#)

[ActivatedRoute](#)

[Music Search App](#)

[First Steps](#)

[The SpotifyService](#)

[The SearchComponent](#)

[Trying the search](#)

[TrackComponent](#)

[Wrapping up music search](#)

[Router Hooks](#)

[AuthService](#)

[LoginComponent](#)

[ProtectedComponent and Route Guards](#)

[Nested Routes](#)

[Configuring Routes](#)

[ProductsComponent](#)

[Summary](#)

[Advanced Components](#)

[Styling](#)

[View \(Style\) Encapsulation](#)

[Shadow DOM Encapsulation](#)

[No Encapsulation](#)

[Creating a Popup - Referencing and Modifying Host Elements](#)

[Popup Structure](#)

[Using ElementRef](#)

[Binding to the host](#)

[Adding a Button using exportAs](#)

[Creating a Message Pane with Transclusion](#)

[Changing the host CSS](#)

[Using ng-content](#)

[Querying Neighbor Directives - Writing Tabs](#)

[Tab Component](#)

[Tabset Component](#)

[Using the Tabset](#)

[Lifecycle Hooks](#)

[OnInit and OnDestroy](#)

[OnChange](#)

[DoCheck](#)

[AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked](#)

[Advanced Templates](#)

[Rewriting ngIf - ngBookIf](#)

[Rewriting ngFor - ngBookRepeat](#)

[Change Detection](#)

[Customizing Change Detection](#)

[Zones](#)

[Observables and OnPush](#)

[Summary](#)

[Converting an Angular 1 App to Angular 2](#)

[Peripheral Concepts](#)

[What We're Building](#)

[Mapping Angular 1 to Angular 2](#)

[Requirements for Interoperability](#)

[The Angular 1 App](#)

[The ng1-app HTML](#)

[Code Overview](#)

[ng1: PinsService](#)

[ng1: Configuring Routes](#)

[ng1: HomeController](#)

[ng1: / HomeController template](#)

[ng1: pin Directive](#)

[ng1: pin Directive template](#)

[ng1: AddController](#)

[ng1: AddController template](#)

[ng1: Summary](#)

[Building A Hybrid](#)

[Hybrid Project Structure](#)

[Bootstrapping our Hybrid App](#)

[What We'll Upgrade](#)

[A Minor Detour: Typing Files](#)

[Writing ng2 PinControlsComponent](#)

[Using ng2 PinControlsComponent](#)

[Downgrading ng2 PinControlsComponent to ng1](#)
[Adding Pins with ng2](#)
[Upgrading ng1 PinsService and \\$state to ng2](#)
[Writing ng2 AddPinComponent](#)
[Using AddPinComponent](#)
[Exposing an ng2 service to ng1](#)
[Writing the AnalyticsService](#)
[Downgrade ng2 AnalyticsService to ng1](#)
[Using AnalyticsService in ng1](#)

[Summary](#)

[References](#)

[Testing](#)

[Test driven?](#)

[End-to-end vs. Unit Testing](#)

[Testing Tools](#)

[Jasmine](#)

[Karma](#)

[Writing Unit Tests](#)

[Angular Unit testing framework](#)

[Setting Up Testing](#)

[Testing Services and HTTP](#)

[HTTP Considerations](#)

[Stubs](#)

[Mocks](#)

[Http MockBackend](#)

[addProviders Hooks](#)

[Testing getTrack](#)

[Testing Routing to Components](#)

[Creating a Router for Testing](#)

[Mocking dependencies](#)

[Spies](#)

[Back to Testing Code](#)

[fakeAsync and advance](#)

[inject](#)

[Testing ArtistComponent's Initialization](#)

[Testing ArtistComponent Methods](#)

[Testing ArtistComponent DOM Template Values](#)

[Testing Forms](#)

[Creating a ConsoleSpy](#)

[Installing the ConsoleSpy](#)

[Testing The Form](#)

[Refactoring Our Form Test](#)

[Testing HTTP requests](#)

[Testing a POST](#)

[Testing DELETE](#)

[Testing HTTP Headers](#)

[Testing YouTubeService](#)

[Conclusion](#)

[Dependency Injection](#)

[Injections Example: PriceService](#)

[“Don't Call Us...”](#)

[Dependency Injection Parts](#)

[Playing with an Injector](#)

[Providers](#)

[Using a Class](#)

[Using a Factory](#)

[Using a Value](#)

[Using an alias](#)

[Dependency Injection in Apps](#)

[Working with Injectors](#)

[Substituting values](#)

[Conclusion](#)

[Changelog](#)

[Revision 35 - 2016-06-30](#)

[Revision 34 - 2016-06-15](#)

[Revision 33 - 2016-05-11](#)

[Revision 32 - 2016-05-06](#)

[Revision 31 - 2016-04-28](#)

[Revision 30 - 2016-04-20](#)

[Revision 29 - 2016-04-08](#)

[Revision 28 - 2016-04-01](#)

[Revision 27 - 2016-03-25](#)

[Revision 26 - 2016-03-24](#)

[Revision 25 - 2016-03-21](#)

[Revision 24 - 2016-03-10](#)

[Revision 23 - 2016-03-04](#)

[Revision 22 - 2016-02-24](#)

[Revision 21 - 2016-02-20](#)

[Revision 20 - 2016-02-11](#)

[Revision 19 - 2016-02-04](#)

[Revision 18 - 2016-01-29](#)

[Revision 17 - 2016-01-28](#)

[Revision 16 - 2016-01-14](#)

[Revision 15 - 2016-01-07](#)

[Revision 14 - 2015-12-23](#)

[Revision 13 - 2015-12-17](#)

[Revision 12 - 2015-11-16](#)

[Revision 11 - 2015-11-09](#)

[Revision 10 - 2015-10-30](#)

[Revision 9 - 2015-10-15](#)

[Revision 8 - 2015-10-08](#)

[Revision 7 - 2015-09-23](#)

[Revision 6 - 2015-08-28](#)

[Revision 5](#)

[Revision 4](#)

[Revision 3](#)

[Revision 2](#)

[Revision 1](#)

[Notes](#)

Book Revision

Revision 35 - Covers up to Angular 2 (2.0.0-rc.4, 2016-06-30)

Prerelease

This book is a prerelease version and a work-in-progress.

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: us@fullstack.io.

Chat With The Community!

We're experimenting with a community chat room for this book using Gitter. If you'd like to hang out with other people learning Angular 2, come [join us on Gitter!](#)

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, [follow @fullstackio](#)

[](#)

Writing your First Angular 2 Web Application

Simple Reddit Clone

In this chapter we're going to build an application that allows the user to **post an article** (with a title and a URL) and then **vote on the posts**.

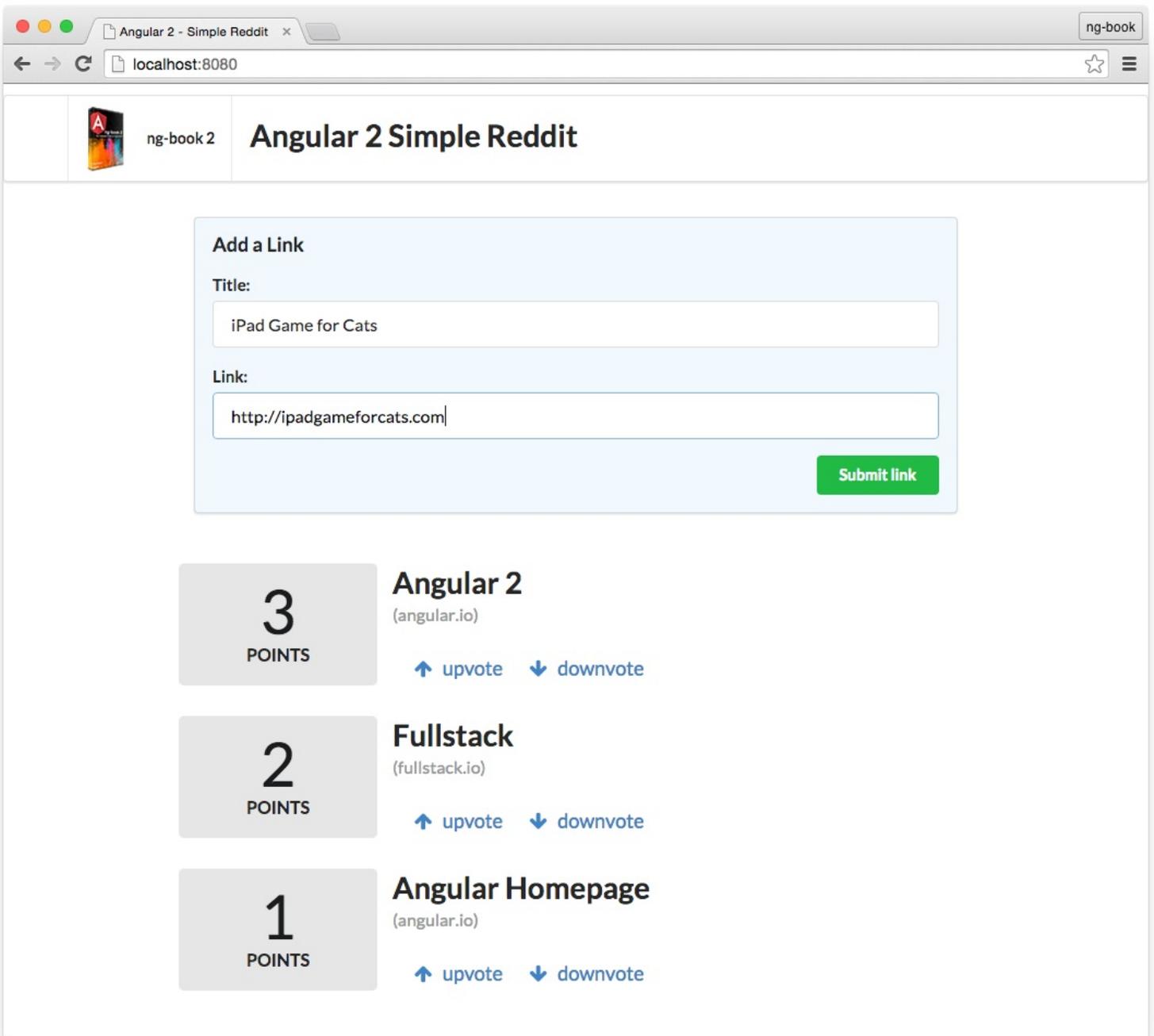
You can think of this app as the beginnings of a site like [Reddit](#) or [Product Hunt](#).

In this simple app we're going to cover most of the essentials of Angular 2 including:

- Building custom components
- Accepting user input from forms
- Rendering lists of objects into views
- Intercepting user clicks and acting on them

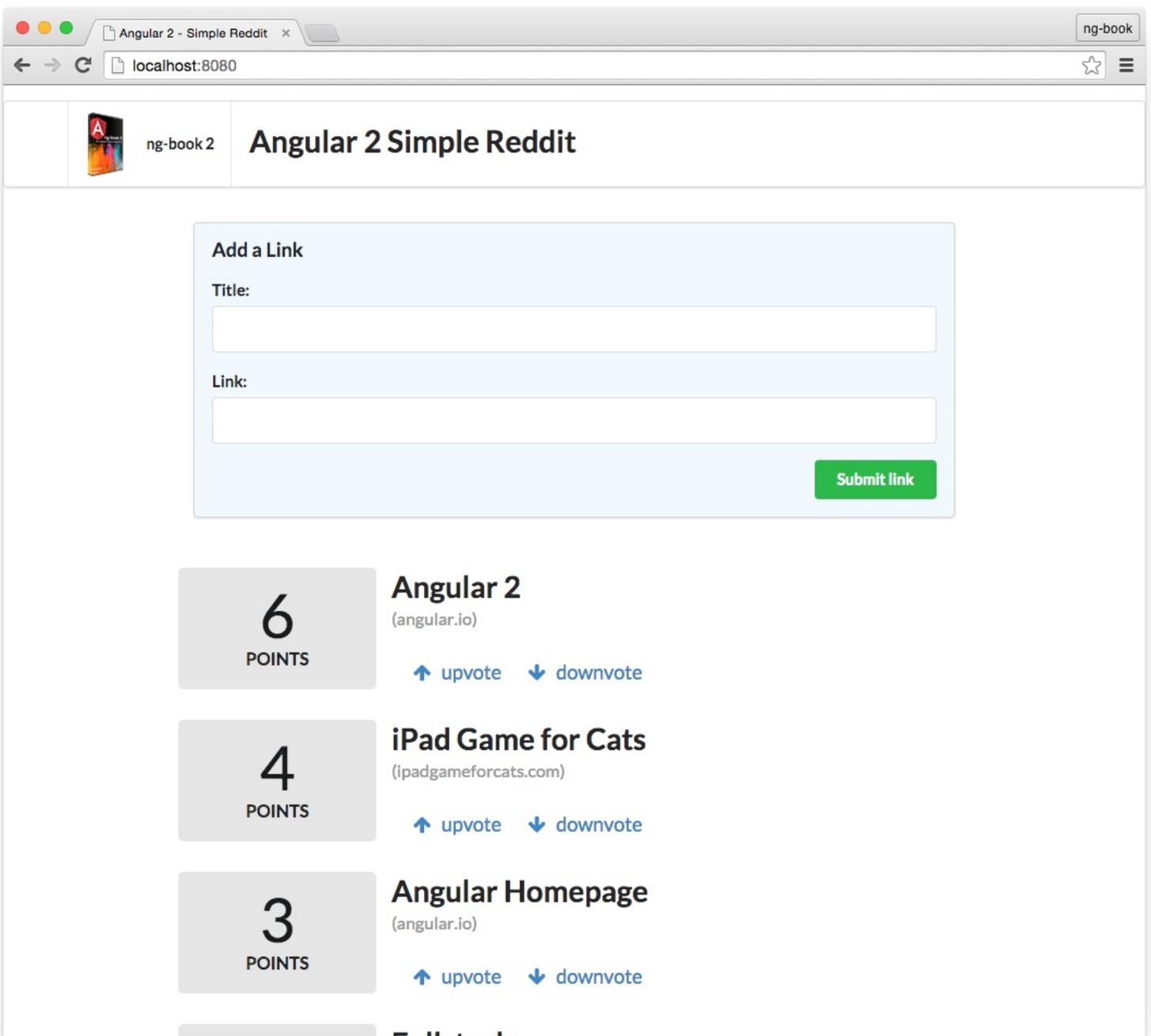
By the time you're finished with this chapter you'll have a good grasp on how to build basic Angular 2 applications.

Here's a screenshot of what our app will look like when it's done:



Completed application

First, a user will submit a new link and after submitting the users will be able to upvote or downvote each article. Each link will have a score and we can vote on which links we find useful.



App with new article

In this project, and throughout the book, we're going to use TypeScript. TypeScript is a superset of JavaScript ES6 that adds types. We're not going to talk about TypeScript in depth in this chapter, but if you're familiar with ES5 ("normal" javascript) / ES6 (ES2015) you should be able to follow along without any problems.

We'll go over TypeScript more in depth in the next chapter. So don't worry if you're having trouble with some of the new syntax.

Getting started

TypeScript

To get started with TypeScript, you'll need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to the Node.js website for detailed information: <https://nodejs.org/download/>.



Do I have to use TypeScript? No, you don't *have* to use TypeScript to use Angular 2, but you probably should. ng2 does have an ES5 API, but Angular 2 is written in TypeScript and generally that's what everyone is using. We're going to use TypeScript in this book because it's great and it makes working with Angular 2 easier. That said, it isn't strictly required.

Once you have Node.js setup, the next step is to install TypeScript. Make sure you install at least version 1.7 or greater. To install it, run the following npm command:

```
1 $ npm install -g 'typescript@1.9.0-dev.20160409'
```



npm is installed as part of Node.js. If you don't have npm on your system, make sure you used a Node.js installer that includes it.



Windows Users: We'll be using Linux/Mac-style commands on the commandline throughout this book. We'd highly recommend you install [Cygwin](#) as it will let you run commands just as we have them written out in this book.

Example Project

Now that you have your environment ready, let's start writing our first Angular2 application!

Open up the code download that came with this book and unzip it. In your terminal, cd into the `first_app/angular2-reddit-base` directory:

```
1 $ cd first_app/angular2-reddit-base
```



If you're not familiar with `cd`, it stands for "change directory". If you're on a Mac try the following:

1. Open up `/Applications/Utilities/Terminal.app`
2. Type `cd`, without hitting enter
3. In the Finder, Drag the `first_app/angular2-reddit-base` folder on to your terminal window
4. Hit Enter Now you are cded into the proper directory and you can move on to the next step!

Let's first use npm to install all the dependencies:

```
1 $ npm install
```

Create a new `index.html` file in the root of the project and add some basic structure:

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Angular 2 - Simple Reddit</title>
5   </head>
6   <body>
7   </body>
8 </html>
```

Your angular2-reddit-base directory should look something like this:

```
1 .
2 |-- README.md           // A helpful readme
3 |-- index.html         // Your index file
4 |-- index-full.html    // Sample index file
5 |-- node_modules/      // installed dependencies
6 |-- package.json       // npm configuration
7 |-- resources/         // images etc.
8 |-- styles.css         // stylesheet
9 |-- tsconfig.json      // compiler configuration
10 `-- tslint.json       // code-style guidelines
```

Angular 2 itself is a javascript file. So we need to add a script tag to our index.html document to include it. But Angular has some dependencies itself:

Angular's Dependencies



You don't strictly need to understand these dependencies in-depth in order to use Angular 2, but you do need to include them. Feel free to [skip this section](#) if you're not that interested in the dependencies, but make sure you copy and paste these script tags.

To run Angular 2, we depend on these four libraries:

- es6-shim
- zone.js
- reflect-metadata
- SystemJS

To include them, add the following inside your <head>

```
1 <script src="node_modules/es6-shim/es6-shim.js"></script>
2 <script src="node_modules/zone.js/dist/zone.js"></script>
3 <script src="node_modules/reflect-metadata/Reflect.js"></script>
4 <script src="node_modules/systemjs/dist/system.src.js"></script>
```



Notice that we're loading these .js files directly from a directory called node_modules. The node_modules directory will be created when you run npm install. If you don't have a node_modules directory, make sure your shell was "in" the directory angular2-reddit-base (e.g. by using cd angular2-reddit-base) when you typed npm install.

ES6 Shim

ES6 provides shims so that legacy JavaScript engines behave as closely as possible to ECMAScript 6. This shim isn't strictly needed for newer versions of Safari, Chrome, etc. but it is required for older

versions of IE.



What's a *shim*? Perhaps you've heard about *shims* or *polyfills* and you're not sure what they are. A *shim* is code that helps adapt between cross browsers to a standardized behavior.

For instance, check out this [ES6 Compatibility Table](#). Not every browser is completely compatible with every feature. By using different *shims* we're able to get standardized behavior across different browsers (and environments).

See also: [What is the difference between a shim and a polyfill?](#)

For more information on es6-shim [checkout the project page](#).

Zones

[Zone.js](#) is an advanced topic that we don't need to worry about much here. For know, just know that it is a library used by Angular, primarily for detecting changes to data. (If you're coming from Angular 1, you can think of zones as an automatic version of `$digest`. If you're not coming from Angular 1, you can ignore it for now.)

Reflect Metadata

Angular itself was written in Typescript, and Typescript provides [annotations](#) for adding metadata to code. Roughly speaking, the `reflect-metadata` package is a polyfill that lets us use this metadata.

SystemJS

SystemJS is a **module loader**. That is, it helps us create modules and resolve dependencies. Module loading in browser-side javascript is surprisingly complicated and SystemJS makes the process much easier.

All Dependencies Loaded

Now that we've added all of the dependencies, here's how our `index.html` should look now:

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Angular 2 - Simple Reddit</title>
5     <!-- Libraries -->
6     <script src="node_modules/es6-shim/es6-shim.js"></script>
7     <script src="node_modules/systemjs/dist/system.src.js"></script>
8     <script src="node_modules/rxjs/bundles/Rx.js"></script>
9     <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
10
11  </head>
12  <body>
13  </body>
14 </html>
```

Adding CSS

We also want to add some CSS styling so that our app isn't completely unstyled. Let's include two stylesheets as well:

```
1 <!doctype html>
2 <html>
3   <head>
```

```

4 <title>Angular 2 - Simple Reddit</title>
5 <!-- Libraries -->
6 <script src="node_modules/es6-shim/es6-shim.js"></script>
7 <script src="node_modules/zone.js/dist/zone.js"></script>
8 <script src="node_modules/reflect-metadata/Reflect.js"></script>
9 <script src="node_modules/systemjs/dist/system.src.js"></script>
10
11 <!-- Stylesheet -->
12 <link rel="stylesheet" type="text/css"
13     href="resources/vendor/semantic.min.css">
14 <link rel="stylesheet" type="text/css" href="styles.css">
15 </head>
16 <body>
17 </body>
18 </html>

```



For this project we're going to be using [Semantic-UI](#) to help with the styling. Semantic-UI is a CSS framework, similar to Foundation or Twitter Bootstrap. We've included it in the sample code download so all you need to do is add the `link` tag.

Our First TypeScript

Let's now create our first TypeScript file. Create a new file called `app.ts` in the same folder and add the following code:



Notice that we suffix our TypeScript file with `.ts` instead of `.js`. The problem is our browser doesn't know how to interpret TypeScript files. To solve this, we'll compile our `.ts` to a `.js` file in just a few minutes.

code/first_app/angular2-reddit-base/app.ts

```

1 import { bootstrap } from "@angular/platform-browser-dynamic";
2 import { Component } from "@angular/core";
3
4 @Component({
5   selector: 'hello-world',
6   template: `
7     <div>
8       Hello world
9     </div>
10 `
11 })
12 class HelloWorld {
13 }
14
15 bootstrap(HelloWorld);

```

This snippet may seem scary at first, but don't worry. We're going to walk through it step by step.

TypeScript is a form of *typed* Javascript. In order to use Angular in our browser, we need to tell the TypeScript compiler where to find some typing files. The reference statements specify the path to some typing files (ending in `.d.ts`). (If you don't understand this quite yet, just copy and paste these lines and don't worry about. We'll talk more about *why* we need to do this later.)

The `import` statement defines the modules we want to use to write our code. Here we're importing two things: `Component`, and `bootstrap`.

We import Component from the module "@angular/core". The "@angular/core" portion tells our program **where to find the dependencies** that we're looking for.

Similarly we import bootstrap from the module "@angular/platform-browser-dynamic".

Notice that the structure of this import is of the format `import { things } from wherever`. In the `{ things }` part what we are doing is called *destructuring*. Destructuring is a feature provided ES6 and we talk more about it in the next chapter.

The idea with the import is a lot like import in Java or require in Ruby: we're pulling in these dependencies from another module and making these dependencies available for use in this file.

Making a component

One of the big ideas behind Angular 2 is the idea of *components*.

In our Angular apps we write HTML markup that becomes our interactive application. But the browser only knows so many tags: the built-ins like `<select>` or `<form>` or `<video>` all have functionality defined by our browser creator. But what if we want to teach the browser new tags? What if we wanted to have a `<weather>` tag that shows the weather? Or what if we wanted to have a `<login>` tag that creates a login panel?

That is the idea behind components: we teach the browser new tags that have new functionality.

 If you have a background in Angular 1, **Components are the new version of directives**.

Let's create our very first component. When we have this component written, we will be able to use it in our HTML document like so:

```
1 <hello-world></hello-world>
```

So how do we actually define a new Component? A basic Component has two parts:

1. A component annotation
2. A component definition class

Let's take these one at a time.

If you've been programming in Javascript for a while then this next statement might seem a little weird:

```
1 @Component({
2   // ...
3 })
```

What is going on here? Well if you have a Java background it may look familiar to you: they are annotations.

Think of annotations as **metadata added to your code**. When we use `@Component` on the `HelloWorld` class, we are “decorating” the `HelloWorld` as a `Component`.

We want to be able to use this component in our markup by using a `<hello-world>` tag. To do that we configure the `@Component` and specify the selector as `hello-world`.

```
1 @Component({
2   selector: 'hello-world'
3 })
```

If you’re familiar with CSS selectors, XPath, or JQuery selectors you’ll know that there are lots of ways to configure a selector. Angular adds its own special sauce to the selector mix, and we’ll cover that later on. For now, just know that in this case we’re **defining a new tag**.

The `selector` property here indicates which DOM element this component is going to use. This way if we have any `<hello-world></hello-world>` tag within a template, it will be compiled using this `Component` class.

Adding a template

We can add a template to our `@Component` by passing the `template` option:

```
1 @Component({
2   selector: 'hello-world',
3   template: `
4     <div>
5       Hello world
6     </div>
7 `
8 })
```

Notice that we’re defining our `template` string between backticks (`` ... ``). This is a new (and fantastic) feature of ES6 that allows us to do **multiline strings**. Using backticks for multiline strings makes it easy to put templates inside your code files.

 **Should I really be putting templates in my code files?** The answer is: it depends. For a long time the commonly held belief was that you should keep your code and templates separate. While this might be easier for some teams, for some projects it adds overhead because you have to switch between a lot of files.

Personally, if my templates are shorter than a page I much prefer to have the templates alongside the code (that is, within the `.ts` file). When I can see both the logic and the view together and it’s easy to understand how they interact with one another.

The biggest drawback to putting your views inlined with your code is that many editors don’t support syntax highlighting of the internal strings (yet). Hopefully we’ll see more editors supporting syntax highlighting HTML within template strings soon.

Booting Our Application

The last line of our file `bootstrap(HelloWorld);` will start the application. The first argument indicates that the “main” component of our application is our `HelloWorld` component.

Once the application is bootstrapped, the `HelloWorld` component will be rendered where the `<hello-world></hello-world>` snippet is on the `index.html` file. Let’s try it out!

Loading our Application

To run our application, we need to do two things:

1. we need to tell our HTML document to import our app file
2. we need to use our `<hello-world>` component

Add the following to the body section:

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>First App - Hello world</title>
5     <!-- Libraries -->
6     <script src="node_modules/es6-shim/es6-shim.js"></script>
7     <script src="node_modules/zone.js/dist/zone.js"></script>
8     <script src="node_modules/reflect-metadata/Reflect.js"></script>
9     <script src="node_modules/systemjs/dist/system.src.js"></script>
10
11     <!-- Stylesheet -->
12     <link rel="stylesheet" type="text/css" href="resources/vendor/semantic.min.c\
13 ss">
14     <link rel="stylesheet" type="text/css" href="styles.css">
15   </head>
16   <body>
17     <script src="resources/systemjs.config.js"></script>
18     <script>
19       System.import('app.js')
20         .then(null, console.error.bind(console));
21     </script>
22
23     <hello-world></hello-world>
24
25   </body>
26 </html>
```

We've added two script tags here that configure our module loader, System.js:

1. We load `resources/systemjs.config.js` - this file tells System.js *how* to load libraries and files. The details aren't important now.
2. We import our `app.js` file.

The important line to understand here is:

```
1 System.import('app.js')
```

This tells System.js that we want to load `app.js` as our main entry point. There's one problem though: we don't have an `app.js` file yet! (Our file is `app.ts`, a TypeScript file.)

Running The App

Compiling the TypeScript Code to .js

Since our application is written in TypeScript, we used a file called `app.ts`. The next step is to compile it to JavaScript, so that the browser can understand it.

In order to do that, let's run the TypeScript compiler command line utility, called `tsc`:

```
1 tsc
```

If you get a prompt back with no error messages, it means that the compilation worked and we should now have the `app.js` file sitting in the same directory.

```
1 ls app.js
2 # app.js should exist
```



Troubleshooting:

Maybe you get the following message: `tsc: command not found`. This means that `tsc` is either not installed or not in your `PATH`. Try using the path to the `tsc` binary in your `node_modules` directory:

```
1 ./node_modules/.bin/tsc
```



You don't need to specify any arguments to the TypeScript compiler `tsc` in this case because it will look for `.ts` files in the current directory. If you don't get an `app.js` file, first make sure you're in the same directory as your `app.ts` file by using `cd` to change to that directory.

You may also get an error when you run `tsc`. For instance, maybe it says `app.ts(2,1): error TS2304: Cannot find name or app.ts(12,1): error TS1068: Unexpected token`.

In this case the compiler is giving you some hints as to where the error is. The section `app.ts(12,1):` is saying that the error is in the file `app.ts` on line 12 character 1. You can also search online for the error code and often you'll get a helpful explanation on how to fix it.



More Troubleshooting:

Maybe you get the following message: `error TS2307: Cannot find module '@angular/platform-browser-dynamic'`. The short answer is: *don't add filename argument* when running `tsc` and it should work.

The long answer is this (and feel free to skip this if you're just getting started with `tsc`): `tsc` it has some non-intuitive functionality: if you run `tsc` with no arguments it does the following:

- look in the current directory (or project directory if you specify the `-p` option) for a `tsconfig.json` file and then
- compile all `.ts` files in that directory

However, if you specify files (e.g. run it like `tsc app.ts` then `tsc` won't read your `tsconfig.json` and you may have to specify several more options in order to get it to compile correctly.

TypeScript requires *typing files* to know the types of certain code. We'll talk a lot more about types and typing files in this book. But for now, just know that the file `@angular/platform-browser-dynamic` is loaded because we specified it in the `tsconfig.json` and if you specify a particular file for `tsc` to compile, it won't know where to get `@angular/platform-browser-dynamic` unless you specify a lot more commandline flags.

Using npm

If your `tsc` command worked above, you can also use `npm` to compile the files. In the `package.json` included in the sample code we've defined a few shortcuts you can use to help compile.

Try running:

```
1 npm run tsc // compiles TypeScript code once and exits
2 npm run tsc:w // watches for changes and compiles on change
```

Serving The App

We have one more step to test our application: we need to run a local webserver to serve our app.

If you did an `npm install` earlier, you've already got a local webserver installed. To run it, just run the following command:

```
1 npm run serve
```

Then open up your browser to <http://localhost:8080>.

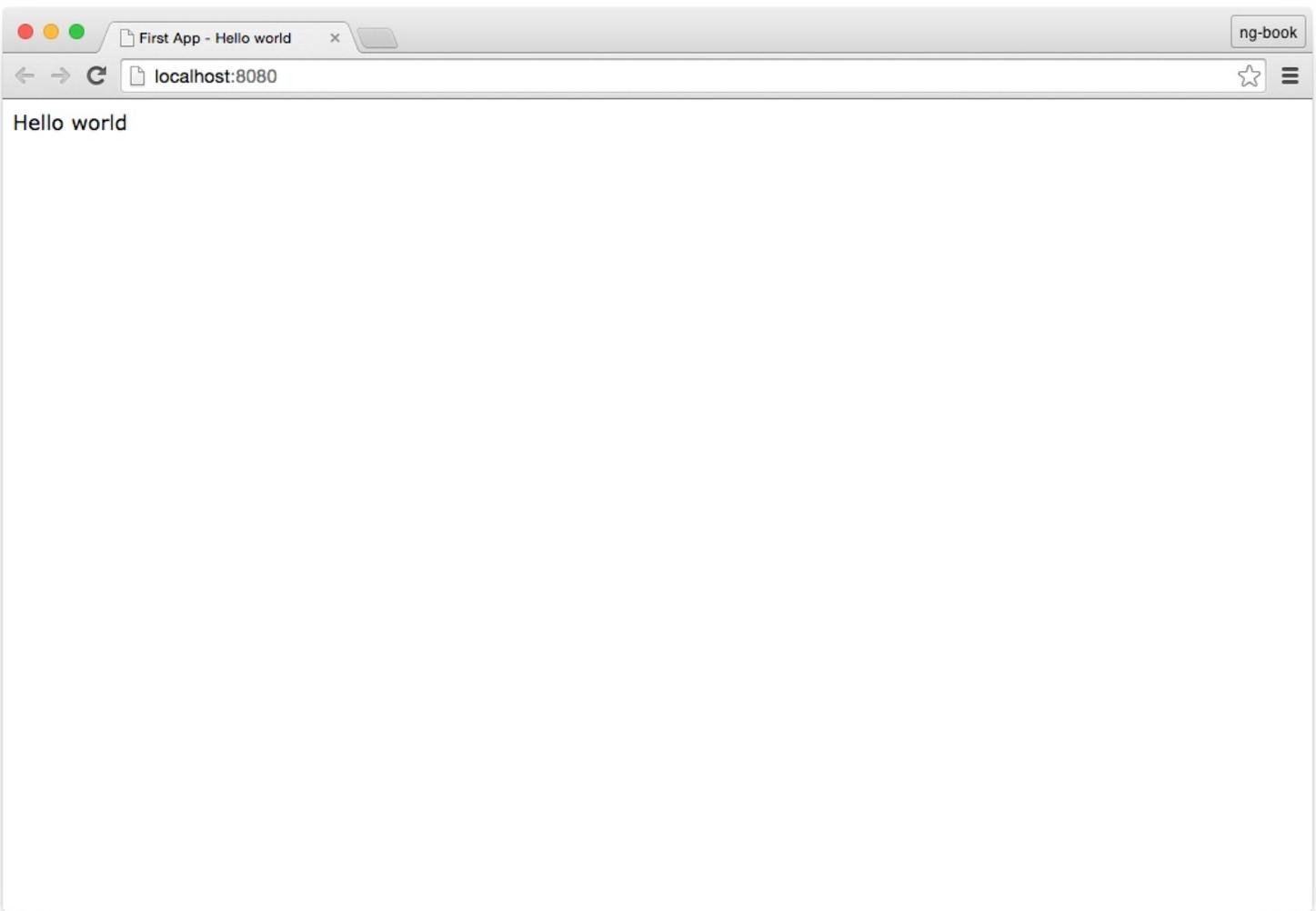
 **Why do I need a webserver?** If you've developed javascript applications before you probably know that sometimes you can simply open up the `index.html` file by double clicking on it and view it in your browser. This won't work for us because we're using SystemJS.

When you open the `index.html` file directly, your browser is going to use a `file:///` URL. Because of security restrictions, your browser will not allow AJAX requests to happen when using the `file:///` protocol (this is a good thing because otherwise javascript could read any file on your system and do something malicious with it).

So instead we run a local webserver that simply serve whatever is on the filesystem. This is really convenient for testing, but not how you would deploy your production application.

 If you have trouble with the hostname (`localhost`) or the port (`8080`) then simply adjust the configuration flags in the `package.json` file. For instance, if you're having trouble reaching `localhost`, try changing your hostname to `127.0.0.1`.

If everything worked correctly, you should see the following:



Completed application



If you're having trouble viewing your application here's a few things to try:

1. Make sure that your `app.js` file was created from the Typescript compiler `tsc`
2. Make sure that your webserver was started in the same directory as your `app.js` file
3. Make sure that your `index.html` file matches our code example above
4. Try opening the page in Chrome, right click, and pick "Inspect Element". Then click the "Console" tab and check for any errors.
5. If all else fails, [join us here to chat on Gitter!](#)

Compiling on every change

We will be making a lot of changes to our application code. Instead of having to run `tsc` everytime we make a change, we can take advantage of the `--watch` option. The `--watch` option will tell `tsc` to stay running and watch for any changes to our TypeScript files and automatically recompile to JavaScript on every change:

```
1 tsc --watch
2 message TS6042: Compilation complete. Watching for file changes.
```

In fact, this is so common that we've created a shortcut that will both

1. recompile on file changes and
2. reload your dev server

```
1 npm run go
```

Now you can just edit your code and changes will be reflected automatically in your browser.

Adding Data to the Component

Right now our component renders a static string, which means our component isn't very interesting.

Let's introduce `name` as a new property of our component. This way we can reuse the same component for different inputs.

Make the following changes:

```
1 @Component({
2   selector: 'hello-world',
3   template: `

Hello {{ name }}</div>`
4 })
5 class HelloWorld {
6   name: string;
7
8   constructor() {
9     this.name = 'Felipe';
10  }
11 }


```

Here we've changed three things:

1. name Property

On the `HelloWorld` class we added a *property*. Notice that the syntax is new relative to ES5 Javascript. When we write `name: string`; it means `name` is the name of the attribute we want to set and `string` is the *type*.

The typing is provided by TypeScript! This sets up a `name` property on *instances* of our `HelloWorld` class and the compiler ensures that `name` is a `string`.

2. A Constructor

On the `HelloWorld` class we define a *constructor*, i.e. function that is called when we create new instances of this class.

In our constructor we can assign our `name` property by using `this.name`

When we write:

```
1   constructor() {
2     this.name = 'Felipe';
3   }
```

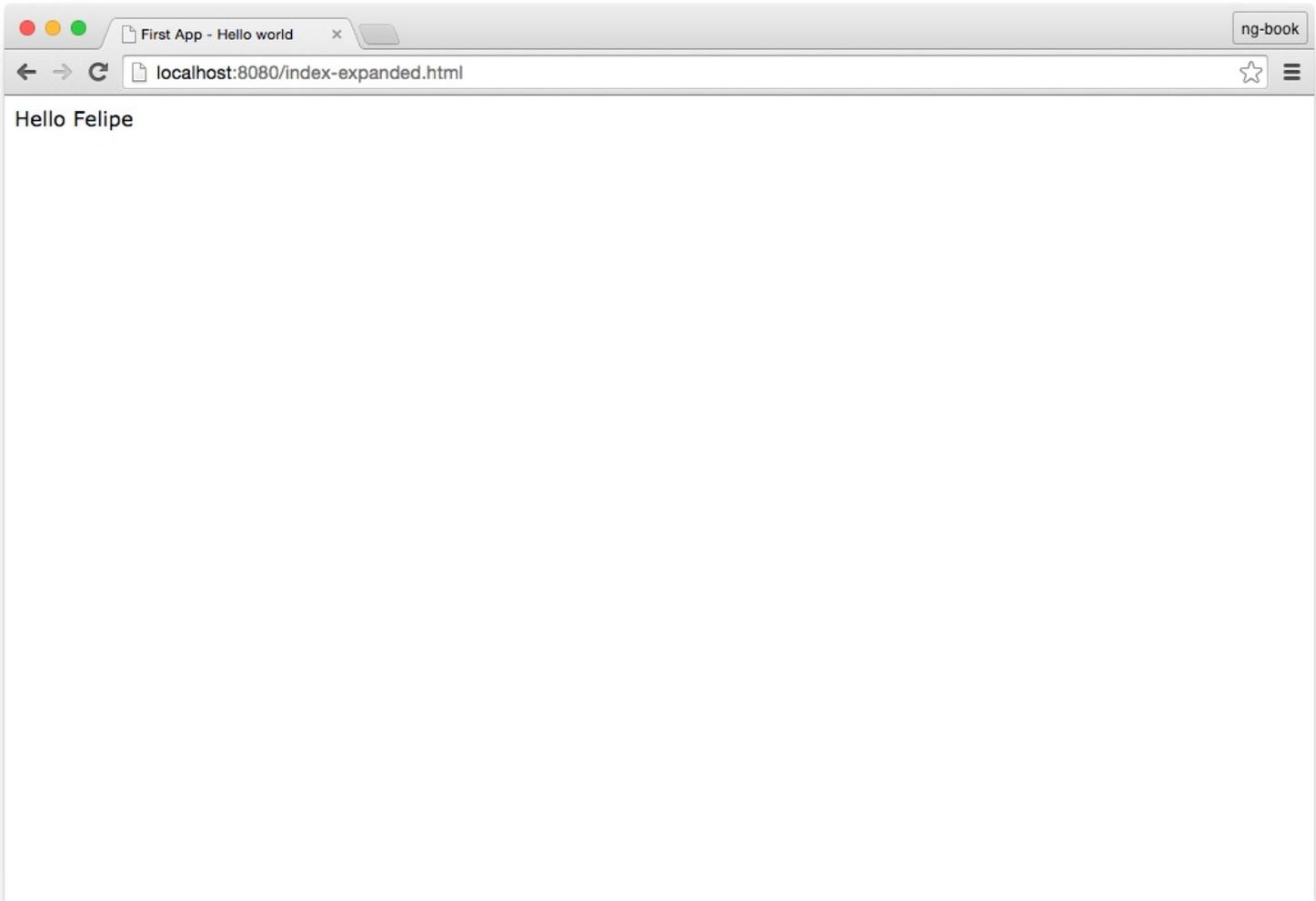
We're saying that whenever a new `HelloWorld` is created, set the `name` to `'Felipe'`.

3. Template Variable

On the template notice that we added a new syntax: `{{ name }}`. The brackets are called “template-tags” (or “mustache tags”). Whatever is between the template tags will be expanded as an *expression*. Here, because the template is *bound* to our Component, the name will expand to the value of `this.name` i.e. 'Felipe'.

Try it out

After making these changes reload the page and the page should display Hello Felipe



Application with Data

Working with arrays

Now we are able to say “Hello” to a single name, but what if we want to say “Hello” to a collection of names?

If you’ve worked with Angular 1 before, you probably used `ng-repeat` directive. In Angular 2, the analogous directive is called `ngFor` (we use it in the markup as `*ngFor`, which we’ll talk about soon). Its syntax is slightly different but they have the same purpose: **repeat the same markup for a collection of objects.**

Let’s make the following changes to our `app.ts` code:

```

1 import { bootstrap } from "@angular/platform-browser-dynamic";
2 import { Component } from "@angular/core";
3
4 @Component({
5   selector: 'hello-world',
6   template: `
7     <ul>
8       <li *ngFor="let name of names">Hello {{ name }}</li>
9     </ul>
10 `
11 })
12 class HelloWorld {
13   names: string[];
14
15   constructor() {
16     this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
17   }
18 }
19
20 bootstrap(HelloWorld);

```

The first change to point out is the new `string[]` property on our `HelloWorld` class. This syntax means that `names` is typed as an Array of strings. Another way to write this would be `Array<string>`.

We changed our constructor to set the value of `this.names` to `['Ari', 'Carlos', 'Felipe', 'Nate']`.

The next thing we changed was our `template`. We now have one `ul` and one `li` with a new `*ngFor="let name of names"` attribute. The `*` and `#` characters can be a little overwhelming at first, so let's break it down:

The `*ngFor` syntax says we want to use the `NgFor` directive on this attribute. You can think of `NgFor` akin to a `for` loop; the idea is that we're creating a new DOM element for every item in a collection.

The value states: `"let name of names"`. `names` is our array of names as specified on the `HelloWorld` object. `let name` is called a *reference*. When we say `"let name of names"` we're saying loop over each element in `names` and assign each one to a variable called `name`.

The `NgFor` directive will render one `li` tag for each entry found on the `names` array, declare a local variable `name` to hold the current item being iterated. This new variable will then be replaced inside the `Hello {{ name }}` snippet.



We didn't have to call the reference variable name. We could just as well have written:

```
1 <li *ngFor="let foobar of names">Hello {{ foobar }}</li>
```

But what about the reverse? Quiz question: what would have happened if we wrote:

```
1 <li *ngFor="let name of foobar">Hello {{ name }}</li>
```

We'd get an error because foobar isn't a property on the component.

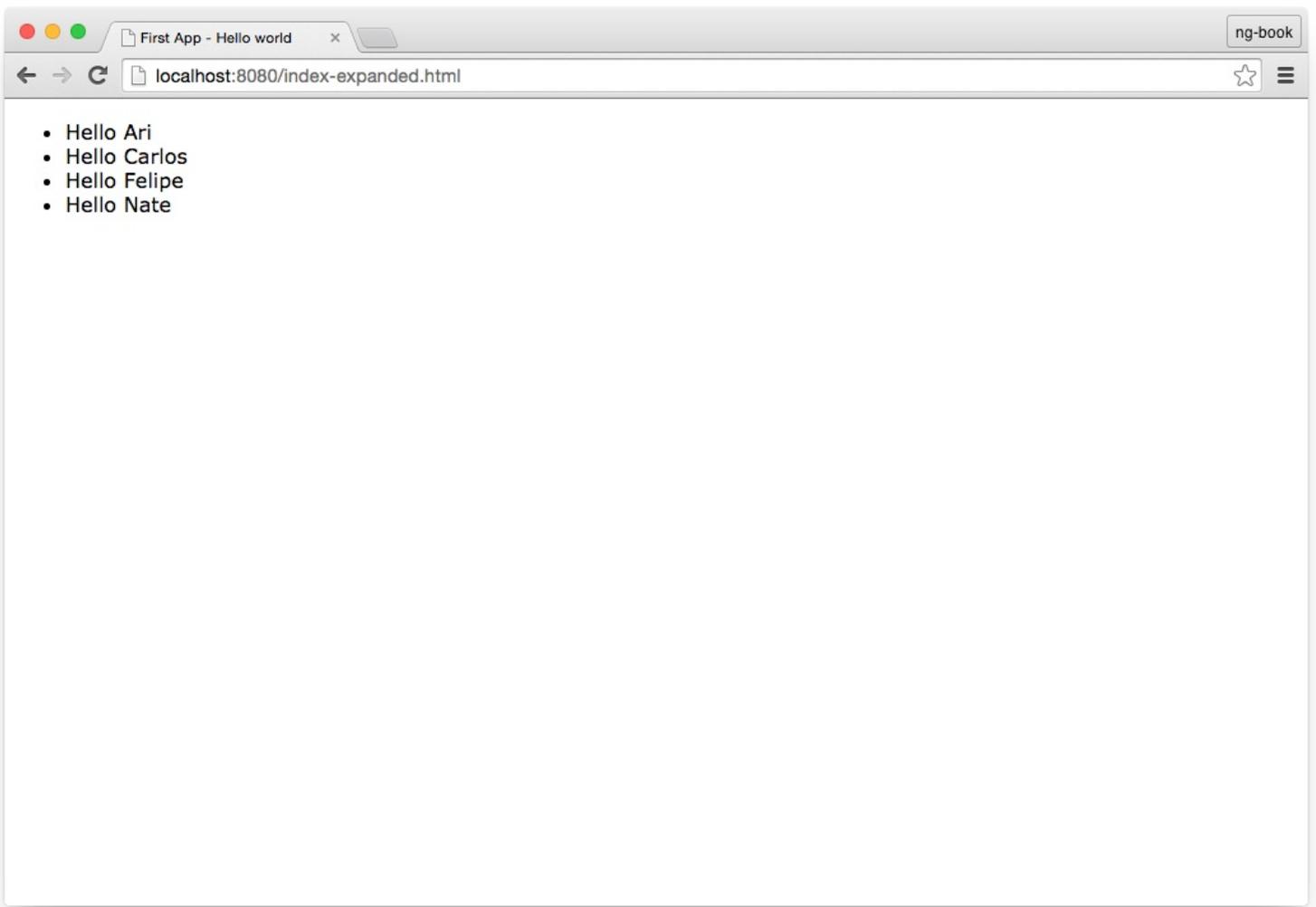


NgFor repeats the element that the ngFor is attached to. That is, we put it on the li tag and **not** the ul tag because we want to repeat the list element (li) and not the list itself (ul).



If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the [NgFor directive here](#)

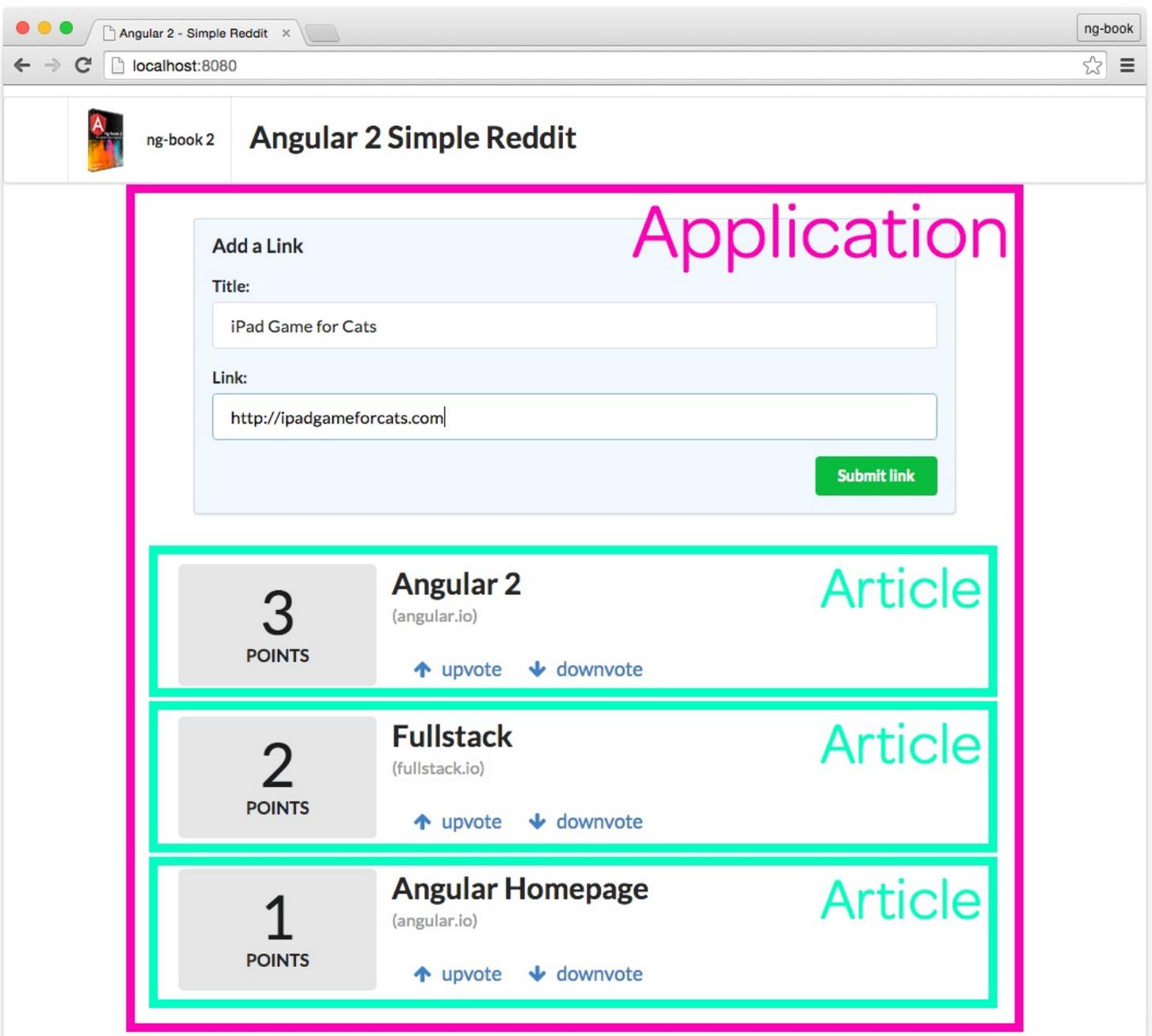
When you reload the page now, you can see that we have one li for each string on the array:



Application with Data

Expanding our Application

Now that we know how to create a basic component, let's revisit our Reddit clone. Before we start coding, it's a good idea to look over our app and break it down into its logical components.



Application with Data

We're going to make two components in this app:

1. The overall application, which contains form used to submit new articles (marked in magenta in the picture).
2. Each article (marked in mint green).



In a larger application, the form for submitting articles would probably become its own component. However, having the form be its own component makes the data passing more complex, so we're going to simplify in this chapter and only have two components.

For now, we'll just make two components, but we'll learn how to deal with more sophisticated data architectures in later chapters of this book.

The Application Component

Let's start building the top-level application component. This is the component that will 1. store our current list of articles and 2. contain the form for submitting new articles.

We're done with our HelloWorld component for now and instead we're going to build a component to represent our whole app: a RedditApp component.

To do that, we'll create a new component with a template:



For this example we're using the [Semantic UI](#) CSS Framework. In our template below when you see classes on the attributes - like `class="ui large form segment"` - these are styles coming from Semantic. It's a great way to have our app look nice without too much extra markup.

```

1 import { bootstrap } from '@angular/platform-browser-dynamic';
2 import { Component } from '@angular/core';
3
4 @Component({
5   selector: 'reddit',
6   template: `
7     <form class="ui large form segment">
8       <h3 class="ui header">Add a Link</h3>
9
10      <div class="field">
11        <label for="title">Title:</label>
12        <input name="title">
13      </div>
14      <div class="field">
15        <label for="link">Link:</label>
16        <input name="link">
17      </div>
18    </form>
19  `
20 })
21 class RedditApp {
22   constructor() {
23   }
24 }
25
26 bootstrap(RedditApp);

```

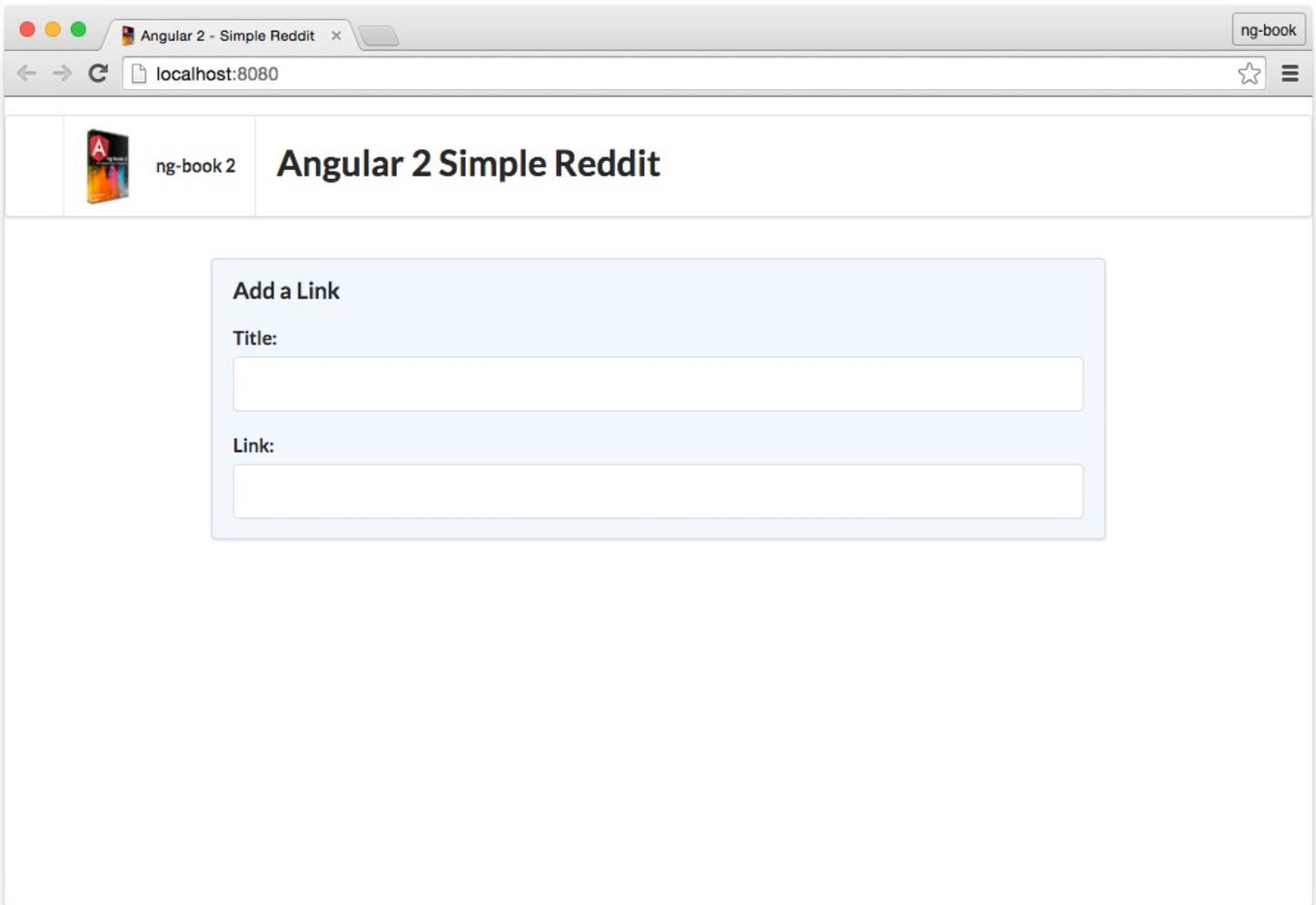
Here we declare a RedditApp component. Our selector is reddit which means we can place it on our page by using `<reddit></reddit>`.

We're creating a template that defines two input tags: one for the title of the article and the other for the link URL.

We can use our new `RedditApp` component by opening up the `index.html` and replacing the `<hello-world></hello-world>` tag with `<reddit></reddit>`.

 If you haven't already, now is a good time to replace your `index.html` with our provided `index-full.html`. Just rename `index-full.html` to `index.html` and you'll have the `<reddit>` tags along with a menu bar and all the dependencies.

When you reload the browser you should see the form rendered:



Form

 You'll notice this screenshot has nice styling and a header. This is because we're using the markup from `index-full.html` and the styles from Semantic UI. You can find the matching markup for this in your sample code.

Adding Interaction

Now we have the form with input tags but we don't have any way to submit the data. Let's add some interaction by adding a submit button to our form:

```
1 @Component({
2   selector: 'reddit',
3   template: `
4     <form class="ui large form segment">
5       <h3 class="ui header">Add a Link</h3>
6
7       <div class="field">
8         <label for="title">Title:</label>
9         <input name="title" #newtitle>
10      </div>
11      <div class="field">
12        <label for="link">Link:</label>
13        <input name="link" #newlink>
14      </div>
15
16      <button (click)="addArticle(newtitle, newlink)"
17              class="ui positive right floated button">
18        Submit link
19      </button>
20    </form>
21  `
22 })
23 class RedditApp {
24   constructor() {
25   }
26
27   addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
28     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
29   }
30 }
```

Notice we've made **four** changes:

1. Created a button tag in our markup that shows the user where to click
2. We created a function named `addArticle` that defines what we want to do when the button is clicked
3. We added a `(click)` attribute on the button that says "call the function `addArticle` when this button is pressed".
4. We added the attribute `#newtitle` and `#newlink` to the `<input>` tags

Let's cover each one of these steps in reverse order:

Binding inputs to values

Notice in our first input tag we have the following:

```
1 <input name="title" #newtitle>
```

This syntax is new. This markup tells Angular to *bind* this `<input>` to the variable `newtitle`. The `#newtitle` syntax is called a *resolve*. The effect is that this makes the variable `newtitle` available to the expressions within this view.

`newtitle` is now an **object** that represents this input DOM element (specifically, the type is `HTMLInputElement`). Because `newtitle` is an object, that means we get the value of the input tag using `newtitle.value`.

Similarly we add `#newlink` to the other `<input>` tag, so that we'll be able to extract the value from it as well.

Binding actions to events

On our button tag we add the attribute `(click)` to define what should happen when the button is clicked on. When the `(click)` event happens we call `addArticle` with two arguments: `newtitle` and `newlink`. Where did this function and two arguments come from?

1. `addArticle` is a function on our component definition class `RedditApp`
2. `newtitle` comes from the resolve `(#newtitle)` on our `<input>` tag named `title`
3. `newlink` comes from the resolve `(#newlink)` on our `<input>` tag named `link`

All together:

```
1 <button (click)="addArticle(newtitle, newlink)"
2     class="ui positive right floated button">
3   Submit link
4 </button>
```



The markup `class="ui positive right floated button"` comes from Semantic UI and it gives the button the pleasant green color.

Defining the Action Logic

On our class `RedditApp` we define a new function called `addArticle`. It takes two arguments: `title` and `link`. Again, it's important to realize that `title` and `link` are both **objects** of type `HTMLInputElement` and *not the input values directly*. To get the value from the input we have to call `title.value`. For now, we're just going to `console.log` out those arguments.

```
1 addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
2   console.log(`Adding article title: ${title.value} and link: ${link.value}`\
3 );
4 }
```

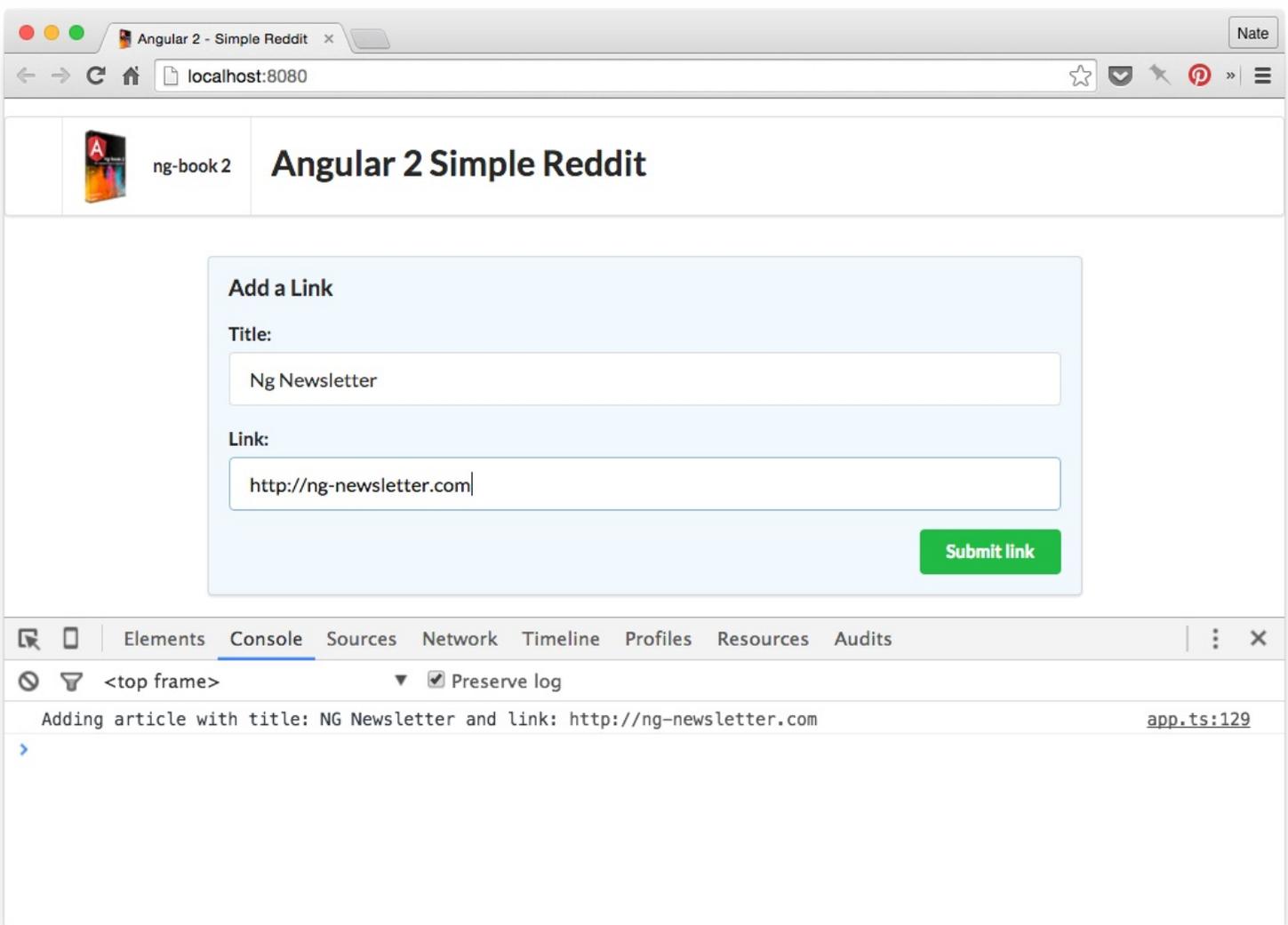


Notice that we're using backtick strings again. This is a really handy feature of ES6: backtick strings will expand template variables!

Here we're putting `${title.value}` in the string and this will be replaced with the value of `title.value` in the string.

Try it out!

Now when you click the submit button, you can see that the message is printed on the console:



Clicking the Button

Adding the Article Component

Now we have a form to submit new articles, but we aren't showing the new articles anywhere. Because every article submitted is going to be displayed as a list on the page, this is the perfect candidate for a new component.

Let's create a new component to represent the submitted articles.

For that, we can create a new component on the same file. Insert the following snippet **above** the declaration of the `RedditApp` component:

code/first_app/angular2-reddit-completed/intermediates/app-02.ts

```
4 @Component({
5   selector: 'reddit-article',
6   host: {
7     class: 'row'
8   },
9   template: `
10     <div class="four wide column center aligned votes">
11       <div class="ui statistic">
12         <div class="value">
13           {{ votes }}
14         </div>
15       <div class="label">
```

```

16     Points
17     </div>
18 </div>
19 </div>
20 <div class="twelve wide column">
21   <a class="ui large header" href="{{ link }}">
22     {{ title }}
23   </a>
24   <ul class="ui big horizontal list voters">
25     <li class="item">
26       <a href (click)="voteUp()">
27         <i class="arrow up icon"></i>
28         upvote
29       </a>
30     </li>
31     <li class="item">
32       <a href (click)="voteDown()">
33         <i class="arrow down icon"></i>
34         downvote
35       </a>
36     </li>
37   </ul>
38 </div>
39
40 })
41 class ArticleComponent {
42   votes: number;
43   title: string;
44   link: string;
45
46   constructor() {
47     this.title = 'Angular 2';
48     this.link = 'http://angular.io';
49     this.votes = 10;
50   }
51
52   voteUp() {
53     this.votes += 1;
54   }
55
56   voteDown() {
57     this.votes -= 1;
58   }
59 }

```

Notice that we have three parts to defining this new component:

1. Describing the Component properties by annotating the class with `@Component`
2. Describing the Component view in the `template` option
3. Creating a component-definition class (`ArticleComponent`) which houses our component logic

Let's talk through each part in detail:

Creating the `reddit-article` Component

`code/first_app/angular2-reddit-completed/intermediates/app-02.ts`

```

4 @Component({
5   selector: 'reddit-article',
6   host: {
7     class: 'row'
8   },

```

First, we define a new Component with `@Component`. The selector says that this component is placed on the page by using the tag `<reddit-article>` (i.e. the selector is a tag name).

So the most essential way to use this component would be to place the following tag in our markup:

```
1 <reddit-article>
2 </reddit-article>
```

These tags will remain in our view when the page is rendered.

We want each `reddit-article` to be on its own row. We're using Semantic UI, and Semantic provides a [CSS class for rows](#) called `row`.

In Angular 2, a component *host* is the element this component is attached to. You'll notice on our `@Component` we're passing the option: `host: { class: 'row' }`. This tells Angular that on the **host element** (the `reddit-article` tag) we want to set the `class` attribute to have "row".



Using the `host` option is nice because it means we can encapsulate the `reddit-article` markup within our component. That is, we don't have to both use a `reddit-article` tag **and** require a `class="row"` in the markup of the parent view. By using the `host` option, we're able to configure our host element from *within* the component.

Creating the `reddit-article` template

Second, we define the template with the `template` option.

`code/first_app/angular2-reddit-completed/intermediates/app-02.ts`

```
9  template: `
10  <div class="four wide column center aligned votes">
11    <div class="ui statistic">
12      <div class="value">
13        {{ votes }}
14      </div>
15      <div class="label">
16        Points
17      </div>
18    </div>
19  </div>
20  <div class="twelve wide column">
21    <a class="ui large header" href="{{ link }}">
22      {{ title }}
23    </a>
24    <ul class="ui big horizontal list voters">
25      <li class="item">
26        <a href (click)="voteUp()">
27          <i class="arrow up icon"></i>
28          upvote
29        </a>
30      </li>
31      <li class="item">
32        <a href (click)="voteDown()">
33          <i class="arrow down icon"></i>
34          downvote
35        </a>
36      </li>
37    </ul>
38  </div>
39  `
```

There's a lot of markup here, so let's break it down:

3
POINTS

Angular 2

(angular.io)

↑ upvote ↓ downvote

A Single reddit-article Row

We have two columns:

1. the number of votes on the left and
2. the article information on the right.

We specify these columns with the CSS classes `four wide column` and `twelve wide column` respectively.

We're showing votes and the title with the template expansion strings `{{ votes }}` and `{{ title }}`. The values come from the value of `votes` and `title` property of the `ArticleComponent` class.

Notice that we can use template strings in **attribute values**, as in the href of the a tag: `href="{{ link }}"`. In this case, the value of the href will be dynamically populated with the value of `link` from the component class

On our upvote/downvote links we have an action. We use `(click)` to bind `voteUp()`/`voteDown()` to their respective buttons. When the upvote button is pressed, the `voteUp()` function will be called on the `ArticleComponent` class (similarly with downvote and `voteDown()`).

Creating the `reddit-article` `ArticleComponent` Definition Class

Finally, we create the `ArticleComponent` definition class:

`code/first_app/angular2-reddit-completed/intermediates/app-02.ts`

```
41 class ArticleComponent {
42   votes: number;
43   title: string;
44   link: string;
45
46   constructor() {
47     this.title = 'Angular 2';
48     this.link = 'http://angular.io';
49     this.votes = 10;
50   }
51
52   voteUp() {
53     this.votes += 1;
54   }
55
56   voteDown() {
57     this.votes -= 1;
58   }
59 }
```

Here we create three properties on `ArticleComponent`:

1. `votes` - a number representing the sum of all upvotes, minus the downvotes
2. `title` - a string holding the title of the article
3. `link` - a string holding the URL of the article

In the constructor () we set some default attributes:

```
code/first_app/angular2-reddit-completed/intermediates/app-02.ts
```

```
46  constructor() {
47    this.title = 'Angular 2';
48    this.link = 'http://angular.io';
49    this.votes = 10;
50  }
```

And we define two functions for voting, one for voting up `voteUp` and one for voting down `voteDown`:

```
code/first_app/angular2-reddit-completed/intermediates/app-02.ts
```

```
52  voteUp() {
53    this.votes += 1;
54  }
55
56  voteDown() {
57    this.votes -= 1;
58  }
```

In `voteUp` we increment `this.votes` by one. Similarly we decrement for `voteDown`.

Using the `reddit-article` Component

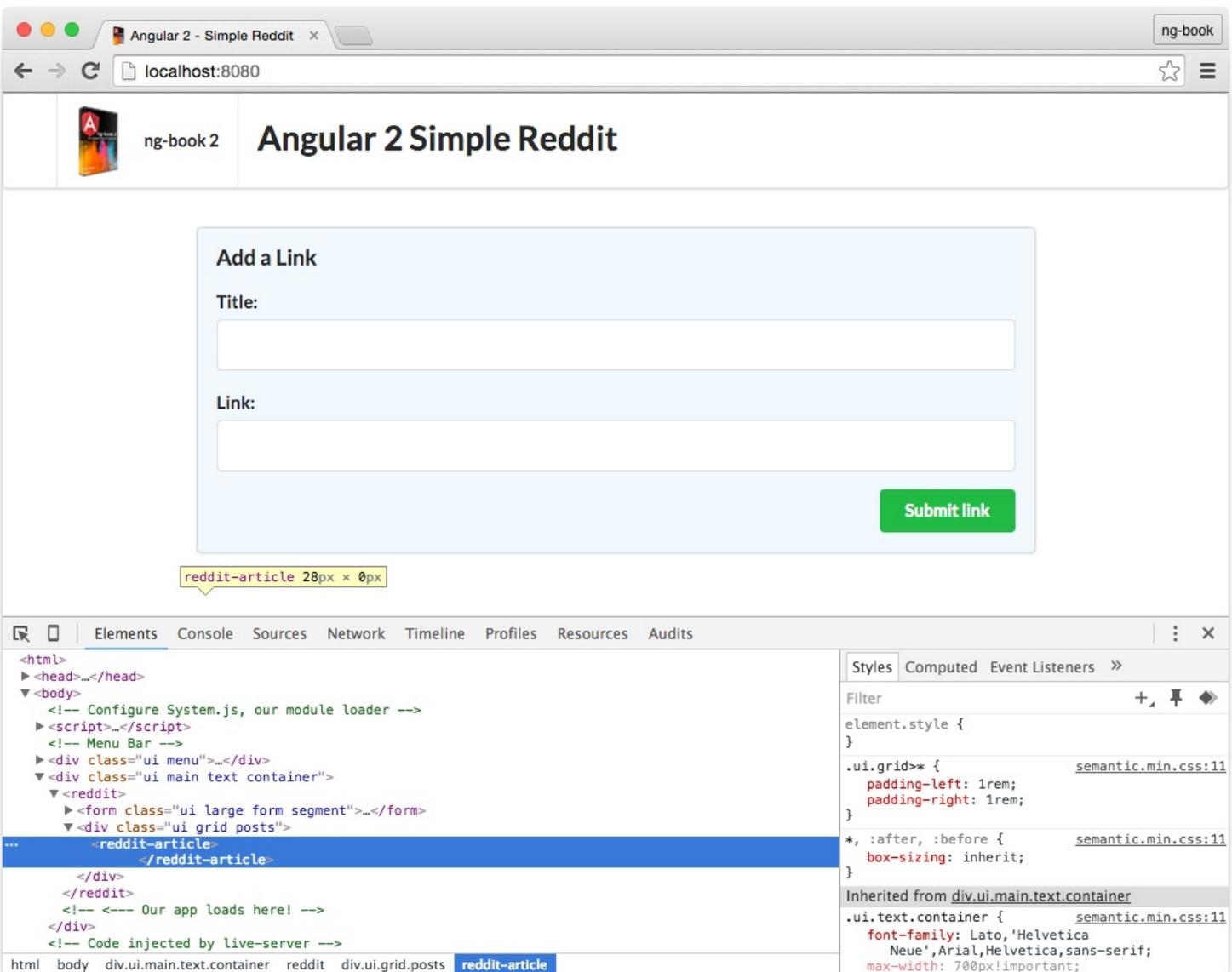
In order to use this component and make the data visible, we have to add a `<reddit-article>` `</reddit-article>` tag somewhere in our markup.

In this case, we want the `RedditApp` component to render this new component, so let's change the code in that component. Add the `<reddit-article>` tag to the `RedditApp`'s template right after the closing `</form>` tag:

```
1  <button (click)="addArticle(newtitle, newlink)"
2    class="ui positive right floated button">
3    Submit link
4  </button>
5 </form>
6
7 <div class="ui grid posts">
8   <reddit-article>
9   </reddit-article>
10 </div>
11 `
```

If we reload the browser now, you will see that the `<reddit-article>` tag wasn't compiled. Oh no!

Whenever hitting a problem like this, the first thing to do is open up your browser's developer console. If we inspect our markup (see screenshot below), we can see that the `reddit-article` tag is on our page, but it hasn't been compiled into markup. Why not?



Unexpanded tag when inspecting the DOM

This happens because the `RedditApp` component **doesn't know about the** `ArticleComponent` component yet.



Angular 1 Note: If you've used Angular 1 it might be surprising that our app doesn't know about our new `reddit-article` component. This is because in Angular 1, directives match globally. However, in Angular 2 you need explicitly specify which components (and therefore, which selectors) you want to use.

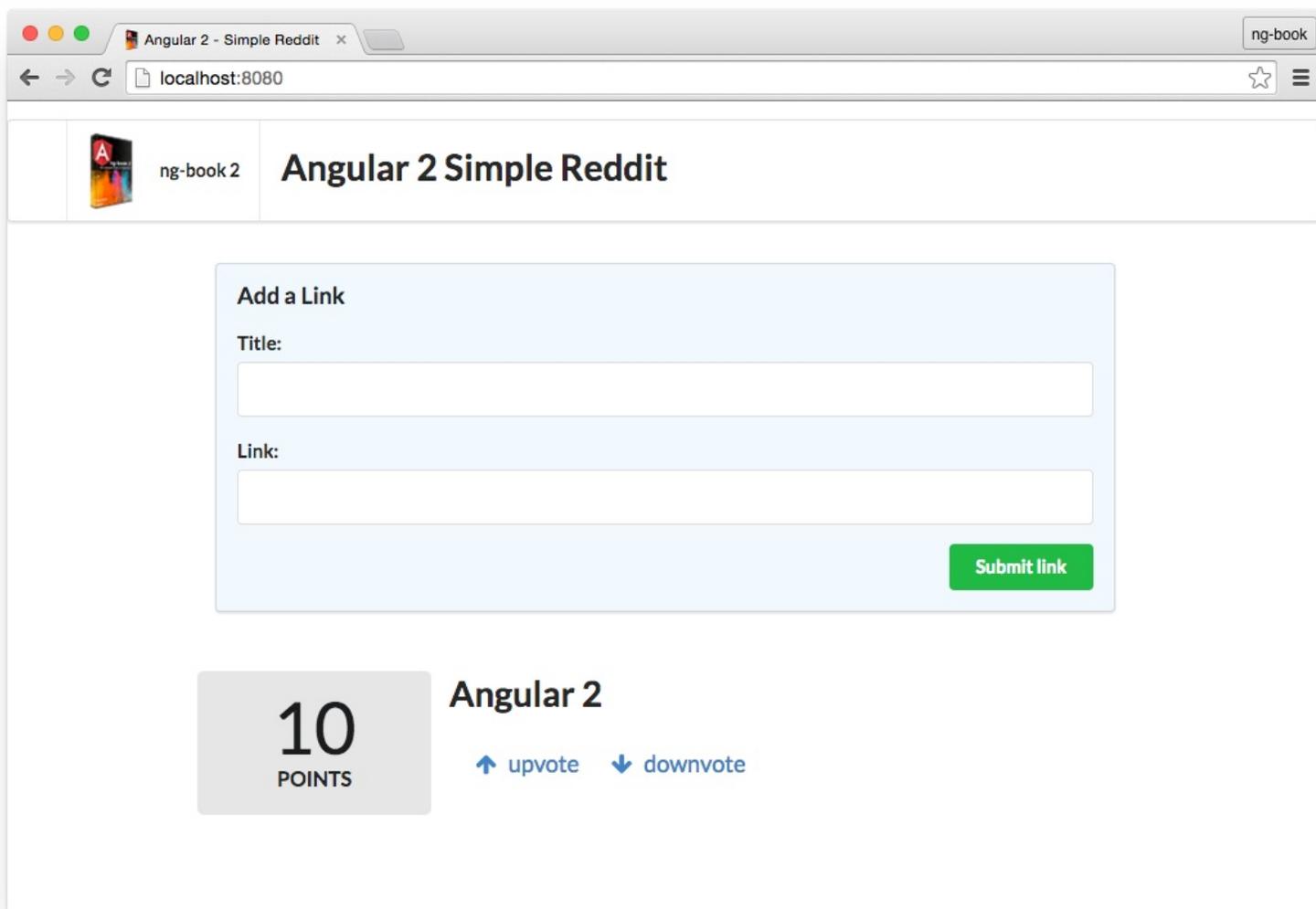
On the one hand, this requires a little more configuration. On the other hand, it's great for building scalable apps because it means you don't have to share your directive selectors in a global namespace.

In order to tell our `RedditApp` about our new `ArticleComponent` component, we need to add the `directives` property on our `RedditApp`:

```
1 // for RedditApp
2 @Component({
3   selector: 'reddit',
```

```
4 directives: [ArticleComponent],
5 template: `
6 // ...
```

And now, when we reload the browser we should see the article properly rendered:



Rendered ArticleComponent component

However, if you try to click the **vote up** or **vote down** links, you'll see that the page unexpectedly reloads.

This is because Javascript, by default, **propagates the click event to all the parent components**. Because the `click` event is propagated to parents, our browser is trying to follow the empty link.

To fix that, we need to make the click event handler to return `false`. This will ensure the browser won't try to refresh the page. We change our code like so:

```
1 voteUp(): boolean {
2   this.votes += 1;
3   return false;
4 }
5
6 voteDown(): boolean {
7   this.votes -= 1;
8   return false;
9 }
```

Now if you click the links, you'll see that the votes increase and decrease properly without a page refresh.

Rendering Multiple Rows

Right now we only have one article on the page and there's no way to render more, unless we paste another `<reddit-article>` tag. And even if we did that all the articles would have the same content, so it wouldn't be very interesting.

Creating an Article class

A good practice when writing Angular code is to try to isolate the data structures you are using from the component code. To do this, let's create a data structure that represents a single article. Add the following code before the `ArticleComponent` component code:

```
1 class Article {
2   title: string;
3   link: string;
4   votes: number;
5
6   constructor(title: string, link: string, votes?: number) {
7     this.title = title;
8     this.link = link;
9     this.votes = votes || 0;
10  }
11 }
```

Here we are creating a new class that represents an `Article`. Note that this is a **plain class and not a component**. In the Model-View-Controller pattern this would be the **Model**.

Each article has a title, a link, and a total for the votes. When creating a new article we need the title and the link. The votes parameter is optional and defaults to zero.

Now let's change the `ArticleComponent` code to use our new `Article` class. Instead of storing the properties directly on the `ArticleComponent` component let's **store the properties on an instance of the Article class**.

code/first_app/angular2-reddit-completed/intermediates/app-03.ts

```
53 class ArticleComponent {
54   article: Article;
55
56   constructor() {
57     this.article = new Article('Angular 2', 'http://angular.io', 10);
58   }
59
60   voteUp(): boolean {
61     this.article.votes += 1;
62     return false;
63   }
64
65   voteDown(): boolean {
66     this.article.votes -= 1;
67     return false;
68   }
69 }
```

Notice what we've changed: instead of storing the title, link, and votes properties directly on the component, instead we're storing a reference to an article. What's neat is that we've defined the type of article to be our new `Article` class.

When it comes to `voteUp` (and `voteDown`), we don't increment votes on the component, but rather, we need to increment the votes on the article.

However this refactoring introduces another change: we need to update our view to get the template variables from the right location. To do that, we need to change our template tags to read from `article`. That is, where before we had `{{ votes }}`, we need to change it to `{{ article.votes }}`:

code/first_app/angular2-reddit-completed/intermediates/app-03.ts

```
21 template: `
22   <div class="four wide column center aligned votes">
23     <div class="ui statistic">
24       <div class="value">
25         {{ article.votes }}
26       </div>
27       <div class="label">
28         Points
29       </div>
30     </div>
31   </div>
32   <div class="twelve wide column">
33     <a class="ui large header" href="{{ article.link }}">
34       {{ article.title }}
35     </a>
36     <ul class="ui big horizontal list voters">
37       <li class="item">
38         <a href (click)="voteUp()">
39           <i class="arrow up icon"></i>
40           upvote
41         </a>
42       </li>
43       <li class="item">
44         <a href (click)="voteDown()">
45           <i class="arrow down icon"></i>
46           downvote
47         </a>
48       </li>
49     </ul>
50   </div>
51 `
```

Reload the browser and everything should still work.

That's good, but something in our code is still off: our `voteUp` and `voteDown` methods break the encapsulation of the `Article` class by changing the article's internal properties directly.

 `voteUp` and `voteDown` current break the [Law of Demeter](#) which says that a given object should assume as little as possible about the structure or properties any other objects. One way to detect this is to be suspicious when you see long method/property chains like `foo.bar.baz.bam`. This pattern of long-method chaining is also affectionately referred to as a “train-wreck”.

The problem is that our `ArticleComponent` component knows too much about the `Article` class internals. To fix that, let's add `voteUp` and `voteDown` methods on the `Article` class.

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
4 class Article {
5   title: string;
6   link: string;
7   votes: number;
8
9   constructor(title: string, link: string, votes?: number) {
10    this.title = title;
11    this.link = link;
```

```
12     this.votes = votes || 0;
13   }
14
15   voteUp(): void {
16     this.votes += 1;
17   }
18
19   voteDown(): void {
20     this.votes -= 1;
21   }
```

Then we'll change ArticleComponent to call these methods:

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
72 class ArticleComponent {
73   article: Article;
74
75   voteUp(): boolean {
76     this.article.voteUp();
77     return false;
78   }
79
80   voteDown(): boolean {
81     this.article.voteDown();
82     return false;
83   }
84 }
```

 Checkout our ArticleComponent component definition now: it's so short! We've moved a lot of logic **out** of our component and into our models. The corresponding MVC guideline here might be [Fat Models, Skinny Controllers](#). The idea is that we want to move most of our domain logic to our models so that our components do the minimum work possible.

After reloading your browser, again, you'll notice everything works the same way, but we now have clearer code.

Storing multiple Articles

Let's write the code that allows us to have a list of multiple Articles.

Start by changing RedditApp to have a collection of articles:

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
116 class RedditApp {
117   articles: Article[];
118
119   constructor() {
120     this.articles = [
121       new Article('Angular 2', 'http://angular.io', 3),
122       new Article('Fullstack', 'http://fullstack.io', 2),
123       new Article('Angular Homepage', 'http://angular.io', 1),
124     ];
125   }
126
127   addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
128     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
```

Notice that our RedditApp has the line:

```
1   articles: Article[];
```

The `Article[]` might look a little unfamiliar. We're saying here that `articles` is an Array of `Articles`. Another way this could be written is `Array<Article>`. The word for this pattern is *generics*. It's a concept seen in Java, C#, and other languages. The idea is that your collection (the Array) is typed. That is, the Array is a collection that will only hold objects of type `Article`.

We can populate this Array by setting `this.articles` in the constructor:

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
119 constructor() {
120     this.articles = [
121         new Article('Angular 2', 'http://angular.io', 3),
122         new Article('Fullstack', 'http://fullstack.io', 2),
123         new Article('Angular Homepage', 'http://angular.io', 1),
124     ];
125 }
```

Configuring the ArticleComponent with inputs

Now that we have a list of `Article models`, how can we pass them to our `ArticleComponent component`?

Here we introduce a new attribute of `Component` called **inputs**. We can configure a `Component` with inputs that are passed to it from its parent.

Previously we had our `ArticleComponent` class defined like this:

```
1 class ArticleComponent {
2     article: Article;
3
4     constructor() {
5         this.article = new Article('Angular 2', 'http://angular.io');
6     }
7 }
```

The problem here is that we've hard coded a particular `Article` in the constructor. The point of making components is not only encapsulation, but also reusability.

What we would really like to do is to configure the `Article` we want to display. If, for instance, we had two articles, `article1` and `article2`, we would like to be able to reuse the `reddit-article` component by passing an `Article` as a “parameter” to the component like this:

```
1 <reddit-article [article]="article1"></reddit-article>
2 <reddit-article [article]="article2"></reddit-article>
```

Angular allows us to do this by using the `inputs` option of `Component`:

```
1 @Component({
2     selector: 'reddit-article',
3     inputs: ['article'],
4     // ... same
5 })
6 class ArticleComponent {
7     article: Article;
8     // ...
```

Now if we have an `Article` in a variable `myArticle` we could pass it to our `ArticleComponent` in our view like this:

```
1 <reddit-article [article]="myArticle"></reddit-article>
```

Notice the syntax here: we put the name of the input in brackets as in: [article] and the value of the attribute is what we want to pass in to that input.

Then, and this is important, the `this.article` on the `ArticleComponent` instance will be set to `myArticle`. You can think of it like `myArticle` is being passed as a *parameter* (i.e. input) to your component (via inputs).

Notice that `inputs` is an Array. This is because you can specify that a component has many inputs.

Here's what our full `reddit-article` component now looks like using inputs:

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
33 @Component({
34   selector: 'reddit-article',
35   inputs: ['article'],
36   host: {
37     class: 'row'
38   },
39   template: `
40     <div class="four wide column center aligned votes">
41       <div class="ui statistic">
42         <div class="value">
43           {{ article.votes }}
44         </div>
45         <div class="label">
46           Points
47         </div>
48       </div>
49     </div>
50     <div class="twelve wide column">
51       <a class="ui large header" href="{{ article.link }}">
52         {{ article.title }}
53       </a>
54
55       <ul class="ui big horizontal list voters">
56         <li class="item">
57           <a href (click)="voteUp()">
58             <i class="arrow up icon"></i>
59             upvote
60           </a>
61         </li>
62         <li class="item">
63           <a href (click)="voteDown()">
64             <i class="arrow down icon"></i>
65             downvote
66           </a>
67         </li>
68       </ul>
69     </div>
70   `
71 })
72 class ArticleComponent {
73   article: Article;
74
75   voteUp(): boolean {
76     this.article.voteUp();
77     return false;
78   }
79
80   voteDown(): boolean {
81     this.article.voteDown();
82     return false;
83   }
84 }
```

Rendering a List of Articles

Earlier we configured our `RedditApp` to store an array of articles. Now let's configure `RedditApp` to *render* all the articles. To do so, instead of having the `<reddit-article>` tag alone, we are going to use the `NgFor` directive to iterate over the list of articles and render a `reddit-article` for each one:

Add this in the template of the `RedditApp` `@Component`, just below the closing `<form>` tag:

```
1   Submit link
2   </button>
3 </form>
4
5 <!-- start adding here -->
6 <div class="ui grid posts">
7   <reddit-article
8     *ngFor="let article of articles"
9     [article]="article">
10  </reddit-article>
11 </div>
12 <!-- end adding here -->
```

Remember when we rendered a list of names as a bullet list using the `NgFor` directive earlier in the chapter? Well, that also works for rendering multiple components.

The `*ngFor="let article of articles"` syntax will iterate through the list of articles and creating the local variable `article` (for each item in the list).

To specify the `article` input on a component we use the `[inputName]="inputValue"` expression. In this case, we're saying that we want to set the `article` input to the value of the local variable `article` set by `ngFor`.



I realize we're using the variable `article` many times in that previous code snippet. It's (potentially) clearer if we rename the temporary variable created by `NgFor` to `foobar`:

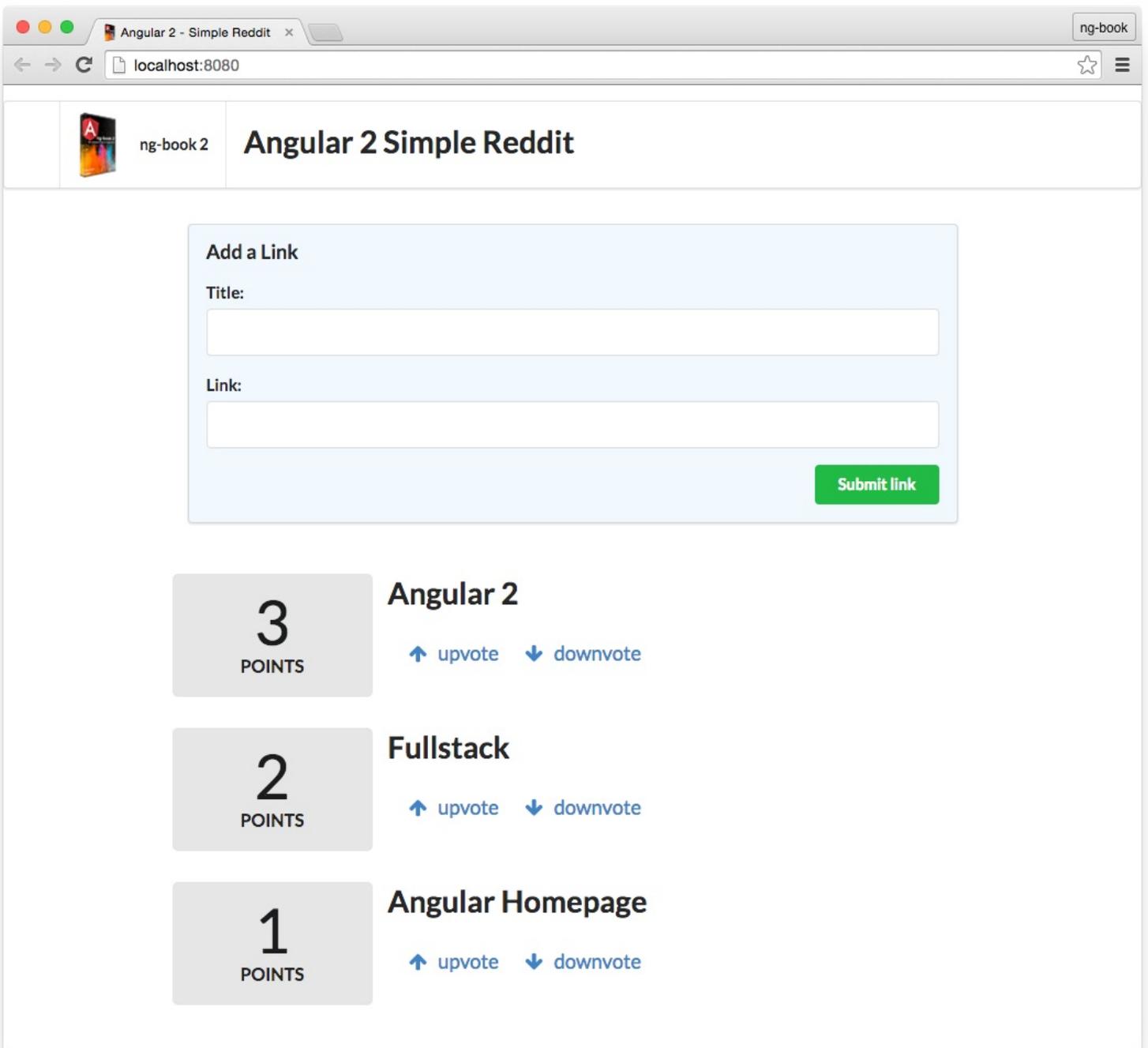
```
1 <reddit-article
2   *ngFor="let foobar of articles"
3   [article]="foobar">
4 </reddit-article>
```

So here we have three variables:

1. `articles` which is an `Array of Articles`, defined on the `RedditApp` component
2. `foobar` which is a single element of `articles` (an `Article`), defined by `NgFor`
3. `article` which is the name of the field defined on inputs of the `ArticleComponent`

Basically, `NgFor` generates a temporary variable `foobar` and then we're passing it in to `reddit-article`

If you reload your browser now, you can see that all articles will be rendered:



Multiple articles being rendered

Adding New Articles

Now we need to change `addArticle` to actually add new articles when the button is pressed. Change the `addArticle` method to match the following:

`code/first_app/angular2-reddit-completed/intermediates/app-04.ts`

```
127 addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
128     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
129     this.articles.push(new Article(title.value, link.value, 0));
130     title.value = '';
131     link.value = '';
132 }
```

This will:

1. create a new `Article` instance with the submitted title and URL
2. add it to the array of `Articles` and
3. clear the input field values



How are we clearing the input field values? Well, if you recall, `title` and `link` are `HTMLInputElement` *objects*. That means we can set their properties. When we change the `value` property, the input tag on our page changes.

If you add a new article and click **Submit Link** you will see the new article added!

Finishing Touches

Displaying the Article Domain

As a nice touch, let's add a hint next to the link that shows the domain where the user will be redirected to when the link is clicked.

Add this domain method to the `Article` class:

code/first_app/angular2-reddit-completed/intermediates/app-04.ts

```
23 domain(): string {
24   try {
25     const link: string = this.link.split('///')[1];
26     return link.split('/')[0];
27   } catch (err) {
28     return null;
29   }
30 }
```

And add it to the `ArticleComponent`'s template:

```
1 <div class="twelve wide column">
2   <a class="ui large header" href="{{ article.link }}">
3     {{ article.title }}
4   </a>
5   <!-- right here -->
6   <div class="meta">{{ article.domain() }}</div>
7   <ul class="ui big horizontal list voters">
8     <li class="item">
9       <a href (click)="voteUp()">
```

And now when we reload the browser, we should see the domain name of each URL.

Re-sorting Based on Score

If you click and vote you'll notice something doesn't feel quite right: our articles don't sort based on score! We definitely want to see the highest-rated items on top.

We're storing the articles in an `Array` in our `RedditApp` class, but that `Array` is unsorted. An easy way to handle this is to create a new method `sortedArticles` on `RedditApp`:

code/first_app/angular2-reddit-completed/app.ts

```
134 sortedArticles(): Article[] {
135   return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
136 }
```

Now in our ngFor we can iterate over sortedArticles() (instead of articles directly):

code/first_app/angular2-reddit-completed/app.ts

```
108 <div class="ui grid posts">
109   <reddit-article
110     *ngFor="let article of sortedArticles()"
111     [article]="article">
112   </reddit-article>
113 </div>
```

Full Code Listing

We've been zooming in to lots of small pieces of code for this chapter. Here's a full listing of the TypeScript code for our app (you can find the whole thing in the sample code download):

code/first_app/angular2-reddit-completed/app.ts

```
1 import { bootstrap } from '@angular/platform-browser-dynamic';
2 import { Component } from '@angular/core';
3
4 class Article {
5   title: string;
6   link: string;
7   votes: number;
8
9   constructor(title: string, link: string, votes?: number) {
10    this.title = title;
11    this.link = link;
12    this.votes = votes || 0;
13  }
14
15  domain(): string {
16    try {
17      const link: string = this.link.split('///')[1];
18      return link.split('/')[0];
19    } catch (err) {
20      return null;
21    }
22  }
23
24  voteUp(): void {
25    this.votes += 1;
26  }
27
28  voteDown(): void {
29    this.votes -= 1;
30  }
31 }
32
33 @Component({
34   selector: 'reddit-article',
35   inputs: ['article'],
36   host: {
37     class: 'row'
38   },
39   template: `
40     <div class="four wide column center aligned votes">
41       <div class="ui statistic">
42         <div class="value">
43           {{ article.votes }}
44         </div>
45         <div class="label">
46           Points
47         </div>
48       </div>
49     </div>
50     <div class="twelve wide column">
51       <a class="ui large header" href="{{ article.link }}">
52         {{ article.title }}
53       </a>
54       <div class="meta">{{ article.domain() }}</div>
55       <ul class="ui big horizontal list voters">
56         <li class="item">
57           <a href (click)="voteUp()">
```

```

58     <i class="arrow up icon"></i>
59     upvote
60     </a>
61 </li>
62 <li class="item">
63     <a href (click)="voteDown()">
64         <i class="arrow down icon"></i>
65         downvote
66     </a>
67 </li>
68 </ul>
69 </div>
70
71 })
72 class ArticleComponent {
73     article: Article;
74
75     voteUp(): boolean {
76         this.article.voteUp();
77         return false;
78     }
79
80     voteDown(): boolean {
81         this.article.voteDown();
82         return false;
83     }
84 }
85
86 @Component({
87     selector: 'reddit',
88     directives: [ArticleComponent],
89     template: `
90         <form class="ui large form segment">
91             <h3 class="ui header">Add a Link</h3>
92
93             <div class="field">
94                 <label for="title">Title:</label>
95                 <input name="title" #newtitle>
96             </div>
97             <div class="field">
98                 <label for="link">Link:</label>
99                 <input name="link" #newlink>
100            </div>
101
102            <button (click)="addArticle(newtitle, newlink)"
103                class="ui positive right floated button">
104                Submit link
105            </button>
106        </form>
107
108        <div class="ui grid posts">
109            <reddit-article
110                *ngFor="let article of sortedArticles()"
111                [article]="article">
112            </reddit-article>
113        </div>
114    `
115 })
116 class RedditApp {
117     articles: Article[];
118
119     constructor() {
120         this.articles = [
121             new Article('Angular 2', 'http://angular.io', 3),
122             new Article('Fullstack', 'http://fullstack.io', 2),
123             new Article('Angular Homepage', 'http://angular.io', 1),
124         ];
125     }
126
127     addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
128         console.log(`Adding article title: ${title.value} and link: ${link.value}`);
129         this.articles.push(new Article(title.value, link.value, 0));
130         title.value = '';
131         link.value = '';
132     }
133
134     sortedArticles(): Article[] {

```

```
135     return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
136   }
137
138 }
139
140 bootstrap(RedditApp);
```

Wrapping Up

We did it! We've created our first Angular 2 App. That wasn't so bad, was it? There's lots more to learn: understanding data flow, making AJAX requests, built-in components, routing, manipulating the DOM etc.

But for now, bask in your success! Much of writing Angular 2 apps is just as we did above:

1. Split your app into components
2. Create the views
3. Define your models
4. Display your models
5. Add interaction

Getting Help

Did you have any trouble with this chapter? Did you find a bug or have trouble getting the code running? We'd love to hear from you!

- Come join our (free!) community and [chat with us on Gitter](#)
- Email us directly at us@fullstack.io

Onward!

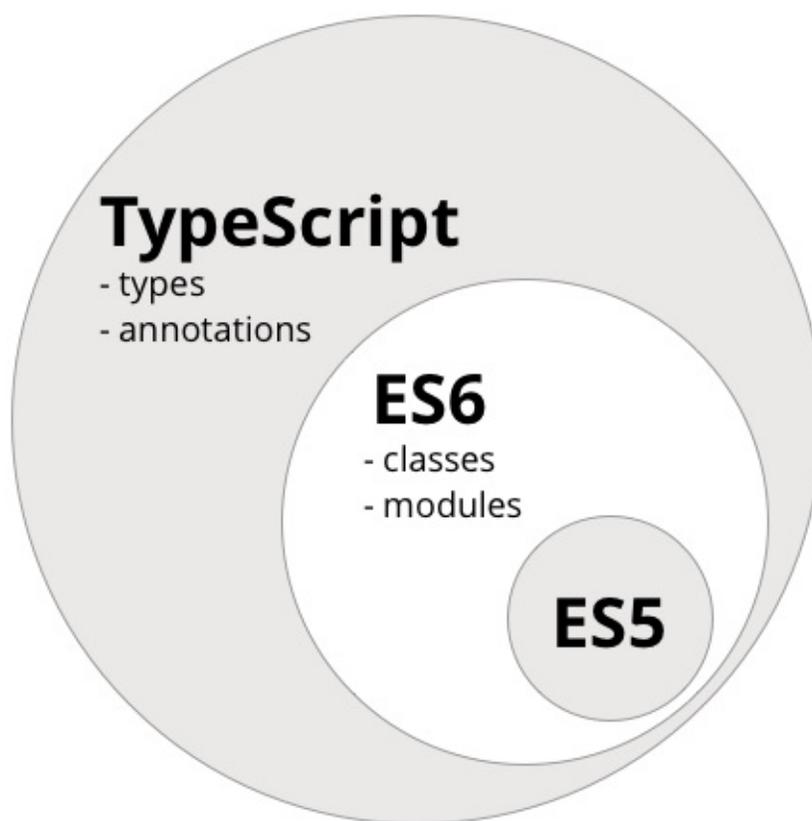
TypeScript

Angular 2 is built in TypeScript

Angular 2 is built in a Javascript-like language called [TypeScript](#).

You might be skeptical of using a new language just for Angular, but it turns out, there are a lot of great reasons to use TypeScript instead of plain Javascript.

TypeScript isn't a completely new language, it's a superset of ES6. If we write ES6 code, it's perfectly valid and compilable TypeScript code. Here's a diagram that shows the relationship between the languages:



ES5, ES6, and TypeScript



What is ES5? What is ES6? ES5 is short for “ECMAScript 5”, otherwise known as “regular Javascript”. ES5 is the normal Javascript we all know and love. It runs in more-or-less every browser. ES6 is the next version of Javascript, which we talk more about below.

At the publishing of this book, very few browsers will run ES6 out of the box, much less TypeScript. To solve this issue we have *transpilers* (or sometimes called *transcompiler*). The TypeScript transpiler takes our TypeScript code as input and outputs ES5 code that nearly all browsers understand.



For converting TypeScript to ES5 there is a single transpiler written by the core TypeScript team. However if we wanted to convert *ES6* code (not TypeScript) to *ES5* there are two major ES6-to-ES5 transpilers: [traceur](#) by Google and [babel](#) created by the JavaScript community. We're not going to be using either directly for this book, but they're both great projects that are worth knowing about.

We installed TypeScript in the last chapter, but in case you're just starting out in this chapter, you can install it like so:

```
npm install -g typescript
```

TypeScript is an official collaboration between Microsoft and Google. That's great news because with two tech heavyweights behind it we know that it will be supported for a long time. Both groups are committed to moving the web forward and as developers we win because of it.

One of the great things about transpilers is that they allow relatively small teams to make improvements to a language without requiring everyone on the internet upgrade their browser.

One thing to point out: we don't *have* to use TypeScript with Angular2. If you want to use ES5 (i.e. "regular" JavaScript), you definitely can. There is an ES5 API that provides access to all functionality of Angular2. Then why should we use TypeScript at all? Because there are some great features in TypeScript that make development a lot better.

What do we get with TypeScript?

There are five big improvements that TypeScript bring over ES5:

- types
- classes
- annotations
- imports
- language utilities (e.g. destructuring)

Let's deal with these one at a time.

Types

The major improvement of TypeScript over ES6, that gives the language its name, is the typing system.

For some people the lack of type checking is considered one of the benefits of using a language like JavaScript. You might be a little skeptical of type checking but I'd encourage you to give it a chance. One of the great things about type checking is that

1. it helps when *writing* code because it can prevent bugs at compile time and
2. it helps when *reading* code because it clarifies your intentions

It's also worth noting that types are optional in TypeScript. If we want to write some quick code or prototype a feature, we can omit types and gradually add them as the code becomes more mature.

TypeScript's basic types are the same ones we've been using implicitly when we write "normal" JavaScript code: strings, numbers, booleans, etc.

Up until ES5, we would define variables with the `var` keyword, like `var name;`

The new TypeScript syntax is a natural evolution from ES5, we still use `var` but now we can optionally provide the variable type along with its name:

```
1 var name: string;
```

When declaring functions we can use types for arguments and return values:

```
1 function greetText(name: string): string {
2   return "Hello " + name;
3 }
```

In the example above we are defining a new function called `greetText` which takes one argument: `name`. The syntax `name: string` says that this function expects `name` to be a `string`. Our code won't compile if we call this function with anything other than a `string` and that's a good thing because otherwise we'd introduce a bug.

Notice that the `greetText` function also has a new syntax after the parentheses: `: string {`. The colon indicates that we will specify the return type for this function, which in this case is a `string`. This is helpful because 1. if we accidentally return anything other than a `string` in our code, the compiler will tell us that we made a mistake and 2. any other developers who want to use this function know precisely what type of object they'll be getting.

Let's see what happens if we try to write code that doesn't conform to our declared typing:

```
1 function hello(name: string): string {
2   return 12;
3 }
```

If we try to compile it, we'll see the following error:

```
1 $ tsc compile-error.ts
2 compile-error.ts(2,12): error TS2322: Type 'number' is not assignable to type 'string'.
3 
```

What happened here? We tried to return `12` which is a number, but we stated that `hello` would return a `string` (by putting the `): string {` after the argument declaration).

In order to correct this, we need to update the function declaration to return a number:

```
1 function hello(name: string): number {
2   return 12;
3 }
```

This is one small example, but already we can see that by using types it can save us from a lot of bugs down the road.

So now that we know how to use types, how can we know what types are available to use? Let's look at the list of built-in types, and then we'll figure out how to create our own.

Trying it out with a REPL

To play with the examples on this chapter, let's install a nice little utility called [TSUN](#) (TypeScript Upgraded Node):

```
1 $ npm install -g tsun
```

Now start tsun:

```
1 $ tsun
2 TSUN : TypeScript Upgraded Node
3 type in TypeScript expression to evaluate
4 type :help for commands in repl
5
6 >
```

That little > is the prompt indicating that TSUN is ready to take in commands.

In most of the examples below, you can copy and paste into this terminal and play long.

Built-in types

String

A string holds text and is declared using the string type:

```
1 var name: string = 'Felipe';
```

Number

A number is any type of numeric value. In TypeScript, all numbers are represented as floating point. The type for numbers is number:

```
1 var age: number = 36;
```

Boolean

The boolean holds either true or false as the value.

```
1 var married: boolean = true;
```

Array

Arrays are declared with the Array type. However, because an Array is a collection, we also need to specify the type of the objects *in* the Array.

We specify the type of the items in the array with either the Array<type> or type[] notations:

```
1 var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
2 var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

Or similarly with a number:

```
1 var jobs: Array<number> = [1, 2, 3];
2 var jobs: number[] = [4, 5, 6];
```

Enums

Enums work by naming numeric values. For instance, if we wanted to have a fixed list of roles a person may have we could write this:

```
1 enum Role {Employee, Manager, Admin};
2 var role: Role = Role.Employee;
```

The default initial value for an enum is 0. You can tweak either the start of the range:

```
1 enum Role {Employee = 3, Manager, Admin};
2 var role: Role = Role.Employee;
```

In the code above, instead of Employee being 0, Employee is 3. The value of the enum increments from there, which means Manager is 4 and Admin is 5, and we can even set individual values:

```
1 enum Role {Employee = 3, Manager = 5, Admin = 7};
2 var role: Role = Role.Employee;
```

You can also look up the name of a given enum, but using its value:

```
1 enum Role {Employee, Manager, Admin};
2 console.log('Roles: ', Role[0], ', ', Role[1], 'and', Role[2]);
```

Any

any is the default type if we omit typing for a given variable. Having a variable of type any allows it to receive any kind of value:

```
1 var something: any = 'as string';
2 something = 1;
3 something = [1, 2, 3];
```

Void

Using void means there's no type expected. This is usually in functions with no return value:

```
1 function setName(name: string): void {
2   this.name = name;
3 }
```

Classes

In Javascript ES5 object oriented programming was accomplished by using prototype-based objects. This model doesn't use classes, but instead relies on *prototypes*.

A number of good practices have been adopted by the JavaScript community to compensate the lack of classes. A good summary of those good practices can be found in [Mozilla Developer Network's JavaScript Guide](#), and you can find a good overview on the [Introduction to Object-Oriented Javascript](#) page.

However, in ES6 we finally have built-in classes in Javascript.

To define a class we use the new `class` keyword and give our class a name and a body:

```
1 class Vehicle {
2 }
```

Classes may have *properties*, *methods*, and *constructors*.

Properties

Properties define data attached to an instance of a class. For example, a class named `Person` might have properties like `first_name`, `last_name` and `age`.

Each property in a class can optionally have a type. For example, we could say that the `first_name` and `last_name` properties are strings and the `age` property is a number.

The result declaration for a `Person` class that looks like this:

```
1 class Person {
2   first_name: string;
3   last_name: string;
4   age: number;
5 }
```

Methods

Methods are functions that run in context of an object. To call a method on an object, we first have to have an instance of that object.

 To instantiate a class, we use the `new` keyword. Use `new Person()` to create a new instance of the `Person` class, for example.

If we wanted to add a way to greet a `Person` using the class above, we would write something like:

```
1 class Person {
2   first_name: string;
3   last_name: string;
4   age: number;
5
6   greet() {
7     console.log("Hello", this.first_name);
8   }
9 }
```

Notice that we're able to access the `first_name` for this `Person` by using the `this` keyword and calling `this.first_name`.

When methods don't declare an explicit returning type and return a value, it's assumed they can return anything (any type). However, in this case we are returning `void`, since there's no explicit return statement.

 Note that a `void` value is also a valid any value.

In order to invoke the `greet` method, you would need to first have an instance of the `Person` class. Here's how we do that:

```
1 // declare a variable of type Person
2 var p: Person;
3
```

```

4 // instantiate a new Person instance
5 p = new Person();
6
7 // give it a first_name
8 p.first_name = 'Felipe';
9
10 // call the greet method
11 p.greet();

```



You can declare a variable and instantiate a class on the same line if you want:

```
1 var p: Person = new Person();
```

Say we want to have a method on the Person class that returns a value. For instance, to know the age of a Person in a number of years from now, we could write:

```

1 class Person {
2     first_name: string;
3     last_name: string;
4     age: number;
5
6     greet() {
7         console.log("Hello", this.first_name);
8     }
9
10    ageInYears(years: number): number {
11        return this.age + years;
12    }
13 }

```

```

1 // instantiate a new Person instance
2 var p: Person = new Person();
3
4 // set initial age
5 p.age = 6;
6
7 // how old will he be in 12 years?
8 p.ageInYears(12);
9
10 // -> 18

```

Constructors

A *constructor* is a special method that is executed when a new instance of the class is being created. Usually, the constructor is where you perform any initial setup for new objects.

Constructor methods must be named `constructor`. They can optionally take parameters but they can't return any values, since they are called when the class is being instantiated (i.e. an instance of the class is being created, no other value can be returned).



In order to instantiate a class we call the class constructor method by using the class name: `new ClassName()`.

When a class has no constructor defined explicitly one will be created automatically:

```

1 class Vehicle {
2 }

```

```
3 var v = new Vehicle();
```

Is the same as:

```
1 class Vehicle {
2   constructor() {
3   }
4 }
5 var v = new Vehicle();
```

 In TypeScript you can have only **one constructor per class**.

That is a departure from ES6 which allows one class to have more than one constructor as long as they have a different number of parameters.

Constructors can take parameters when we want to parameterize our new instance creation.

For example, we can change Person to have a constructor that initializes our data:

```
1 class Person {
2   first_name: string;
3   last_name: string;
4   age: number;
5
6   constructor(first_name: string, last_name: string, age: number) {
7     this.first_name = first_name;
8     this.last_name = last_name;
9     this.age = age;
10  }
11
12  greet() {
13    console.log("Hello", this.first_name);
14  }
15
16  ageInYears(years: number): number {
17    return this.age + years;
18  }
19 }
```

It makes our previous example a little easier to write:

```
1 var p: Person = new Person('Felipe', 'Coury', 36);
2 p.greet();
```

This way the person's names and age are set for us when the object is created.

Inheritance

Another important aspect of object oriented programming is inheritance. Inheritance is a way to indicate that a class receives behavior from a parent class. Then we can override, modify or augment those behaviors on the new class.

 If you want to have a deeper understanding of how inheritance used to work in ES5, take a look at the Mozilla Developer Network article about it: [Inheritance and the prototype chain](#).

TypeScript fully supports inheritance and, unlike ES5, it's built into the core language. Inheritance is achieved through the `extends` keyword.

To illustrate, let's say we've created a `Report` class:

```
1  class Report {
2    data: Array<string>;
3
4    constructor(data: Array<string>) {
5      this.data = data;
6    }
7
8    run() {
9      this.data.forEach(function(line) { console.log(line); });
10   }
11 }
```

This report has a property `data` which is an `Array` of strings. When we call `run` we loop over each element of `data` and print them out using `console.log`

 `.forEach` is a method on `Array` that accepts a function as an argument and calls that function for each element in the `Array`.

This `Report` works by adding lines and then calling `run` to print out the lines:

```
1 var r: Report = new Report(['First line', 'Second line']);
2 r.run();
```

Running this should show:

```
1 First line
2 Second line
```

Now let's say we want to have a second report that takes some headers and some data but we still want to reuse how the `Report` class presents the data to the user.

To reuse that behavior from the `Report` class we can use inheritance with the `extends` keyword:

```
1  class TabbedReport extends Report {
2    headers: Array<string>;
3
4    constructor(headers: string[], values: string[]) {
5      this.headers = headers;
6      super(values)
7    }
8
9    run() {
10     console.log(headers);
11     super.run();
12   }
13 }

```

```
1 var headers: string[] = ['Name'];
2 var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
3 var r: TabbedReport = new TabbedReport(headers, data)
4 r.run();
```

Utilities

ES6, and by extension TypeScript provides a number of syntax features that make programming really enjoyable. Two important ones are:

- fat arrow function syntax
- template strings

Fat Arrow Functions

Fat arrow => functions are a shorthand notation for writing functions.

In ES5, whenever we want to use a function as an argument we have to use the `function` keyword along with `{}` braces like so:

```
1 // ES5-like example
2 var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
3 data.forEach(function(line) { console.log(line); });
```

However with the => syntax we can instead rewrite it like so:

```
1 // Typescript example
2 var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
3 data.forEach( (line) => console.log(line) );
```

The => syntax can be used both as an expression:

```
1 var evens = [2,4,6,8];
2 var odds = evens.map(v => v + 1);
```

Or as a statement:

```
1 data.forEach( line => {
2   console.log(line.toUpperCase())
3 });
```

One important feature of the => syntax is that it shares the same `this` as the surrounding code. This is **important** and different than what happens when you normally create a function in Javascript. Generally when you write a function in Javascript that function is given its own `this`. Sometimes in Javascript we see code like this:

```
1 var nate = {
2   name: "Nate",
3   guitars: ["Gibson", "Martin", "Taylor"],
4   printGuitars: function() {
5     var self = this;
6     this.guitars.forEach(function(g) {
7       // this.name is undefined so we have to use self.name
8       console.log(self.name + " plays a " + g);
9     });
10  }
11 };
```

Because the fat arrow shares `this` with its surrounding code, we can instead write this:

```
1 var nate = {
2   name: "Nate",
3   guitars: ["Gibson", "Martin", "Taylor"],
4   printGuitars: function() {
5     this.guitars.forEach( (g) => {
6       console.log(this.name + " plays a " + g);
7     });
8   }
9 };
```

```
8 }  
9 };
```

Arrows are a great way to cleanup your inline functions. It makes it even easier to use higher-order functions in Javascript.

Template Strings

In ES6 new template strings were introduced. The two great features of template strings are

1. Variables within strings (without being forced to concatenate with +) and
2. Multi-line strings

Variables in strings

This feature is also called “string interpolation.” The idea is that you can put variables right in your strings. Here’s how:

```
1 var firstName = "Nate";  
2 var lastName = "Murray";  
3  
4 // interpolate a string  
5 var greeting = `Hello ${firstName} ${lastName}`;  
6  
7 console.log(greeting);
```

Note that to use string interpolation you must enclose your string in **backticks** not single or double quotes.

Multiline strings

Another **great** feature of backtick strings is multi-line strings:

```
1 var template = `  
2 <div>  
3   <h1>Hello</h1>  
4   <p>This is a great website</p>  
5 </div>  
6 `;  
7  
8 // do something with `template`
```

Multiline strings are a huge help when we want to put strings in our code that are a little long, like templates.

Wrapping up

There are a variety of other features in TypeScript/ES6 such as:

- Interfaces
- Generics
- Importing and Exporting Modules
- Annotations
- Destructuring

We’ll be touching on these concepts as we use them throughout the book, but for now these basics should get you started.

Let's get back to Angular!

How Angular Works

In this chapter, we're going to talk about the high-level concepts of Angular 2. We're going to take a step back so that we can see how all the pieces fit together.



If you've used Angular 1, you'll notice that Angular 2 has a new mental-model for building applications. Don't panic! As Angular 1 users we've found Angular 2 to be both straightforward and familiar. A little later in this book we're going to talk specifically about how to convert your Angular 1 apps to Angular 2.

In the chapters that follow, we'll be taking a deep dive into each concept, but here we're just going to give an overview and explain the foundational ideas.

The first big idea is that an Angular 2 application is made up of *Components*. One way to think of Components is a way to teach the browser new tags. If you have an Angular 1 background, Components are analogous to *directives* in Angular 1 (it turns out, Angular 2 has directives too, but we'll talk more about this distinction later on).

However, Angular 2 Components have some significant advantages over Angular 1 directives and we'll talk about that below. First, let's start at the top: the Application.

Application

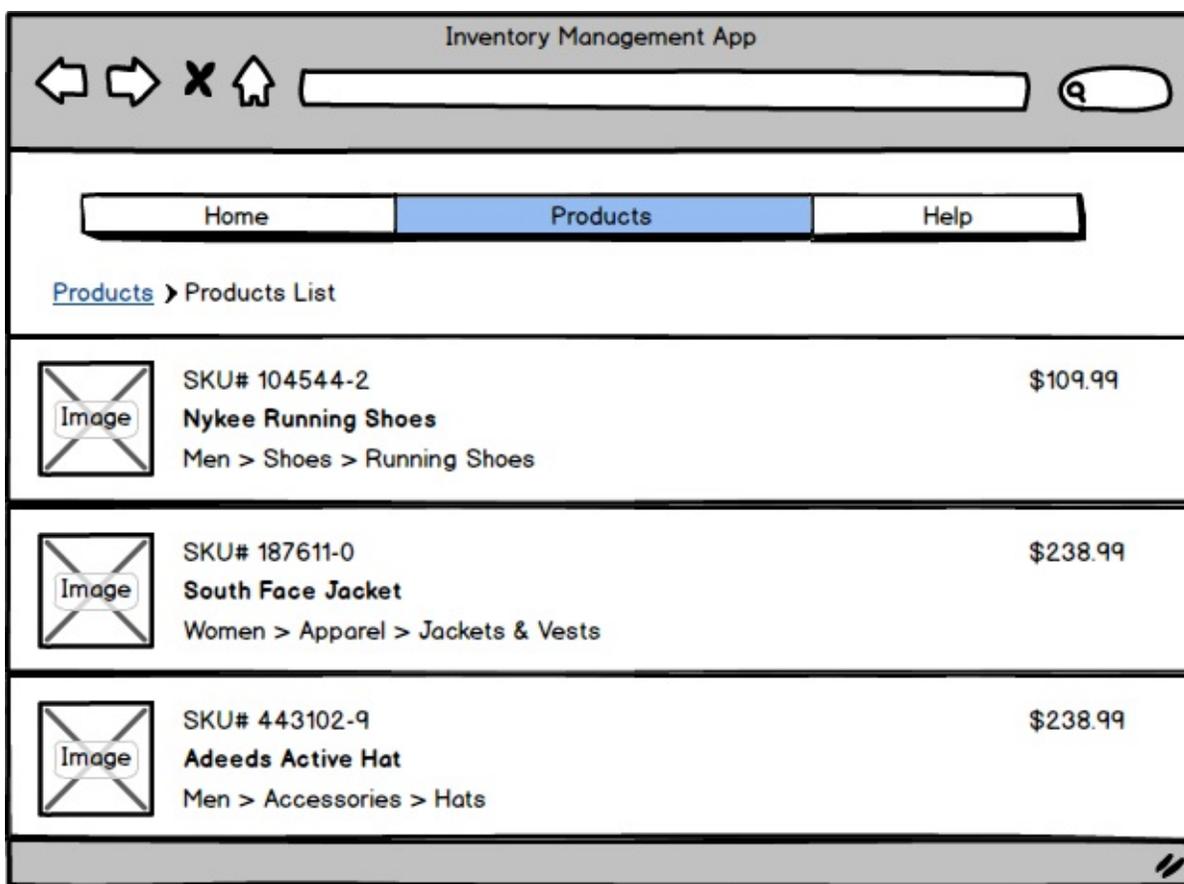
An Angular 2 Application is nothing more than a tree of Components.

At the root of that tree, the top level Component is the application itself. And that's what the browser will render when "booting" (a.k.a *bootstrapping*) the app.

One of the great things about Components is that they're **composable**. This means that we can build up larger Components from smaller ones. The Application is simply a Component that renders other Components.

Because Components are structured in a parent/child tree, when each Component renders, it recursively renders its children Components.

For example, let's create a simple inventory management application that is represented by the following page mockup:



Inventory Management App

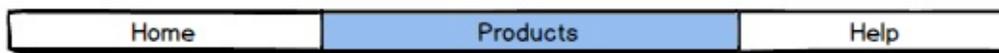
Given this mockup, to write this application the first thing we want to do is split it into components.

In this example, we could group the page into three high level components

1. The Navigation Component
2. The Breadcrumbs Component
3. The Product Info Component

The Navigation Component

This component would render the navigation section. This would allow the user to visit other areas of the application.



Navigation Component

The Breadcrumbs Component

This would render a hierarchical representation of where in the application the user currently is.



Breadcrumbs Component

The Product List Component

The Products List component would be a representation of collection of products.

	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 South Face Jacket Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 Adeeds Active Hat Men > Accessories > Hats	\$238.99

Product List Component

Breaking this component down into the next level of smaller components, we could say that the Product List is composed of multiple Product Rows.

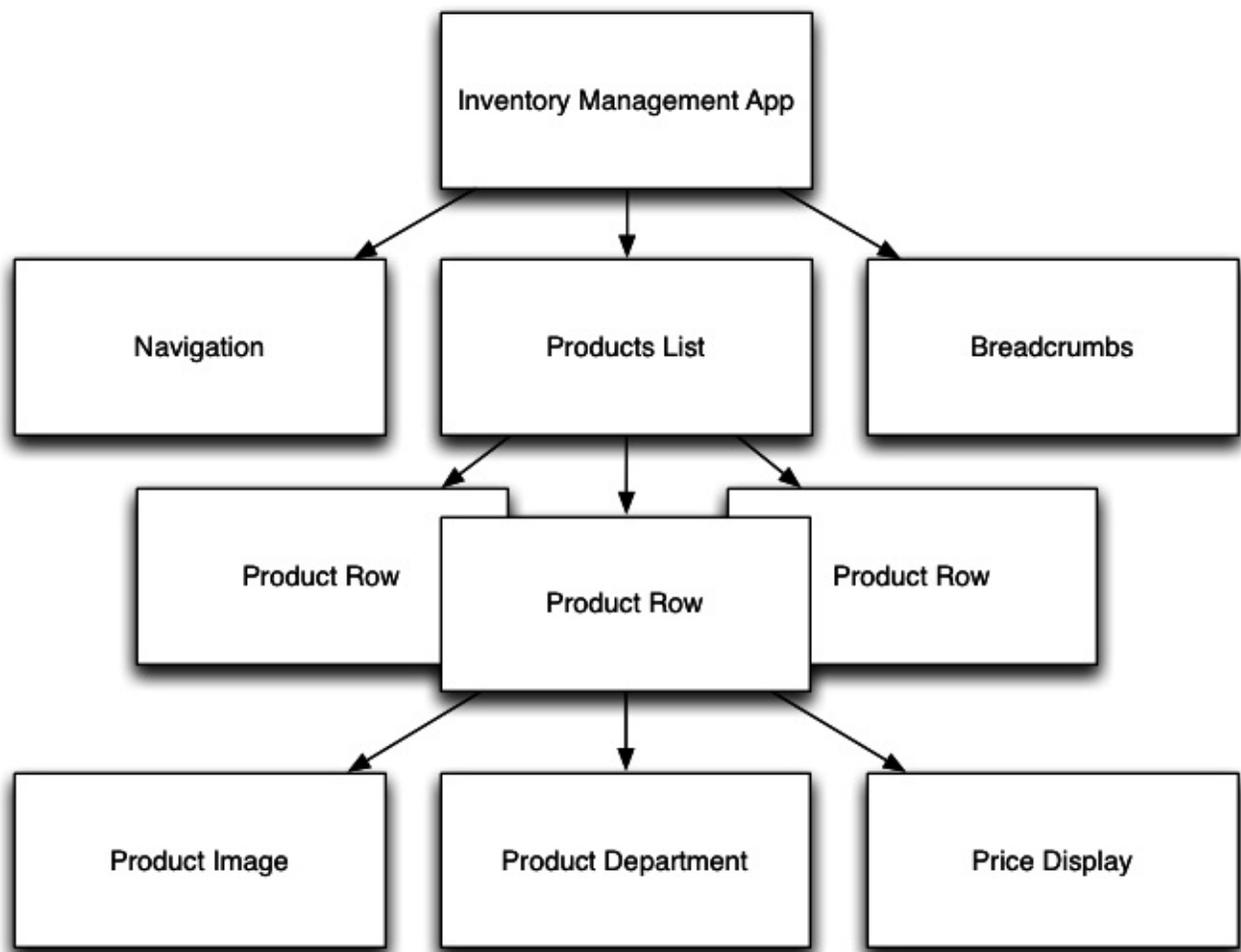
	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
---	--	----------

Product Row Component

And of course, we could continue one step further, breaking each Product Row into smaller pieces:

- the **Product Image** component would be responsible for rendering a product image, given its image name
- the **Product Department** component would render the department tree, like *Men > Shoes > Running Shoes*
- the **Price Display** component would render the price. Imagine that our implementation customizes the pricing if the user is logged in to include system-wide tier discounts or include shipping for instance. We could implement all this behavior into this component.

Finally, putting it all together into a tree representation, we end up with the following diagram:



App Tree Diagram

At the top we see **Inventory Management App**: that's our application.

Under the application we have the Navigation, the Breadcrumb and the Products List components.

The Products List component has Product Rows, one for each product.

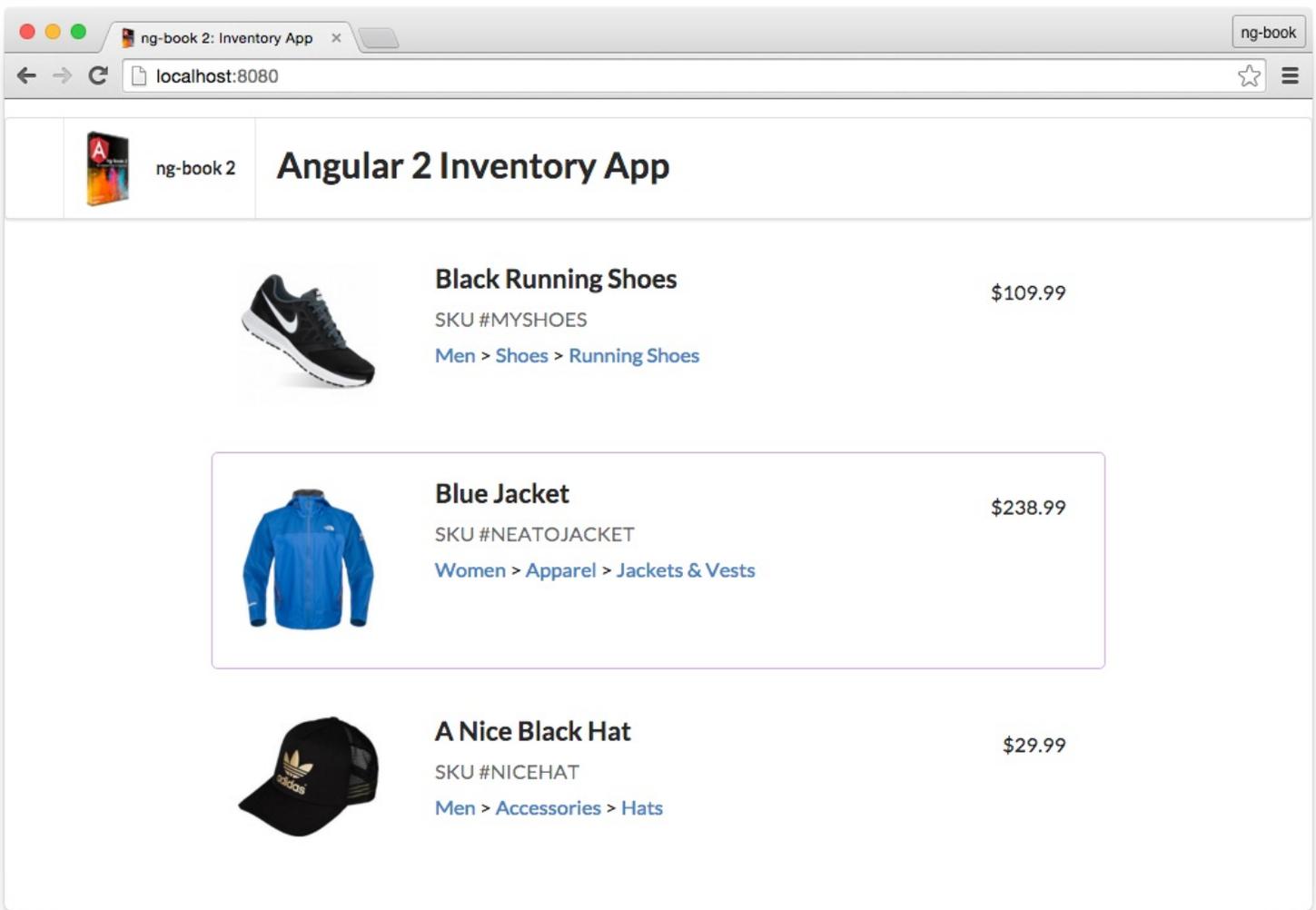
And the Product Row uses three components itself: one for the image, the department, and the price.

Let's work together to build this application.



You can find the full code listing for this chapter in the downloads under `how_angular_works/inventory_app`.

Here's a screenshot of what our app will look like when we're done:



Completed Inventory App

Product Model

One of the key things to realize about Angular is that it **doesn't prescribe a particular model library**.

Angular is flexible enough to support many different kinds of models (and data architectures). However, this means the choice is left to you as the user to determine how to implement these things.

We'll have **a lot** to say about data architectures in [future chapters](#). For now, though, we're going to have our models be plain JavaScript objects.

code/how_angular_works/inventory_app/app.ts

```
16 /**
17  * Provides a `Product` object
18  */
19 class Product {
20   constructor(
21     public sku: string,
22     public name: string,
23     public imageUrl: string,
24     public department: string[],
25     public price: number) {
26   }
27 }
```

If you're new to ES6/TypeScript this syntax might be a bit unfamiliar.

We're creating a new `Product` class and the constructor takes 5 arguments. When we write `public sku: string`, we're saying two things:

- there is a `public` variable on instances of this class called `sku`
- `sku` is of type `string`.



If you're already familiar with JavaScript, you can quickly catch up on some of the differences, including the `public` constructor shorthand, [here at learnxinyminutes](#)

This `Product` class doesn't have any dependencies on Angular, it's just a model that we'll use in our app.

Components

As we mentioned before, Components are the fundamental building block of Angular 2 applications. The "application" itself is just the top-level Component. Then we break our application into smaller child Components.



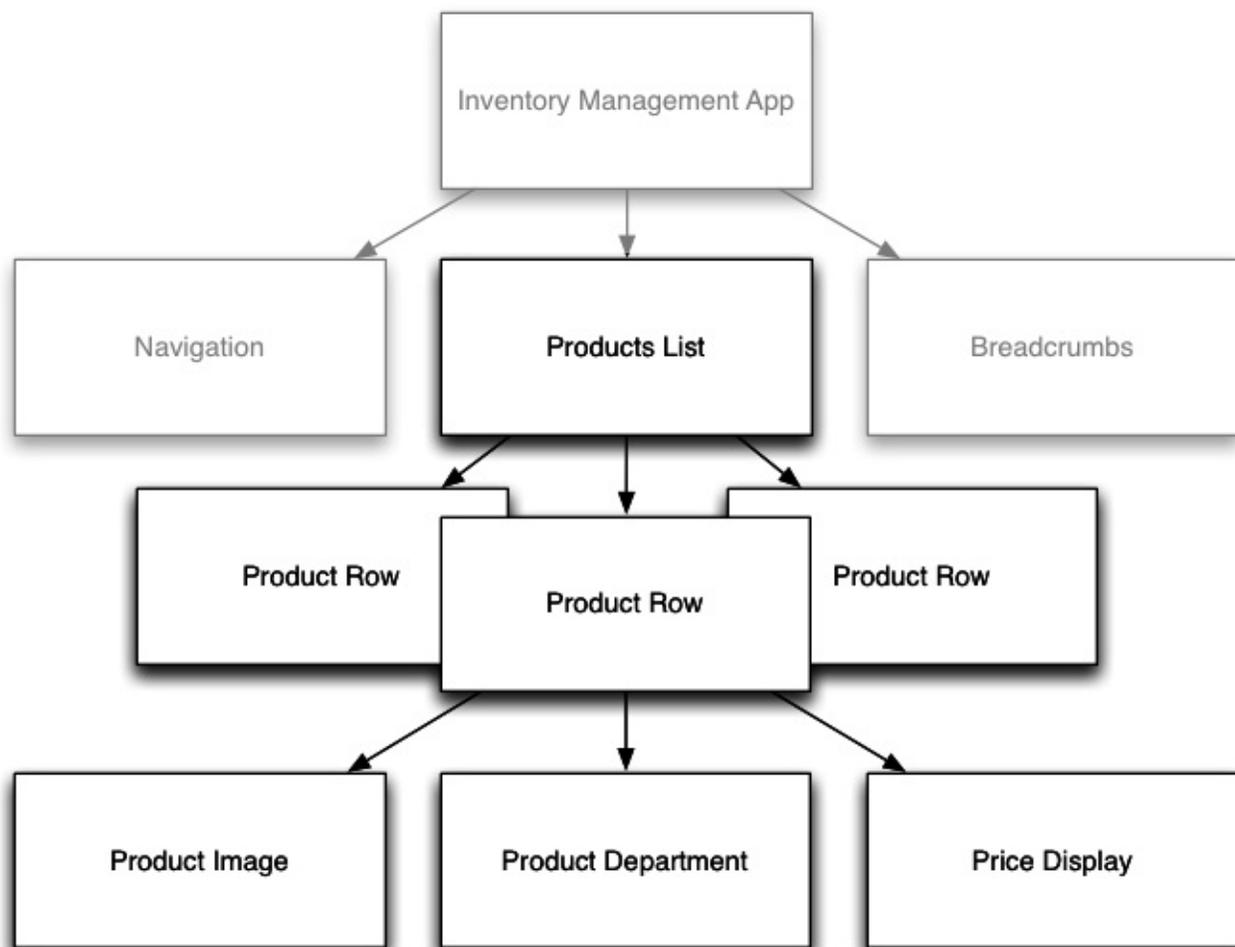
TIP: When building a new Angular application, mockup the design and then break it down into Components.

We'll be using Components a lot, so it's worth looking at them more closely.

Each components is composed of three parts:

- Component *Decorator*
- A View
- A Controller

To illustrate the key concepts we need to understand about components, we'll start with the top level Inventory App and then focus on the **Products List** and child components:



Products List Component

Here's what a basic, top-level InventoryApp looks like:

```

1 @Component({
2   selector: 'inventory-app',
3   template: `
4     <div class="inventory-app">
5       (Products will go here soon)
6     </div>
7   `
8 })
9 class InventoryApp {
10 }
11
12 bootstrap(InventoryApp);
  
```

If you've been using Angular 1 the syntax might look pretty foreign! But the ideas are pretty similar, so let's take them step by step:

The `@Component` is called a **decorator**. It adds metadata to the class that follows it (`InventoryApp`). The `@Component` annotation specifies:

- a selector, which tells Angular what element to match
- a template, which defines the view

The Component **controller** is defined by a class, the `InventoryApp` class, in this case.

Let's take a look into each part now in more detail.

Component Decorator

The `@Component` decorator is where you configure your component. Primarily, `@Component` will configure how the outside world will interact with your component.

There are lots of options available to configure a component, which we cover in the [Components chapter](#) (forthcoming). In this chapter we're just going to touch on some of the basics.

Component selector

With the `selector` key, you indicate how your component will be recognized when rendering HTML templates. The idea is similar to CSS or XPath selectors. The `selector` is a way to define what elements in the HTML will match this component. In this case, by saying `selector: 'inventory-app'`, we're saying that in our HTML we want to match the `inventory-app` tag, that is, we're defining a new tag that has new functionality whenever we use it. E.g. when we put this in our HTML:

```
1 <inventory-app></inventory-app>
```

Angular will use the `InventoryApp` component to implement the functionality.

Alternatively, with this selector, we can also use a regular `div` and specify the component as an attribute:

```
1 <div inventory-app></div>
```

Component template

The view is the visual part of the component. By using the `template` option on `@Component`, we declare the HTML template that the component will have.

```
1 @Component({
2   selector: 'inventory-app',
3   template: `
4     <div class="inventory-app">
5       (Products will go here soon)
6     </div>
7   `
8 })
```

For this template, notice that we're using TypeScript's backtick multi-line string syntax. Our template so far is pretty sparse: just a `div` with some placeholder text.

Adding A Product

Our app isn't very interesting without Products to view. Let's add some now.

We can create a new Product like this:

```
1 let newProduct = new Product(
2   'NICEHAT', 'A Nice Black Hat',
3   '/resources/images/products/black-hat.jpg',
4   ['Men', 'Accessories', 'Hats'],
5   29.99);
```

Our constructor for Product takes 5 arguments. We can create a new Product by using the new keyword.



Normally, I probably wouldn't pass more than 5 arguments to a function. Another option here is to configure the Product class to take an Object in the constructor, then if we wouldn't have to remember the order of the arguments. That is, Product could be changed to do something like this:

```
1 new Product({sku: "MYHAT", name: "A green hat"})
```

But for now, a 5 argument constructor is fine.

We want to be able to show this Product in the view. In order to make properties accessible to our template **we add them as instance variables to the Component.**

For instance, if we want to access newProduct in our view we would write:

```
1 class InventoryApp {
2   product: Product;
3
4   constructor() {
5     let newProduct = new Product(
6       'NICEHAT', 'A Nice Black Hat',
7       '/resources/images/products/black-hat.jpg',
8       ['Men', 'Accessories', 'Hats'],
9       29.99);
10
11    this.product = newProduct;
12  }
13 }
```

or more concisely:

```
1 class InventoryApp {
2   product: Product;
3
4   constructor() {
5     this.product = new Product(
6       'NICEHAT', 'A Nice Black Hat',
7       '/resources/images/products/black-hat.jpg',
8       ['Men', 'Accessories', 'Hats'],
9       29.99);
10  }
11 }
```

Notice that we did three things here:

1. **We added a constructor** - When Angular creates a new instance of this Component, it calls the constructor function. This is where we can put setup for this Component.
2. **We described an instance variable** - On InventoryApp, when we write: product: Product, we're specifying that the InventoryApp instances have a property product which is a Product object.
3. **We assigned a Product to product** - In the constructor we create an instance of Product and assigned it to the instance variable

Viewing the Product with Template Binding

Now that we have product assigned, we can use that variable in our view. Let's change our template to the following:

```
1 @Component({
2   selector: 'inventory-app',
3   template: `
4     <div class="inventory-app">
5       <h1>{{ product.name }}</h1>
6       <span>{{ product.sku }}</span>
7     </div>
8   `
9 })
```

Using the `{{...}}` syntax is called *template binding*. It tells the view we want to use the value of the expression inside the brackets at this location in our template.

So in this case, we have two bindings:

- `{{ product.name }}`
- `{{ product.sku }}`

The product variable comes from the instance variable `product` on our Component instance of `InventoryApp`.

What's neat about template binding is that the code inside the brackets is *an expression*. That means you can do things like this:

- `{{ count + 1 }}`
- `{{ myFunction(myArguments) }}`

In the first case, we're using an operator to change the displayed value of `count`. In the second case, we're able to replace the tags with the value of the function `myFunction(myArguments)`. Using template binding tags is the main way that you'll show data in your Angular applications.

Adding More Products

We actually don't want to show only a single product in our app - we actually want to show a whole list of products. So let's change our `InventoryApp` to store an array of `Products` rather than a single `Product`:

```
1 class InventoryApp {
2   products: Product[];
3
4   constructor() {
5     this.products = [];
6   }
7 }
```

Notice that we've renamed the variable `product` to `products`, and we've changed the type to `Product[]`. The `[]` characters at the end mean we want `products` to be an Array of `Products`. We also could have written this as: `Array<Product>`.

Now that our `InventoryApp` holds an array of `Products`. Let's create some `Products` in the constructor:

`code/how_angular_works/inventory_app/app.ts`

```

176 class InventoryApp {
177   products: Product[];
178
179   constructor() {
180     this.products = [
181       new Product(
182         'MYSHOES', 'Black Running Shoes',
183         '/resources/images/products/black-shoes.jpg',
184         ['Men', 'Shoes', 'Running Shoes'],
185         109.99),
186       new Product(
187         'NEATOJACKET', 'Blue Jacket',
188         '/resources/images/products/blue-jacket.jpg',
189         ['Women', 'Apparel', 'Jackets & Vests'],
190         238.99),
191       new Product(
192         'NICEHAT', 'A Nice Black Hat',
193         '/resources/images/products/black-hat.jpg',
194         ['Men', 'Accessories', 'Hats'],
195         29.99)
196     ];
197   }

```

This code will give us some Products to work with in our app.

Selecting a Product

We want to support user interaction in our app. For instance, the user might *select* a particular product to view more information about the product, add it to the cart, etc.

Let's add some functionality here in our InventoryApp to handle what happens when a new Product is selected. To do that, let's define a new function, `productWasSelected`:

code/how_angular_works/inventory_app/app.ts

```

199 productWasSelected(product: Product): void {
200   console.log('Product clicked: ', product);
201 }

```

Listing products using <products-list>

Now that we have our top-level InventoryApp component, we need to add a new component for rendering a list of products. In the next section we'll create the implementation of a `ProductsList` component that matches the selector `products-list`. Before we dive into the implementation details, here's how we will *use* this new component:

code/how_angular_works/inventory_app/app.ts

```

164 @Component({
165   selector: 'inventory-app',
166   directives: [ProductsList],
167   template: `
168     <div class="inventory-app">
169       <products-list
170         [productList]="products"
171         (onProductSelected)="productWasSelected($event)">
172     </products-list>
173   </div>
174 `
175 })

```

There's some new syntax and options here, so let's talk about each of them:

directives option

Notice that we've added the `directives` option to our `@Component` configuration. This specifies the other components we want to be able to use in this view. This option takes an Array of classes.

Unlike Angular 1, where all directives are essentially globals, in Angular 2 you must specifically say which directives you're going to be using. Here we say we're going to use the `ProductList` directive, which we will define in a minute.

Inputs and Outputs

When we use `products-list` we're using a key feature of Angular components: inputs and outputs:

```
1 <products-list
2   [productList]="products"           <!-- input -->
3   (onProductSelected)="productWasSelected($event)" > <!-- output -->
4 </products-list>
```

The `[squareBrackets]` pass inputs and the `(parenthesis)` handle outputs.

Data flows *in* to your component via *input bindings* and events flow *out* of your component through *output bindings*.

Think of the set of input + output bindings as defining the **public API** of your component.

`[squareBrackets]` pass inputs

In Angular, you pass data into child components via *inputs*.

In our code where we show:

```
1 <products-list
2   [productList]="products"
```

We're using an *input* of the `ProductList` component.

It can be tricky to understand where `products/productList` are coming from. There are two sides to this attribute:

- `[productList]` (the left-hand side) and
- `"products"` (the right-hand side)

The left-hand side `[productList]` says we want to use the `productList` *input* of the `products-list` component

The right-hand side `"products"` says we want to send the *value of the expression* `products`. That is, the array `this.products` in the `InventoryApp` class.



You might ask, “how would I know that `productList` is a valid input to the `products-list` component? The answer is: you’d read the docs for that component. The inputs (and outputs) are part of the “public API” of a component.

You’d know the inputs for a component that you’re using in the same way that you’d know what the arguments are for a function that you’re using.

(parens) handle outputs

In Angular, you send data out of components via *outputs*.

In our code where we show:

```
1 <products-list
2   ...
3   (onProductSelected)="productWasSelected($event)">
```

We’re saying that we want to listen to the `onProductSelected` *output* from the `ProductsList` component.

That is:

- `(onProductSelected)`, the left-hand side is the name of the output we want to “listen” on
- `"productWasSelected"`, the right-hand side is the function we want to call when something new is on this output
- `$event` is a special variable here that represents the thing emitted on the output.

Now, we haven’t talked about how to define inputs or outputs on our own components yet, but we will shortly when we define the `ProductsList` component.

Full InventoryApp Listing

Here’s the full code listing of our `InventoryApp` component:

[code/how_angular_works/inventory_app/app.ts](#)

```
161 /**
162  * @InventoryApp: the top-level component for our application
163  */
164 @Component({
165   selector: 'inventory-app',
166   directives: [ProductsList],
167   template: `
168     <div class="inventory-app">
169       <products-list
170         [productList]="products"
171         (onProductSelected)="productWasSelected($event)">
172     </products-list>
173   </div>
174 `
175 })
176 class InventoryApp {
177   products: Product[];
178
179   constructor() {
180     this.products = [
181       new Product(
182         'MYSHOES', 'Black Running Shoes',
183         '/resources/images/products/black-shoes.jpg',
184         ['Men', 'Shoes', 'Running Shoes'],
185         109.99),
```

```

186     new Product(
187         'NEATOJACKET', 'Blue Jacket',
188         '/resources/images/products/blue-jacket.jpg',
189         ['Women', 'Apparel', 'Jackets & Vests'],
190         238.99),
191     new Product(
192         'NICEHAT', 'A Nice Black Hat',
193         '/resources/images/products/black-hat.jpg',
194         ['Men', 'Accessories', 'Hats'],
195         29.99)
196 ];
197 }
198
199 productWasSelected(product: Product): void {
200     console.log('Product clicked: ', product);
201 }
202 }
203
204 bootstrap(InventoryApp);

```

The ProductsList Component

Now that we have our top-level application component, let’s write the ProductsList component, which will render a list of product rows.

We want to allow the user to select **one** Product and we want to keep track of which Product is the currently selected one. The ProductsList component is a great place to do this because it “knows” all of the Products at the same time.

Let’s write the ProductsList Component in three steps:

- Configuring the ProductsList @Component options
- Writing the ProductsList controller class
- Writing the ProductsList view template

Configuring the ProductsList @Component Options

Let’s take a look at the @Component configuration for ProductsList:

[code/how_angular_works/inventory_app/app.ts](#)

```

105 /**
106  * @ProductsList: A component for rendering all ProductRows and
107  * storing the currently selected Product
108  */
109 @Component({
110     selector: 'products-list',
111     directives: [ProductRow],
112     inputs: ['productList'],
113     outputs: ['onProductSelected'],

```

We start our ProductsList Component with a familiar option: selector. This selector means we can place our ProductsList component with the tag <products-list>.

The directives option should also be familiar: we’re specifying that we want to use the component ProductRow in this template. We haven’t defined ProductRow, but we will in a minute. We’ll use one ProductRow for each Product.

There are two new options though: inputs and outputs.

Component inputs

With the `inputs` option, we're specifying the parameters we expect our component to receive. `inputs` takes an array of strings which specify the input keys.

When we specify that a Component takes an input, it is expected that the definition class **will have an instance variable** that will receive the value. For example, say we have the following code:

```
1  @Component({
2    selector: 'my-component',
3    inputs: ['name', 'age']
4  })
5  class MyComponent {
6    name: string;
7    age: number;
8  }
```

The `name` and `age` inputs map to the `name` and `age` properties on instances of the `MyComponent` class.

If we want to use `MyComponent` from another template, we write something like: `<my-component [name]="myName" [age]="myAge"></my-component>`.

Notice that the attribute name matches the input name, which in turn matches the `MyComponent` property name. They don't always have to match.

For instance, say we wanted our attribute key and instance property to differ. That is, we want to use our component like this:

```
1 <my-component [shortName]="myName" [oldAge]="myAge"></my-component>
```

To do this, we would change the format of the string in the `inputs` option:

```
1  @Component({
2    selector: 'my-component',
3    inputs: ['name: shortName', 'age: oldAge']
4  })
5  class MyComponent {
6    name: string;
7    age: number;
8  }
```

More generally, `inputs` strings can have the format `'componentProperty: exposedProperty'`.

For instance we could have a different component that looks like this:

```
1  @Component({
2    //...
3    inputs: ['name', 'age', 'enabled']
4    //...
5  })
6  class MyComponent {
7    name: string;
8    age: number;
9    enabled: boolean;
10 }
```

However, if we wanted to represent the exposed property `enabled` in my component as `isEnabled`, we could use the alternative notation, like this:

```

1  @Component({
2  //...
3  inputs: [
4    'name: name',
5    'age: age',
6    'isEnabled: enabled'
7  ]
8  //...
9  })
10 class MyComponent {
11   name: string;
12   age: number;
13   isEnabled: boolean;
14 }

```

And going a little further, since the only property that requires an explicit mapping is enabled to isEnabled, we could even simplify and write it like this:

```

1  @Component({
2  //...
3  inputs: ['name', 'age', 'isEnabled: enabled']
4  //...
5  })
6  class MyComponent {
7   name: string;
8   age: number;
9   isEnabled: boolean;
10 }

```

In the inputs array, when the strings are in the key: value format, each have a specific meaning:

- The **key** (name, age and isEnabled) represent how that incoming property will be **visible (“bound”) in the controller**.
- The **value** (name, age and enabled) configures how the property is **visible to the outside world**.

Passing products through via the inputs

If you recall, in our InventoryApp, we passed products to our products-list via the [productList] input:

code/how_angular_works/inventory_app/app.ts

```

161 /**
162  * @InventoryApp: the top-level component for our application
163  */
164 @Component({
165   selector: 'inventory-app',
166   directives: [ProductsList],
167   template: `
168     <div class="inventory-app">
169       <products-list
170         [productList]="products"
171         (onProductSelected)="productWasSelected($event)">
172     </products-list>
173   </div>
174 `
175 })
176 class InventoryApp {
177   products: Product[];
178
179   constructor() {
180     this.products = [
181     new Product(

```

Hopefully this now makes a bit more sense: we’re passing this.products in via an input on ProductsList.

Component outputs

When you want to send data from your component to the outside world, you use *output bindings*.

Let's say a component we're writing has a button and we need to do something when that button is clicked.

The way to do this is by binding the *click* output of the button to a method declared on our component's controller. You do that using the `(output)="action"` notation.

Here's an example where we keep a counter and increment (or decrement) based on which button is pressed:

```
1 @Component({
2   selector: 'counter',
3   template: `
4     {{ value }}
5     <button (click)="increase()">Increase</button>
6     <button (click)="decrease()">Decrease</button>
7   `
8 })
9 class Counter {
10  value: number;
11
12  constructor() {
13    this.value = 1;
14  }
15
16  increase() {
17    this.value = this.value + 1;
18  }
19
20  decrease() {
21    this.value = this.value - 1;
22  }
23 }
```

In this example we're saying that every time the first button is clicked, we want the `increase()` method on our controller to be invoked. And, similarly, when the second button is clicked, we want to call the `decrease()` method.

The parentheses attribute syntax looks like this: `(output)="action"`. In this case, the output we're listening for is `click` event on this button. There are many other built-in events you can listen to: `mousedown`, `mousemove`, `dbl-click`, etc.

In this example, the event is internal to the component. When creating our own components we can also expose "public events" (component outputs) that allow the component to talk to the outside world.

The key thing to understand here is that in a view, we can listen to an event by using the `(output)="action"` syntax.

Emitting Custom Events

Let's say we want to create a component that emits a custom event, like `click` or `mousedown` above. To create a custom output event we do three things:

1. Specify outputs in the `@Component` configuration
2. Attach an `EventEmitter` to the output property

3. Emit an event from the EventEmitter, at the right time



Perhaps EventEmitter is unfamiliar to you. Don't panic! It's not too hard.

An EventEmitter is simply an object that helps you implement the [Observer Pattern](#). That is, it's an object that can maintain a list of subscribers and publish events to them. That's it.

Here's a short and sweet example of how you can use EventEmitter

```
1 let ee = new EventEmitter();
2 ee.subscribe((name: string) => console.log(`Hello ${name}`));
3 ee.emit("Nate");
4
5 // -> "Hello Nate"
```

When we assign an EventEmitter to an output *Angular automatically subscribes* for us. You don't need to do the subscription yourself (necessarily, though you can add your own subscriptions if you want to).

Here's a code example of how we write a component that has outputs:

```
1 @Component({
2   selector: 'single-component',
3   outputs: ['putRingOnIt'],
4   template: `
5     <button (click)="liked()">Like it?</button>
6   `
7 })
8 class SingleComponent {
9   putRingOnIt: EventEmitter<string>;
10
11   constructor() {
12     this.putRingOnIt = new EventEmitter();
13   }
14
15   liked(): void {
16     this.putRingOnIt.emit("oh oh oh");
17   }
18 }
```

Notice that we did all three steps: 1. specified outputs, 2. created an EventEmitter that we attached to the output property putRingOnIt and 3. Emitted an event when liked is called.

If we wanted to use this output in a parent component we could do something like this:

```
1 @Component({
2   selector: 'club',
3   template: `
4     <div>
5       <single-component
6         (putRingOnIt)="ringWasPlaced($event)"
7       ></single-component>
8     </div>
9   `
10 })
11 class ClubComponent {
12   ringWasPlaced(message: string) {
13     console.log(`Put your hands up: ${message}`);
14   }
15 }
16
17 // logged -> "Put your hands up: oh oh oh"
```

Again, notice that:

- putRingOnIt comes from the outputs of SingleComponent
- ringWasPlaced is a function on the ClubComponent
- \$event contains the thing that was emitted, in this case a string

Writing the ProductsList Controller Class

Back to our store example, our ProductsList controller class needs three instance variables:

- One to hold the list of Products (that come from the productList input)
- One to output events (that emit from the onProductSelected output)
- One to hold a reference to the currently selected product

Here's how we define those in code:

code/how_angular_works/inventory_app/app.ts

```

125 class ProductsList {
126   /**
127    * @input productList - the Product[] passed to us
128    */
129   productList: Product[];
130
131   /**
132    * @output onProductSelected - outputs the current
133    *     Product whenever a new Product is selected
134    */
135   onProductSelected: EventEmitter<Product>;
136
137   /**
138    * @property currentProduct - local state containing
139    *     the currently selected `Product`
140    */
141   currentProduct: Product;
142
143   constructor() {
144     this.onProductSelected = new EventEmitter();
145   }

```

Notice that our productList is an Array of Products - this comes in from the inputs.

onProductSelected is our output.

currentProduct is a property internal to ProductsList. You might also hear this being referred to as “local component state”. It's only used here within the component.

Writing the ProductsList View Template

Here's the template for our products-list component:

code/how_angular_works/inventory_app/app.ts

```

114   template: `
115     <div class="ui items">
116       <product-row
117         *ngFor="let myProduct of productList"
118         [product]="myProduct"
119         (click)='clicked(myProduct)'
120         [class.selected]="isSelected(myProduct)">
121       </product-row>
122     </div>
123   `
124 })

```

Here we're using the `product-row` tag, which comes from the `ProductRow` component, which we'll define in a minute.

We're using `ngFor` to iterate over each `Product` in `productList`. We've talked about `ngFor` before in this book, but just as a reminder the `let thing of things` syntax says, "iterate over things and create a copy of this element for each item, and assign each item to the variable `thing`".

So in this case, we're iterating over the `Products` in `productList` and generating a local variable `myProduct` for each one.



Stylistically, I probably wouldn't call this variable `myProduct` in a real app. I'd probably just call it `product`, or even `p`. But I want to be explicit about what we're passing around, and so `myProduct` is slightly clearer.

The interesting thing to note about this `myProduct` variable is that we can now use it *even on the same tag*. As you can see, we do this on the following three lines.

The line that reads `[product]="myProduct"` says that we want to pass `myProduct` (the local variable) to the input `product` of the `product-row`. (We'll define this input when we define the `ProductRow` component below.)

The `(click)="clicked(myProduct)"` line describes what we want to do when this element is clicked. `click` is a built-in event that is triggered when the host element is clicked on. In this case, we want to call the component function `clicked` on `ProductsList` whenever this element is clicked on.

The line `[class.selected]="isSelected(myProduct)"` is a fun one: Angular allows us to set classes conditionally on an element using this syntax. This syntax says "add the CSS class `selected` if `isSelected(myProduct)` returns `true`." This is a really handy way for us to mark the currently selected product.

You may have noticed that we didn't define `clicked` nor `isSelected` yet, so let's do that now (in `ProductsList`):

`clicked`

code/how_angular_works/inventory_app/app.ts

```
147 clicked(product: Product): void {
148     this.currentProduct = product;
149     this.onProductSelected.emit(product);
150 }
```

This function does two things:

1. Set `this.currentProduct` to the `Product` that was passed in.
2. Emit the `Product` that was clicked on our output

`isSelected`

code/how_angular_works/inventory_app/app.ts

```
152 isSelected(product: Product): boolean {
153     if (!product || !this.currentProduct) {
```

```
154     return false;
155   }
156   return product.sku === this.currentProduct.sku;
157 }
```

This function accepts a Product and returns true if product's sku matches the currentProduct's sku. It returns false otherwise.

The Full ProductsList Component

Here's the full code listing so we can see everything in context:

code/how_angular_works/inventory_app/app.ts

```
105 /**
106  * @ProductsList: A component for rendering all ProductRows and
107  * storing the currently selected Product
108  */
109 @Component({
110   selector: 'products-list',
111   directives: [ProductRow],
112   inputs: ['productList'],
113   outputs: ['onProductSelected'],
114   template: `
115     <div class="ui items">
116       <product-row
117         *ngFor="let myProduct of productList"
118         [product]="myProduct"
119         (click)="clicked(myProduct)"
120         [class.selected]="isSelected(myProduct)">
121     </product-row>
122   </div>
123 `
124 })
125 class ProductsList {
126   /**
127    * @input productList - the Product[] passed to us
128    */
129   productList: Product[];
130
131   /**
132    * @output onProductSelected - outputs the current
133    * Product whenever a new Product is selected
134    */
135   onProductSelected: EventEmitter<Product>;
136
137   /**
138    * @property currentProduct - local state containing
139    * the currently selected `Product`
140    */
141   currentProduct: Product;
142
143   constructor() {
144     this.onProductSelected = new EventEmitter();
145   }
146
147   clicked(product: Product): void {
148     this.currentProduct = product;
149     this.onProductSelected.emit(product);
150   }
151
152   isSelected(product: Product): boolean {
153     if (!product || !this.currentProduct) {
154       return false;
155     }
156     return product.sku === this.currentProduct.sku;
157   }
158 }
159 }
```

The ProductRow Component



A Selected Product Row Component

Our ProductRow displays our Product. ProductRow will have its own template, but will also be split up into three smaller Components:

- ProductImage - for the image
- ProductDepartment - for the department “breadcrumbs”
- PriceDisplay - for showing the product’s price

Here’s a visual of the three Components that will be used within the ProductRow:



ProductRow’s Sub-components

Let’s take a look at the ProductRow’s Component configuration, definition class, and template:

ProductRow Component Configuration

code/how_angular_works/inventory_app/app.ts

```
79 /**
80  * @ProductRow: A component for the view of single Product
81  */
82 @Component({
83   selector: 'product-row',
84   inputs: ['product'],
85   host: {'class': 'item'},
86   directives: [ProductImage, ProductDepartment, PriceDisplay],
```

We start by defining the selector of product-row. We’ve seen this several times now - this defines that this component will match the tag product-row.

Next we define that this row takes an input of product. This will be the Product that was passed in from our parent Component.

The third option `host` is a new one. The `host` option lets us set attributes on the host element. In this case, we're using the [Semantic UI item class](#). Here when we say `host: {'class': 'item'}` we're saying that we want to attach the CSS class "item" to the host element.



Using `host` is nice because it means we can configure our host element from *within* the component. This is great because otherwise we'd require the host element to specify the CSS tag and that is bad because we would then make assigning a CSS class part of the requirement to using the Component.

Next we specify the directives we're going to be using within our template. We haven't defined these directives yet, but we will in a minute.

ProductRow Component Definition Class

The ProductRow Component definition class is straightforward:

code/how_angular_works/inventory_app/app.ts

```
101 class ProductRow {
102   product: Product;
103 }
```

Here we're specifying that the ProductRow will have an instance variable `product`. Because we specified an input of `product`, when Angular creates an instance of this Component, it will automatically assign the `product` for us. We don't need to do it manually, and we don't need a constructor.

ProductRow template

Now let's take a look at the template:

code/how_angular_works/inventory_app/app.ts

```
87 template: `
88 <product-image [product]="product"></product-image>
89 <div class="content">
90   <div class="header">{{ product.name }}</div>
91   <div class="meta">
92     <div class="product-sku">SKU #{{ product.sku }}</div>
93   </div>
94   <div class="description">
95     <product-department [product]="product"></product-department>
96   </div>
97 </div>
98 <price-display [price]="product.price"></price-display>
99 `
```

Our template doesn't have anything conceptually new.

In the first line we use our `product-image` directive and we pass our `product` to the `product` input of the `ProductImage` component. We use the `product-department` directive in the same way.

We use the price-display directive slightly differently in that we pass the `product.price`, instead of the `product` directly.

The rest of the template is standard HTML elements with custom CSS classes and some template bindings.

ProductRow Full Listing

Here's the ProductRow component all together:

code/how_angular_works/inventory_app/app.ts

```
79 /**
80  * @ProductRow: A component for the view of single Product
81  */
82 @Component({
83   selector: 'product-row',
84   inputs: ['product'],
85   host: {'class': 'item'},
86   directives: [ProductImage, ProductDepartment, PriceDisplay],
87   template: `
88     <product-image [product]="product"></product-image>
89     <div class="content">
90       <div class="header">{{ product.name }}</div>
91       <div class="meta">
92         <div class="product-sku">SKU #{{ product.sku }}</div>
93       </div>
94       <div class="description">
95         <product-department [product]="product"></product-department>
96       </div>
97     </div>
98     <price-display [price]="product.price"></price-display>
99 `
100 })
101 class ProductRow {
102   product: Product;
103 }
```

Now let's talk about the three components we used. They're pretty short.

The ProductImage Component

code/how_angular_works/inventory_app/app.ts

```
29 /**
30  * @ProductImage: A component to show a single Product's image
31  */
32 @Component({
33   selector: 'product-image',
34   host: {'class': 'ui small image'},
35   inputs: ['product'],
36   template: `
37     <img class="product-image" [src]="product.imageUrl">
38 `
39 })
40 class ProductImage {
41   product: Product;
42 }
```

The one thing to note here is in the `img` tag, notice how we use the `[src]` input to `img`.

We could have written this tag this way:

```
1 <!-- wrong, don't do it this way -->
2 
```

Why is that wrong? Well, because in the case where your browser loads that template before Angular has run, your browser will try to load the image with the literal string `{{ product.imageUrl }}` and then it will get a 404 not found, which can show a broken image on your page until Angular runs.

By using the `[src]` attribute, we're telling Angular that we want to use the `[src]` *input* on this `img` tag. Angular will then replace the value of the `src` attribute once the expression is resolved.

The PriceDisplay Component

Next, let's look at PriceDisplay:

code/how_angular_works/inventory_app/app.ts

```
64 /**
65  * @PriceDisplay: A component to show the price of a
66  * Product
67  */
68 @Component({
69   selector: 'price-display',
70   inputs: ['price'],
71   template: `
72     <div class="price-display">\${{ price }}</div>
73   `
74 })
75 class PriceDisplay {
76   price: number;
77 }
```

It's pretty straightforward, but one thing to note is that we're escaping the dollar sign `$` because this is a backtick string and the dollar sign is used for template variables.

The ProductDepartment Component

code/how_angular_works/inventory_app/app.ts

```
44 /**
45  * @ProductDepartment: A component to show the breadcrumbs to a
46  * Product's department
47  */
48 @Component({
49   selector: 'product-department',
50   inputs: ['product'],
51   template: `
52     <div class="product-department">
53       <span *ngFor="let name of product.department; let i=index">
54         <a href="#">{{ name }}</a>
55         <span>{{ i < (product.department.length-1) ? '>' : '' }}</span>
56       </span>
57     </div>
58   `
59 })
60 class ProductDepartment {
61   product: Product;
62 }
```

The thing to note about the ProductDepartment Component is the `ngFor` and the `span` tag.

Our `ngFor` loops over `product.department` and assigns each department string to `name`. The new part is the second expression that says: `#i=index`. This is how you get the iteration number out of `ngFor`.

In the `span` tag, we use the `i` variable to determine if we should show the greater-than `>` symbol.

The idea is that given a department, we want to show the department string like:

The expression `{{i < (product.department.length-1) ? '>' : ''}}` says that we only want to use the '>' character if we're not the last department. On the last department just show an empty string ''.

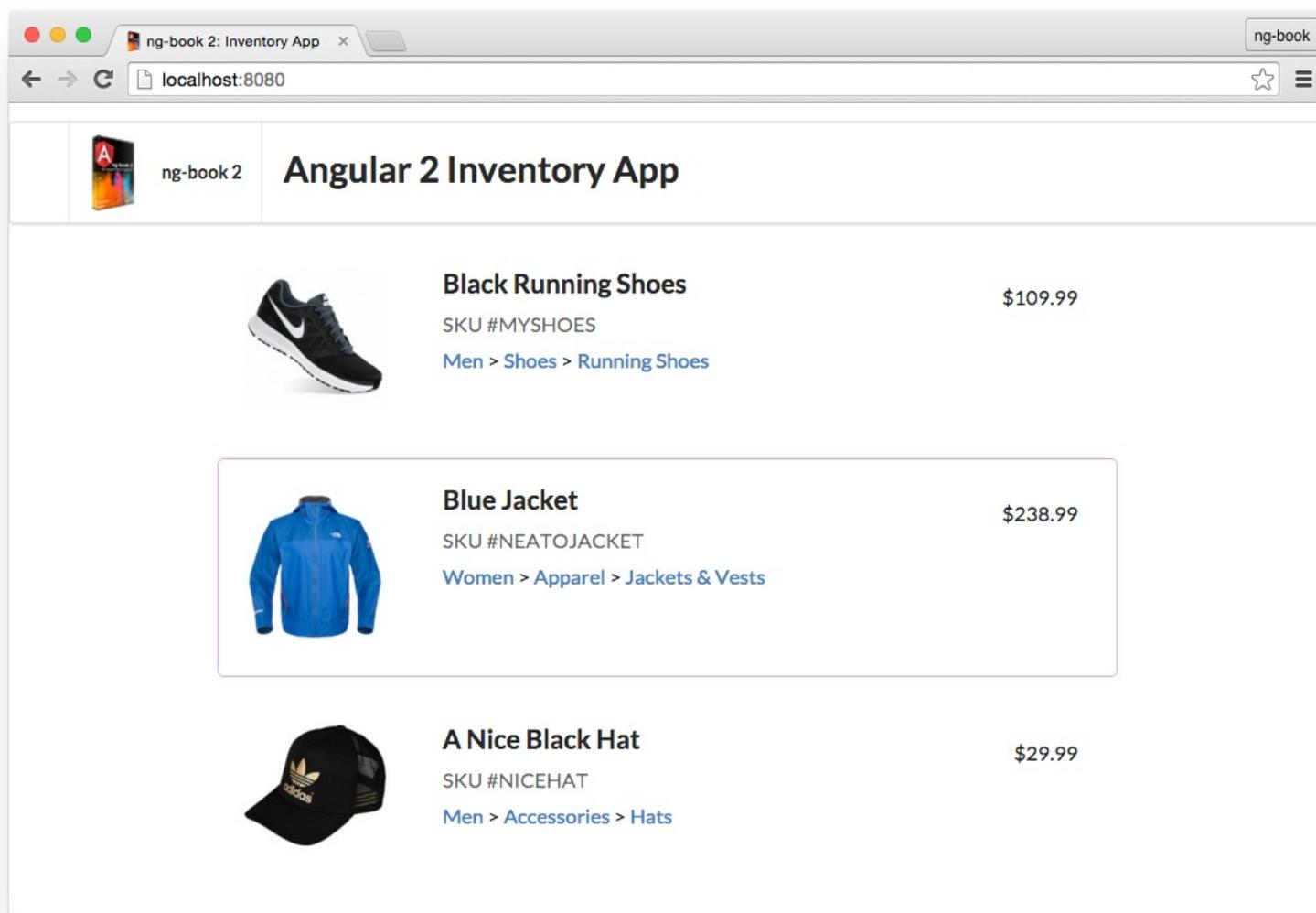


This format: `test ? valueIfTrue : valueIfFalse` is called the *ternary operator*.

The Completed Project

Now we have all the pieces we need for the working project!

Here's what it will look like when we're done:



Completed Inventory App



You can run the code example in `how_angular_works/inventory_app`. See the [README](#) there.

Now you can click to select a particular product and have it render a nice purple outline when selected. If you add new Products in your code, you'll see them rendered.

A Word on Data Architecture

You might be wondering at this point how we would manage the data flow if we started adding more functionality to this app.

For instance, say we wanted to add a shopping cart view and then we would add items to our cart. How could we implement this?

The only tools we've talked about are emitting output events. When we click add-to-cart do we simply bubble up an `addedToCart` event and handle at the root component? That feels a bit awkward.

Data architecture is a large topic with many opinions. Thankfully, Angular is flexible enough to handle a wide variety of data architectures, but that means that you have to decide for yourself which to use.

In Angular 1, the default option was two-way data binding. Two-way data binding is super easy to get started: your controllers have data, your forms manipulate that data directly, and your views show the data.

The problem with two-way data binding is that it often causes cascading effects throughout your application and makes it really difficult to trace data flow as your project grows.

Another problem with two-way data binding is that because you're passing data down through components it often forces your "data layout tree" to match your "dom view tree". In practice, these two things should really be separate.

One way you might handle this scenario would be to create a `ShoppingCartService`, which would be a singleton that would hold the list of the current items in the cart. This service could notify any interested objects when the items in the cart changes.

The idea is easy enough, but in practice there's a lot of details to be worked out.

The recommended way in Angular 2, and in many modern web frameworks (such as React) is to adopt a pattern of **one-way data binding**. That is, your data flows only **down** through components. If you need to make changes, you emit events that cause changes to happen "at the top" which then trickle down.

One-way data binding can seem like it adds some overhead in the beginning but it saves *a lot* of complication around change detection and it makes your systems easier to reason about.

Thankfully there are two major contenders for managing your data architecture:

1. Use an Observables-based architecture like RxJS
2. Use a Flux-based architecture

Later in this book we'll talk about how to implement a scalable data architecture for your app. For now, bask in the joy of your new Component-based application!

Built-in Components

Introduction

Angular 2 provides a number of built-in components. In this chapter, we're going to cover each built-in component and show you examples of how to use them.

 The built-in components are imported and made available to your components automatically, so you don't need to inject it as a directive like you would do with your own components.

NgIf

The `ngIf` directive is used when you want to display or hide an element based on a condition. The condition is determined by the result of the *expression* that you pass in to the directive.

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

Some examples are:

```
1 <div *ngIf="false"></div>           <!-- never displayed -->
2 <div *ngIf="a > b"></div>         <!-- displayed if a is more than b -->
3 <div *ngIf="str == 'yes'"></div>  <!-- displayed if str holds the string "yes" -\
4 ->
5 <div *ngIf="myFunc()"></div>     <!-- displayed if myFunc returns a true value \
6 -->
```

 If you have experience with Angular 1, you probably used `ngIf` directive before. You can think of the Angular 2 version as a direct substitute. On the other hand, Angular 2 offers no built-in alternative for `ng-show`. So, if your goal is to just change the CSS visibility of an element, you should look into either the `ngStyle` or the `class` directives, described later in this chapter.

NgSwitch

Sometimes you need to render different elements depending on a given condition.

When you run into this situation, you could use `ngIf` several times like this:

```
1 <div class="container">
2   <div *ngIf="myVar == 'A'">Var is A</div>
3   <div *ngIf="myVar == 'B'">Var is B</div>
4   <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something else</div>
5 </div>
```

But as you can see, the scenario where `myVar` is neither A nor B is pretty verbose, all we're really trying to express is an `else`. And as we add more values the last `ngIf` condition will become more complex.

To illustrate this growth in complexity, let's say we wanted to handle a new hypothetical C value.

In order to do that, we'd have to not only add the new element with `ngIf`, but also change the last case:

```
1 <div class="container">
2   <div *ngIf="myVar == 'A'">Var is A</div>
3   <div *ngIf="myVar == 'B'">Var is B</div>
4   <div *ngIf="myVar == 'C'">Var is C</div>
5   <div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something else</div>
6 </div>
7 </div>
```

For cases like this, Angular 2 introduces the `ngSwitch` directive.

If you're familiar with the `switch` statement then you'll feel very at home.

The idea behind this directive is the same: allow a single evaluation of an expression, and then display nested elements based on the value that resulted from that evaluation.

Once we have the result then we can:

- Describe the known results, using the `ngSwitchWhen` directive
- Handle all the other unknown cases with `ngSwitchDefault`

Let's rewrite our example using this new set of directives:

```
1 <div class="container" [ngSwitch]="myVar">
2   <div *ngSwitchWhen="'A'">Var is A</div>
3   <div *ngSwitchWhen="'B'">Var is B</div>
4   <div *ngSwitchDefault>Var is something else</div>
5 </div>
```

Then if we want to handle the new value C we insert a single line:

```
1 <div class="container" [ngSwitch]="myVar">
2   <div *ngSwitchWhen="'A'">Var is A</div>
3   <div *ngSwitchWhen="'B'">Var is B</div>
4   <div *ngSwitchWhen="'C'">Var is C</div>
5   <div *ngSwitchDefault>Var is something else</div>
6 </div>
```

And we don't have to touch the default (i.e. *fallback*) condition.

Having the `ngSwitchDefault` element is optional. If we leave it out, nothing will be rendered when `myVar` fails to match any of the expected values.

You can also declare the same `*ngSwitchWhen` value for different elements. Here's an example:

[code/built_in_components/ng_switch/app.ts](#)

```
4 @Component({
5   selector: 'switch-sample-app',
6   template: `
7     <h4 class="ui horizontal divider header">
8       Current choice is {{ choice }}
9     </h4>
10
11     <div class="ui raised segment">
12       <ul [ngSwitch]="choice">
13         <li *ngSwitchWhen="1">First choice</li>
14         <li *ngSwitchWhen="2">Second choice</li>
15         <li *ngSwitchWhen="3">Third choice</li>
```

```

16     <li *ngSwitchWhen="4">Fourth choice</li>
17     <li *ngSwitchWhen="2">Second choice, again</li>
18     <li *ngSwitchDefault>Default choice</li>
19   </ul>
20 </div>
21
22 <div style="margin-top: 20px;">
23   <button class="ui primary button" (click)="nextChoice()">
24     Next choice
25   </button>
26 </div>
27
28 })

```

Another nice feature of `ngSwitchWhen` is that you're not limited to matching only a single time. For instance, in the example above when the choice is 2, both the second and fifth `li`s will be rendered.

NgStyle

With the `NgStyle` directive, you can set a given DOM element CSS properties from Angular expressions.

The simplest way to use this directive is by doing `[style.<cssproperty>]="value"`. For example:

```

code/built_in_components/ng_style/app.ts
11 <div [style.background-color]='yellow'>
12   Uses fixed yellow background
13 </div>

```

This snippet is using the `NgStyle` directive to set the `background-color` CSS property to the literal string `'yellow'`.

Another way to set fixed values is by using the `NgStyle` attribute and using key value pairs for each property you want to set, like this:

```

code/built_in_components/ng_style/app.ts
19 <div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
20   Uses fixed white text on blue background
21 </div>

```



Notice that in the `ng-style` specification we have single quotes around `background-color` but not around `color`. Why is that? Well, the argument to `ng-style` is a Javascript object and `color` is a valid key, without quotes. With `background-color`, however, the dash character isn't allowed in an object key, unless it's a string so we have to quote it.

Generally I'd leave out quoting as much as possible in object keys and only quote keys when we have to.

Here we are setting both the `color` and the `background-color` properties.

But the real power of the `NgStyle` directive comes with using dynamic values.

In our example, we are defining two input boxes:

```

code/built_in_components/ng_style/app.ts
62 <div class="ui input">
63   <input type="text" name="color" value="{{color}}" #colorinput>
64 </div>

```

```
65 <div class="ui input">
66   <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
67 </div>
```

And then using their values to set the CSS properties for three elements.

On the first one, we're setting the font size based on the input value:

code/built_in_components/ng_style/app.ts

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

It's important to note that we have to specify units where appropriate. For instance, it isn't valid CSS to set a font-size of 12 - we have to specify a unit such as 12px or 1.2em. Angular provides a handy syntax for specifying units: here we used the notation [style.fontSize.px].

The .px suffix indicates that we're setting the font-size property value in pixels. You could easily replace that by [style.fontSize.em] to express the font size in ems or even in percentage using [style.fontSize.%].

The other two elements use the #colorinput to set the text and background colors:

code/built_in_components/ng_style/app.ts

```
39 <h4 class="ui horizontal divider header">
40   ngStyle with object property from variable
41 </h4>
42
43 <div>
44   <span [ngStyle]="{color: colorinput.value}">
45     {{ colorinput.value }} text
46   </span>
47 </div>
48
49 <h4 class="ui horizontal divider header">
50   style from variable
51 </h4>
52
53 <div [style.background-color]="colorinput.value"
54   style="color: white;">
55   {{ colorinput.value }} background
56 </div>
```

This way, when we click the **Apply settings** button, we call a method that sets the new values:

code/built_in_components/ng_style/app.ts

```
97 apply(color, fontSize) {
98   this.color = color;
99   this.fontSize = fontSize;
100 }
```

And with that, both the color and the font size will be applied to the elements using the NgStyle directive.

NgClass

The `NgClass` directive, represented by a `ngClass` attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.



If you're coming from Angular 1, the `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

The first way to use this directive is by passing in an object literal. The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

Let's assume we have a CSS class called `bordered` that adds a dashed black border to an element:

code/built_in_components/class/styles.css

```
14 .bordered {
15   border: 1px dashed black;
16   background-color: #eee;
17 }
```

Let's add two `div` elements: one always having the `bordered` class (and therefore always having the border) and another one never having it:

code/built_in_components/ng_class/app.ts

```
7 <div [ngClass]="{bordered: false}">This is never bordered</div>
8 <div [ngClass]="{bordered: true}">This is always bordered</div>
```

As expected, this is how those two `div`s would be rendered:

This is never bordered

This is always bordered

Simple class directive usage

Of course, it's a lot more useful to use the `NgClass` directive to make class assignments dynamic.

To make it dynamic we add a variable as the value for the object value, like this:

code/built_in_components/ng_class/app.ts

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

Alternatively, we can define the object in our component:

code/built_in_components/ng_class/app.ts

```
58 toggleBorder() {
59   this.isBordered = !this.isBordered;
60   this.classesObj = {
61     bordered: this.isBordered
62   };
63 }
```

And use the object directly:

code/built_in_components/ng_class/app.ts

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



Again, be careful when you have class names that contains dashes, like bordered-box. JavaScript objects don't allow literal keys to have dashes. If you need to use them, you must make the key a string like this:

```
1 <div [ng-class]="{'bordered-box': false}">...</div>
```

We can also use a list of class names to specify which class names should be added to the element. For that, we can either pass in an array literal:

code/built_in_components/ng_class/app.ts

```
36 <div class="base" [ngClass]="['blue', 'round']">
37   This will always have a blue background and
38   round corners
39 </div>
```

Or declare an array variable in our component:

```
1   this.classList = ['blue', 'round'];
```

And passing it in:

code/built_in_components/ng_class/app.ts

```
41 <div class="base" [ngClass]="classList">
42   This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
43   and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
44 </div>
```

In this last example, the [class] assignment works alongside existing values assigned by the HTML class attribute.

The resulting classes added to the element will always be the set of the classes provided by usual class HTML attribute and the result of the evaluation of the [class] directive.

In this example:

code/built_in_components/ng_class/app.ts

```
36 <div class="base" [ngClass]="['blue', 'round']">
37   This will always have a blue background and
38   round corners
39 </div>
```

The element will have all three classes: base from the class HTML attribute and also blue and round from the [class] assignment:

```
Elements Console Sources Network Timeline Profiles Resources Audits
<button>Toggle</button>
▶ <div class="selectors">...</div>
... <div class="base blue round">
  This will always have a blue background and
  round corners
</div>
<div class="base blue round">
  This is blue
  and round
</div>
</style-sample-app>
<!-- Our app loads here -->
</div>
<!-- Code injected by live-server -->
html body div.ui.main.text.container style-sample-app div.base.blue.round
```

Classes from both the attribute and directive

NgFor

The role of this directive is to **repeat a given DOM element** (or a collection of DOM elements), each time passing it a different value from an array.

 This directive is the successor of ng1's ng-repeat.

The syntax is `*ngFor="let item of items"`.

- The `let item` syntax specifies a (template) variable that's receiving each element of the `items` array;
- The `items` is the collection of items from your controller.

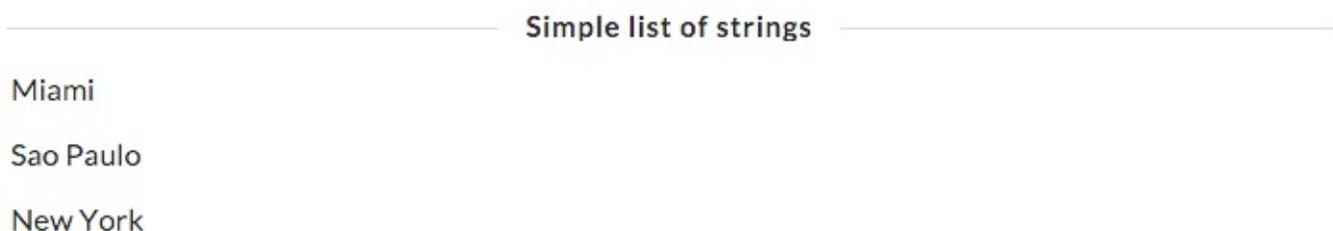
To illustrate, we can take a look at the code example. We declare an array of cities on our component controller:

```
1 this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

And then, in our template we can have the following HTML snippet:

```
code/built_in_components/ng_for/app.ts
7 <h4 class="ui horizontal divider header">
8   Simple list of strings
9 </h4>
10
11 <div class="ui list" *ngFor="let c of cities">
12   <div class="item">{{ c }}</div>
13 </div>
```

And it will render each city inside the div as you would expect:



We can also iterate through an array of objects like these:

code/built_in_components/ng_for/app.ts

```
71   this.people = [
72     { name: 'Anderson', age: 35, city: 'Sao Paulo' },
73     { name: 'John', age: 12, city: 'Miami' },
74     { name: 'Peter', age: 22, city: 'New York' }
75   ];
```

And then render a table based on each row of data:

code/built_in_components/ng_for/app.ts

```
15   <h4 class="ui horizontal divider header">
16     List of objects
17   </h4>
18
19   <table class="ui celled table">
20     <thead>
21       <tr>
22         <th>Name</th>
23         <th>Age</th>
24         <th>City</th>
25       </tr>
26     </thead>
27     <tr *ngFor="let p of people">
28       <td>{{ p.name }}</td>
29       <td>{{ p.age }}</td>
30       <td>{{ p.city }}</td>
31     </tr>
32   </table>
```

Getting the following result:

List of objects

Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

Rendering array of objects

We can also work with nested arrays. If we wanted to have the same table as above, broken down by city, we could easily declare a new array of objects:

code/built_in_components/ng_for/app.ts

```
76   this.peopleByCity = [
77     {
78       city: 'Miami',
79       people: [
80         { name: 'John', age: 12 },
81         { name: 'Angel', age: 22 }
82       ]
83     },
84     {
85       city: 'Sao Paulo',
```

```
86     people: [  
87       { name: 'Anderson', age: 35 },  
88       { name: 'Felipe', age: 36 }  
89     ]  
90   }  
91 ];  
92 };
```

And then we could use NgFor to render one h2 for each city:

code/built_in_components/ng_for/app.ts

```
38 <div *ngFor="let item of peopleByCity">  
39   <h2 class="ui header">{{ item.city }}</h2>
```

And use a nested directive to iterate through the people for a given city:

code/built_in_components/ng_for/app.ts

```
41 <table class="ui celled table">  
42   <thead>  
43     <tr>  
44       <th>Name</th>  
45       <th>Age</th>  
46     </tr>  
47   </thead>  
48   <tr *ngFor="let p of item.people">  
49     <td>{{ p.name }}</td>  
50     <td>{{ p.age }}</td>  
51   </tr>  
52 </table>
```

Resulting in the following template code:

code/built_in_components/ng_for/app.ts

```
34 <h4 class="ui horizontal divider header">  
35   Nested data  
36 </h4>  
37  
38 <div *ngFor="let item of peopleByCity">  
39   <h2 class="ui header">{{ item.city }}</h2>  
40  
41   <table class="ui celled table">  
42     <thead>  
43       <tr>  
44         <th>Name</th>  
45         <th>Age</th>  
46       </tr>  
47     </thead>  
48     <tr *ngFor="let p of item.people">  
49       <td>{{ p.name }}</td>  
50       <td>{{ p.age }}</td>  
51     </tr>  
52   </table>  
53 </div>
```

And it would render one table for each city:

Miami

Name	Age
John	12
Angel	22

Sao Paulo

Name	Age
Anderson	35
Felipe	36

Rendering nested arrays

Getting an index

There are times that we need the index of each item when we're iterating an array.

We can get the index by appending the syntax `let idx = index` to the value of our `ngFor` directive, separated by a semi-colon. When we do this, `ng2` will assign the current index into the variable we provide (in this case, the variable `idx`).



Note that, like JavaScript, the index is always zero based. So the index for first element is 0, 1 for the second and so on...

Making some changes to our first example, adding the `let num = index` snippet like below:

code/built_in_components/ng_for/app.ts

```
59 <div class="ui list" *ngFor="let c of cities; let num = index">
60   <div class="item">{{ num+1 }} - {{ c }}</div>
61 </div>
```

It will add the position of the city before the name, like this:

- 1 - Miami
- 2 - Sao Paulo
- 3 - New York

Using an index

NgNonBindable

We use `ngNonBindable` when we want tell Angular **not** to compile or bind a particular section of our page.

Let's say we want to render the literal text `{{ content }}` in our template. Normally that text will be *bound* to the value of the `content` variable because we're using the `{{ }}` template syntax.

So how can we render the exact text `{{ content }}`? We use the `ngNonBindable` directive.

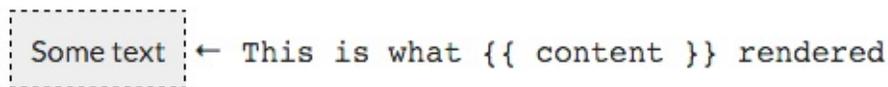
Let's say we want to have a `div` that renders the contents of that `content` variable and right after we want to point that out by outputting `<- this is what {{ content }} rendered` next to the actual value of the variable.

To do that, here's the template we'd have to use:

code/built_in_components/ng_non_bindable/app.ts

```
7 <div>
8   <span class="bordered">{{ content }}</span>
9   <span class="pre" ngNonBindable>
10     &larr; This is what {{ content }} rendered
11   </span>
12 </div>
```

And with that `ngNonBindable` attribute, `ng2` will not compile within that second `span`'s context, leaving it intact:



Some text ← This is what {{ content }} rendered

Result of using `ngNonBindable`

Conclusion

Angular 2 has only a few core directives, but we can combine these simple pieces to create dynamic apps.

Forms in Angular 2

Forms are Crucial, Forms are Complex

Forms are probably the most crucial aspect of your web application. While we often get events from clicking on links or moving the mouse, it's through *forms* where we get the majority of our rich data input from users.

On the surface, forms seem straightforward: you make an `input` tag, the user fills it out, and hits submit. How hard could it be?

It turns out, forms can end up being really complex. Here's a few reasons why:

- Form inputs are meant to modify data, both on the page and the server
- Changes often need to be reflected elsewhere on the page
- Users have a lot of leeway in what they enter, so you need to validate values
- The UI needs to clearly state expectations and errors, if any
- Dependent fields can have complex logic
- We want to be able to test our forms, without relying on DOM selectors

Thankfully, Angular 2 has tools to help with all of these things.

- **FormControls** encapsulate the inputs in our forms and give us objects to work with them
- **Validators** give us the ability to validate inputs, any way we'd like
- **Observers** let us watch our form for changes and respond accordingly

In this chapter we're going to walk through building forms, step by step. We'll start with some simple forms and build up to more complicated logic.

FormControls and FormGroups

The two fundamental objects in ng2 forms are `FormControl` and `FormGroup`.

FormControl

A `FormControl` represents a single input field - it is the smallest unit of an Angular form.

`FormControls` encapsulate the field's value, and states such as if it is valid, dirty (changed), or has errors.

For instance, here's how we might use a `FormControl` in TypeScript:

```
1 // create a new FormControl with the value "Nate"
2 let nameControl = new FormControl("Nate");
3
4 let name = nameControl.value; // -> Nate
5
6 // now we can query this control for certain values:
```

```
7 nameControl.errors // -> StringMap<string, any> of errors
8 nameControl.dirty  // -> false
9 nameControl.valid  // -> true
10 // etc.
```

To build up forms we create `FormControls` (and groups of `FormControls`) and then attach metadata and logic to them.

Like many things in Angular, we have a class (`FormControl`, in this case) that we attach to the DOM with an attribute (`formControl`, in this case). For instance, we might have the following in our form:

```
1 <!-- part of some bigger form -->
2 <input type="text" [formControl]="name" />
```

This will create a new `FormControl` object within the context of our form. We'll talk more about how that works below.

FormGroup

Most forms have more than one field, so we need a way to manage multiple `FormControls`. If we wanted to check the validity of our form, it's cumbersome to iterate over an array of `FormControls` and check each `FormControl` for validity. `FormGroups` solve this issue by providing a wrapper interface around a collection of `FormControls`.

Here's how you create a `FormGroup`:

```
1 let personInfo = new FormGroup({
2   firstName: new FormControl("Nate"),
3   lastName:  new FormControl("Murray"),
4   zip:      new FormControl("90210")
5 })
```

`FormGroup` and `FormControl` have a common ancestor ([AbstractControl](#)). That means we can check the status or value of `personInfo` just as easily as a single `FormControl`:

```
1 personInfo.value; // -> {
2   //   firstName: "Nate",
3   //   lastName:  "Murray",
4   //   zip:      "90210"
5 // }
6
7 // now we can query this control group for certain values, which have sensible
8 // values depending on the children FormControl's values:
9 personInfo.errors // -> StringMap<string, any> of errors
10 personInfo.dirty  // -> false
11 personInfo.valid  // -> true
12 // etc.
```

Notice that when we tried to get the value from the `FormGroup` we received an **object** with key-value pairs. This is a really handy way to get the full set of values from our form without having to iterate over each `FormControl` individually.

Our First Form

There are lots of moving pieces to create a form, and several important ones we haven't touched on. Let's jump in to a full example and I'll explain each piece as we go along.



You can find the full code listing for this section in the code download under `forms/`

Here's a screenshot of the very first form we're going to build:

The screenshot shows a web form with the title "Demo Form: Sku". Below the title is a label "SKU" followed by a text input field containing the value "SKU". At the bottom left of the form is a "Submit" button.

Demo Form with Sku: Simple Version

In our imaginary application we're creating an e-commerce-type site where we're listing products for sale. In this app we need to store the product's SKU, so let's create a simple form that takes the SKU as the only input field.



SKU is an abbreviation for "stockkeeping unit". It's a term for a unique id for a product that is going to be tracked in inventory. When we talk about a SKU, we're talking about a human-readable item ID.

Our form is super simple: we have a single input for sku (with a label) and a submit button.

Let's turn this form into a Component. If you recall, there are three parts to defining a component:

- Configure the `@Component()` annotation
- Create the template
- Implement custom functionality in the component definition class

Let's take these in turn:

Simple SKU Form: @Component Annotation

In order to use the new forms library we need to first make sure we bootstrap our app to use the forms library. To do this, we do the following in our `app.ts` where we bootstrap the app:

```
1 import { bootstrap } from '@angular/platform-browser-dynamic';
2 import { provideForms } from '@angular/forms';
3 //
4 // further down...
5 //
6 bootstrap(FormsDemoApp, [
7   provideForms()
8 ])
9 .catch((err: any) => console.error(err));
```

This ensures that we're able to use a set of directives named `FORM_DIRECTIVES` in our views. `FORM_DIRECTIVES` is a constant that Angular provides for us as a shorthand to several directives that are all useful in a form. `FORM_DIRECTIVES` includes:

- `formControl`
- `ngFormGroup`
- `ngForm`
- `ngModel`

... and several more. We haven't talked about how to use these directives or what they do, but we will shortly. For now, just know that by injecting `FORM_DIRECTIVES`, that means we can *use any of the directives in that list* in our view template.

Now we can start creating our component:

code/forms/app/forms/demo_form_sku.ts

```
1 import { Component } from '@angular/core';
2
3
4 @Component({
5   selector: 'demo-form-sku',
```

Here we define a selector of `demo-form-sku`. If you recall, `selector` tells Angular what elements this component will bind to. In this case we can use this component by having a `demo-form-sku` tag like so:

```
1 <demo-form-sku></demo-form-sku>
```

Simple SKU Form: template

Let's look at our template:

code/forms/app/ts/forms/demo_form_sku.ts

```
4 @Component({
5   selector: 'demo-form-sku',
6
7   template: `
8     <div class="ui raised segment">
9       <h2 class="ui header">Demo Form: Sku</h2>
10      <form #f="ngForm"
11        (ngSubmit)="onSubmit(f.value)"
12        class="ui form">
13
14        <div class="field">
15          <label for="skuInput">SKU</label>
16          <input type="text"
17            id="skuInput"
18            placeholder="SKU"
19            name="sku" ngModel>
20
21        </div>
22
23        <button type="submit" class="ui button">Submit</button>
24      </form>
25    </div>
26  `)
```

form & NgForm

Now things get interesting: because we injected `FORM_DIRECTIVES`, that makes `NgForm` available to our view. Remember that whenever we make directives available to our view, they will **get attached to any**

element that matches their selector.

NgForm does something handy but **non-obvious**: it includes the form tag in its selector (instead of requiring you to explicitly add ngForm as an attribute). What this means is that if you inject FORM_DIRECTIVES, NgForm will get *automatically* attached to any <form> tags you have in your view. This is really useful but potentially confusing because it happens behind the scenes.

There are two important pieces of functionality that NgForm gives us:

1. A FormGroup named ngForm
2. A (**ngSubmit**) output

You can see that we use both of these in the <form> tag in our view:

code/forms/app/ts/forms/demo_form_sku.ts

```
10 <form #f="ngForm"  
11     (ngSubmit)="onSubmit(f.value)"
```

First we have #f="ngForm". The #v=thing syntax says that we want to create a local variable for this view.

Here we're creating an alias to ngForm, for this view, bound to the variable #f. Where did ngForm come from in the first place? It came from the NgForm directive.

And what type of object is ngForm? It is a FormGroup. That means we can use f as a FormGroup in our view. And that's exactly what we do in the (ngSubmit) output.



Astute readers might notice that I just said above that NgForm is automatically attached to <form> tags (because of the default NgForm selector), which means we don't have to add an ngForm attribute to use NgForm. But here we're putting ngForm in an attribute (value) tag. Is this a typo?

No, it's not a typo. If ngForm were the *key* of the attribute then we would be telling Angular that we want to use NgForm on this attribute. In this case, we're using ngForm as the *attribute* when we're assigning a *_reference_*. That is, we're saying the value of the evaluated expression ngForm should be assigned to a local template variable f`.

ngForm is already on this element and you can think of it as if we are "exporting" this FormGroup so that we can reference it elsewhere in our view.

We bind to the ngSubmit action of our form by using the syntax: (ngSubmit)="onSubmit(f.value)".

- (ngSubmit) - comes from NgForm
- onSubmit() - will be implemented in our component definition class (below)
- f.value - f is the FormGroup that we specified above. And .value will return the key/value pairs of this FormGroup

Put it all together and that line says "when I submit the form, call onSubmit on my component instance, passing the value of the form as the arguments".

input & NgModel

Our input tag has a few things we should touch on before we talk about NgModel:

code/forms/app/ts/forms/demo_form_sku.ts

```
10 <form #f="ngForm"  
11   (ngSubmit)="onSubmit(f.value)"  
12   class="ui form">  
13  
14   <div class="field">  
15     <label for="skuInput">SKU</label>  
16     <input type="text"  
17       id="skuInput"  
18       placeholder="SKU"  
19       name="sku" ngModel>  
20   </div>
```

- `class="ui form"` and `class="field"` - these two classes are totally optional. They come from the [CSS framework Semantic UI](#). I've added them in some of our examples just to give them a nice coat of CSS but they're not part of Angular.
- The `label "for"` attribute and the `input "id"` attribute are to match, as [per W3C standard](#)
- We set a `placeholder` of "SKU", which is just a hint to the user for what this input should say when it is blank

The `NgModel` directive specifies a selector of `ngModel`. This means we can attach it to our input tag by adding this sort of attribute: `ngModel="whatever"`. In this case, we specify `ngModel` with no attribute value.

There are a couple of different ways to specify `ngModel` in your templates and this is the first. When we use `ngModel` with no attribute value we are specifying:

1. a *one-way* data binding
2. we want to create a `FormControl` on this form with the name `sku` (because of the name attribute on the input tag)

`NgModel` **creates a new `FormControl`** that is **automatically added** to the parent `FormGroup` (in this case, on the form) and then binds a DOM element to that new `FormControl`. That is, it sets up an association between the input tag in our view and the `FormControl` and the association is matched by a name, in this case "sku".



NgModel vs. ngModel: what's the difference? Generally, when we use PascalCase, like `NgModel`, we're specifying the *class* and referring to the object as it's defined in code. The lower case (CamelCase), as in `ngModel`, comes from the *selector* of the directive and it's only used in the DOM / template.

It's also worth pointing out that `NgModel` and `FormControl` are separate objects. `NgModel` is the *directive* that you use in your view, whereas `FormControl` is the object used for representing the data and validations in your form.



Sometimes we want to do *two-way* binding with `ngModel` like we used to do in Angular 1. We'll look at how to do that towards the end of this chapter.

Simple SKU Form: Component Definition Class

Now let's look at our class definition:

code/forms/app/ts/forms/demo_form_sku.ts

```
27 export class DemoFormSku {
28   onSubmit(form: any): void {
29     console.log('you submitted value:', form);
30   }
31 }
```

Here our class defines one function: `onSubmit`. This is the function that is called when the form is submitted. For now, we'll just `console.log` out the value that is passed in.

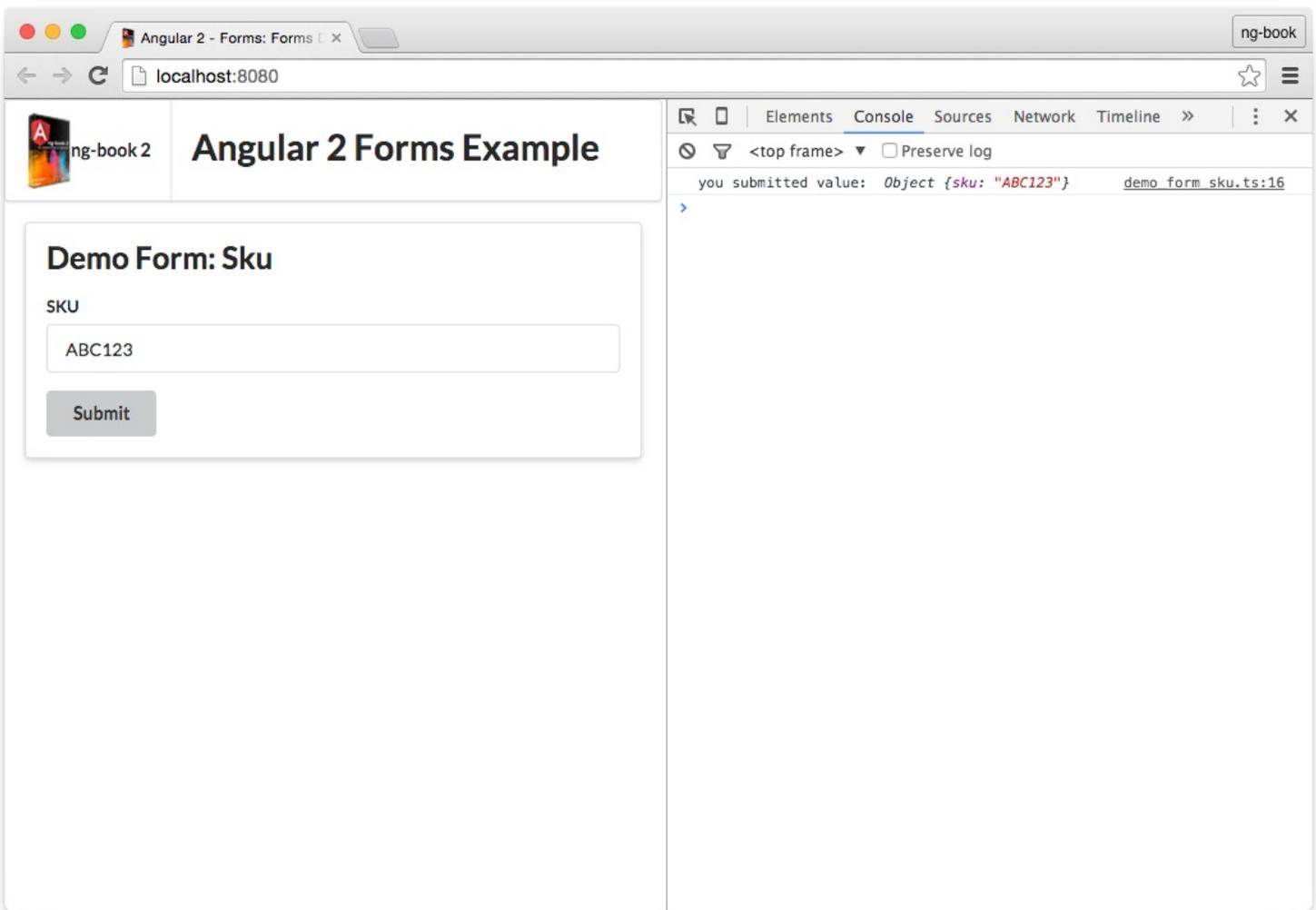
Try it out!

Putting it all together, here's what our code listing looks like:

code/forms/app/ts/forms/demo_form_sku.ts

```
1 import { Component } from '@angular/core';
2
3
4 @Component({
5   selector: 'demo-form-sku',
6
7   template: `
8     <div class="ui raised segment">
9       <h2 class="ui header">Demo Form: Sku</h2>
10      <form #f="ngForm"
11        (ngSubmit)="onSubmit(f.value)"
12        class="ui form">
13
14        <div class="field">
15          <label for="skuInput">SKU</label>
16          <input type="text"
17            id="skuInput"
18            placeholder="SKU"
19            name="sku" ngModel>
20        </div>
21
22        <button type="submit" class="ui button">Submit</button>
23      </form>
24    </div>
25  `
26 })
27 export class DemoFormSku {
28   onSubmit(form: any): void {
29     console.log('you submitted value:', form);
30   }
31 }
```

If we try this out in our browser, here's what it looks like:



Demo Form with Sku: Simple Version, Submitted

Using FormBuilder

Building our `FormControls` and `FormGroups` implicitly using `ngForm` and `ngControl` is convenient, but doesn't give us a lot of customization options. A more flexible and common way to configure forms is to use a `FormBuilder`.

`FormBuilder` is an aptly-named helper class that helps us build forms. As you recall, forms are made up of `FormControls` and `FormGroups` and the `FormBuilder` helps us make them (you can think of it as a "factory" object).

Let's add a `FormBuilder` to our previous example. Let's look at:

- how to use the `FormBuilder` in our component definition class
- how to use our custom `FormGroup` on a form in the view

Injecting `REACTIVE_FORM_DIRECTIVES`

For this component we're going to be using the `formGroup` and `formControl` directives which means we need to import `REACTIVE_FORM_DIRECTIVES` like so:

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
1 import { Component } from '@angular/core';
2 import {
3   FORM_DIRECTIVES,
4   REACTIVE_FORM_DIRECTIVES,
5   FormBuilder,
6   FormGroup
7 } from '@angular/forms';
8
9 @Component({
10  selector: 'demo-form-sku-builder',
11  directives: [FORM_DIRECTIVES, REACTIVE_FORM_DIRECTIVES],
```

Using FormBuilder

We inject FormBuilder by creating an argument in the constructor of our component class:



What does inject mean? We haven't talked much about dependency injection (DI) or how DI relates to the hierarchy tree, so that last sentence may not make a lot of sense. We talk a lot more about dependency injection in [the Dependency Injection chapter](#), so go there if you'd like to learn more about it in depth.

At a high level, Dependency Injection is a way to tell Angular what dependencies this component needs to function properly.

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
31 })
32 export class DemoFormSkuBuilder {
33   myForm: FormGroup;
34
35   constructor(fb: FormBuilder) {
36     this.myForm = fb.group({
37       'sku': ['ABC123']
38     });
39   }
40
41   onSubmit(value: string): void {
42     console.log('you submitted value: ', value);
43   }
```

During injection an instance of FormBuilder will be created and we assign it to the fb variable (in the constructor).

There are two main functions we'll use on FormBuilder:

- control - creates a new FormControl
- group - creates a new FormGroup

Notice that we've setup a new *instance variable* called myForm on this class. (We could have just as easily called it form, but I want to differentiate between our FormGroup and the form we had before.)

myForm is typed to be a FormGroup. We create a FormGroup by calling fb.group(). group takes an object of key-value pairs that specify the FormControls in this group.

In this case, we're setting up one control sku, and the value is ["ABC123"] - this says that the default value of this control is "ABC123". (You'll notice that is an array. That's because we'll be adding more configuration options there later.)

Now that we have `myForm` we need to use that in the view (i.e. we need to *bind* it to our form element).

Using `myForm` in the view

We want to change our `<form>` to use `myForm`. If you recall, in the last section we said that `ngForm` is applied for us automatically when we used `FORM_DIRECTIVES`. We also mentioned that `ngForm` creates its own `FormGroup`. Well, in this case, we **don't** want to use an outside `FormGroup`. Instead we want to use our instance variable `myForm`, which we created with our `FormBuilder`. How can we do that?

Angular provides another directive that we use **when we have an existing `FormGroup`**: it's called `formGroup` and we use it like this:

```
code/forms/app/ts/forms/demo_form_sku_with_builder.ts
```

```
14 <h2 class="ui header">Demo Form: Sku with Builder</h2>
15 <form [formGroup]="myForm">
```

Here we're telling Angular that we want to use `myForm` as the `FormGroup` for this form.



Remember how earlier we said that when using `FORM_DIRECTIVES` that `NgForm` will be automatically applied to a `<form>` element? There is an exception: `NgForm` won't be applied to a `<form>` that has `formGroup`.

If you're curious, the selector for `NgForm` is:

```
1 form:not([ngNoForm]):not([formGroup]),ngForm,[ngForm]
```

This means you *could* have a form that doesn't get `NgForm` applied by using the `ngNoForm` attribute.

We also need to change `onSubmit` to use `myForm` instead of `f`, because now it is `myForm` that has our configuration and values.

There's one last thing we need to do to make this work: bind our `FormControl` to the `input` tag. Remember that `ngControl` creates a new **`FormControl` object**, and attaches it to the parent `FormGroup`. But in this case, we used `FormBuilder` to create our own `FormControls`.

When we want to bind an **existing `FormControl`** to an input we use `formControl`:

```
code/forms/app/ts/forms/demo_form_sku_with_builder.ts
```

```
20 <label for="skuInput">SKU</label>
21 <input type="text"
22       id="skuInput"
23       placeholder="SKU">
```

Here we are instructing the `formControl` directive to look at `myForm.controls` and use the existing `sku FormControl` for this input.

Try it out!

Here's what it looks like all together:

```
code/forms/app/ts/forms/demo_form_sku_with_builder.ts
```

```

1 import { Component } from '@angular/core';
2 import {
3   FORM_DIRECTIVES,
4   REACTIVE_FORM_DIRECTIVES,
5   FormBuilder,
6   FormGroup
7 } from '@angular/forms';
8
9 @Component({
10  selector: 'demo-form-sku-builder',
11  directives: [FORM_DIRECTIVES, REACTIVE_FORM_DIRECTIVES],
12  template: `
13    <div class="ui raised segment">
14      <h2 class="ui header">Demo Form: Sku with Builder</h2>
15      <form [formGroup]="myForm"
16        (ngSubmit)="onSubmit(myForm.value)"
17        class="ui form">
18
19        <div class="field">
20          <label for="skuInput">SKU</label>
21          <input type="text"
22            id="skuInput"
23            placeholder="SKU"
24            [formControl]="myForm.controls['sku']">
25
26        </div>
27
28        <button type="submit" class="ui button">Submit</button>
29      </form>
30    </div>
31  `
32 })
33 export class DemoFormSkuBuilder {
34   myForm: FormGroup;
35
36   constructor(fb: FormBuilder) {
37     this.myForm = fb.group({
38       'sku': ['ABC123']
39     });
40   }
41
42   onSubmit(value: string): void {
43     console.log('you submitted value: ', value);
44   }
45 }

```

Remember:

To create a new FormGroup and FormControl implicitly use:

- ngForm and
- ngModel

But to bind to an existing FormGroup and FormControl use:

- formGroup and
- formControl

Adding Validations

Our users aren't always going to enter data in exactly the right format. If someone enters data in the wrong format, we want to give them feedback and not allow the form to be submitted. For this we use *validators*.

Validators are provided by the `validators` module and the simplest validator is `Validators.required` which simply says that the designated field is required or else the `FormControl` will be considered

invalid.

To use validators we need to do two things:

1. Assign a validator to the `FormControl` object
2. Check the status of the validator in the view and take action accordingly

To assign a validator to a `FormControl` object we simply pass it as the second argument to our `FormControl` constructor:

```
1 let control = new FormControl('sku', Validators.required);
```

Or in our case, because we're using `FormBuilder` we will use the following syntax:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
47 constructor(fb: FormBuilder) {  
48   this.myForm = fb.group({  
49     'sku': ['', Validators.required]  
49   });  
}
```

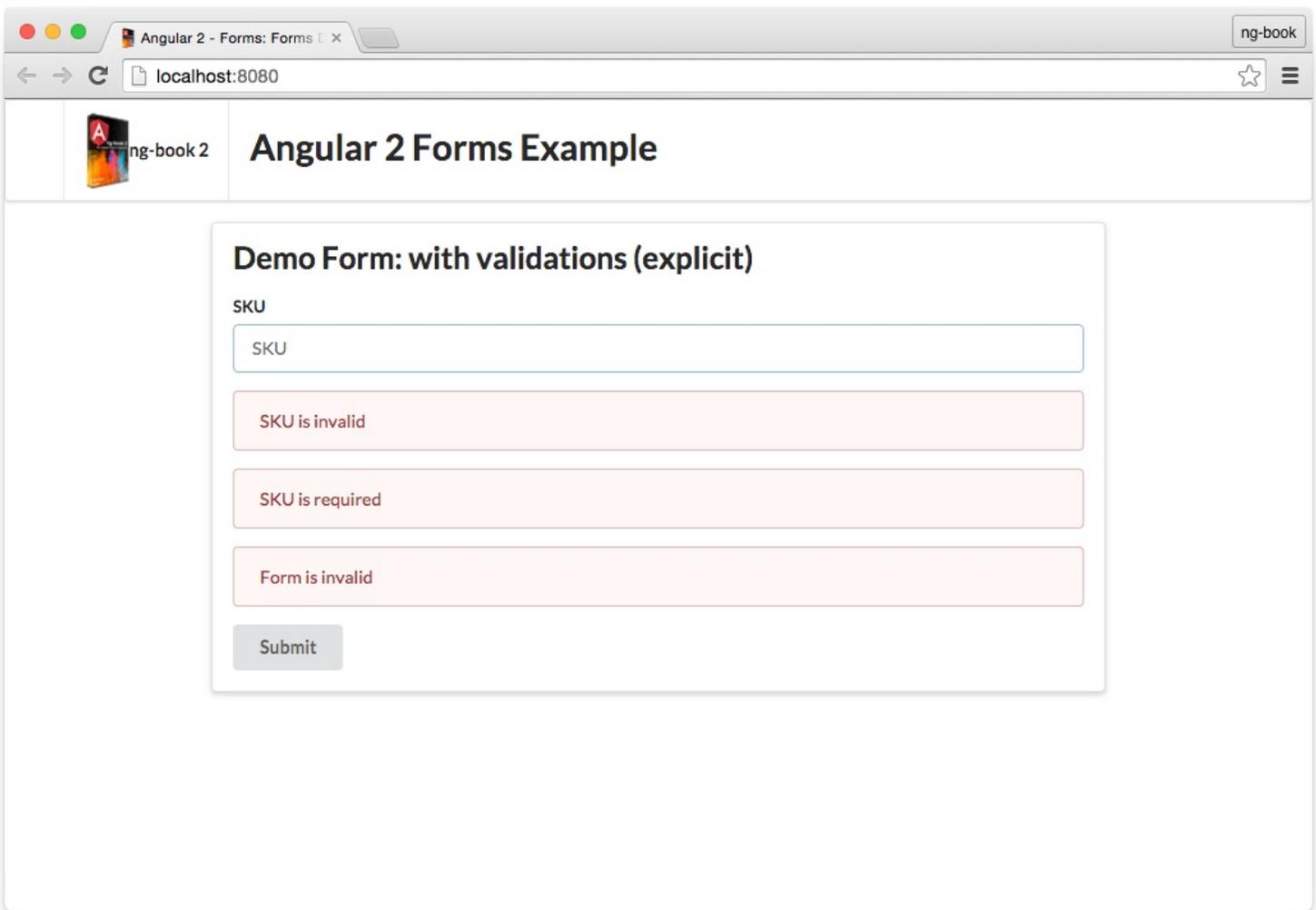
Now we need to use our validation in the view. There are two ways we can access the validation value in the view:

1. We can explicitly assign the `FormControl` `sku` to an instance variable of the class - which is more verbose, but gives us easy access to the `FormControl` in the view.
2. We can lookup the `FormControl` `sku` from `myForm` in the view. This requires less work in the component definition class, but is slightly more verbose in the view.

To make this difference clearer, let's look at this example both ways:

Explicitly setting the `sku FormControl` as an instance variable

Here's a screenshot of what our form is going to look like with validations:



Demo Form with Validations

The most flexible way to deal with individual `FormControls` in your view is to set each `FormControl` up as an instance variable in your component definition class. Here's how we could setup `sku` in our class:

[code/forms/app/ts/forms/demo_form_with_validations_explicit.ts](#)

```
42 })
43 export class DemoFormWithValidationsExplicit {
44   myForm: FormGroup;
45   sku: AbstractControl;
46
47   constructor(fb: FormBuilder) {
48     this.myForm = fb.group({
49       'sku': ['', Validators.required]
50     });
51
52     this.sku = this.myForm.controls['sku'];
53   }
54
55   onSubmit(value: string): void {
56     console.log('you submitted value: ', value);
57   }
}
```

Notice that:

1. We setup `sku: AbstractControl` at the top of the class and
2. We assign `this.sku` after we've created `myForm` with the `FormBuilder`

This is great because it means we can reference sku anywhere in our component view. The downside is that by doing it this way, we'd have to setup an instance variable **for every field in our form**. For large forms, this can get pretty verbose.

Now that we have our sku being validated, I want to look at four different ways we can use it in our view:

1. Checking the validity of our whole form and displaying a message
2. Checking the validity of our individual field and displaying a message
3. Checking the validity of our individual field and coloring the field red if it's invalid
4. Checking the validity of our individual field on a particular requirement and displaying a message

Form message

We can check the validity of our whole form by looking at `myForm.valid`:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
34 <div *ngIf="!myForm.valid"
```

Remember, `myForm` is a `FormGroup` and a `FormGroup` is valid if all of the children `FormControls` are also valid.

Field message

We can also display a message for the specific field if that field's `FormControl` is invalid:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
28 [formControl]="sku">
29 <div *ngIf="!sku.valid"
```

Field coloring

I'm using the Semantic UI CSS Framework's CSS class `.error`, which means if I add the class `error` to the `<div class="field">` it will show the input tag with a red border.

To do this, we can use the property syntax to set conditional classes:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
21 <div class="field"
22 [class.error]="!sku.valid && sku.touched">
```

Notice here that we have two conditions for setting the `.error` class: We're checking for `!sku.valid` and `sku.touched`. The idea here is that we only want to show the error state if the user has tried editing the form ("touched" it) and it's now invalid.

To try this out, enter some data into the `input` tag and then delete the contents of the field.

Specific validation

A form field can be invalid for many reasons. We often want to show a different message depending on the reason for a failed validation.

To look up a specific validation failure we use the `hasError` method:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
30     class="ui error message">SKU is invalid</div>
31 <div *ngIf="sku.hasError('required')"
```

Note that `hasError` is defined on both `FormControl` and `FormGroup`. This means you can pass a second argument of path to lookup a specific field from `FormGroup`. For example, we could have written the previous example as:

```
1     <div *ngIf="myForm.hasError('required', 'sku')"
2         class="error">SKU is required</div>
```

Putting it together

Here's the full code listing of our form with validations with the `FormControl` set as an instance variable:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
1  /* tslint:disable:no-string-literal */
2  import { Component } from '@angular/core';
3  import {
4    FORM_DIRECTIVES,
5    REACTIVE_FORM_DIRECTIVES,
6    FormBuilder,
7    FormGroup,
8    Validators,
9    AbstractControl
10 } from '@angular/forms';
11
12 @Component({
13   selector: 'demo-form-with-validations-explicit',
14   directives: [FORM_DIRECTIVES, REACTIVE_FORM_DIRECTIVES],
15   template: `
16     <div class="ui raised segment">
17       <h2 class="ui header">Demo Form: with validations (explicit)</h2>
18       <form [formGroup]="myForm"
19           (ngSubmit)="onSubmit(myForm.value)"
20           class="ui form">
21
22         <div class="field"
23             [class.error]="!sku.valid && sku.touched">
24           <label for="skuInput">SKU</label>
25           <input type="text"
26               id="skuInput"
27               placeholder="SKU"
28               [formControl]="sku">
29           <div *ngIf="!sku.valid"
30               class="ui error message">SKU is invalid</div>
31           <div *ngIf="sku.hasError('required')"
32               class="ui error message">SKU is required</div>
33         </div>
34
35         <div *ngIf="!myForm.valid"
36             class="ui error message">Form is invalid</div>
37
38         <button type="submit" class="ui button">Submit</button>
39       </form>
40     </div>
41 `
42 })
43 export class DemoFormWithValidationsExplicit {
44   myForm: FormGroup;
45   sku: AbstractControl;
46
47   constructor(fb: FormBuilder) {
48     this.myForm = fb.group({
49       'sku': ['', Validators.required]
50     });
51
52     this.sku = this.myForm.controls['sku'];
53   }
54
55   onSubmit(value: string): void {
56     console.log('you submitted value: ', value);
```

```
57 }  
58 }
```

Removing the sku instance variable

In the example above we set `sku: FormControl` as an instance variable. We often won't want to create an instance variable for each `FormControl`, so how would we reference this `FormControl` in our view without an instance variable?

Instead we can use the `myForm.controls` property as in:

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```
23 <input type="text"  
24       id="skuInput"  
25       placeholder="SKU"  
26       [FormControl]="myForm.controls['sku']">  
27 <div *ngIf="!myForm.controls['sku'].valid"  
28       class="ui error message">SKU is invalid</div>  
29 <div *ngIf="myForm.controls['sku'].hasError('required')"  
30       class="ui error message">SKU is required</div>
```

In this way we can access the sku control without being forced to explicitly add it as an instance variable on the component class.

Custom Validations

We often are going to want to write our own custom validations. Let's take a look at how to do that.

To see how validators are implemented, let's look at `Validators.required` from the Angular core source:

```
1 export class Validators {  
2   static required(c: FormControl): StringMap<string, boolean> {  
3     return isNaN(c.value) || c.value == "" ? {"required": true} : null;  
4   }  
}
```

A validator: - Takes a `FormControl` as its input and - Returns a `StringMap<string, boolean>` where the key is "error code" and the value is true if it fails

Writing the Validator

Let's say we have specific requirements for our sku. For example, say our sku needs to begin with 123. We could write a validator like so:

code/forms/app/ts/forms/demo_form_with_custom_validations.ts

```
21 function skuValidator(control: FormControl): { [s: string]: boolean } {  
22   if (!control.value.match(/^123/)) {  
23     return {invalidSku: true};  
24   }  
25 }
```

This validator will return an error code `invalidSku` if the input (the `control.value`) does not begin with 123.

Assigning the Validator to the FormControl

Now we need to add the validator to our `FormControl`. However, there's one small problem: we already have a validator on sku. How can we add multiple validators to a single field?

For that, we use `Validators.compose`:

```
code/forms/app/ts/forms/demo_form_with_custom_validations.ts
```

```
64     this.myForm = fb.group({
65       'sku': ['', Validators.compose([
66         Validators.required, skuValidator])]
67     });
```

`Validators.compose` wraps our two validators and lets us assign them both to the `FormControl`. The `FormControl` is not valid unless both validations are valid.

Now we can use our new validator in the view:

```
code/forms/app/ts/forms/demo_form_with_custom_validations.ts
```

```
48     <div *ngIf="sku.hasError('invalidSku')">
49       <span class="ui error message">SKU must begin with <tt>123</tt></div>
```

 Note that in this section, I'm using “explicit” notation of adding an instance variable for each `FormControl`. That means that in the view in this section, `sku` refers to a `FormControl`.

If you run the sample code, one neat thing you'll notice is that if you type something in to the field, the required validation will be fulfilled, but the `invalidSku` validation may not. This is great - it means we can partially-validate our fields and show the appropriate messages.

Watching For Changes

So far we've only extracted the value from our form by calling `onSubmit` when the form is submitted. But often we want to watch for any value changes on a control.

Both `FormGroup` and `FormControl` have an `EventEmitter` that we can use to observe changes.

 `EventEmitter` is an *Observable*, which means it conforms to a defined specification for watching for changes. If you're interested in the *Observable* spec, [you can find it here](#)

To watch for changes on a control we:

1. get access to the `EventEmitter` by calling `control.valueChanges`. Then we
2. add an *observer* using the `.observer` method

Here's an example:

```
code/forms/app/ts/forms/demo_form_with_events.ts
```

```
48     this.myForm = fb.group({
49       'sku': ['', Validators.required]
50     });
51
52     this.sku = this.myForm.controls['sku'];
53
54     this.sku.valueChanges.subscribe(
```

```
55     (value: string) => {
56         console.log('sku changed to:', value);
57     }
58 );
59
60 this.myForm.valueChanges.subscribe(
61     (form: any) => {
62         console.log('Form changed to:', form);
63     }
64 );
```

Here we're observing two separate events: changes on the sku field and changes on the form as a whole.

The observable that we pass in is an object with a single key: `next` (there are other keys you can pass in, but we're not going to worry about those now). `next` is the function we want to call with the new value whenever the value changes.

If we type 'kj' into the text box we will see in our console:

```
1 sku changed to: k
2 form changed to: Object {sku: "k"}
3 sku changed to: kj
4 form changed to: Object {sku: "kj"}
```

As you can see each keystroke causes the control to change, so our observable is triggered. When we observe the individual `FormControl` we receive a value (e.g. `kj`), but when we observe the whole form, we get an object of key-value pairs (e.g. `{sku: "kj"}`).

ngModel

`NgModel` is a special directive: it binds a model to a form. `ngModel` is special in that it **implements two-way data binding**. Two-way data binding is almost always more complicated and difficult to reason about vs. one-way data binding. Angular 2 is built to generally have data flow one-way: top-down. However, when it comes to forms, there are times where it is easier to opt-in to a two-way bind.



Just because you've used `ng-model` in Angular 1 in the past, don't rush to use `ngModel` right away. There are good reasons to avoid two-way data binding. Of course, `ngModel` can be really handy, but know that we don't necessarily rely on two-way data binding it as much as we did in Angular 1.

Let's change our form a little bit and say we want to input `productName`. We're going to use `ngModel` to keep the component instance variable in sync with the view.

First, here's our component definition class:

```
code/forms/app/ts/forms/demo_form_ng_model.ts
```

```
42 export class DemoFormNgModel {
43     myForm: FormGroup;
44     productName: string;
45
46     constructor(fb: FormBuilder) {
47         this.myForm = fb.group({
48             'productName': ['', Validators.required]
49         });
50     }
51
52     onSubmit(value: string): void {
```

```
53     console.log('you submitted value: ', value);
54   }
55 }
```

Notice that we're simply storing `productName: string` as an instance variable.

Next, let's use `ngModel` on our input tag:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
27     <input type="text"
28           id="productNameInput"
29           placeholder="Product Name"
30           [formControl]="myForm.find('productName')"
31           [(ngModel)]="productName">
```

Now notice something - the syntax for `ngModel` is funny: we are using both brackets and parenthesis around the `ngModel` attribute! The idea this is intended to invoke is that we're using both the *input* `[]` brackets and the *output* `()` parenthesis. It's an indication of the two-way bind.

Notice something else here: we're still using `formControl` to specify that this input should be bound to the `FormControl` on our form. We do this because `ngModel` is only binding the input to the instance variable - the `FormControl` is completely separate. But because we still want to validate this value and submit it as part of the form, we keep the `formControl` directive.

Last, let's display our `productName` value in the view:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
17     <div class="ui info message">
18       The product name is: {{productName}}
19     </div>
```

Here's what it looks like:

The screenshot shows a web browser window with the title "Angular 2 - Forms: Forms" and the URL "localhost:8080". The page content includes a header with "ng-book 2" and "Angular 2 Forms Example". Below the header is a "Demo Form: with ng-model" section. The form displays "The product name is: Blue Widget" in a light blue box. Below this is a "Product Name" label, a text input field containing "Blue Widget", and a "Submit" button. To the right of the form is a Chrome DevTools console window showing the following log entry:

```
you submitted value: Object {productName: "Blue Widget"} demo_form ng_model.ts:19
```

Demo Form with ngModel

Easy!

Wrapping Up

Forms have a lot of moving pieces, but Angular 2 makes it fairly straightforward. Once you get a handle on how to use `FormGroups`, `FormControls`, and `Validations`, it's pretty easy going from there!

Data Architecture in Angular 2

An Overview of Data Architecture

Managing data can be one of the trickiest aspects of writing a maintainable app. There are tons of ways to get data into your application:

- AJAX HTTP Requests
- Websockets
- Indexeddb
- LocalStorage
- Service Workers
- etc.

The problem of data architecture addresses questions like:

- How can we aggregate all of these different sources into a coherent system?
- How can we avoid bugs caused by unintended side-effects?
- How can we structure the code sensibly so that it's easier to maintain and on-board new team members?
- How can we make the app run as fast as possible when data changes?

For many years MVC was a standard pattern for architecting data in applications: the Models contained the domain logic, the View displayed the data, and the Controller tied it all together. The problem is, we've learned that MVC doesn't translate directly into client-side web applications very well.

There has been a renaissance in the area of data architectures and many new ideas are being explored. For instance:

- **MVW / Two-way data binding:** *Model-View-Whatever* is a term used¹ to describe Angular 1's default architecture. The `$scope` provides a two-way data-binding - the whole application shares the same data structures and a change in one area propagates to the rest of the app.
- **Flux:** uses a unidirectional data flow. In Flux, Stores hold data, Views render what's in the Store, and Actions change the data in the Store. There is a bit more ceremony to setup Flux, but the idea is that because data only flows in one direction, it's easier to reason about.
- **Observables:** Observables give us streams of data. We subscribe to the streams and then perform operations to react to changes. [RxJs](#) is the most popular reactive streams library for Javascript and it gives us powerful operators for composing operations on streams of data.



There are a lot of variations on these ideas. For instance:

- Flux is a pattern, and not an implementation. There are **many** different implementations of Flux (just like there are many implementations of MVC)
- Immutability is a common variant on all of the above data architectures.
- [Falcor](#) is a powerful framework that helps bind your client-side models to the server-side data. Falcor often used with an Observables-type data architecture.

Data Architecture in Angular 2

Angular 2 is extremely flexible in what it allows for data architecture. A data strategy that works for one project doesn't necessarily work for another. So Angular doesn't prescribe a particular stack, but instead tries to make it easy to use whatever architecture we choose (while still retaining fast performance).

The benefit of this is that you have flexibility to fit Angular into almost any situation. The downside is that you have to make your own decisions about what's right for your project.

Don't worry, we're not going to leave you to make this decision on your own! In the chapters that follow, we're going to cover how to build applications using some of these patterns.

Data Architecture with Observables - Part 1: Services

Observables and RxJS

In Angular, we can structure our application to use Observables as the backbone of our data architecture. Using Observables to structure our data is called *Reactive Programming*.

But what are Observables, and Reactive Programming anyway? Reactive Programming is a way to work with asynchronous streams of data. Observables are the main data structure we use to implement Reactive Programming. But I'll admit, those terms may not be that clarifying. So we'll look at concrete examples through the rest of this chapter that should be more enlightening.

Note: Some RxJS Knowledge Required

I want to point out **this book is not primarily about Reactive Programming**. There are several other good resources that can teach you the basics of Reactive Programming and you should read them. We've listed a few below.

Consider this chapter a tutorial on how to work with RxJS and Angular rather than an exhaustive introduction to RxJS and Reactive Programming.

In this chapter, I'll **explain in detail the RxJS concepts and APIs that we encounter**. But know that you may need to supplement the content here with other resources if RxJS is still new to you.



Use of Underscore.js in this chapter

[Underscore.js](#) is a popular library that provides functional operators on Javascript data structures such as Array and Object. We use it a bunch in this chapter alongside RxJS. If you see the `_` in code, such as `_.map` or `_.sortBy` know that we're using the Underscore.js library. You can find [the docs for Underscore.js here](#).

Learning Reactive Programming and RxJS

If you're just learning RxJS I recommend that you read this article first:

- [The introduction to Reactive Programming you've been missing](#) by Andre Staltz

After you've become a bit more familiar with the concepts behind RxJS, here are a few more links that can help you along the way:

- [Which static operators to use to create streams?](#)
- [Which instance operators to use on streams?](#)
- [RxMarbles](#) - Interactive diagrams of the various operations on streams

Throughout this chapter I'll provide links to the API documentation of RxJS. The RxJS docs have tons of great example code that shed light on how the different streams and operators work.



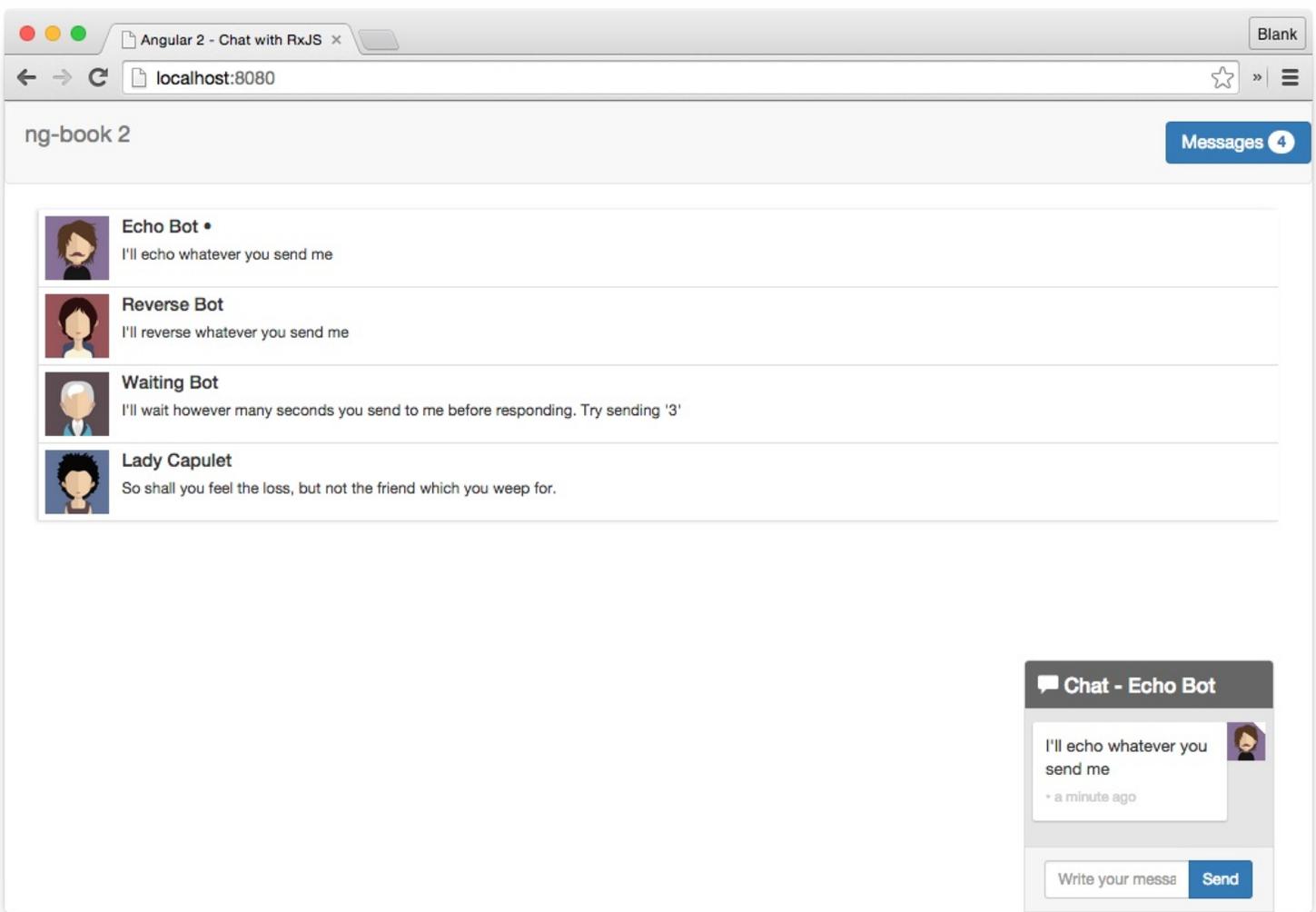
Do I have to use RxJS to use Angular 2? - No, you definitely don't. Observables are just one pattern out of many that you can use with Angular 2. We talk more about [other data patterns you can use here](#).

I want to give you fair warning: learning RxJS can be a bit mind-bending at first. But trust me, you'll get the hang of it and it's worth it. Here's a few big ideas about streams that you might find helpful:

1. **Promises emit a single value whereas streams emit many values.** - Streams fulfill the same role in your application as promises. If you've made the jump from callbacks to promises, you know that promises are a big improvement in readability and data maintenance vs. callbacks. In the same way, streams improve upon the promise pattern in that we can continuously respond to data changes on a stream (vs. a one-time resolve from a promise)
2. **Imperative code "pulls" data whereas reactive streams "push" data** - In Reactive Programming our code subscribes to be notified of changes and the streams "push" data to these subscribers
3. **RxJS is *functional*** - If you're a fan of functional operators like `map`, `reduce`, and `filter` then you'll feel right at home with RxJS because streams are, in some sense, lists and so the powerful functional operators all apply
4. **Streams are composable** - Think of streams like a pipeline of operations over your data. You can subscribe to any part of your stream and even combine them to create new streams

Chat App Overview

In this chapter, we're going to use RxJS to build a chat app. Here's a screenshot:



Completed Chat Application



Usually we try to show every line of code here in the book text. However, this chat application has a lot of moving parts, so in this chapter we're not going to have every single line of code in the text. You can find the sample code for this chapter in the folder `code/rxjs/chat`. We'll call out each filter where you can view the context, where appropriate.

In this application we've provided a few bots you can chat with. Open up the code and try it out:

```
1 cd code/rxjs/chat
2 npm install
3 npm run go
```

Now open your browser to `http://localhost:8080`.



If the above URL doesn't work, try this URL: `http://localhost:8080/webpack-dev-server/index.html`



Some Windows users may have trouble doing an `npm install` on this repo. If this causes problems for you, make sure you're running these commands inside [Cygwin](#).

Notice a few things about this application:

- You can click on the threads to chat with another person
- The bots will send you messages back, depending on their personality
- The unread message count in the top corner stays in sync with the number of unread messages

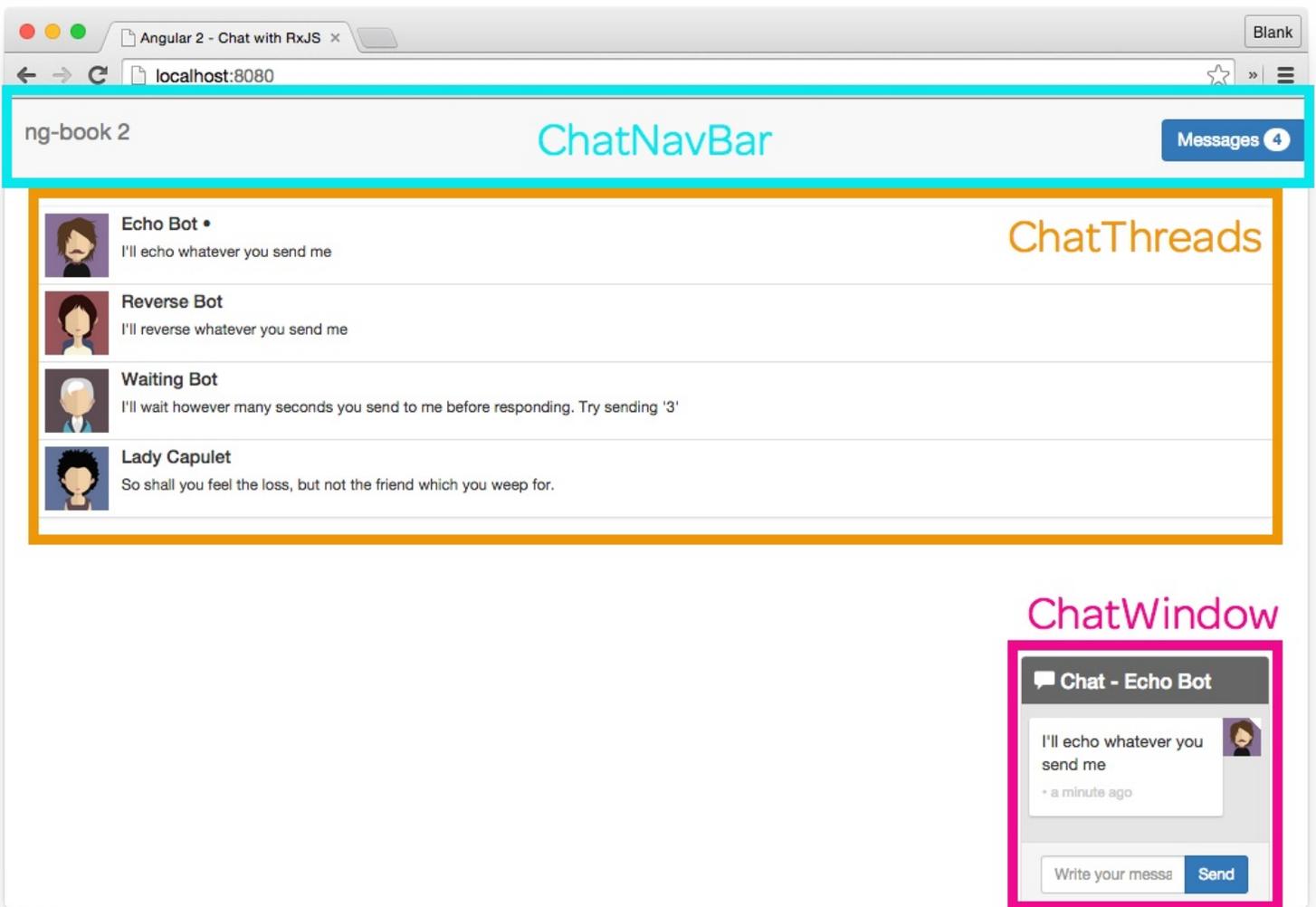
Let's look at an overview of how this app is constructed. We have

- 3 top-level Angular Components
- 3 models
- and 3 services

Let's look at them one at a time.

Components

The page is broken down into three top-level components:



Chat Top-Level Components

- ChatNavBar - contains the unread messages count
- ChatThreads - shows a clickable list of threads, along with the most recent message and the conversation avatar
- ChatWindow - shows the messages in the current thread with an input box to send new messages

Models

This application also has three models:



Chat Models

- User - stores information about a chat participant

- Message - stores an individual message
- Thread - stores a collection of Messages as well as some data about the conversation

Services

In this app, each of our models has a corresponding *service*. The services are singleton objects that play two roles:

1. **Provide streams** of data that our application can subscribe to
2. **Provide operations** to add or modify data

For instance, the `UserService`:

- publishes a stream that emits the current user and
- offers a `setCurrentUser` function which will set the current user (that is, emit the current user from the `currentUser` stream)

Summary

At a high level, the application data architecture is straightforward:

- The **services** maintain streams which emit models (e.g. Messages)
- The **components** subscribe to those streams and render according to the most recent values

For instance, the `ChatThreads` component listens for the most recent list of threads from the `ThreadService` and the `ChatWindow` subscribes for the most recent list of messages.

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular 2 and RxJS. We'll start by implementing our models, then look at how we create Services to manage our streams, and then finally implement the Components.

Implementing the Models

Let's start with the easy stuff and take a look at the models.

User

Our `User` class is straightforward. We have an `id`, `name`, and `avatarSrc`.

code/rxjs/chat/app/ts/models.ts

```
3 export class User {
4   id: string;
5
6   constructor(public name: string,
7               public avatarSrc: string) {
8     this.id = uuid();
9   }
10 }
```



Notice above that we're using a TypeScript shorthand in the constructor. When we say `public name: string` we're telling TypeScript that 1. we want `name` to be a public property on this class and 2. assign the argument value to that property when a new instance is created.

Thread

Similarly, `Thread` is also a straightforward TypeScript class:

`code/rxjs/chat/app/ts/models.ts`

```
12 export class Thread {
13   id: string;
14   lastMessage: Message;
15   name: string;
16   avatarSrc: string;
17
18   constructor(id?: string,
19               name?: string,
20               avatarSrc?: string) {
21     this.id = id || uuid();
22     this.name = name;
23     this.avatarSrc = avatarSrc;
24   }
25 }
```

Note that we store a reference to the `lastMessage` in our `Thread`. This lets us show a preview of the most recent message in the threads list.

Message

`Message` is also a simple TypeScript class, however in this case we use a slightly different form of constructor:

`code/rxjs/chat/app/ts/models.ts`

```
27 export class Message {
28   id: string;
29   sentAt: Date;
30   isRead: boolean;
31   author: User;
32   text: string;
33   thread: Thread;
34
35   constructor(obj?: any) {
36     this.id = obj && obj.id || uuid();
37     this.isRead = obj && obj.isRead || false;
38     this.sentAt = obj && obj.sentAt || new Date();
39     this.author = obj && obj.author || null;
40     this.text = obj && obj.text || null;
41     this.thread = obj && obj.thread || null;
42   }
43 }
```

The pattern you see here in the constructor allows us to simulate using keyword arguments in the constructor. Using this pattern, we can create a new `Message` using whatever data we have available and we don't have to worry about the order of the arguments. For instance we could do this:

```
1 let msg1 = new Message();
2
3 # or this
4
5 let msg2 = new Message({
```

```
6     text: "Hello Nate Murray!"
7   })
```

Now that we've looked at our models, let's take a look at our first service: the `UserService`.

Implementing `userService`

The point of the `UserService` is to provide a place where our application can learn about the current user and also notify the rest of the application if the current user changes.

The first thing we need to do is create a TypeScript class and make it *injectable* by using the `@Injectable` annotation.

```
code/rxjs/chat/app/ts/services/UserService.ts
```

```
9 @Injectable()
10 export class UserService {
```



When we make something *injectable* that means we will be able to use it as a dependency to other components in our application. Briefly, two benefits of dependency-injection are:

1. we let Angular handle the lifecycle of the object and
2. it's easier to test injected components.

We talk more about `@Injectable` in the [chapter on dependency injection](#) (forthcoming), but the result is that now we can inject it as a dependency to our components like so:

```
1 class MyComponent {
2   constructor(public userService: UserService) {
3     // do something with `userService` here
4   }
5 }
```

`currentUser` stream

Next we setup a stream which we will use to manage our current user:

```
code/rxjs/chat/app/ts/services/UserService.ts
```

```
12 currentUser: Subject<User> = new BehaviorSubject<User>(null);
```

There's a lot going on here, so let's break it down:

- We're defining an instance variable `currentUser` which is a `Subject` stream.
- Concretely, `currentUser` is a `BehaviorSubject` which will contain `User`.
- However, the first value of this stream is `null` (the constructor argument).

If you haven't worked with RxJS much, then you may not know what `Subject` or `BehaviorSubject` are. You can think of a `Subject` as a "read/write" stream.

 Technically a [Subject](#) inherits from both [Observable](#) and [Observer](#)

One consequence of streams is that, because messages are published immediately, a new subscriber risks missing the latest value of the stream. `BehaviourSubject` compensates for this.

[BehaviourSubject](#) has a special property in that it **stores the last value**. Meaning that any subscriber to the stream will receive the latest value. This is great for us because it means that any part of our application can subscribe to the `UserService.currentUser` stream and immediately know who the current user is.

Setting a new user

We need a way to publish a new user to the stream whenever the current user changes (e.g. logging in).

There's two ways we can expose an API for doing this:

1. Add new users to the stream directly:

The most straightforward way to update the current user is to have clients of the `UserService` simply publish a new `User` directly to the stream like this:

```
1 userService.subscribe((newUser) => {
2   console.log('New User is: ', newUser.name);
3 })
4
5 // => New User is: originalUserName
6
7 let u = new User('Nate', 'anImgSrc');
8 userService.currentUser.next(u);
9
10 // => New User is: Nate
```

 Note here that we use the `next` method on a `Subject` to push a new value to the stream

The pro here is that we're able to reuse the existing API from the stream, so we're not introducing any new code or APIs

2. Create a `setCurrentUser(newUser: User)` method

The other way we could update the current user is to create a helper method on the `UserService` like this:

`code/rxjs/chat/app/ts/services/UserService.ts`

```
14 public setCurrentUser(newUser: User): void {
15   this.currentUser.next(newUser);
16 }
```

You'll notice that we're still using the `next` method on the `currentUser` stream, so why bother doing this?

Because there is value in decoupling the implementation of the `currentUser` from the implementation of the stream. By wrapping the next in the `setCurrentUser` call we give ourselves room to change the implementation of the `UserService` without breaking our clients.

In this case, I wouldn't recommend one method very strongly over the other, but it can make a big difference on the maintainability of larger projects.



A third option could be to have the updates expose streams of their own (that is, a stream where we place the action of changing the current user). We explore this pattern in the `MessagesService` below.

UserService.ts

Putting it together, our `UserService` looks like this:

code/rxjs/chat/app/ts/services/UserService.ts

```
1 import {Injectable, bind} from '@angular/core';
2 import {Subject, BehaviorSubject} from 'rxjs';
3 import {User} from '../models';
4
5
6 /**
7  * UserService manages our current user
8  */
9 @Injectable()
10 export class UserService {
11   // `currentUser` contains the current user
12   currentUser: Subject<User> = new BehaviorSubject<User>(null);
13
14   public setCurrentUser(newUser: User): void {
15     this.currentUser.next(newUser);
16   }
17 }
18
19 export var userServiceInjectables: Array<any> = [
20   bind(UserService).toClass(UserService)
21 ];
```

The MessagesService

The `MessagesService` is the backbone of this application. In our app, all messages flow through the `MessagesService`.

Our `MessagesService` has much more sophisticated streams compared to our `UserService`. There are five streams that make up our `MessagesService`: 3 “data management” streams and 2 “action” streams.

The three data management streams are:

- `newMessages` - emits each new `Message` only once
- `messages` - emits **an array** of the current `Messages`
- `updates` - performs operations on messages

the `newMessages` stream

`newMessages` is a `Subject` that will publish each new `Message` only once.

code/rxjs/chat/app/ts/services/MessageService.ts

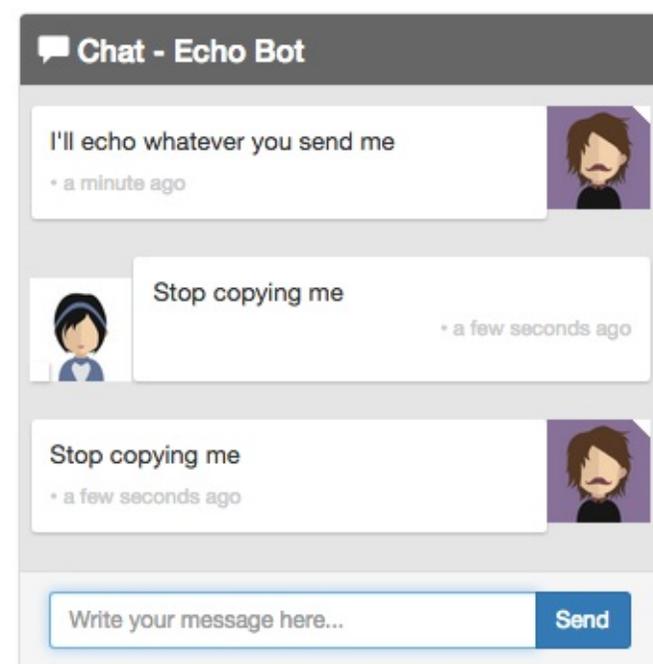
```
11 @Injectable()
12 export class MessageService {
13   // a stream that publishes new messages only once
14   newMessages: Subject<Message> = new Subject<Message>();
```

If we want, we can define a helper method to add Messages to this stream:

code/rxjs/chat/app/ts/services/MessageService.ts

```
88 addMessage(message: Message): void {
89   this.newMessages.next(message);
90 }
```

It would also be helpful to have a stream that will get all of the messages from a thread that are not from a particular user. For instance, consider the Echo Bot:



Real mature, Echo Bot

When we are implementing the Echo Bot, we don't want to enter an infinite loop and repeat back the bot's messages to itself.

To implement this we can subscribe to the newMessages stream and filter out all messages that are

1. part of this thread and
2. not written by the bot.

You can think of this as saying, for a given Thread I want a stream of the messages that are "for" this User.

code/rxjs/chat/app/ts/services/MessageService.ts

```
92 messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
93   return this.newMessages
94     .filter((message: Message) => {
95       // belongs to this thread
96       return (message.thread.id === thread.id) &&
97         // and isn't authored by this user
98         (message.author.id !== user.id);
99     });
100 }
```

messagesForThreadUser takes a Thread and a User and returns a new stream of Messages that are filtered on that Thread and not authored by the User. That is, it is a stream of “everyone else’s” messages in this Thread.

the messages stream

Whereas newMessages emits individual Messages, the messages stream emits **an Array of the most recent Messages**.

code/rxjs/chat/app/ts/services/MessageService.ts

```
16 // `messages` is a stream that emits an array of the most up to date messages
17 messages: Observable<Message[]>;
```

 The type Message[] is the same as Array<Message>. Another way of writing the same thing would be: Observable<Array<Message>>. When we define the type of messages to be Observable<Message[]> we mean that this stream emits an **Array** (of Messages), not individual Messages.

So how does messages get populated? For that we need to talk about the updates stream and a new pattern: the Operation stream.

The Operation Stream Pattern

Here’s the idea:

- We’ll maintain state in messages which will hold an Array of the most current Messages
- We use an updates stream which is a **stream of functions** to apply to messages

You can think of it this way: any function that is put on the updates stream will change the list of the current messages. A function that is put on the updates stream should **accept a list of Messages** and then **return a list of Messages**. Let’s formalize this idea by creating an interface in code:

code/rxjs/chat/app/ts/services/MessageService.ts

```
7 interface IMessageOperation extends Function {
8   (messages: Message[]): Message[];
9 }
```

Let’s define our updates stream:

code/rxjs/chat/app/ts/services/MessageService.ts

```
19 // `updates` receives operations to be applied to our `messages`
20 // it's a way we can perform changes on *all* messages (that are currently
21 // stored in `messages`)
22 updates: Subject<any> = new Subject<any>();
```

Remember, updates receives *operations* that will be applied to our list of messages. But how do we make that connection? We do (in the constructor of our MessageService) like this:

code/rxjs/chat/app/ts/services/MessageService.ts

```
28 constructor() {
29   this.messages = this.updates
30     // watch the updates and accumulate operations on the messages
31     .scan((messages: Message[],
```

```
32     operation: IMessagesOperation) => {
33         return operation(messages);
34     },
35     initialMessages)
```

This code introduces a new stream function: [scan](#). If you're familiar with functional programming, scan is a lot like reduce: it runs the function for each element in the incoming stream and **accumulates a value**. What's special about scan is that it will **emit a value for each intermediate result**. That is, it doesn't wait for the stream to complete before emitting a result, which is exactly what we want.

When we call `this.updates.scan`, we are creating a new stream that is subscribed to the updates stream. On each pass, we're given:

1. the messages we're accumulating and
2. the new operation to apply.

and then we return the new `Message[]`.

Sharing the Stream

One thing to know about streams is that they aren't shareable by default. That is, if one subscriber reads a value from a stream, it can be gone forever. In the case of our messages, we want to 1. share the same stream among many subscribers and 2. replay the last value for any subscribers who come "late".

To do that, we use two operators: `publishReplay` and `refCount`.

- `publishReplay` let's us share a subscription between multiple subscribers and replay n number of values to future subscribers. (see [publish](#) and [replay](#))
- [refCount](#) - makes it easier to use the return value of `publish`, by managing when the observable will emit values



Wait, so what does `refCount` do?

`refCount` can be a little tricky to understand because it relates to how one manages "hot" and "cold" observables. We're not going to dive deep into explaining how this works and we direct the reader to:

- [RxJS docs on refCount](#)
- [Introduction to Rx: Hot and Cold observables](#)
- [RefCount Marble Diagram](#)

code/rxjs/chat/app/ts/services/MessageService.ts

```
30     // watch the updates and accumulate operations on the messages
31     .scan((messages: Message[],
32         operation: IMessagesOperation) => {
33         return operation(messages);
34     },
35     initialMessages)
36     // make sure we can share the most recent list of messages across anyone
37     // who's interested in subscribing and cache the last known list of
38     // messages
39     .publishReplay(1)
40     .refCount();
```

Adding Messages to the messages Stream

Now we could add a Message to the messages stream like so:

```
1 var myMessage = new Message(/* params here... */);
2
3 updates.next( (messages: Message[]): Message[] => {
4   return messages.concat(myMessage);
5 })
```

Above, we're adding an operation to the updates stream. messages is subscribe to that stream and so it will apply that operation which will concat our newMessage on to the accumulated list of messages.

 It's okay if this takes a few minutes to mull over. It can feel a little foreign if you're not used to this style of programming.

One problem with the above approach is that it's a bit verbose to use. It would be nice to not have to write that inner function every time. We could do something like this:

```
1 addMessage(newMessage: Message) {
2   updates.next( (messages: Message[]): Message[] => {
3     return messages.concat(newMessage);
4   })
5 }
6
7 // somewhere else
8
9 var myMessage = new Message(/* params here... */);
10 MessagesService.addMessage(myMessage);
```

This is a little bit better, but it's not “the reactive way”. In part, because this action of creating a message isn't composable with other streams. (Also this method is circumventing our newMessages stream. More on that later.)

A reactive way of creating a new message would be **to have a stream that accepts Messages to add to the list**. Again, this can be a bit new if you're not used to thinking this way. Here's how you'd implement it:

First we make an “action stream” called create. (The term “action stream” is only meant to describe its role in our service. The stream itself is still a regular Subject):

code/rxjs/chat/app/ts/services/MessagesService.ts

```
24 // action streams
25 create: Subject<Message> = new Subject<Message>();
```

Next, in our constructor we configure the create stream:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
56 this.create
57   .map( function(message: Message): IMessagesOperation {
58     return (messages: Message[]) => {
59       return messages.concat(message);
60     });
61 })
```

The [map](#) operator is a lot like the built-in `Array.map` function in Javascript except that it works on streams. That is, it runs the function once for each item in the stream and emits the return value of the function.

In this case, we're saying "for each Message we receive as input, return an `IMessagesOperation` that adds this message to the list". Put another way, this stream will emit a **function** which accepts the list of Messages and adds this Message to our list of messages.

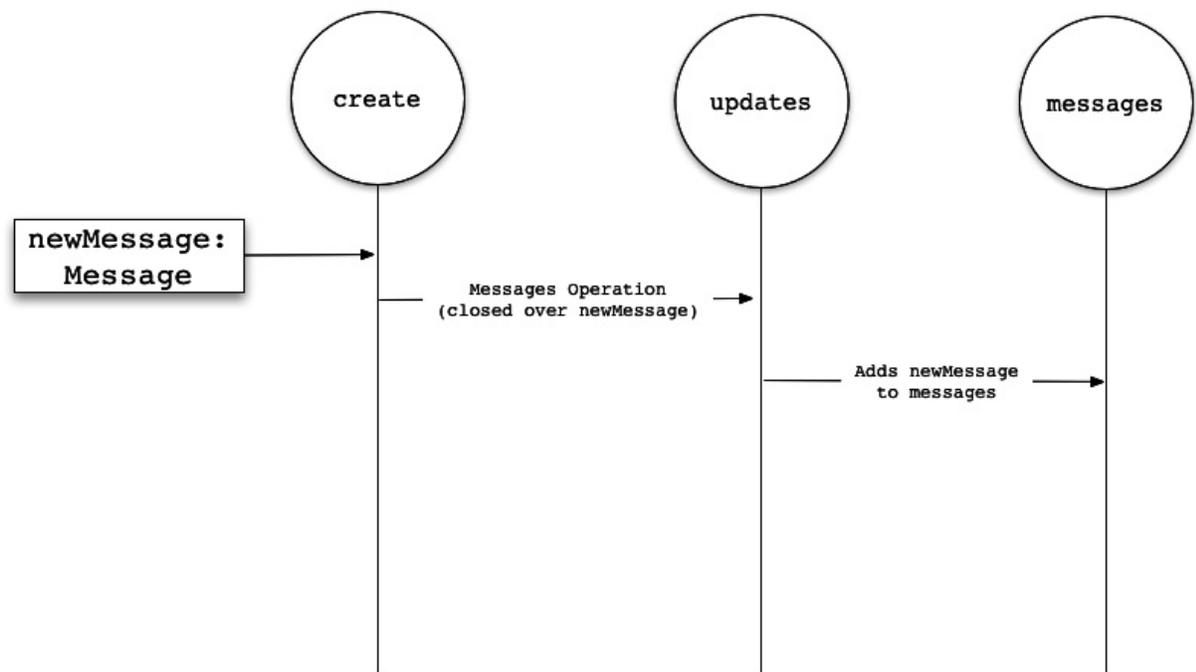
Now that we have the create stream, we still have one thing left to do: we need to actually hook it up to the updates stream. We do that by using [subscribe](#).

code/rxjs/chat/app/ts/services/MessagesService.ts

```
56     this.create
57     .map( function(message: Message): IMessagesOperation {
58         return (messages: Message[]) => {
59             return messages.concat(message);
60         });
61     })
62     .subscribe(this.updates);
```

What we're doing here is *subscribing* the updates stream to listen to the create stream. This means that if create receives a Message it will emit an `IMessagesOperation` that will be received by updates and then the Message will be added to messages.

Here's a diagram that shows our current situation:



Creating a new message, starting with the create stream

This is great because it means we get a few things:

1. The current list of messages from messages
2. A way to process operations on the current list of messages (via updates)
3. An easy-to-use stream to put create operations on our updates stream (via create)

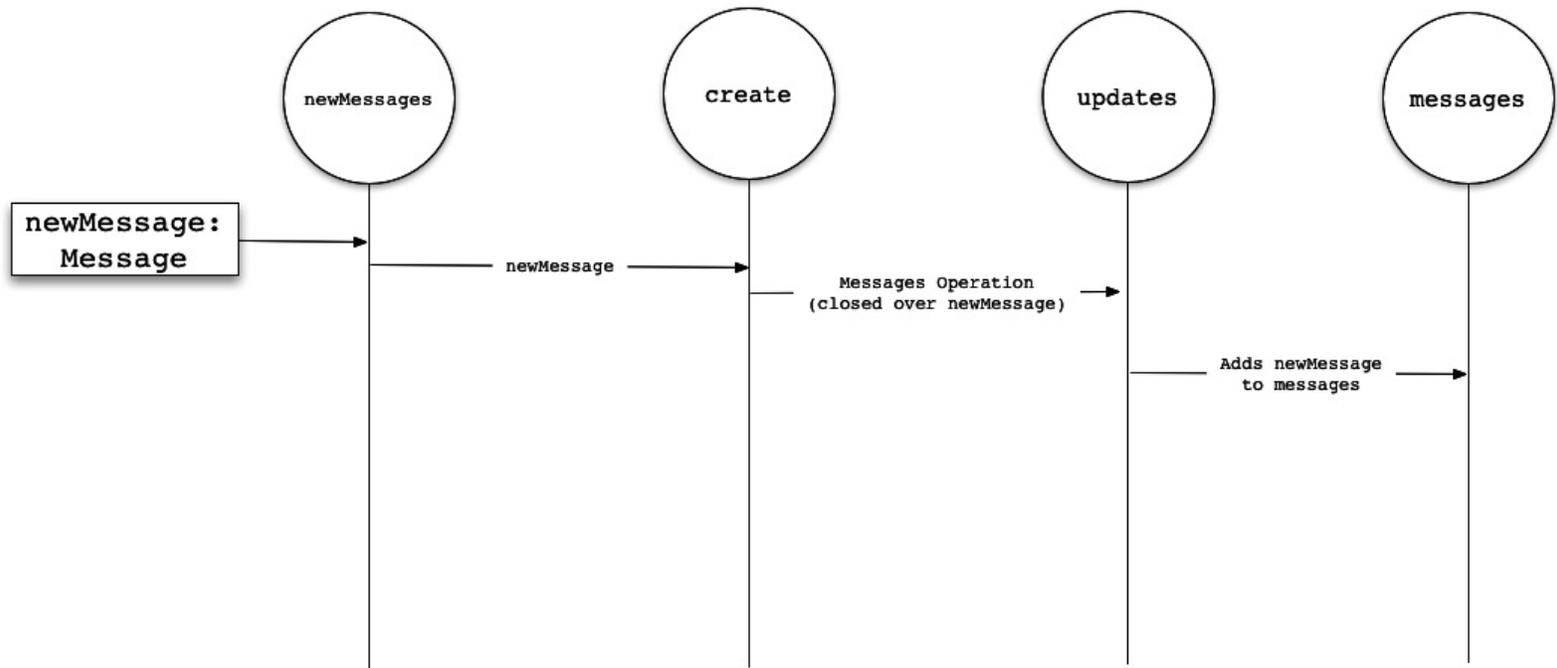
Anywhere in our code, if we want to get the most current list of messages, we just have to go to the messages stream. But we have a problem, **we still haven't connected this flow to the newMessages stream.

It would be great if we had a way to easily connect this stream with any Message that comes from newMessages. It turns out, it's really easy:

code/rxjs/chat/app/ts/services/MessageService.ts

```
64 this.newMessages
65   .subscribe(this.create);
```

Now our diagram looks like this:



Creating a new message, starting with the newMessages stream

Now our flow is complete! It's the best of both worlds: we're able to subscribe to the stream of individual messages through newMessages, but if we just want the most up-to-date list, we can subscribe to messages.



It's worth pointing out some implications of this design: if you subscribe to newMessages directly, you have to be careful about changes that may happen downstream. Here are three things to consider:

First, you obviously won't get any downstream updates that are applied to the Messages.

Second, in this case, we have **mutable** Message objects. So if you subscribe to newMessages and store a reference to a Message, that Message's attributes may change.

Third, in the case where you want to take advantage of the mutability of our Messages you may not be able to. Consider the case where we could put an operation on the updates queue that makes a copy of each Message and then mutates the copy. (This is probably a better design than what we're doing here.) In this case, you couldn't rely on any Message emitted directly from newMessages being in its "final" state.

That said, as long as you keep these considerations in mind, you shouldn't have too much trouble.

Our completed MessagesService

Here's what the completed MessagesService looks like:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
1 import {Injectable, bind} from '@angular/core';
2 import {Subject, Observable} from 'rxjs';
3 import {User, Thread, Message} from '../models';
4
5 let initialMessages: Message[] = [];
6
7 interface IMessageOperation extends Function {
8   (messages: Message[]): Message[];
9 }
10
11 @Injectable()
12 export class MessagesService {
13   // a stream that publishes new messages only once
14   newMessages: Subject<Message> = new Subject<Message>();
15
16   // `messages` is a stream that emits an array of the most up to date messages
17   messages: Observable<Message[]>;
18
19   // `updates` receives _operations_ to be applied to our `messages`
20   // it's a way we can perform changes on *all* messages (that are currently
21   // stored in `messages`)
22   updates: Subject<any> = new Subject<any>();
23
24   // action streams
25   create: Subject<Message> = new Subject<Message>();
26   markThreadAsRead: Subject<any> = new Subject<any>();
27
28   constructor() {
29     this.messages = this.updates
30       // watch the updates and accumulate operations on the messages
31       .scan((messages: Message[],
32           operation: IMessageOperation) => {
33         return operation(messages);
34       },
35         initialMessages)
36       // make sure we can share the most recent list of messages across anyone
37       // who's interested in subscribing and cache the last known list of
38       // messages
39       .publishReplay(1)
40       .refCount();
41
42     // `create` takes a Message and then puts an operation (the inner function)
43     // on the `updates` stream to add the Message to the list of messages.
44     //
45     // That is, for each item that gets added to `create` (by using `next`)
46     // this stream emits a concat operation function.
47     //
48     // Next we subscribe `this.updates` to listen to this stream, which means
49     // that it will receive each operation that is created
50     //
51     // Note that it would be perfectly acceptable to simply modify the
52     // "addMessage" function below to simply add the inner operation function to
53     // the update stream directly and get rid of this extra action stream
54     // entirely. The pros are that it is potentially clearer. The cons are that
55     // the stream is no longer composable.
56     this.create
57       .map( function(message: Message): IMessageOperation {
58         return (messages: Message[]) => {
59           return messages.concat(message);
60         };
61       })
62       .subscribe(this.updates);
63
64     this.newMessages
65       .subscribe(this.create);
66
67     // similarly, `markThreadAsRead` takes a Thread and then puts an operation
68     // on the `updates` stream to mark the Messages as read
69     this.markThreadAsRead
70       .map( (thread: Thread) => {
71         return (messages: Message[]) => {
```

```

72     return messages.map( (message: Message) => {
73         // note that we're manipulating `message` directly here. Mutability
74         // can be confusing and there are lots of reasons why you might want
75         // to, say, copy the Message object or some other 'immutable' here
76         if (message.thread.id === thread.id) {
77             message.isRead = true;
78         }
79         return message;
80     });
81 };
82 });
83 .subscribe(this.updates);
84
85 }
86
87 // an imperative function call to this action stream
88 addMessage(message: Message): void {
89     this.newMessages.next(message);
90 }
91
92 messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
93     return this.newMessages
94         .filter((message: Message) => {
95             // belongs to this thread
96             return message.thread.id === thread.id) &&
97             // and isn't authored by this user
98             (message.author.id !== user.id);
99         });
100 }
101 }
102
103 export var messagesServiceInjectables: Array<any> = [
104     bind(MessagesService).toClass(MessagesService)
105 ];

```

Trying out MessagesService

If you haven't already, this would be a good time to open up the code and play around with the MessagesService to get a feel for how it works. We've got an example you can start with in `test/services/MessagesService.spec.ts`.



To run the tests in this project, open up your terminal then:

```

1 cd /path/to/code/rxjs/chat // <-- your path will vary
2 npm install
3 karma start

```

Let's start by creating a few instances of our models to use:

```

code/rxjs/chat/test/services/MessagesService.spec.ts
7 import {Message, User, Thread} from '../..app/ts/models';
8
9 describe('MessagesService', () => {
10     it('should test', () => {
11
12         let user: User = new User('Nate', '');
13         let thread: Thread = new Thread('t1', 'Nate', '');
14         let m1: Message = new Message({
15             author: user,
16             text: 'Hi!',
17             thread: thread
18         });

```

Next let's subscribe to a couple of our streams:

```

21     author: user,
22     text: 'Bye!',
23     thread: thread
24   });
25
26   let messageService: MessageService = new MessageService();
27
28   // listen to each message individually as it comes in
29   messageService.newMessages
30     .subscribe( (message: Message) => {
31       console.log('=> newMessages: ' + message.text);
32     });
33
34   // listen to the stream of most current messages
35   messageService.messages
36     .subscribe( (messages: Message[]) => {
37       console.log('=> messages: ' + messages.length);
38     });
39
40   messageService.addMessage(m1);
41   messageService.addMessage(m2);

```

Notice that even though we subscribed to `newMessages` first and `newMessages` is called directly by `addMessage`, our `messages` subscription is logged first. The reason for this is because `messages` subscribed to `newMessages` earlier than our subscription in this test (when `MessageService` was instantiated). (You shouldn't be relying on the ordering of independent streams in your code, but why it works this way is worth thinking about.)

Play around with the `MessageService` and get a feel for the streams there. We're going to be using them in the next section where we build the `ThreadsService`.

The ThreadsService

On our `ThreadsService` we're going to define four streams that emit respectively:

1. A map of the current set of `Threads` (in `threads`)
2. A chronological list of `Threads`, newest-first (in `orderedthreads`)
3. The currently selected `Thread` (in `currentThread`)
4. The list of `Messages` for the currently selected `Thread` (in `currentThreadMessages`)

Let's walk through how to build each of these streams, and we'll learn a little more about RxJS along the way.

A map of the current set of `Threads` (in `threads`)

Let's start by defining our `ThreadsService` class and the instance variable that will emit the `Threads`:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1 import {Injectable, bind} from '@angular/core';
2 import {Subject, BehaviorSubject, Observable} from 'rxjs';
3 import {Thread, Message} from '../models';
4 import {MessageService} from './MessageService';
5 import * as _ from 'underscore';
6
7 @Injectable()
8 export class ThreadsService {
9
10   // `threads` is an observable that contains the most up to date list of threads
11   threads: Observable<{ [key: string]: Thread }>;

```

Notice that this stream will emit a map (an object) with the id of the Thread being the string key and the Thread itself will be the value.

To create a stream that maintains the current list of threads, we start by attaching to the `messagesService.messages` stream:

```
code/rxjs/chat/app/ts/services/ThreadsService.ts
25   this.threads = messagesService.messages
```

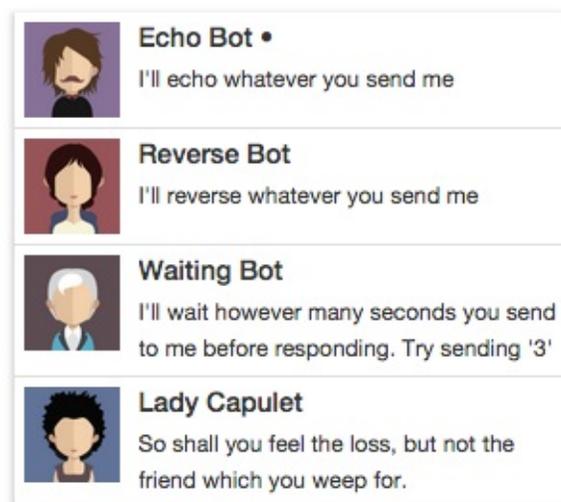
Recall that each time a new Message is added to the steam, messages will emit an array of the current Messages. We're going to look at each Message and we want to return a unique list of the Threads.

```
code/rxjs/chat/app/ts/services/ThreadsService.ts
26   this.threads = messagesService.messages
27   .map( (messages: Message[]) => {
28     let threads: {[key: string]: Thread} = {};
29     // Store the message's thread in our accumulator `threads`
30     messages.map((message: Message) => {
31       threads[message.thread.id] = threads[message.thread.id] ||
32       message.thread;

```

Notice above that each time we will create a new list of threads. The reason for this is because we might delete some messages down the line (e.g. leave the conversation). Because we're recalculating the list of threads each time, we naturally will "delete" a thread if it has no messages.

In the threads list, we want to show a preview of the chat by using the text of the most recent Message in that Thread.



List of Threads with Chat Preview

In order to do that, we'll store the most recent Message for each Thread. We know which Message is newest by comparing the `sentAt` times:

```
code/rxjs/chat/app/ts/services/ThreadsService.ts
33   // Cache the most recent message for each thread
34   let messagesThread: Thread = threads[message.thread.id];
35   if (!messagesThread.lastMessage ||
36       messagesThread.lastMessage.sentAt < message.sentAt) {
37     messagesThread.lastMessage = message;
38   }
39   });
```

```
40     return threads;
41   });
```

Putting it all together, threads looks like this:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
26   this.threads = messagesService.messages
27   .map( (messages: Message[]) => {
28     let threads: {[key: string]: Thread} = {};
29     // Store the message's thread in our accumulator `threads`
30     messages.map((message: Message) => {
31       threads[message.thread.id] = threads[message.thread.id] ||
32         message.thread;
33
34       // Cache the most recent message for each thread
35       let messagesThread: Thread = threads[message.thread.id];
36       if (!messagesThread.lastMessage ||
37         messagesThread.lastMessage.sentAt < message.sentAt) {
38         messagesThread.lastMessage = message;
39       }
40     });
41   return threads;
42 });
```

Trying out the ThreadsService

Let's try out our ThreadsService. First we'll create a few models to work with:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
8 import * as _ from 'underscore';
9
10 describe('ThreadsService', () => {
11   it('should collect the Threads from Messages', () => {
12
13     let nate: User = new User('Nate Murray', '');
14     let felipe: User = new User('Felipe Coury', '');
15
16     let t1: Thread = new Thread('t1', 'Thread 1', '');
17     let t2: Thread = new Thread('t2', 'Thread 2', '');
18
19     let m1: Message = new Message({
20       author: nate,
21       text: 'Hi!',
22       thread: t1
23     });
24
25     let m2: Message = new Message({
26       author: felipe,
27       text: 'Where did you get that hat?',
28       thread: t1
29     });
```

Now let's create an instance of our services:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
32     author: nate,
33     text: 'Did you bring the briefcase?'
```



Notice here that we're passing `messagesService` as an argument to the constructor of our `ThreadsService`. Normally we let the Dependency Injection system handle this for us. But in our test, we can provide the dependencies ourselves.

Let's subscribe to threads and log out what comes through:

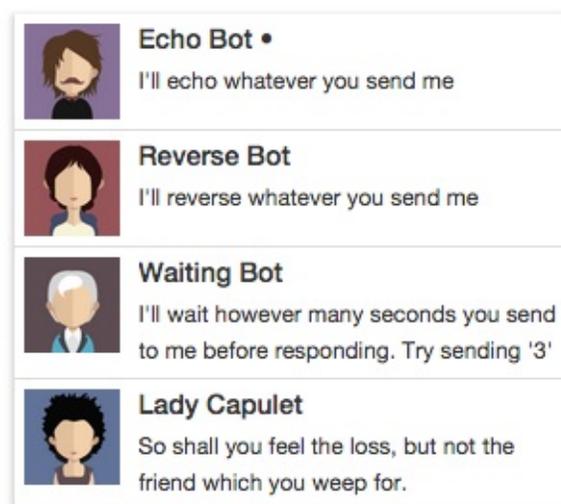
```

35     });
36
37     let messagesService: MessagesService = new MessagesService();
38     let threadsService: ThreadsService = new ThreadsService(messagesService);
39
40     threadsService.threads
41       .subscribe( (threadIdx: { [key: string]: Thread }) => {
42         let threads: Thread[] = _.values(threadIdx);
43         let threadNames: string = _.map(threads, (t: Thread) => t.name)
44           .join(', ');
45         console.log(`=> threads (${threads.length}): ${threadNames} `);
46       });
47
48     messagesService.addMessage(m1);
49     messagesService.addMessage(m2);
50     messagesService.addMessage(m3);
51
52     // => threads (1): Thread 1

```

A chronological list of threads, newest-first (in orderedthreads)

threads gives us a map which acts as an “index” of our list of threads. But we want the threads view to be ordered according the most recent message.



Time Ordered List of Threads

Let’s create a new stream that returns an Array of Threads ordered by the most recent Message time:

We’ll start by defining orderedThreads as an instance property:

```
code/rxjs/chat/app/ts/services/ThreadsService.ts
```

```

13 // `orderedThreads` contains a newest-first chronological list of threads
14 orderedThreads: Observable<Thread[]>;

```

Next, in the constructor we’ll define orderedThreads by subscribing to threads and ordered by the most recent message:

```
code/rxjs/chat/app/ts/services/ThreadsService.ts
```

```

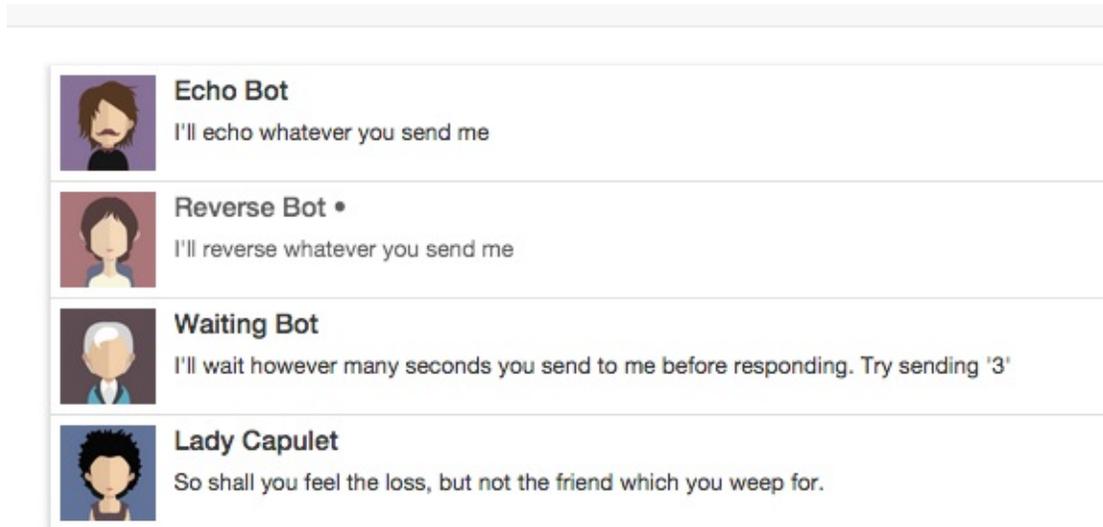
44 this.orderedThreads = this.threads
45   .map((threadGroups: { [key: string]: Thread }) => {
46     let threads: Thread[] = _.values(threadGroups);
47     return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
48   });

```

The currently selected thread (in currentThread)

Our application needs to know which Thread is the currently selected thread. This lets us know:

1. which thread should be shown in the messages window
2. which thread should be marked as the current thread in the list of threads



The current thread is marked by a '•' symbol

Let's create a BehaviorSubject that will store the currentThread:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
16 // `currentThread` contains the currently selected thread
17 currentThread: Subject<Thread> =
18   new BehaviorSubject<Thread>(new Thread());
```

Notice that we're issuing an empty Thread as the default value. We don't need to configure the currentThread any further.

Setting the Current Thread

To set the current thread we can have clients either

1. submit new threads via next directly or
2. add a helper method to do it.

Let's define a helper method setCurrentThread that we can use to set the next thread:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
69 setCurrentThread(newThread: Thread): void {
70   this.currentThread.next(newThread);
71 }
```

Marking the Current Thread as Read

We want to keep track of the number of unread messages. If we switch to a new Thread then we want to mark all of the Messages in that Thread as read. We have the parts we need to do this:

1. The `messagesService.makeThreadAsRead` accepts a `Thread` and then will mark all `Messages` in that `Thread` as read
2. Our `currentThread` emits a single `Thread` that represents the current `Thread`

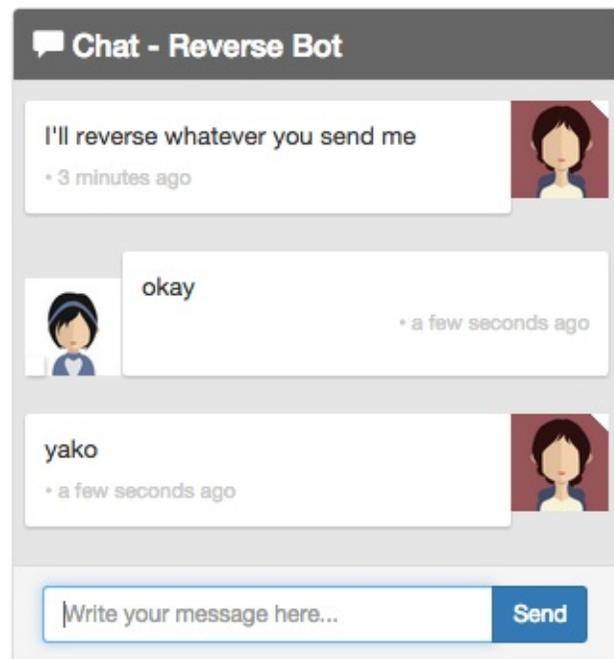
So all we need to do is hook them together:

`code/rxjs/chat/app/ts/services/ThreadsService.ts`

```
65 this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

The list of messages for the currently selected thread (in `currentThreadMessages`)

Now that we have the currently selected thread, we need to make sure we can show the list of `Messages` in that `Thread`.



The current list of messages is for the Reverse Bot

Implementing this is a little bit more complicated than it may seem at the surface. Say we implemented it like this:

```
1 var theCurrentThread: Thread;
2
3 this.currentThread.subscribe((thread: Thread) => {
4   theCurrentThread = thread;
5 })
6
7 this.currentThreadMessages.map(
8   (messages: Message[]) => {
9     return _.filter(messages,
10      (message: Message) => {
11        return message.thread.id == theCurrentThread.id;
12      })
13   })
```

What's wrong with this approach? Well, if the `currentThread` changes, `currentThreadMessages` won't know about it and so we'll have an outdated list of `currentThreadMessages`!

What if we reversed it, and stored the current list of messages in a variable and subscribed to the changing of `currentThread`? We'd have the same problem only this time we would know when the thread changes but not when a new message came in.

How can we solve this problem?

It turns out, RxJS has a set of operators that we can use to **combine multiple streams**. In this case we want to say “if *either* `currentThread` **or** `messagesService.messages` changes, then we want to emit something.” For this we use the [combineLatest](#) operator.

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
50   this.currentThreadMessages = this.currentThread
51     .combineLatest(messagesService.messages,
52                   (currentThread: Thread, messages: Message[]) => {
```

When we’re combining two streams one or the other will arrive first and there’s no guarantee that we’ll have a value on both streams, so we need to check to make sure we have what we need otherwise we’ll just return an empty list.

Now that we have both the current thread and messages, we can filter out just the messages we’re interested in:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
50   this.currentThreadMessages = this.currentThread
51     .combineLatest(messagesService.messages,
52                   (currentThread: Thread, messages: Message[]) => {
53     if (currentThread && messages.length > 0) {
54       return _.chain(messages)
55         .filter((message: Message) =>
56               (message.thread.id === currentThread.id))
```

One other detail, since we’re already looking at the messages for the current thread, this is a convenient area to mark these messages as read.

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
54     return _.chain(messages)
55       .filter((message: Message) =>
56             (message.thread.id === currentThread.id))
57       .map((message: Message) => {
58         message.isRead = true;
59         return message; })
60       .value();
```



Whether or not we should be marking messages as read here is debatable. The biggest drawback is that we’re mutating objects in what is, essentially, a “read” thread. i.e. this is a read operation with a side effect, which is generally a Bad Idea. That said, in this application the `currentThreadMessages` only applies to the `currentThread` and the `currentThread` should always have its messages marked as read. That said, the “read with side-effects” is not a pattern I recommend in general.

Putting it together, here’s what `currentThreadMessages` looks like:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
50   this.currentThreadMessages = this.currentThread
51     .combineLatest(messagesService.messages,
52                   (currentThread: Thread, messages: Message[]) => {
53     if (currentThread && messages.length > 0) {
54       return _.chain(messages)
55         .filter((message: Message) =>
56               (message.thread.id === currentThread.id))
57         .map((message: Message) => {
58           message.isRead = true;
59           return message; })
```

```
        .value();
61     } else {
62         return [];
63     }
64 });
```

Our Completed ThreadsService

Here's what our ThreadService looks like:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1  import {Injectable, bind} from '@angular/core';
2  import {Subject, BehaviorSubject, Observable} from 'rxjs';
3  import {Thread, Message} from '../models';
4  import {MessagesService} from '../MessagesService';
5  import * as _ from 'underscore';
6
7  @Injectable()
8  export class ThreadsService {
9
10     // `threads` is a observable that contains the most up to date list of threads
11     threads: Observable<{ [key: string]: Thread }>;
12
13     // `orderedThreads` contains a newest-first chronological list of threads
14     orderedThreads: Observable<Thread[]>;
15
16     // `currentThread` contains the currently selected thread
17     currentThread: Subject<Thread> =
18         new BehaviorSubject<Thread>(new Thread());
19
20     // `currentThreadMessages` contains the set of messages for the currently
21     // selected thread
22     currentThreadMessages: Observable<Message[]>;
23
24     constructor(public messagesService: MessagesService) {
25
26         this.threads = messagesService.messages
27             .map((messages: Message[]) => {
28                 let threads: {[key: string]: Thread} = {};
29                 // Store the message's thread in our accumulator `threads`
30                 messages.map((message: Message) => {
31                     threads[message.thread.id] = threads[message.thread.id] ||
32                         message.thread;
33
34                     // Cache the most recent message for each thread
35                     let messagesThread: Thread = threads[message.thread.id];
36                     if (!messagesThread.lastMessage ||
37                         messagesThread.lastMessage.sentAt < message.sentAt) {
38                         messagesThread.lastMessage = message;
39                     }
40                 });
41                 return threads;
42             });
43
44         this.orderedThreads = this.threads
45             .map((threadGroups: { [key: string]: Thread }) => {
46                 let threads: Thread[] = _.values(threadGroups);
47                 return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
48             });
49
50         this.currentThreadMessages = this.currentThread
51             .combineLatest(messagesService.messages,
52                 (currentThread: Thread, messages: Message[]) => {
53                     if (currentThread && messages.length > 0) {
54                         return _.chain(messages)
55                             .filter((message: Message) =>
56                                 (message.thread.id === currentThread.id))
57                             .map((message: Message) => {
58                                 message.isRead = true;
59                                 return message; })
60                             .value();
61                     } else {
62                         return [];
63                     }
64                 });
65     }
```

```
64     });
65
66     this.currentThread.subscribe(this.messagesService.markThreadAsRead);
67 }
68
69 setCurrentThread(newThread: Thread): void {
70     this.currentThread.next(newThread);
71 }
72
73 }
74
75 export var threadsServiceInjectables: Array<any> = [
76     bind(ThreadsService).toClass(ThreadsService)
77 ];
```

Data Model Summary

Our data model and services are complete! Now we have everything we need now to start hooking it up to our view components! In the next chapter we'll build out our 3 major components to render and interact with these streams.

Data Architecture with Observables - Part 2: View Components

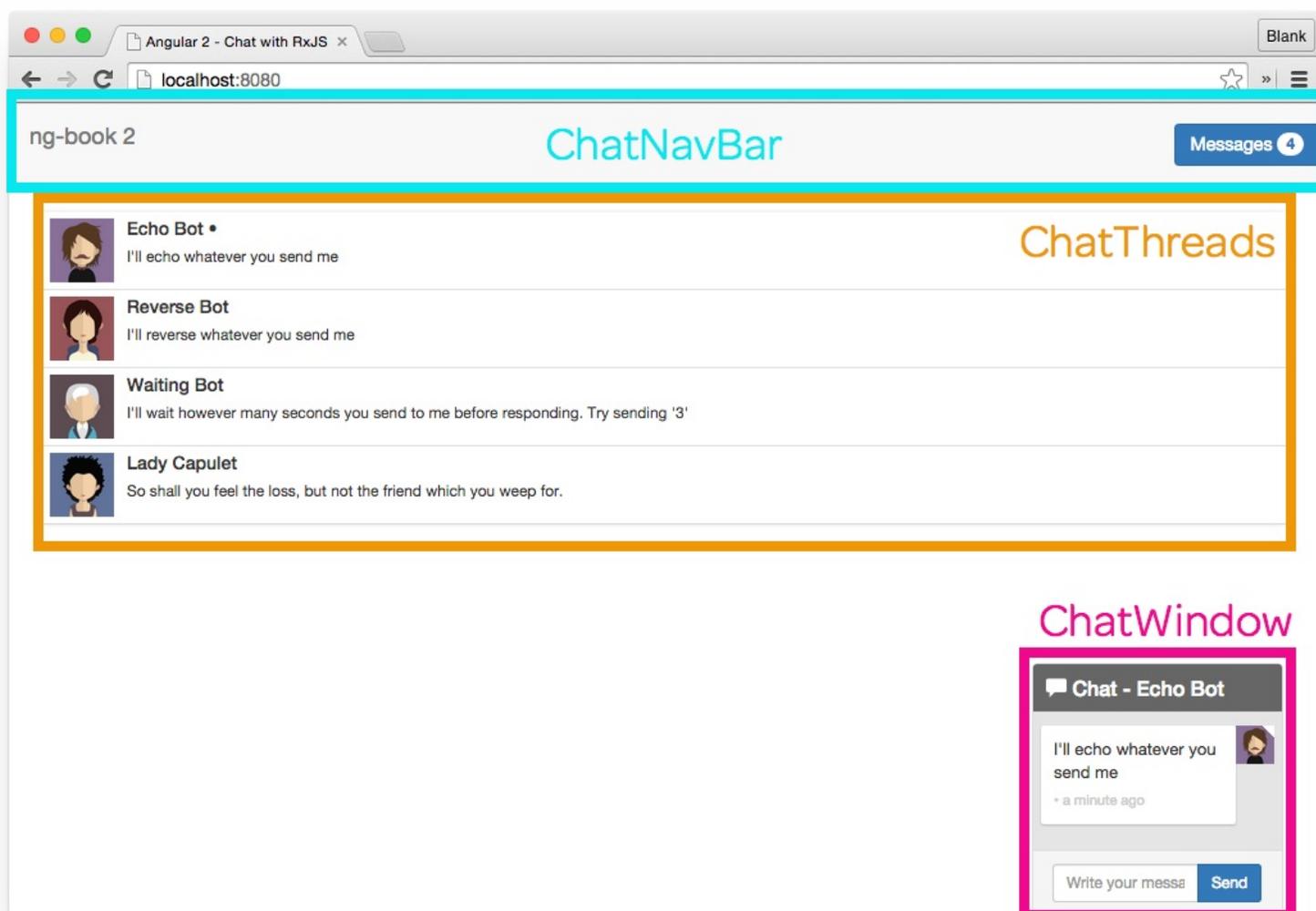
Building Our Views: The chatApp Top-Level Component

Let's turn our attention to our app and implement our view components.

i For the sake of clarity and space, in the following sections I'll be leaving out some `import` statements, CSS, and a few other similar things lines of code. If you're curious about each line of those details, open up the sample code because it contains everything we need to run this app.

The first thing we're going to do is create our top-level component `chat-app`

As we talked about earlier, the page is broken down into three top-level components:



- ChatNavBar - contains the unread messages count
- ChatThreads - shows a clickable list of threads, along with the most recent message and the conversation avatar
- ChatWindow - shows the messages in the current thread with an input box to send new messages

Here's what our component looks like in code:

code/rxjs/chat/app/ts/app.ts

```
43 @Component({
44   selector: 'chat-app',
45   directives: [ChatNavBar,
46               ChatThreads,
47               ChatWindow],
48   template: `
49     <div>
50       <nav-bar></nav-bar>
51       <div class="container">
52         <chat-threads></chat-threads>
53         <chat-window></chat-window>
54       </div>
55     </div>
56   `
57 })
58 class ChatApp {
59   constructor(public messagesService: MessagesService,
60               public threadsService: ThreadsService,
61               public userService: UserService) {
62     ChatExampleData.init(messagesService, threadsService, userService);
63   }
64 }
65
66 bootstrap(ChatApp, [ servicesInjectables, utilInjectables ]);
```

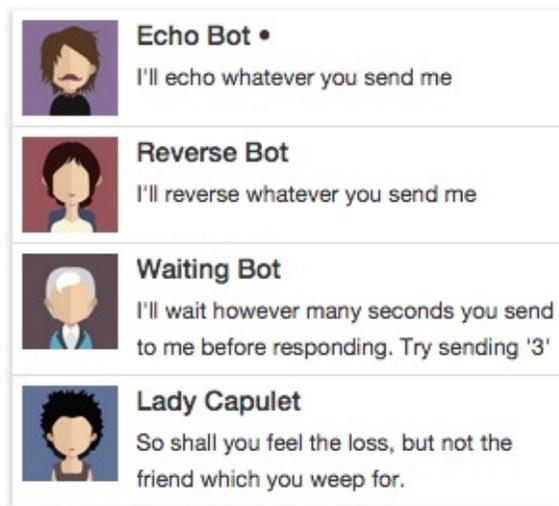
Take a look at the constructor. Here we're injecting our three services: the MessagesService, ThreadsService, and UserService. We're using those services to initialize our example data.



If you're interested in the example data you can find it in `code/rxjs/chat/app/ts/ChatExampleData.ts`.

The chatThreads Component

Next let's build our thread list in the ChatThreads component.



Time Ordered List of Threads

Our selector is straightforward, we want to match chat - threads.

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
52 @Component({
53   selector: 'chat-threads',
```

chatThreads Controller

Take a look at our component controller ChatThreads:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
70 export class ChatThreads {
71   threads: Observable<any>;
72
73   constructor(public threadsService: ThreadsService) {
74     this.threads = threadsService.orderedThreads;
75   }
76 }
```

Here we're injecting ThreadsService and then we're keeping a reference to the orderedThreads.

ChatThreads template

Lastly, let's look at the template and its configuration:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
52 @Component({
53   selector: 'chat-threads',
54   directives: [ChatThread],
55   changeDetection: ChangeDetectionStrategy.OnPush,
56   template: `
57     <!-- conversations -->
58     <div class="row">
59       <div class="conversation-wrap">
60
61         <chat-thread
62           *ngFor="let thread of threads | async"
63           [thread]="thread">
64         </chat-thread>
65
66       </div>
67     </div>
68 `
69 })
```

There's three things to look at here: NgFor with the async pipe, the ChangeDetectionStrategy and ChatThread.

The ChatThread directive component (which matches chat-thread in the markup) will show the view for the Threads. We'll define that in a moment.

The NgFor iterates over our threads, and passes the input [thread] to our ChatThread directive. But you probably notice something new in our *ngFor: the pipe to async.

async is implemented by AsyncPipe and it lets us use an RxJS observable here in our view. What's great about async is that it lets us use our async observable as if it was a sync collection. This is super convenient and really cool.

On this component we specify a custom changeDetection. Angular 2 has a flexible and efficient change detection system. One of the benefits is that if we have a component which has immutable or observable bindings, then we're able to give the change detection system hints that will make our application run very efficiently.

In this case, instead of watching for changes on an array of Threads, Angular will subscribe for changes to the threads observable - and trigger an update when a new event is emitted.

Here's what our total ChatThreads component looks like:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
52 @Component({
53   selector: 'chat-threads',
54   directives: [ChatThread],
55   changeDetection: ChangeDetectionStrategy.OnPush,
56   template: `
57     <!-- conversations -->
58     <div class="row">
59       <div class="conversation-wrap">
60
61         <chat-thread
62           *ngFor="let thread of threads | async"
63           [thread]="thread">
64         </chat-thread>
65
66       </div>
67     </div>
68 `
69 })
70 export class ChatThreads {
71   threads: Observable<any>;
72
73   constructor(public threadsService: ThreadsService) {
74     this.threads = threadsService.orderedThreads;
75   }
76 }
```

The Single chatThread Component

Let's look at our ChatThread component. This is the component that will be used to display a **single thread**. Starting with the @Component:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
10 @Component({
11   inputs: ['thread'],
12   selector: 'chat-thread',
```

We'll come back and look at the template in a minute, but first let's look at the component definition controller.

chatThread Controller and ngOnInit

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
29 class ChatThread implements OnInit {
30   thread: Thread;
31   selected: boolean = false;
32
33   constructor(public threadsService: ThreadsService) {
34   }
35
36   ngOnInit(): void {
37     this.threadsService.currentThread
38       .subscribe( (currentThread: Thread) => {
39         this.selected = currentThread &&
40           this.thread &&
41             (currentThread.id === this.thread.id);
42       });
43   }
44
45   clicked(event: any): void {
46     this.threadsService.setCurrentThread(this.thread);
47     event.preventDefault();
48   }
49 }
```

Notice that we're implementing a new interface here: `OnInit`. Angular components can declare that they listen for certain lifecycle events. We talk more about lifecycle events [here in the Components chapter](#) (forthcoming).

In this case, because we declared that we implement `OnInit`, the method `ngOnInit` will be called on our component after the component has been checked for changes the first time.

A key reason we will use `ngOnInit` is because **our thread property won't be available in the constructor**.

Above you can see that in `ngOnInit` we subscribe to `threadsService.currentThread` and if the `currentThread` matches the `thread` property of this component, we set `selected` to `true` (conversely, if the `Thread` doesn't match, we set `selected` to `false`).

We also setup an event handler `clicked`. This is how we handle selecting the current thread. In our template (below), we will bind `clicked()` to clicking on the thread view. If we receive `clicked()` then we tell the `threadsService` we want to set the current thread to the `Thread` of this component.

ChatThread template

Here's the code for our template:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
10 @Component({
11   inputs: ['thread'],
12   selector: 'chat-thread',
13   template: `
14     <div class="media conversation">
15       <div class="pull-left">
16         
```

```

18 </div>
19 <div class="media-body">
20   <h5 class="media-heading contact-name">{{thread.name}}
21     <span *ngIf="selected">&bull;</span>
22   </h5>
23   <small class="message-preview">{{thread.lastMessage.text}}</small>
24 </div>
25 <a (click)="clicked($event)" class="div-link">Select</a>
26 </div>
27 `
28 })

```

Notice we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`, and `{{thread.lastMessage.text}}`.

We've got an `*ngIf` which will show the `•` symbol only if this is the selected thread.

Lastly, we're binding to the `(click)` event to call our `clicked()` handler. Notice that when we call `clicked` we're passing the argument `$event`. This is a special variable provided by Angular that describes the event. We use that in our `clicked` handler by calling `event.preventDefault();`. This makes sure that we don't navigate to a different page.

chatThread Complete Code

Here's the whole of the ChatThread component:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```

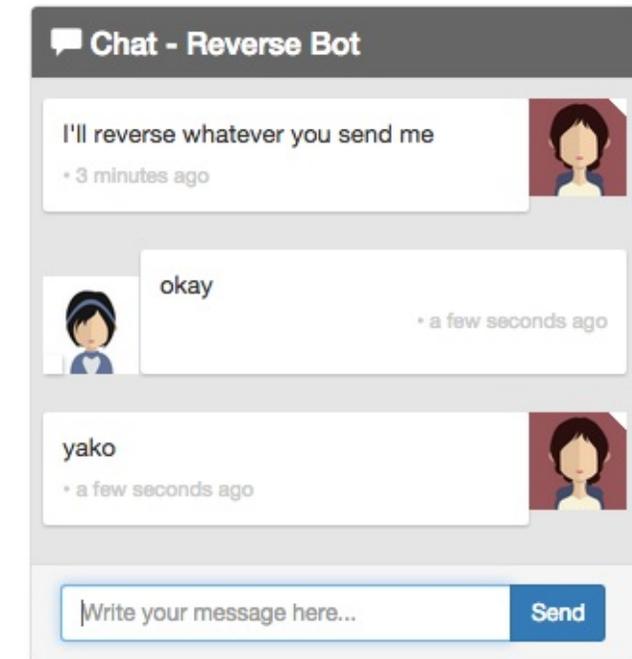
1  import {
2    Component,
3    OnInit,
4    ChangeDetectionStrategy
5  } from '@angular/core';
6  import {ThreadsService} from '../services/services';
7  import {Observable} from 'rxjs';
8  import {Thread} from '../models';
9
10 @Component({
11   inputs: ['thread'],
12   selector: 'chat-thread',
13   template: `
14 <div class="media conversation">
15   <div class="pull-left">
16     
18   </div>
19   <div class="media-body">
20     <h5 class="media-heading contact-name">{{thread.name}}
21       <span *ngIf="selected">&bull;</span>
22     </h5>
23     <small class="message-preview">{{thread.lastMessage.text}}</small>
24   </div>
25   <a (click)="clicked($event)" class="div-link">Select</a>
26 </div>
27 `
28 })
29 class ChatThread implements OnInit {
30   thread: Thread;
31   selected: boolean = false;
32
33   constructor(public threadsService: ThreadsService) {
34   }
35
36   ngOnInit(): void {
37     this.threadsService.currentThread
38       .subscribe( (currentThread: Thread) => {
39         this.selected = currentThread &&
40           this.thread &&
41           (currentThread.id === this.thread.id);

```

```
42     });
43   }
44
45   clicked(event: any): void {
46     this.threadsService.setCurrentThread(this.thread);
47     event.preventDefault();
48   }
49 }
```

The chatwindow Component

The Chatwindow is the most complicated component in our app. Let's take it one section at a time:



The Chat Window

We start by defining our @Component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
64 @Component({
65   selector: 'chat-window',
66   directives: [ChatMessage,
67               FORM_DIRECTIVES],
68   changeDetection: ChangeDetectionStrategy.OnPush,
```

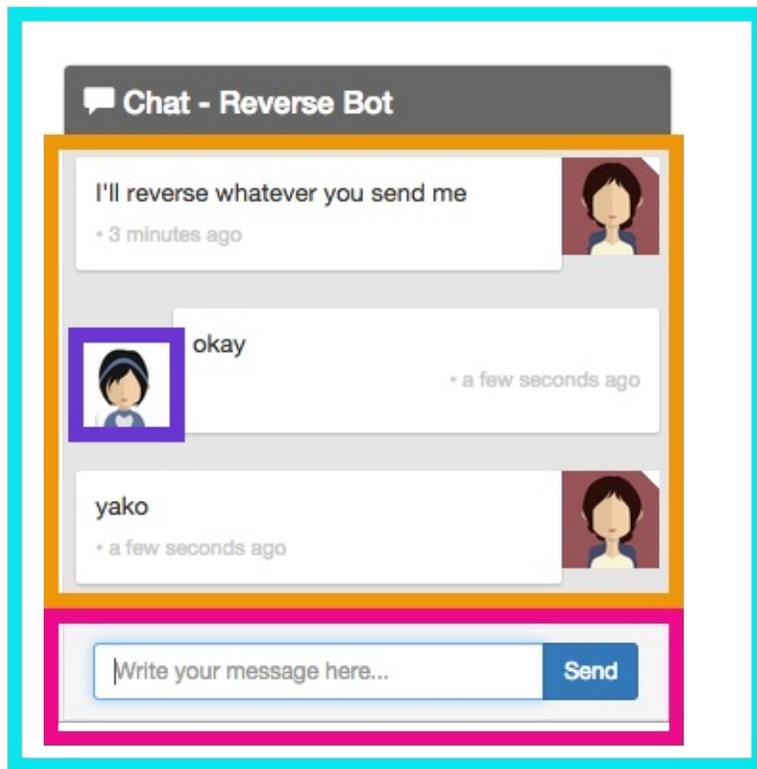
chatwindow Component Class Properties

Our Chatwindow class has four properties:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
115 export class ChatWindow implements OnInit {
116   messages: Observable<any>;
117   currentThread: Thread;
118   draftMessage: Message;
119   currentUser: User;
```

Here's a diagram of where each one is used:



currentThread

messages

currentUser

draftMessage

Chat Window Properties

In our constructor we're going to inject four things:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
121 constructor(public messagesService: MessagesService,  
122             public threadsService: ThreadsService,  
123             public userService: UserService,  
124             public el: ElementRef) {  
125 }
```

The first three are our services. The fourth, `e1` is an `ElementRef` which we can use to get access to the host DOM element. We'll use that when we scroll to the bottom of the chat window when we create and receive new messages.



Remember: by using `public messagesService: MessagesService` in the constructor, we are not only injecting the `MessagesService` but setting up an instance variable that we can use later in our class via `this.messagesService`

ChatWindow ngOnInit

We're going to put the initialization of this component in `ngOnInit`. The main thing we're going to be doing here is setting up the subscriptions on our observables which will then change our component properties.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
127 ngOnInit(): void {  
128   this.messages = this.threadsService.currentThreadMessages;  
129  
130   this.draftMessage = new Message();
```

First, we'll save the currentThreadMessages into messages. Next we create an empty Message for the default draftMessage.

When we send a new message we need to make sure that Message stores a reference to the sending Thread. The sending thread is always going to be the current thread, so let's store a reference to the currently selected thread:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
132     this.threadsService.currentThread.subscribe(  
133       (thread: Thread) => {  
134         this.currentThread = thread;  
135       });
```

We also want new messages to be sent from the current user, so let's do the same with currentUser:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
137     this.userService.currentUser  
138       .subscribe(  
139         (user: User) => {  
140           this.currentUser = user;  
141         });
```

ChatWindow sendMessage

Since we're talking about it, let's implement a sendMessage function that will send a new message:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
157     sendMessage(): void {  
158       let m: Message = this.draftMessage;  
159       m.author = this.currentUser;  
160       m.thread = this.currentThread;  
161       m.isRead = true;  
162       this.messagesService.addMessage(m);  
163       this.draftMessage = new Message();  
164     }
```

The sendMessage function above takes the draftMessage, sets the author and thread using our component properties. Every message we send has "been read" already (we wrote it) so we mark it as read.

Notice here that we're not updating the draftMessage text. That's because we're going to bind the value of the messages text in the view in a few minutes.

After we've updated the draftMessage properties we send it off to the messagesService and then **create a new Message** and set that new Message to this.draftMessage. We do this to make sure we don't mutate an already sent message.

ChatWindow onEnter

In our view, we want to send the message in two scenarios

1. the user hits the "Send" button or
2. the user hits the Enter (or Return) key.

Let's define a function that will handle that event:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
152 onEnter(event: any): void {
153     this.sendMessage();
154     event.preventDefault();
155 }
```

ChatWindow scrollToBottom

When we send a message, or when a new message comes in, we want to scroll to the bottom of the chat window. To do that, we're going to set the `scrollTop` property of our host element:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
166 scrollToBottom(): void {
167     let scrollPane: any = this.el
168     .nativeElement.querySelector('.msg-container-base');
169     scrollPane.scrollTop = scrollPane.scrollHeight;
170 }
```

Now that we have a function that will scroll to the bottom, we have to make sure that we call it at the right time. Back in `ngOnInit` let's subscribe to the list of `currentThreadMessages` and scroll to the bottom any time we get a new message:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
143 this.messages
144     .subscribe(
145     (messages: Array<Message>) => {
146         setTimeout(() => {
147             this.scrollToBottom();
148         });
149     });
```



Why do we have the `setTimeout`?

If we call `scrollToBottom` immediately when we get a new message then what happens is we scroll to the bottom before the new message is rendered. By using a `setTimeout` we're telling Javascript that we want to run this function when it is finished with the current execution queue. This happens **after** the component is rendered, so it does what we want.

ChatWindow template

The opening of our template should look familiar, we start by defining some markup and the panel header:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
64 @Component({
65     selector: 'chat-window',
66     directives: [ChatMessage,
67                 FORM_DIRECTIVES],
68     changeDetection: ChangeDetectionStrategy.OnPush,
69     template: `
70         <div class="chat-window-container">
71             <div class="chat-window">
72                 <div class="panel-container">
73                     <div class="panel panel-default">
74
75                         <div class="panel-heading top-bar">
76                             <div class="panel-title-container">
77                                 <h3 class="panel-title">
78                                     <span class="glyphicon glyphicon-comment"></span>
79                                     Chat - {{currentThread.name}}
80                                 </h3>
81                             </div>
82                             <div class="panel-buttons-container">
```

```
83         <!-- you could put minimize or close buttons here -->
84     </div>
85 </div>
```

Next we show the list of messages. Here we use `ngFor` along with the `async` pipe to iterate over our list of messages. We'll describe the individual chat-message component in a minute.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
87     <div class="panel-body msg-container-base">
88         <chat-message
89             *ngFor="let message of messages | async"
90             [message]="message">
91         </chat-message>
92     </div>
```

Lastly we have the message input box and closing tags:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
94     <div class="panel-footer">
95         <div class="input-group">
96             <input type="text"
97                 class="chat-input"
98                 placeholder="Write your message here..."
99                 (keydown.enter)="onEnter($event)"
100                [(ngModel)]="draftMessage.text" />
101             <span class="input-group-btn">
102                 <button class="btn-chat"
103                     (click)="onEnter($event)"
104                     >Send</button>
105             </span>
106         </div>
107     </div>
108
109 </div>
110 </div>
111 </div>
112 </div>
113
```

The message input box is the most interesting part of this view, so let's talk about two interesting properties: 1. `(keydown.enter)` and 2. `[(ngModel)]`.

Handling keystrokes

Angular provides a straightforward way to handle keyboard actions: we bind to the event on an element. In this case, we're binding to `keydown.enter` which says if "Enter" is pressed, call the function in the expression, which in this case is `onEnter($event)`.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
96     <input type="text"
97         class="chat-input"
98         placeholder="Write your message here..."
99         (keydown.enter)="onEnter($event)"
100        [(ngModel)]="draftMessage.text" />
```

Using ngModel

As we've talked about before, Angular doesn't have a general model for two-way binding. However it can be very useful to have a two-way binding between a component and its view. As long as the side-effects are kept local to the component, it can be a very convenient way to keep a component property in sync with the view.

In this case, we're establishing a two-way bind **between the value of the input tag and `draftMessage.text`**. That is, if we type into the input tag, `draftMessage.text` will automatically be set to the value of that input. Likewise, if we were to update `draftMessage.text` in our code, the value in the input tag would change in the view.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
96         <input type="text"
97             class="chat-input"
98             placeholder="Write your message here..."
99             (keydown.enter)="onEnter($event)"
100             [(ngModel)]="draftMessage.text" />
```

Clicking "Send"

On our "Send" button we bind the `(click)` property to the `onEnter` function of our component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
101         <span class="input-group-btn">
102             <button class="btn-chat"
103                 (click)="onEnter($event)"
104                 >Send</button>
105         </span>
```

The Entire `chatWindow` Component

Here's the code listing for the entire `ChatWindow` Component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
64 @Component({
65   selector: 'chat-window',
66   directives: [ChatMessage,
67               FORM_DIRECTIVES],
68   changeDetection: ChangeDetectionStrategy.OnPush,
69   template: `
70     <div class="chat-window-container">
71       <div class="chat-window">
72         <div class="panel-container">
73           <div class="panel panel-default">
74
75             <div class="panel-heading top-bar">
76               <div class="panel-title-container">
77                 <h3 class="panel-title">
78                   <span class="glyphicon glyphicon-comment"></span>
79                   Chat - {{currentThread.name}}
80                 </h3>
81               </div>
82               <div class="panel-buttons-container">
83                 <!-- you could put minimize or close buttons here -->
84               </div>
85             </div>
86
87             <div class="panel-body msg-container-base">
88               <chat-message
89                 *ngFor="let message of messages | async"
90                 [message]="message">
91             </chat-message>
92           </div>
93
94           <div class="panel-footer">
95             <div class="input-group">
96               <input type="text"
97                 class="chat-input"
98                 placeholder="Write your message here..."
99                 (keydown.enter)="onEnter($event)"
100                 [(ngModel)]="draftMessage.text" />
101             <span class="input-group-btn">
102               <button class="btn-chat"
103                 (click)="onEnter($event)"
104                 >Send</button>
105             </span>
```

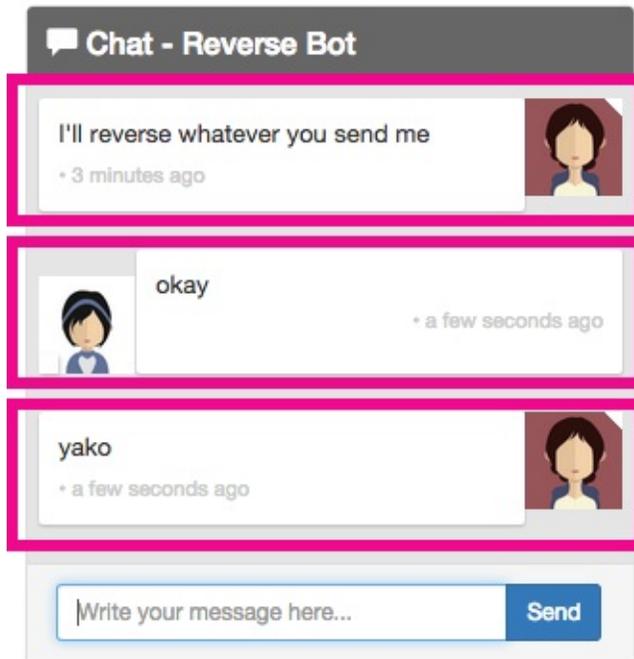
```

106         </div>
107     </div>
108
109     </div>
110 </div>
111 </div>
112 </div>
113
114 })
115 export class ChatWindow implements OnInit {
116     messages: Observable<any>;
117     currentThread: Thread;
118     draftMessage: Message;
119     currentUser: User;
120
121     constructor(public messagesService: MessagesService,
122                 public threadsService: ThreadsService,
123                 public userService: UserService,
124                 public el: ElementRef) {
125     }
126
127     ngOnInit(): void {
128         this.messages = this.threadsService.currentThreadMessages;
129
130         this.draftMessage = new Message();
131
132         this.threadsService.currentThread.subscribe(
133             (thread: Thread) => {
134                 this.currentThread = thread;
135             });
136
137         this.userService.currentUser
138             .subscribe(
139                 (user: User) => {
140                     this.currentUser = user;
141                 });
142
143         this.messages
144             .subscribe(
145                 (messages: Array<Message>) => {
146                     setTimeout(() => {
147                         this.scrollToBottom();
148                     });
149                 });
150     }
151
152     onEnter(event: any): void {
153         this.sendMessage();
154         event.preventDefault();
155     }
156
157     sendMessage(): void {
158         let m: Message = this.draftMessage;
159         m.author = this.currentUser;
160         m.thread = this.currentThread;
161         m.isRead = true;
162         this.messagesService.addMessage(m);
163         this.draftMessage = new Message();
164     }
165
166     scrollToBottom(): void {
167         let scrollPane: any = this.el
168             .nativeElement.querySelector('.msg-container-base');
169         scrollPane.scrollTop = scrollPane.scrollHeight;
170     }
171
172 }

```

The chatMessage Component

Each Message is rendered by the ChatMessage component.



ChatMessage

ChatMessage

ChatMessage

The ChatMessage Component

This component is relatively straightforward. The main logic here is rendering a slightly different view depending on if the message was authored by the current user. If the Message was **not** written by the current user, then we consider the message incoming.

We start by defining the @Component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
17 @Component({
18   inputs: ['message'],
19   selector: 'chat-message',
20   pipes: [FromNowPipe],
```

Setting incoming

Remember that each ChatMessage belongs to one Message. So in ngOnInit we will subscribe to the currentUser stream and set incoming depending on if this Message was written by the current user:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
43 export class ChatMessage implements OnInit {
44   message: Message;
45   currentUser: User;
46   incoming: boolean;
47
48   constructor(public userService: UserService) {
49   }
50
51   ngOnInit(): void {
52     this.userService.currentUser
53       .subscribe(
54         (user: User) => {
55           this.currentUser = user;
56           if (this.message.author && user) {
57             this.incoming = this.message.author.id !== user.id;
58           }
59         });
60   }
```

The chatMessage template

In our template we have two interesting ideas:

1. the FromNowPipe
2. [ngClass]

First, here's the code:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
17 @Component({
18   inputs: ['message'],
19   selector: 'chat-message',
20   pipes: [FromNowPipe],
21   template: `
22     <div class="msg-container"
23       [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">
24
25       <div class="avatar"
26         *ngIf="!incoming">
27         
28       </div>
29
30       <div class="messages"
31         [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
32         <p>{{message.text}}</p>
33         <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
34       </div>
35
36       <div class="avatar"
37         *ngIf="incoming">
38         
39       </div>
40     </div>
41 `
42 })
```

The FromNowPipe is a pipe that casts our Messages sent-at time to a human-readable “x seconds ago” message. You can see that we use it by: `{{message.sentAt | fromNow}}`



FromNowPipe uses the excellent [moment.js](#) library. If you'd like to learn about creating your own custom pipes, checkout the [Pipes](#) chapter (forthcoming). You can also read the source of the FromNowPipe in `code/rxjs/chat/app/ts/util/FromNowPipe.ts`

We also make extensive use of ngClass in this view. The idea is, when we say:

```
1 [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

We're asking Angular to apply the msg-receive class if incoming is truthy (and apply msg-sent if incoming is falsey).

By using the incoming property, we're able to display incoming and outgoing messages differently.

The Complete chatMessage Code Listing

Here's our completed ChatMessage component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 import {
2   Component,
3   OnInit,
4   ElementRef,
5   ChangeDetectionStrategy
6 } from '@angular/core';
7 import {FORM_DIRECTIVES} from '@angular/common';
8 import {
9   MessagesService,
10  ThreadsService,
11  UserService
12 } from '../services/services';
13 import {FromNowPipe} from '../util/FromNowPipe';
14 import {Observable} from 'rxjs';
15 import {User, Thread, Message} from '../models';
16
17 @Component({
18   inputs: ['message'],
19   selector: 'chat-message',
20   pipes: [FromNowPipe],
21   template: `
22 <div class="msg-container"
23   [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">
24
25   <div class="avatar"
26     *ngIf="!incoming">
27     
28   </div>
29
30   <div class="messages"
31     [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
32     <p>{{message.text}}</p>
33     <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
34   </div>
35
36   <div class="avatar"
37     *ngIf="incoming">
38     
39   </div>
40 </div>
41 `
42 })
43 export class ChatMessage implements OnInit {
44   message: Message;
45   currentUser: User;
46   incoming: boolean;
47
48   constructor(public userService: UserService) {
49   }
50
51   ngOnInit(): void {
52     this.userService.currentUser
53       .subscribe(
54         (user: User) => {
55           this.currentUser = user;
56           if (this.message.author && user) {
57             this.incoming = this.message.author.id !== user.id;
58           }
59         });
60   }
61 }
62 }
```

The chatNavBar Component

The last component we have to talk about is the ChatNavBar. In the nav-bar we'll show an unread messages count to the user.

Echo Bot •

The Unread Count in the ChatNavBar Component



The best way to try out the unread messages count is to use the “Waiting Bot”. If you haven’t already, try sending the message ‘3’ to the Waiting Bot and then switch to another window. The Waiting Bot will then wait 3 seconds before sending you a message and you will see the unread messages counter increment.

The chatNavBar @Component

First we define a pretty plain @Component configuration:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
6 @Component({
7   selector: 'nav-bar',
```

The chatNavBar Controller

The only thing the ChatNavBar controller needs to keep track of is the unreadMessagesCount. This is slightly more complicated than it seems on the surface.

The most straightforward way would be to simply listen to messagesService.messages and sum the number of Messages where isRead is false. This works fine for all messages outside of the current thread. However new messages in the current thread aren’t guaranteed to be marked as read by the time messages emits new values.

The safest way to handle this is to combine the messages and currentThread streams and make sure we don’t count any messages that are part of the current thread.

We do this using the combineLatest operator, which we’ve already used earlier in the chapter:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
26 export class ChatNavBar implements OnInit {
27   unreadMessagesCount: number;
28
29   constructor(public messagesService: MessagesService,
30               public threadsService: ThreadsService) {
31   }
32
33   ngOnInit(): void {
34     this.messagesService.messages
35       .combineLatest(
36         this.threadsService.currentThread,
37         (messages: Message[], currentThread: Thread) =>
38           [currentThread, messages] )
39
40     .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
41       this.unreadMessagesCount =
42         _reduce(
43           messages,
44           (sum: number, m: Message) => {
45             let messageIsInCurrentThread: boolean = m.thread &&
46               currentThread &&
47               (currentThread.id === m.thread.id);
48             if (m && !m.isRead && !messageIsInCurrentThread) {
```

```

49         sum = sum + 1;
50     }
51     return sum;
52 },
53     0);
54 });
55 }
56 }

```

If you're not an expert in TypeScript you might find the above syntax a little bit hard to parse. In the `combineLatest` callback function we're returning an array with `currentThread` and `messages` as its two elements.

Then we subscribe to that stream and we're *destructuring* those objects in the function call. Next we reduce over the messages and count the number of messages that are unread and not in the current thread.

The chatNavBar template

In our view, the only thing we have left to do is display our `unreadMessagesCount`:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```

6 @Component({
7   selector: 'nav-bar',
8   template: `
9     <nav class="navbar navbar-default">
10      <div class="container-fluid">
11        <div class="navbar-header">
12          <a class="navbar-brand" href="https://ng-book.com/2">
13            
14            ng-book 2
15          </a>
16        </div>
17        <p class="navbar-text navbar-right">
18          <button class="btn btn-primary" type="button">
19            Messages <span class="badge">{{unreadMessagesCount}}</span>
20          </button>
21        </p>
22      </div>
23    </nav>
24  `
25 })

```

The Completed chatNavBar

Here's the full code listing for ChatNavBar:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```

1 import {Component, OnInit} from '@angular/core';
2 import {MessagesService, ThreadsService} from '../services/services';
3 import {Message, Thread} from '../models';
4 import * as _ from 'underscore';
5
6 @Component({
7   selector: 'nav-bar',
8   template: `
9     <nav class="navbar navbar-default">
10      <div class="container-fluid">
11        <div class="navbar-header">
12          <a class="navbar-brand" href="https://ng-book.com/2">
13            
14            ng-book 2
15          </a>
16        </div>
17        <p class="navbar-text navbar-right">
18          <button class="btn btn-primary" type="button">
19            Messages <span class="badge">{{unreadMessagesCount}}</span>

```

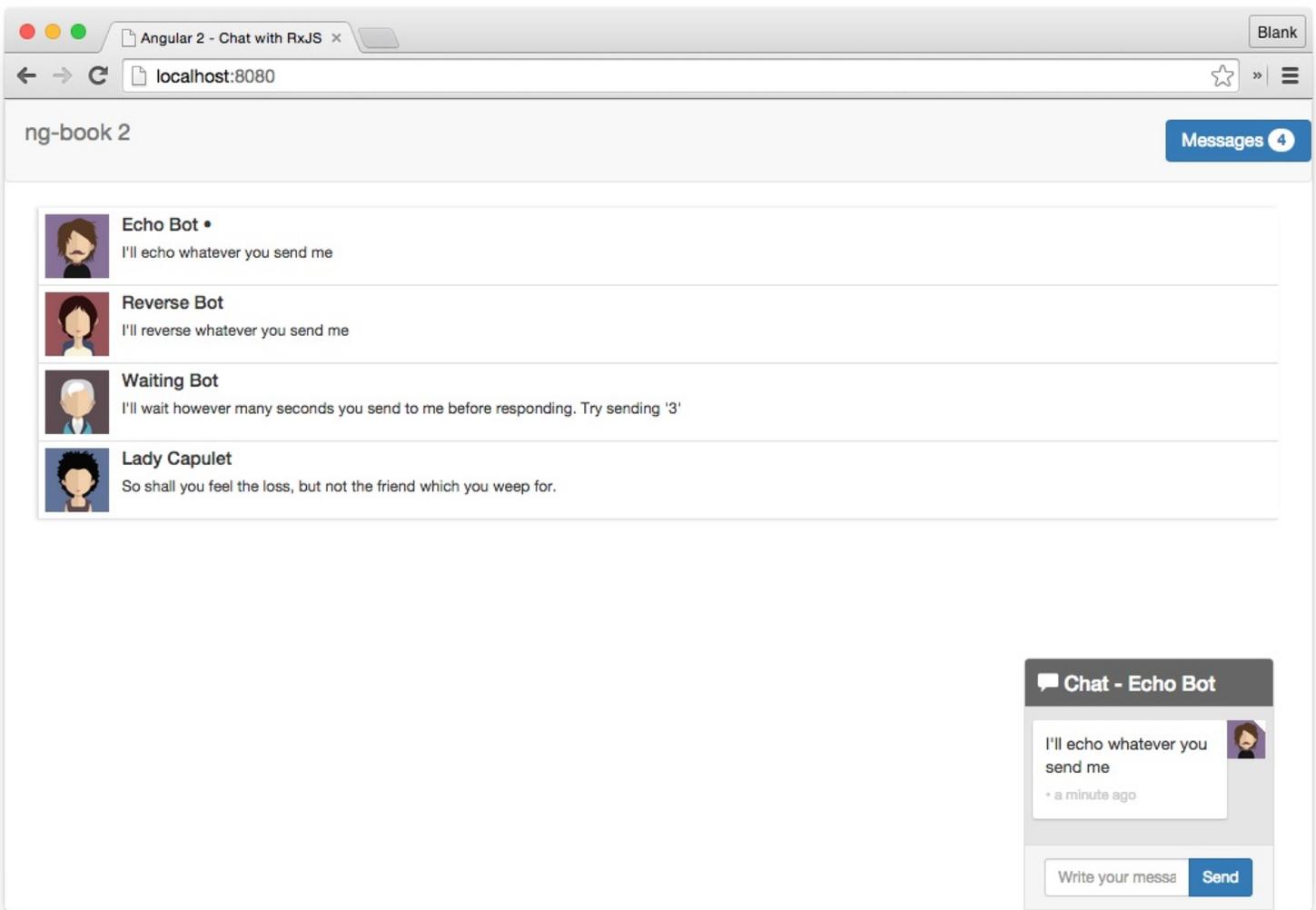
```

20     </button>
21     </p>
22 </div>
23 </nav>
24
25 })
26 export class ChatNavBar implements OnInit {
27     unreadMessagesCount: number;
28
29     constructor(public messagesService: MessagesService,
30                 public threadsService: ThreadsService) {
31     }
32
33     ngOnInit(): void {
34         this.messagesService.messages
35             .combineLatest(
36                 this.threadsService.currentThread,
37                 (messages: Message[], currentThread: Thread) =>
38                 [currentThread, messages] )
39
40         .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
41             this.unreadMessagesCount =
42                 _.reduce(
43                     messages,
44                     (sum: number, m: Message) => {
45                         let messageIsInCurrentThread: boolean = m.thread &&
46                             currentThread &&
47                             (currentThread.id === m.thread.id);
48                         if (m && !m.isRead && !messageIsInCurrentThread) {
49                             sum = sum + 1;
50                         }
51                         return sum;
52                     },
53                     0);
54         });
55     }
56 }

```

Summary

There we go, if we put them all together we've got a fully functional chat app!



Completed Chat Application

If you checkout `code/rxjs/chat/app/ts/ChatExampleData.ts` you'll see we've written a handful of bots for you that you can chat with. Here's a code excerpt from the Reverse Bot:

```
1 let rev: User = new User("Reverse Bot", require("images/avatars/female-avatar-4.\n2 png"));  
3 let tRev: Thread = new Thread("tRev", rev.name, rev.avatarSrc);
```

`code/rxjs/chat/app/ts/ChatExampleData.ts`

```
86 // reverse bot  
87 messagesService.messagesForThreadUser(tRev, rev)  
88   .forEach( (message: Message): void => {  
89     messagesService.addMessage(  
90       new Message({  
91         author: rev,  
92         text: message.text.split('').reverse().join(''),  
93         thread: tRev  
94       })  
95     );
```

Above you can see that we've subscribed to the messages for the "Reverse Bot" by using `messagesForThreadUser`. Try writing a few bots of your own.

Next Steps

Some ways to improve this chat app would be to become stronger at RxJS and then hook it up to an actual API. We'll talk about how to make API requests in the [HTTP Chapter](#). For now, enjoy your fancy chat application!

HTTP

Introduction

Angular comes with its own HTTP library which we can use to call out to external APIs.

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. To achieve this effect, our HTTP requests are *asynchronous*.

Dealing with *asynchronous* code is, historically, more tricky than dealing with synchronous code. In Javascript, there are generally three approaches to dealing with async code:

1. Callbacks
2. Promises
3. Observables

In Angular 2, the preferred method of dealing with async code is using Observables, and so that's what we'll cover in this chapter.

 **There's a whole chapter on RxJS and Observables:** In this chapter we're going to be using Observables and not explaining them much. If you're just starting to read this book at this chapter, you should know that there's [a whole chapter on Observables](#) that goes into RxJS in more detail.

In this chapter we're going to:

1. show a basic example of http
2. create a YouTube search-as-you-type component
3. discuss API details about the http library

 **Sample Code** The complete code for the examples in this chapter can be found in the http folder of the sample code. That folder contains a README.md which gives instructions for building and running the project.

Try running the code while reading the chapter and feel free play around to get a deeper insight about how it all works.

Using @angular/http

HTTP has been split into a separate module in Angular 2. This means that to use it you need to import constants from @angular/http. For instance, we might import constants from @angular/http like this:

```
1 import { Http, Response, RequestOptions, Headers } from '@angular/http';
```

```
import from @angular/http
```

In our app.ts we're going to import HTTP_PROVIDERS which is a convenience collection of modules.

code/http/app/ts/app.ts

```
4 import {  
5   Component  
6 } from '@angular/core';  
7 import { bootstrap } from '@angular/platform-browser-dynamic';  
8 import { HTTP_PROVIDERS } from '@angular/http';
```

When we bootstrap our app we will add HTTP_PROVIDERS as a dependency. The effect is that we will be able to inject Http (and a few other modules) into our components.

```
1 bootstrap(HttpApp, [ HTTP_PROVIDERS ]);
```

Now we can inject the Http service into our components (or anywhere we use DI, actually).

```
1 class MyFooComponent {  
2   constructor(public http: Http) {  
3   }  
4  
5   makeRequest(): void {  
6     // do something with this.http ...  
7   }  
8 }
```

A Basic Request

The first thing we're going to do is make a simple GET request to the [jsonplaceholder API](#).

What we're going to do is:

1. Have a button that calls makeRequest
2. makeRequest will call the http library to perform a GET request on our API
3. When the request returns, we'll update this.data with the results of the data, which will be rendered in the view.

Here's a screenshot of our example:

Basic Request

Make Request

```
{  
  "userId": 1,  
  "id": 1,  
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",  
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"  
}
```

Building the simpleHTTPComponent @Component

The first thing we're going to do is import a few modules and then specify a selector for our @Component:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
4 import {Component} from '@angular/core';
5 import {Http, Response} from '@angular/http';
6
7 @Component({
8   selector: 'simple-http',
```

Building the simpleHTTPComponent template

Next we build our view:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
7 @Component({
8   selector: 'simple-http',
9   template: `
10 <h2>Basic Request</h2>
11 <button type="button" (click)="makeRequest()">Make Request</button>
12 <div *ngIf="loading">loading...</div>
13 <pre>{{data | json}}</pre>
14 `
15 })
```

First we specify that we're going to use the NgIf directive.

Our template has three interesting parts:

1. The button
2. The loading indicator
3. The data

On the button we bind to (click) to call the makeRequest function in our controller, which we'll define in a minute.

We want to indicate to the user that our request is loading, so to do that we will show loading... if the instance variable loading is true, using ngIf.

The data is an Object. A great way to debug objects is to use the json pipe as we do here. We've put this in a pre tag to give us nice, easy to read formatting.

Building the simpleHTTPComponent Controller

We start by defining a new class for our SimpleHTTPComponent:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
16 export class SimpleHTTPComponent {
17   data: Object;
18   loading: boolean;
```

We have two instance variables: `data` and `loading`. This will be used for our API return value and loading indicator respectively.

Next we define our constructor:

```
code/http/app/ts/components/SimpleHTTPComponent.ts
```

```
20 constructor(public http: Http) {
21 }
```

The constructor body is empty, but we inject one key module: `Http`.



Remember that when we use the `public` keyword in `public http: Http` TypeScript will assign `http` to `this.http`. It's a shorthand for:

```
1 // other instance variables here
2 http: Http;
3
4 constructor(http: Http) {
5     this.http = http;
6 }
```

Now let's make our first HTTP request by implementing the `makeRequest` function:

```
code/http/app/ts/components/SimpleHTTPComponent.ts
```

```
23 makeRequest(): void {
24     this.loading = true;
25     this.http.request('http://jsonplaceholder.typicode.com/posts/1')
26         .subscribe((res: Response) => {
27             this.data = res.json();
28             this.loading = false;
29         });
30 }
31 }
```

When we call `makeRequest`, the first thing we do is set `this.loading = true`. This will turn on the loading indicator in our view.

To make an HTTP request is straightforward: we call `this.http.request` and pass the URL to which we want to make a GET request.

`http.request` returns an `Observable`. We can subscribe to changes (akin to using `then` from a `Promise`) using `subscribe`.

```
code/http/app/ts/components/SimpleHTTPComponent.ts
```

```
26         .subscribe((res: Response) => {
27             this.data = res.json();
28             this.loading = false;
29         });
```

When our `http.request` returns (from the server) the stream will emit a `Response` object. We extract the body of the response as an `Object` by using `json` and then we set `this.data` to that `Object`.

Since we have a response, we're not loading anymore so we set `this.loading = false`



.subscribe can also handle failures and stream completion by passing a function to the second and third arguments respectively. In a production app it would be a good idea to handle those cases, too. That is, this.loading should also be set to false if the request fails (i.e. the stream emits an error).

Full SimpleHTTPComponent

Here's what our SimpleHTTPComponent looks like altogether:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 /*
2  * Angular
3  */
4 import {Component} from '@angular/core';
5 import {Http, Response} from '@angular/http';
6
7 @Component({
8   selector: 'simple-http',
9   template: `
10 <h2>Basic Request</h2>
11 <button type="button" (click)="makeRequest()">Make Request</button>
12 <div *ngIf="loading">loading...</div>
13 <pre>{{data | json}}</pre>
14 `
15 })
16 export class SimpleHTTPComponent {
17   data: Object;
18   loading: boolean;
19
20   constructor(public http: Http) {
21   }
22
23   makeRequest(): void {
24     this.loading = true;
25     this.http.request('http://jsonplaceholder.typicode.com/posts/1')
26       .subscribe((res: Response) => {
27         this.data = res.json();
28         this.loading = false;
29       });
30   }
31 }
```

Writing a YouTubeSearchComponent

The last example was a minimal way to get the data from an API server into your code. Now let's try to build a more involved example.

In this section, we're going to build a way to search YouTube as you type. When the search returns we'll show a list of video thumbnail results, along with a description and link to each video.

Here's a screenshot of what happens when I search for "cats playing ipads":

YouTube Search

cats playing ipads|



Funny Cats Playing On iPads Compilation - Funny Videos 2015

You may or may not be surprised, but there are many animals playing on tablet computer. New video funny 2015 Thanks for watching, rating the video and ...

Watch



Animals Playing On iPads Compilation

You may or may not be surprised, but there are many animals playing on tablet computer. Join Us On Facebook
<http://www.facebook.com/Compilariz>
No ...

Watch



Cute cats try to catch a mouse from an iPad

Cute cats try to catch a mouse from an iPad.

Watch



Charlie The Cat - Kitten Playing iPad 2 !!! Game For Cats Cute Funny Clever Pets Bloopers

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos - <http://youtu.be/xZHwYNrfWd0>
Check My Other Videos Kitten HArlem Shake ...

Watch



Cats playing "Game for Cats" with Apple iPad

Two Siberian cats like to play "Game for Cats" with Apple iPad :) Note that the iPad has Invisible Shield screen protector. Siperiankissat leikkivät



White Tiger Plays iPad - Game for Cats Gone Wild! Lions, servals, and more!

<http://www.ipadgameforcats.com>
and
<http://www.conservatorscenter.org/>



Cat Plays with iPad - Friskies Games for Cats

Mr. Kitty playing Cat Fishing on my girlfriends 1st gen iPad, via Friskies Games for Cats
<http://www.gamesforcats.com>.



Cute Cat plays on iPad

Cute Cat plays on iPad.

Watch

Can I get my cat to write Angular 2?

For this example we're going to write several things:

1. A `SearchResult` object that will hold the data we want from each result
2. A `YouTubeService` which will manage the API request to YouTube and convert the results to a stream of `SearchResult[]`
3. A `SearchBox` component which will call out to the YouTube service as the user types
4. A `SearchResultComponent` which will render a specific `SearchResult`
5. A `YouTubeSearchComponent` which will encapsulate our whole YouTube searching app and render the list of results

Let's handle each part one at a time.



Patrick Stapleton has an excellent repository named [angular2-webpack-starter](#). This repo has an RxJS example which autocompletes Github repositories. Some of the ideas in this section are inspired from that example. It's a fantastic project with lots of examples and you should check it out.

Writing a SearchResult

First let's start with writing a basic SearchResult class. This class is just a convenient way to store the specific fields we're interested in from our search results.

code/http/app/ts/components/YouTubeSearchComponent.ts

```
31 class SearchResult {
32   id: string;
33   title: string;
34   description: string;
35   thumbnailUrl: string;
36   videoUrl: string;
37
38   constructor(obj?: any) {
39     this.id = obj && obj.id || null;
40     this.title = obj && obj.title || null;
41     this.description = obj && obj.description || null;
42     this.thumbnailUrl = obj && obj.thumbnailUrl || null;
43     this.videoUrl = obj && obj.videoUrl ||
44       `https://www.youtube.com/watch?v=${this.id}`;
45   }
46 }
```

This pattern of taking an `obj?: any` lets us simulate keyword arguments. The idea is that we can create a new SearchResult and just pass in an object containing the keys we want to specify.

The only thing to point out here is that we're constructing the `videoUrl` using a hard-coded URL format. You're welcome to change this to a function which takes more arguments, or use the `video id` directly in your view to build this URL if you need to.

Writing the YouTubeService

The API

For this example we're going to be using [the YouTube v3 search API](#).



In order to use this API you need to have an API key. I've included an API key in the sample code which you can use. However, by the time you read this, you may find it's over the rate limits. If that happens, you'll need to issue your own key.

To issue your own key [see this documentation](#). For simplicities sake, I've registered a server key, but you should probably use a browser key if you're going to put your javascript code online.

We're going to setup two constants for our YouTubeService mapping to our API key and the API URL:

```
1 let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
2 let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

Eventually we're going to want to test our app. One of the things we find when testing is that we don't always want to test against production - we often want to test against staging or a development API.

To help with this environment configuration, one of the things we can do is **make these constants injectable**.

Why should we inject these constants instead of just using them in the normal way? Because if we make them injectable we can

1. have code that injects the right constants for a given environment at deploy time and
2. replace the injected value easily at test-time

By injecting these values, we have a lot more flexibility about their values down the line.

In order to make these values injectable, we use the `bind(...).toValue(...)` syntax like this:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
83 export var youtubeServiceInjectables: Array<any> = [  
84   bind(YouTubeService).toClass(YouTubeService),  
85   bind(YOUTUBE_API_KEY).toValue(YOUTUBE_API_KEY),  
86   bind(YOUTUBE_API_URL).toValue(YOUTUBE_API_URL)  
87 ];
```

Here we're specifying that we want to bind `YOUTUBE_API_KEY` "injectably" to the value of `YOUTUBE_API_KEY`. (Same for `YOUTUBE_API_URL`, and we'll define `YouTubeService` in a minute.)

If you recall, to make something available to be injected throughout our application, we need to make it a dependency at bootstrap. Since we're exporting `youtubeServiceInjectables` here we can import it in our `app.ts`

```
1 // http/app.ts  
2 import { youtubeServiceInjectables } from "components/YouTubeSearchComponent";  
3  
4 // ....  
5 // further down  
6 bootstrap(HttpApp, [ HTTP_PROVIDERS, youtubeServiceInjectables ]);
```

Now we can inject `YOUTUBE_API_KEY` instead of using the variable directly.

YouTubeService constructor

We create our `YouTubeService` by making a class and annotating it as `@Injectable`:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
48 /**  
49  * YouTubeService connects to the YouTube API  
50  * See: * https://developers.google.com/youtube/v3/docs/search/list  
51  */  
52 @Injectable()  
53 export class YouTubeService {  
54   constructor(public http: Http,  
55               @Inject(YOUTUBE_API_KEY) private apiKey: string,  
56               @Inject(YOUTUBE_API_URL) private apiUrl: string) {  
57   }
```

In the constructor we inject three things:

1. Http
2. YOUTUBE_API_KEY
3. YOUTUBE_API_URL

Notice that we make instance variables from all three arguments, meaning we can access them as `this.http`, `this.apiKey`, and `this.apiUrl` respectively.

Notice that we explicitly inject using the `@Inject(YOUTUBE_API_KEY)` notation.

YouTubeService search

Next let's implement the search function. `search` takes a query string and returns an Observable which will emit a stream of `SearchResult[]`. That is, each item emitted is an *array* of `SearchResults`.

code/http/app/ts/components/YouTubeSearchComponent.ts

```
59 search(query: string): Observable<SearchResult[]> {
60   let params: string = [
61     `q=${query}`,
62     `key=${this.apiKey}`,
63     `part=snippet`,
64     `type=video`,
65     `maxResults=10`
66   ].join('&');
67   let queryUrl: string = `${this.apiUrl}?${params}`;
```

We're building the `queryUrl` in a manual way here. We start by simply putting the query params in the `params` variable. (You can find the meaning of each of those values by [reading the search API docs](#).)

Then we build the `queryUrl` by concatenating the `apiUrl` and the `params`.

Now that we have a `queryUrl` we can make our request:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
68 return this.http.get(queryUrl)
69   .map((response: Response) => {
70     return (<any>response.json()).items.map(item => {
71       // console.log("raw item", item); // uncomment if you want to debug
72       return new SearchResult({
73         id: item.id.videoId,
74         title: item.snippet.title,
75         description: item.snippet.description,
76         thumbnailUrl: item.snippet.thumbnails.high.url
77       });
78     });
79   });
```

Here we take the return value of `http.get` and use `map` to get the `Response` from the request. From that response we extract the body as an object using `.json()` and then we iterate over each item and convert it to a `SearchResult`.



If you'd like to see what the raw `item` looks like, just uncomment the `console.log` and inspect it in your browser's developer console.



Notice that we're calling `(<any>response.json()).items`. What's going on here? We're telling TypeScript that we're not interested in doing strict type checking.

When working with a JSON API, we don't generally have typing definitions for the API responses, and so TypeScript won't know that the `Object` returned even has an `items` key, so the compiler will complain.

We could call `response.json()["items"]` and then cast that to an `Array` etc., but in this case (and in creating the `SearchResult`, it's just cleaner to use an `any` type, at the expense of strict type checking

YouTubeService Full Listing

Here's the full listing of our `YouTubeService`:

`code/http/app/ts/components/YouTubeSearchComponent.ts`

```

48 /**
49  * YouTubeService connects to the YouTube API
50  * See: * https://developers.google.com/youtube/v3/docs/search/list
51  */
52 @Injectable()
53 export class YouTubeService {
54   constructor(public http: Http,
55               @Inject(YOUTUBE_API_KEY) private apiKey: string,
56               @Inject(YOUTUBE_API_URL) private apiUrl: string) {
57   }
58
59   search(query: string): Observable<SearchResult[]> {
60     let params: string = [
61       `q=${query}`,
62       `key=${this.apiKey}`,
63       `part=snippet`,
64       `type=video`,
65       `maxResults=10`
66     ].join('&');
67     let queryUrl: string = `${this.apiUrl}?${params}`;
68     return this.http.get(queryUrl)
69       .map((response: Response) => {
70         return (<any>response.json()).items.map(item => {
71           // console.log("raw item", item); // uncomment if you want to debug
72           return new SearchResult({
73             id: item.id.videoId,
74             title: item.snippet.title,
75             description: item.snippet.description,
76             thumbnailUrl: item.snippet.thumbnails.high.url
77           });
78         });
79       });
80   }
81 }

```

Writing the searchBox

The `SearchBox` component plays a key role in our app: it is the mediator between our UI and the `YouTubeService`.

The `SearchBox` will:

1. Watch for keyup on an input and submit a search to the YouTubeService
2. Emit a loading event when we're loading (or not)
3. Emit a results event when we have new results

SearchBox @Component Definition

Let's define our SearchBox @Component:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
89 /**
90  * SearchBox displays the search box and emits events based on the results
91  */
92
93 @Component({
94   outputs: ['loading', 'results'],
95   selector: 'search-box',
```

The selector we've seen many times before: this allows us to create a <search-box> tag.

The outputs key specifies events that will be emitted from this component. That is, we can use the (output)="callback()" syntax in our view to listen to events on this component. For example, here's how we will use the search-box tag in our view later on:

```
1 <search-box
2   (results)="updateResults($event)"
3   (loading)="loading = $event"
4 ></search-box>
```

In this example, when the SearchBox component emits a loading event, we will set the variable loading in the parent context. Likewise, when the SearchBox emits a results event, we will call the updateResults() function, with the value, in the parent's context.

In the @Component configuration we're simply specifying the names of the events with the strings "loading" and "results". In this example, each event will have a corresponding EventEmitter as an *instance variable of the controller class*. We'll implement that in a few minutes.

For now, remember that @Component is like the public API for our component, so here we're just specifying the name of the events, and we'll worry about implementing the EventEmitters later.

SearchBox template Definition

Our template is straightforward. We have one input tag:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
93 @Component({
94   outputs: ['loading', 'results'],
95   selector: 'search-box',
96   template: `
97     <input type="text" class="form-control" placeholder="Search" autofocus>
98   `
99 })
```

searchBox Controller Definition

Our SearchBox controller is a new class:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
100 class SearchBox implements OnInit {
101   loading: EventEmitter<boolean> = new EventEmitter<boolean>();
```

We say that this class implements `OnInit` because we want to use the `ngOnInit` lifecycle callback. If a class implements `OnInit` then the `ngOnInit` function will be called after the first change detection check.

`ngOnInit` is a good place to do initialization (vs. the constructor) because inputs set on a component are not available in the constructor.

SearchBox Controller Definition constructor

Let's talk about the `SearchBox` constructor:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
104 constructor(public youtube: YouTubeService,
105             private el: ElementRef) {
106 }
```

In our constructor we inject:

1. Our `YouTubeService` and
2. The element `el` that this component is attached to. `el` is an object of type `ElementRef`, which is an Angular wrapper around a native element.

We set both injections as instance variables.

SearchBox Controller Definition ngOnInit

On this input box we want to watch for `keyup` events. The thing is, if we simply did a search after every `keyup` that wouldn't work very well. There are three things we can do to improve the user experience:

1. Filter out any empty or short queries
2. "debounce" the input, that is, don't search on every character but only after the user has stopped typing after a short amount of time
3. discard any old searches, if the user has made a new search

We could manually bind to `keyup` and call a function on each `keyup` event and then implement filtering and debouncing from there. However, there is a better way: turn the `keyup` events into an observable stream.

`RxJS` provides a way to listen to events on an element using `Rx.Observable.fromEvent`. We can use it like so:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
108 ngOnInit(): void {
109     // convert the `keyup` event into an observable stream
110     Observable.fromEvent(this.el.nativeElement, 'keyup')
```

Notice that in `fromEvent`:

- the first argument is `this.el.nativeElement` (the native DOM element this component is attached to)

- the second argument is the string 'keyup', which is the name of the event we want to turn into a stream

We can now perform some RxJS magic over this stream to turn it into `SearchResults`. Let's walk through step by step.

Given the stream of `keyup` events we can chain on more methods. In the next few paragraphs we're going to chain several functions on to our stream which will transform the stream. Then at the end we'll show the whole example together.

First, let's extract the value of the input tag:

```
1 .map((e: any) => e.target.value) // extract the value of the input
```

Above says, map over each `keyup` event, then find the event target (`e.target`, that is, our input element) and extract the `value` of that element. This means our stream is now a stream of strings.

Next:

```
1 .filter((text: string) => text.length > 1)
```

This `filter` means the stream will not emit any search strings for which the length is less than one. You could set this to a higher number if you want to ignore short searches.

```
1 .debounceTime(250)
```

`debounceTime` means we will throttle requests that come in faster than 250ms. That is, we won't search on every keystroke, but rather after the user has paused a small amount.

```
1 .do(() => this.loading.next(true)) // enable loading
```

Using `do` on a stream is way to perform a function mid-stream for each event, but it does not change anything in the stream. The idea here is that we've got our search, it has enough characters, and we've debounced, so now we're about to search, so we turn on loading.

`this.loading` is an `EventEmitter`. We "turn on" loading by emitting `true` as the next event. We emit something on an `EventEmitter` by calling `next`. Writing `this.loading.next(true)` means, emit a `true` event on the loading `EventEmitter`. When we listen to the loading event on this component, the `$event` value will now be `true` (we'll look more closely at using `$event` below).

```
1 .map((query: string) => this.youtube.search(query))
2 .switch()
```

We use `.map` to call perform a search for each query that is emitted. By using `switch` we're, essentially, saying "ignore all search events but the most recent" [1](#). That is, if a new search comes in, we want to use the most recent and discard the rest.

For each query that comes in, we're going to perform a search on our `YouTubeService`.

Putting the chain together we have this:

<code/http/app/ts/components/YouTubeSearchComponent.ts>

```

110 Observable.fromEvent(this.el.nativeElement, 'keyup')
111 .map((e: any) => e.target.value) // extract the value of the input
112 .filter((text: string) => text.length > 1) // filter out if empty
113 .debounceTime(250) // only once every 250ms
114 .do(() => this.loading.next(true)) // enable loading
115 // search, discarding old events if new input comes in
116 .map((query: string) => this.youtube.search(query))
117 .switch()

```

The API of RxJS can be a little intimidating because the API surface area is large. That said, we've implemented a sophisticated event-handling stream in very few lines of code!

Because we are calling out to our YouTubeService our stream is now a stream of SearchResult[]. We can subscribe to this stream and perform actions accordingly.

subscribe takes three arguments: onSuccess, onError, onCompletion.

code/http/app/ts/components/YouTubeSearchComponent.ts

```

119 .subscribe(
120   (results: SearchResult[]) => { // on success
121     this.loading.next(false);
122     this.results.next(results);
123   },
124   (err: any) => { // on error
125     console.log(err);
126     this.loading.next(false);
127   },
128   () => { // on completion
129     this.loading.next(false);
130   }
131 );

```

The first argument specifies what we want to do when the stream emits a regular event. Here we emit an event on both of our EventEmitters:

1. We call this.loading.next(false), indicating we've stopped loading
2. We call this.results.next(results), which will emit an event containing the list of results

The second argument specifies what should happen when the stream has an event. Here we set this.loading.next(false) and log out the error.

The third argument specifies what should happen when the stream completes. Here we also emit that we're done loading.

SearchBox Component: Full Listing

All together, here's the full listing of our SearchBox Component:

code/http/app/ts/components/YouTubeSearchComponent.ts

```

89 /**
90  * SearchBox displays the search box and emits events based on the results
91  */
92
93 @Component({
94   outputs: ['loading', 'results'],
95   selector: 'search-box',
96   template: `
97     <input type="text" class="form-control" placeholder="Search" autofocus>
98   `
99 })
100 class SearchBox implements OnInit {
101   loading: EventEmitter<boolean> = new EventEmitter<boolean>();
102   results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>();

```

```

103 constructor(public youtube: YouTubeService,
104             private el: ElementRef) {
105 }
106
107
108 ngOnInit(): void {
109     // convert the `keyup` event into an observable stream
110     Observable.fromEvent(this.el.nativeElement, 'keyup')
111     .map((e: any) => e.target.value) // extract the value of the input
112     .filter((text: string) => text.length > 1) // filter out if empty
113     .debounceTime(250) // only once every 250ms
114     .do(() => this.loading.next(true)) // enable loading
115     // search, discarding old events if new input comes in
116     .map((query: string) => this.youtube.search(query))
117     .switch()
118     // act on the return of the search
119     .subscribe(
120         (results: SearchResult[]) => { // on success
121             this.loading.next(false);
122             this.results.next(results);
123         },
124         (err: any) => { // on error
125             console.log(err);
126             this.loading.next(false);
127         },
128         () => { // on completion
129             this.loading.next(false);
130         }
131     );
132 }
133 }
134 }

```

Writing SearchResultComponent

The SearchBox was pretty complicated. Let's handle a **much** easier component now: the SearchResultComponent. The SearchResultComponent's job is to render a single SearchResult.

There's not really any new ideas here, so let's take it all at once:

code/http/app/ts/components/YouTubeSearchComponent.ts

```

136 @Component({
137     inputs: ['result'],
138     selector: 'search-result',
139     template: `
140         <div class="col-sm-6 col-md-3">
141             <div class="thumbnail">
142                 
143                 <div class="caption">
144                     <h3>{{result.title}}</h3>
145                     <p>{{result.description}}</p>
146                     <p><a href="{{result.videoUrl}}">
147                         class="btn btn-default" role="button">watch</a></p>
148                 </div>
149             </div>
150         </div>
151     `
152 })
153 export class SearchResultComponent {
154     result: SearchResult;
155 }

```

A few things:

The @Component takes a single input result, on which we will put the SearchResult assigned to this component.



Charlie The Cat - Kitten Playing iPad 2 !!! Game For Cats Cute Funny Clever Pets Bloopers

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos - <http://youtu.be/xZHwYNrFwD0> Check My Other Videos Kitten HArlem Shake ...

Watch

Single Search Result Component

The template shows the title, description, and thumbnail of the video and then links to the video via a button.

The `SearchResultComponent` simply stores the `SearchResult` in the instance variable `result`.

Writing `YouTubeSearchComponent`

The last component we have to implement is the `YouTubeSearchComponent`. This is the component that ties everything together.

`YouTubeSearchComponent @Component`

`code/http/app/ts/components/YouTubeSearchComponent.ts`

```
157 @Component({
158   selector: 'youtube-search',
159   directives: [SearchBox, SearchResultComponent],
```

Our `@Component` annotation is straightforward: use the selector `youtube-search`.

We also specify that we want to use our two custom components: `SearchBox` and `SearchResultComponent` as directives.

`YouTubeSearchComponent Controller`

Before we look at the template, let's take a look at the `YouTubeSearchComponent` controller:

`code/http/app/ts/components/YouTubeSearchComponent.ts`

```
189 export class YouTubeSearchComponent {
190   results: SearchResult[];
191
192   updateResults(results: SearchResult[]): void {
193     this.results = results;
194     // console.log("results:", this.results); // uncomment to take a look
195   }
196 }
```

This component holds one instance variable: `results` which is an array of `SearchResults`.

We also define one function: `updateResults`. `updateResults` simply takes whatever new `SearchResult[]` it's given and sets `this.results` to the new value.

We'll use both `results` and `updateResults` in our template.

`YouTubeSearchComponent template`

Our view needs to do three things:

1. Show the loading indicator, if we're loading
2. Listen to events on the search-box
3. Show the search results

Next let's look at our template. Let's build some basic structure and show the loading gif next to the header:

`code/http/app/ts/components/YouTubeSearchComponent.ts`

```
160 template: `
161 <div class='container'>
162   <div class="page-header">
163     <h1>YouTube Search
164     <img
165       style="float: right;"
166       *ngIf="loading"
167       src='${loadingGif}' />
168   </h1>
169 </div>
```



Notice that our `img` has a `src` of `${loadingGif}` - that `loadingGif` variable came from a `require` statement earlier in the program. Here we're taking advantage of webpack's image loading feature. If you want to learn more about how this works, take a look at the webpack config in the sample code for this chapter or checkout [image-webpack-loader](#).

We only want to show this loading image if `loading` is true, so we use `ngIf` to implement that functionality.

Next, let's look at the markup where we use our `search-box`:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
171 <div class="row">
172   <div class="input-group input-group-lg col-md-12">
173     <search-box
174       (loading)="loading = $event"
175       (results)="updateResults($event)"
176     ></search-box>
177   </div>
178 </div>
```

The interesting part here is how we bind to the `loading` and `results` outputs. Notice, that we use the `(output)="action()"` syntax here.

For the `loading` output, we run the expression `loading = $event`. `$event` will be substituted with the value of the event that is emitted from the `EventEmitter`. That is, in our `SearchBox` component, when we call `this.loading.next(true)` then `$event` will be true.

Similarly, for the `results` output, we call the `updateResults()` function whenever a new set of results are emitted. This has the effect of updating our component's `results` instance variable.

Lastly, we want to take the list of `results` in this component and render a `search-result` for each one:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
180 <div class="row">
181   <search-result
182     *ngFor="let result of results"
183     [result]="result">
184   </search-result>
185 </div>
186 </div>
187 `
188 })
```

YouTubeSearchComponent Full Listing

Here's the full listing for the `YouTubeSearchComponent`:

```
157 @Component({
158   selector: 'youtube-search',
159   directives: [SearchBox, SearchResultComponent],
160   template: `
161     <div class='container'>
162       <div class="page-header">
163         <h1>YouTube Search
164         <img
165           style="float: right;"
166           *ngIf="loading"
167           src='${loadingGif}' />
168       </h1>
169     </div>
170
171     <div class="row">
172       <div class="input-group input-group-lg col-md-12">
173         <search-box
174           (loading)="loading = $event"
175           (results)="updateResults($event)"
176         ></search-box>
177       </div>
178
179       <div class="row">
180         <search-result
181           *ngFor="let result of results"
182           [result]="result">
183         </search-result>
184       </div>
185     </div>
186   `
187 })
188 export class YouTubeSearchComponent {
189   results: SearchResult[];
190
191   updateResults(results: SearchResult[]): void {
192     this.results = results;
193     // console.log("results:", this.results); // uncomment to take a look
194   }
195 }
196 }
```

There we have it! A functional search-as-you-type implemented for YouTube videos! Try running it from the code examples if you haven't already.

@angular/http API

Of course, all of the HTTP requests we've made so far have simply been GET requests. It's important that we know how we can make other requests too.

Making a post request

Making POST request with @angular/http is very much like making a GET request except that we have one additional parameter: a body.

[jsonplaceholder API](#) also provides a convent URL for testing our POST requests, so let's use it for a POST:

```
code/http/app/ts/components/MoreHTTPRequests.ts
```

```
33 makePost(): void {
34   this.loading = true;
35   this.http.post(
36     'http://jsonplaceholder.typicode.com/posts',
37     JSON.stringify({
38       body: 'bar',
39       title: 'foo',
40       userId: 1
```

```
41     })))
42     .subscribe((res: Response) => {
43       this.data = res.json();
44       this.loading = false;
45     });
46   }
```

Notice in the second argument we're taking an Object and converting it to a JSON string using `JSON.stringify`.

PUT / PATCH / DELETE / HEAD

There are a few other fairly common HTTP requests and we call them in much the same way.

- `http.put` and `http.patch` map to PUT and PATCH respectively and both take a URL and a body
- `http.delete` and `http.head` map to DELETE and HEAD respectively and both take a URL (no body)

Here's how we might make a DELETE request:

`code/http/app/ts/components/MoreHTTPRequests.ts`

```
48 makeDelete(): void {
49   this.loading = true;
50   this.http.delete('http://jsonplaceholder.typicode.com/posts/1')
51     .subscribe((res: Response) => {
52       this.data = res.json();
53       this.loading = false;
54     });
55 }
```

RequestOptions

All of the `http` methods we've covered so far also take an optional last argument: `RequestOptions`. The `RequestOptions` object encapsulates:

- method
- headers
- body
- mode
- credentials
- cache
- url
- search

Let's say we want to craft a GET request that uses a special `X-API-TOKEN` header. We can create a request with this header like so:

`code/http/app/ts/components/MoreHTTPRequests.ts`

```
57 makeHeaders(): void {
58   let headers: Headers = new Headers();
59   headers.append('X-API-TOKEN', 'ng-book');
60
61   let opts: RequestOptions = new RequestOptions();
62   opts.headers = headers;
63
64   this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
65     .subscribe((res: Response) => {
66       this.data = res.json();
67     });
68 }
```

Summary

@angular/http is still young but it's already full-featured enough for a wide variety of APIs.

One of the great things about @angular/http is that it has support for mocking the backend which is very useful in testing. To learn about testing HTTP, flip on over to [the testing chapter](#).

Routing

In web development, *routing* means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

For instance, if we visit the / path of a website, we may be visiting the **home route** of that website. Or if we visit /about we want to render the “about page”, and so on.

Why Do We Need Routing?

Defining routes in our application is useful because we can:

- separate different areas of the app;
- maintain the state in the app;
- protect areas of the app based on certain rules;

For example, imagine we are writing an inventory application similar to the one we described in previous chapters.

When we first visit the application, we might see a search form where we can enter a search term and get a list of products that match that term.

After that, we might click a given product to visit that product’s details page.

Because our app is client-side, it’s not technically required that we change the URL when we change “pages”. But it’s worth thinking about for a minute: what would be the consequences of using the same URL for all pages?

- You wouldn’t be able to refresh the page and keep your location within the app
- You wouldn’t be able to bookmark a page and come back to it later
- You wouldn’t be able to share the URL of that page with others

Or put in a positive light, routing lets us define a URL string that specifies where within our app a user should be.

In our inventory example we could determine a series of different routes for each activity, for instance:

The initial root URL could be represented by `http://our-app/`. When we visit this page, we could be redirected to our “home” route at `http://our-app/home`.

When accessing the ‘About Us’ area, the URL could become `http://our-app/about`. This way if we sent the URL `http://our-app/about` to another user they would see same page.

How client-side routing works

Perhaps you've written server-side routing code before (though, it isn't necessary to complete this chapter). Generally with server-side routing, the HTTP request comes in and the server will render a different controller depending on the incoming URL.

For instance, with [Express.js](#) you might write something like this:

```
1 var express = require('express');
2 var router = express.Router();
3
4 // define the about route
5 router.get('/about', function(req, res) {
6   res.send('About us');
7 });
```

Or with [Ruby on Rails](#) you might have:

```
1 # routes.rb
2 get '/about', to: 'pages#about'
3
4 # PagesController.rb
5 class PagesController < ActionController::Base
6   def about
7     render
8   end
9 end
```

The pattern varies per framework, but in both of these cases you have a **server** that accepts a request and *routes* to a **controller** and the controller runs a specific **action**, depending on the path and parameters.

Client-side routing is very similar in concept but different in implementation. With client-side routing **we're not necessarily making a request to the server** on every URL change. With our Angular apps, we refer to them as “Single Page Apps” (SPA) because our server only gives us a single page and it's our JavaScript that renders the different pages.

So how can we have different routes in our JavaScript code?

The beginning: using anchor tags

Client-side routing started out with a clever hack: Instead of using the page page, instead use the *anchor tag* as the client-side URL.

As you may already know, anchor tags were traditionally used to link directly to a place *within* the webpage and make the browser scroll all the way to where that anchor was defined. For instance, if we define an anchor tag in an HTML page:

```
1 <!-- ... lots of page content here ... -->
2 <a name="about"><h1>About</h1></a>
```

And we visited the URL `http://something/#about`, the browser would jump straight to that H1 tag that identified by the about anchor.

The clever move for client-side frameworks used for SPAs was to take the anchor tags and use them represent the routes within the app by formatting them as paths.

For example, the about route for an SPA would be something like `http://something/#/about`. This is

what is known as **hash-based routing**.

What's neat about this trick is that it looks like a "normal" URL because we're starting our anchor with a slash (/about).

The evolution: HTML5 client-side routing

With the introduction of HTML5, browsers acquired the ability to programmatically create new browser history entries that change the displayed URL *without the need for a new request*.

This is achieved using the `history.pushState` method that exposes the browser's navigational history to JavaScript.

So now, instead of relying on the anchor hack to navigate routes, modern frameworks can rely on `pushState` to perform history manipulation without reloads.



Angular 1 Note: This way of routing already works in Angular 1, but it needs to be explicitly enabled using `$locationProvider.html5Mode(true)`.

In Angular 2, however, the HTML5 is the default mode. Later in this chapter we show how to change from HTML5 mode to the old anchor tag mode.



There's two things you need to be aware of when using HTML5 mode routing, though

1. Not all browsers support HTML5 mode routing, so if you need to support older browsers you might be stuck with hash-based routing for a while.
2. **The server has to support HTML5 based routing.**

It may not be immediately clear why the server has to support HTML5 based-routing, we'll talk more about why later in this chapter.

Writing our first routes

In Angular we configure routes by mapping *paths* to the component that will handle them.

Let's create a small app that has multiple routes. On this sample application we will have 3 routes:

- A main page route, using the `/#/home` path;
- An about page, using the `/#/about` path;
- A contact us page, using the `/#/contact` path;

And when the user visits the root path (`/#/`), it will redirect to the home path.

Components of Angular 2 routing

There are three main components that we use to configure routing in Angular:

- RouterConfig describes the routes our application supports
- RouterOutlet is a “placeholder” component that gets expanded to each route’s content
- RouterLink directive is used to link to routes

Let’s look at each one more closely.

RouterConfig

To define routes for our application, create a RouterConfig and then use provideRouter() to provide our application with the dependencies necessary to use the router:

code/routes/basic/app/ts/app.ts

```
46 const routes: RouterConfig = [  
47   { path: '', redirectTo: 'home', terminal: true },  
48   { path: 'home', component: HomeComponent },  
49   { path: 'about', component: AboutComponent },  
50   { path: 'contact', component: ContactComponent },  
51   { path: 'contactus', redirectTo: 'contact' },  
52 ];  
53  
54 bootstrap(RoutesDemoApp, [  
55   provideRouter(routes), // <-- installs our routes  
56   provide(LocationStrategy, { useClass: HashLocationStrategy })  
57 ]);
```

Notice a few things about the routes:

- path specifies the URL this route will handle
- component is what ties a given route path to a component that will handle the route
- the optional redirectTo is used to redirect a given path to an existing route

As a summary, the goal of routes is to specify which component will handle a given path.

Redirections

When we use redirectTo on a route definition, it will tell the router that when we visit the path of the route, we want the browser to be redirected to another route.

In our sample code above, if we visit the root path at <http://localhost:8080/#/>, we’ll be redirected to the route home.

Another example is the contactus route:

code/routes/basic/app/ts/app.ts

```
51 { path: 'contactus', redirectTo: 'contact' },
```

In this case, if we visit the URL <http://localhost:8080/#/contactus>, we’ll see that the browser redirects to /contact.



Sample Code The complete code for the examples in this section can be found in the `routes/basic` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

There are many different `imports` required for routing and we don't list every single one in every code example below. However we do list the filename and line number from which almost every example is taken from. If you're having trouble figuring out how to `import` a particular class, open up the code using your editor to see the entire code listing.

Try running the code while reading this section and feel free play around to get a deeper insight about how it all works.

RouterOutlet using `<router-outlet>`

Our component `@View` has a template which specifies some `div` structure, a section for Navigation, and a directive called `router-outlet`.

When we change routes, we want to keep our outer “layout” template and only substitute the “inner section” of the page with the route's component.

In order to describe to Angular where in our page we want to render the contents for each route, we use the `RouterOutlet` directive.

The `router-outlet` element indicates where the contents of each route component will be rendered.

To use it, declare the `ROUTER_DIRECTIVES` as one of the directives your component needs and add a `<router-outlet>` tag to your HTML:



`ROUTER_DIRECTIVES` is an array that holds all router directives we need to import, including `RouterOutlet`.

`code/routes/basic/app/ts/app.ts`

```
25 @Component({
26   selector: 'router-app',
27   directives: [ROUTER_DIRECTIVES],
28   template: `
29     <div>
30       <nav>
31         <a>Navigation:</a>
32         <ul>
33           <li><a [routerLink]='['home']">Home</a></li>
34           <li><a [routerLink]='['about']">About</a></li>
35           <li><a [routerLink]='['contact']">Contact us</a></li>
36         </ul>
37       </nav>
38
39       <router-outlet></router-outlet>
40     </div>
41   `
42 })
43 class RoutesDemoApp {
44 }
```

If we look at the template contents above, you will note the `router-outlet` element right below the navigation menu. When we visit `/home`, that's where `HomeComponent` template will be rendered. The

same happens for the other components.

RouterLink using [routerLink]

Now that we know where route templates will be rendered, how do we tell Angular 2 to navigate to a given route?

We might try linking to the routes directly using pure HTML:

```
1 <a href="/#/home">Home</a>
```

But if we do this, we'll notice that clicking the link triggers a page reload and that's definitely not what we want when programming single page apps.

To solve this problem, Angular 2 provides a solution that can be used to link to routes **with no page reload**: the RouterLink directive.

This directive allows you to write links using a special syntax:

code/routes/basic/app/ts/app.ts

```
31 <a>Navigation:</a>
32 <ul>
33   <li><a [routerLink]='['home']">Home</a></li>
34   <li><a [routerLink]='['about']">About</a></li>
35   <li><a [routerLink]='['contact']">Contact us</a></li>
36 </ul>
```

We can see on the left-hand side the [routerLink] that applies the directive to the current element (in our case a tags).

Now, on the right-hand side we have an array with the route path as the first element, like "['home']" or "['about']" that will indicate which route to navigate to when we click the element.

It might seem a little odd that the value of routerLink is a string with an array containing a string (" ['home']", for example). This is because there are more things you can provide when linking to routes, but we'll look at this into more detail when we talk about child routes and route parameters.

For now, we're only using routes names from the root app component.

Putting it all together

So now that we have all the basic pieces, let's make them work together to transition from one route to the other.

The first thing we need to write for our application is the index.html file.

Here's the full code for that:

code/routes/basic/app/index.html

```
1 <!doctype html>
2 <html>
3   <head>
4     <base href="/">
5     <title>ng-book 2: Angular 2 Router</title>
6
```

```

7   {% for (var css in o.htmlWebpackPlugin.files.css) { %}
8     <link href="{%=o.htmlWebpackPlugin.files.css[css] %}" rel="stylesheet">
9     {% } %}
10  </head>
11  <body>
12    <router-app></router-app>
13    <script src="/core.js"></script>
14    <script src="/vendor.js"></script>
15    <script src="/bundle.js"></script>
16  </body>
17 </html>

```



The section describing `htmlWebpackPlugin` comes from the [webpack module bundler](#). We're using webpack in this chapter because it's a tool for bundling your assets

The code should be familiar by now, with the exception of this line:

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

This line declares the base HTML tag. This tag is traditionally used to tell the browser where to look for images and other resources declared using relative paths.

It turns out Angular Router also relies on this tag to determine how to construct its routing information.

For instance, if we have a route with a path of `/hello` and our base element declares `href="/app"`, the application will use `/app/#` as the concrete path.

Sometimes though, coders of an Angular application don't have access to the head section of the application HTML. This is true for instance, when reusing headers and footers of a larger, pre-existing application.

Fortunately there is a workaround for this cases. You can declare the application base path programmatically, when bootstrapping the Angular application:

```

1 bootstrap(RoutesDemoApp, [
2   ROUTER_PROVIDERS,
3   provide(APP_BASE_HREF, {useValue: '/'})
4 ]);

```

Injecting the result from `provide(APP_BASE_HREF, {useValue: '/'})` is the equivalent of using `<base href="/">` on our application HTML header.

Creating the Components

Before we get to the main app component, let's create 3 simple components, one for each of the routes.

HomeComponent

The HomeComponent will just have an `h1` tag that says "welcome!". Here's the full code for our HomeComponent:

`code/routes/basic/app/ts/components/HomeComponent.ts`

```

1 /*
2  * Angular

```

```
3  */
4  import {Component} from '@angular/core';
5
6  @Component({
7    selector: 'home',
8    template: `<h1>Welcome!</h1>`
9  })
10 export class HomeComponent {
11 }
```

AboutComponent

Similarly, the AboutComponent will just have a basic h1:

code/routes/basic/app/ts/components/AboutComponent.ts

```
1  /*
2   * Angular
3   */
4  import {Component} from '@angular/core';
5
6  @Component({
7    selector: 'about',
8    template: `<h1>About</h1>`
9  })
10 export class AboutComponent {
11 }
```

ContactComponent

And, likewise with AboutComponent:

code/routes/basic/app/ts/components/ContactComponent.ts

```
1  /*
2   * Angular
3   */
4  import {Component} from '@angular/core';
5
6  @Component({
7    selector: 'contact',
8    template: `<h1>Contact Us</h1>`
9  })
10 export class ContactComponent {
11 }
```

Nothing really very interesting about those components, so let's move on to the main app.ts file.

Application component

Now we need to create the root-level “application” component that will tie everything together.

We start with the imports we'll need, both from the core and router bundles:

code/routes/basic/app/ts/app.ts

```
1  /*
2   * Angular Imports
3   */
4  import {provide, Component} from '@angular/core';
5  import {bootstrap} from '@angular/platform-browser-dynamic';
6  import {
7    ROUTER_DIRECTIVES,
8    provideRouter,
9    RouterConfig,
10 } from '@angular/router';
11 import {LocationStrategy, HashLocationStrategy} from '@angular/common';
```

Next step is to import the three components we created above:

```
code/routes/basic/app/ts/app.ts
```

```
16 import {HomeComponent} from 'components/HomeComponent';
17 import {AboutComponent} from 'components/AboutComponent';
18 import {ContactComponent} from 'components/ContactComponent';
```

Now let's get to the real component code. We start with the declaration of the component selector, directives and template:

```
code/routes/basic/app/ts/app.ts
```

```
25 @Component({
26   selector: 'router-app',
27   directives: [ROUTER_DIRECTIVES],
28   template: `
29     <div>
30       <nav>
31         <a>Navigation:</a>
32         <ul>
33           <li><a [routerLink]='['home']">Home</a></li>
34           <li><a [routerLink]='['about']">About</a></li>
35           <li><a [routerLink]='['contact']">Contact us</a></li>
36         </ul>
37       </nav>
38
39       <router-outlet></router-outlet>
40     </div>
41 `
42 })
43 class RoutesDemoApp {
44 }
```

For this component, we're going to use two router directives: RouterOutlet and the RouterLink. Those directives, along with all other *common* router directives are declared on a special array called ROUTER_DIRECTIVES. When we import that constant on our directives area, it's basically importing all of those directives for us, without the need to do it one by one.

As a recap, the RouterOutlet directive is then used to indicate where in our template the route contents should be rendered. That's represented by the <router-outlet></router-outlet> snippet in our template code.

The RouterLink directive is used to create navigation links to our routes:

```
code/routes/basic/app/ts/app.ts
```

```
31     <a>Navigation:</a>
32     <ul>
33       <li><a [routerLink]='['home']">Home</a></li>
34       <li><a [routerLink]='['about']">About</a></li>
35       <li><a [routerLink]='['contact']">Contact us</a></li>
36     </ul>
```

Using [routerLink] will instruct Angular to take ownership of the click event and then initiate a route switch to the right place, based on the route definition.

Configuring the Routes

Next, we declare the routes creating an array of objects that conform to the RouterConfig type:

```
code/routes/basic/app/ts/app.ts
```

```
46 const routes: RouterConfig = [
47   { path: '', redirectTo: 'home', terminal: true },
48   { path: 'home', component: HomeComponent },
49   { path: 'about', component: AboutComponent },
```

```
50 { path: 'contact', component: ContactComponent },
51 { path: 'contactus', redirectTo: 'contact' },
52 ];
```

In the last section of the `app.ts` file, we bootstrap the application:

```
code/routes/basic/app/ts/app.ts
```

```
5 import {bootstrap} from '@angular/platform-browser-dynamic';
```

Just like we have been doing so far, we are now bootstrapping the app and telling that `RoutesDemoApp` is the root component.

The difference now is that we're providing a second parameter to the `bootstrap` method. This second parameter is an array of injectables you're injecting into our application.

The first thing we're injecting is: `provideRouter(routes)`. `provideRouter()` is a function that will take our routes, configure the router, and return a list of dependencies like `RouteRegistry`, `Location`, and several other classes that are necessary to make routing work.

The second seems a little more complicated:

```
1 provide(LocationStrategy, {useClass: HashLocationStrategy})
```

Let's take an in depth look of what we want to achieve with this line.

Routing Strategies

The way the Angular application parses and creates paths from and to route definitions is now *location strategy*.

 In Angular 1 this is called *routing modes* instead

The default strategy is `PathLocationStrategy`, which is what we call HTML5 routing. While using this strategy, routes are represented by regular paths, like `/home` or `/contact`.

We can change the location strategy used for our application by binding the `LocationStrategy` class to a new, concrete strategy class.

Instead of using the default `PathLocationStrategy` we can also use the `HashLocationStrategy`.

The reason we're using the hash strategy as a default is because if we were using HTML5 routing, our URLs would end up being regular paths (that is, not using hash/anchor tags).

This way, the routes would work when you click a link and navigate on the client side, let's say from `/about` to `/contact`.

If we were to refresh the page, instead of asking the server for the root URL, which is what is being served, instead we'd be asking for `/about` or `/contact`. Because there's no known page at `/about` the

server would return a 404.

This default strategy works with hash based paths, like `/#/home` or `/#/contact` that the server understands as being the `/` path. (This is also the default mode in Angular 1.)

 Let's say you want to use HTML5 mode in production, what can you do?

In order to use HTML5 mode routing, you have to configure your server to redirect every “missing” route to the root URL.

In the `routes/basic` project we've included a script you can use to develop with `webpack-dev-server` and use HTML5 paths at the same time.

To use it `cd routes/basic` and run `node htm15-dev-server.js`.

Finally, in order to make our example application work with this new strategy, first we have to import `LocationStrategy` and `HashLocationStrategy`:

`code/routes/basic/app/ts/app.ts`

```
11 import {LocationStrategy, HashLocationStrategy} from '@angular/common';
```

and then just add that link to the bootstrap section:

`code/routes/basic/app/ts/app.ts`

```
5 import {bootstrap} from '@angular/platform-browser-dynamic';
```

 **You could write your own strategy if you wanted to.** All you need to do is extend the `LocationStrategy` class and implement the methods. A good way to start is reading the Angular 2 source for the `HashLocationStrategy` or `PathLocationStrategy` classes.

Path location strategy

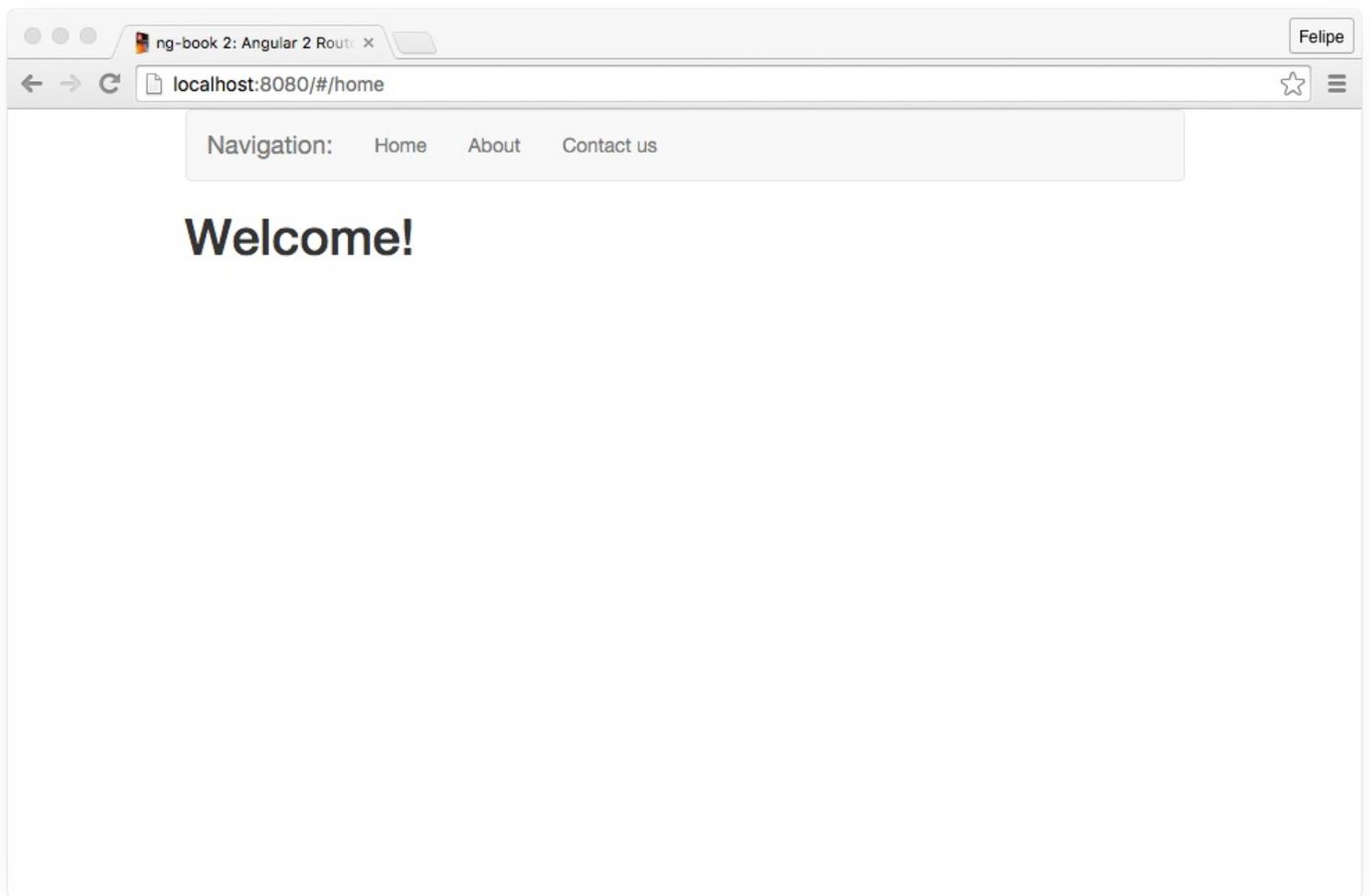
In our sample application folder, you'll find a file called `app/ts/app.html5.ts`.

If we want to play with the default `PathLocationStrategy`, we just need to copy the contents of that file to `app/ts/app.ts`, then reload the application.

Running the application

You can now go into the application root folder (`code/routes`) and run `npm run server` to boot the application.

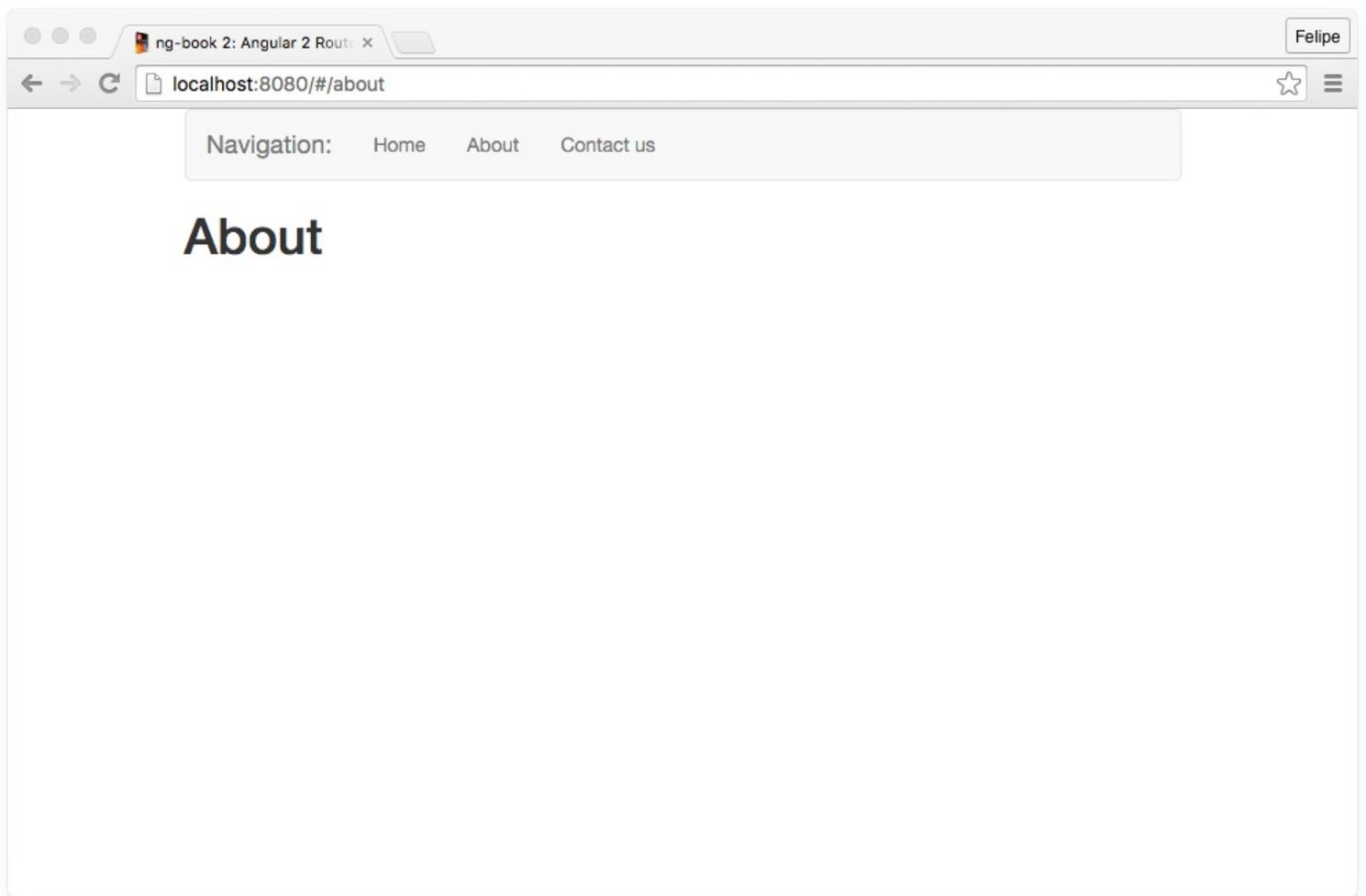
When you type <http://localhost:8080/> into your browser you should see the home route rendered:



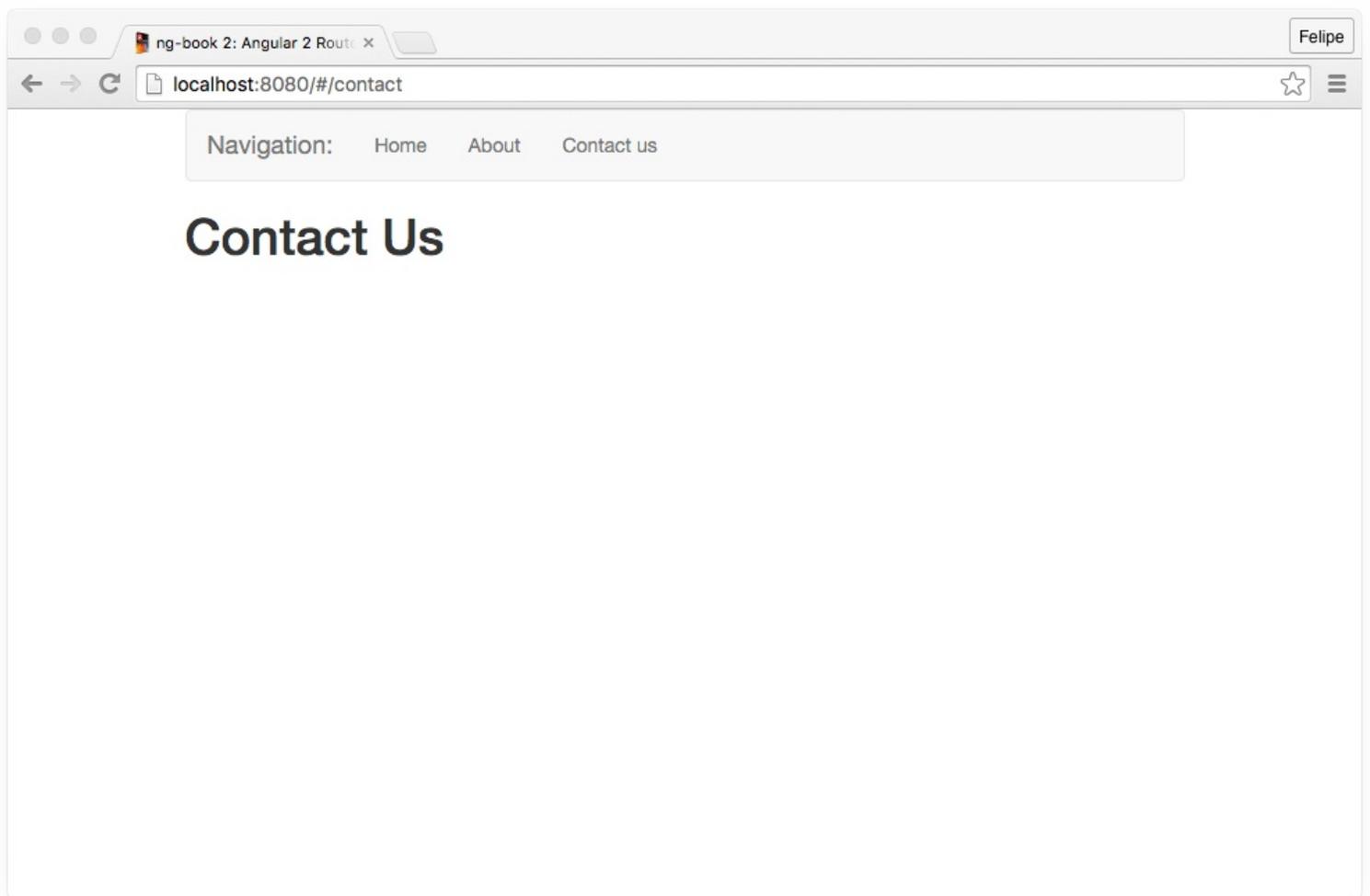
Home Route

Notice that the URL in the browser got redirected to <http://localhost:8080/#/home>.

Now clicking each link will render the appropriate routes:



About Route



Contact Us Route

Route Parameters

In our apps we often want to navigate to a specific resource. For instance, say we had a news website and we had many articles. Each article may have an ID, and if we had an article with ID 3 then we might navigate to that article by visiting the URL:

```
/articles/3
```

And if we had an article with an ID of 4 we would access it at

```
/articles/4
```

and so on.

Obviously we're not going to want to write a route for each article, but instead we want to use a variable, or *route parameter*. We can specify that a route takes a parameter by putting a colon `:` in front of the path segment like this:

```
/route/:param
```

So in our example news site, we might specify our route as:

/articles/:id

To add a parameter to our router configuration, we specify the route path like this:

code/routes/music/app/ts/app.ts

```
51 const routes: RouterConfig = [  
52   { path: '', redirectTo: 'search', terminal: true },  
53   { path: 'search', component: SearchComponent },  
54   { path: 'artists/:id', component: ArtistComponent },  
55   { path: 'tracks/:id', component: TrackComponent },  
56   { path: 'albums/:id', component: AlbumComponent },  
57 ];
```

When we visit the route /artist/123, the 123 part will be passed as the id route parameter to our route.

But how can we retrieve the parameter for a given route? That's where we use route parameters.

ActivatedRoute

In order to use route parameters, we need to first import ActivatedRoute:

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

Next, we inject the ActivatedRoute into the constructor of our component. For example, let's say we have a RouterConfig that specifies the following:

```
1 const routes: RouterConfig = [  
2   { path: 'articles/:id', component: ArticlesComponent }  
3 ];
```

Then when we write the ArticleComponent, we add the ActivatedRoute as one of the constructor arguments:

```
1 export class ArticleComponent {  
2   id: string;  
3  
4   constructor(private route: ActivatedRoute) {  
5     route.params.subscribe(params => { this.id = params['id']; });  
6   }  
7 }
```

Notice that route.params is an *observable*. We can extract the value of the param into a hard value by using .subscribe. In this case, we assign the value of params['id'] to the id instance variable on the component.

Now when we visit /articles/230, our component's id attribute should receive 230.

Music Search App

Let's now work on a more complex application. We will build a music search application that has the following features:

1. **Search for tracks** that match a given term
2. Show **matching tracks** in a grid
3. Show **singer details** when the singer name is clicked
4. Show **album details** and show a list of tracks when the album name is clicked

5. Show **song details** allow the user to **play a preview** when the song name is clicked

Sportify music for active people

Search

Results



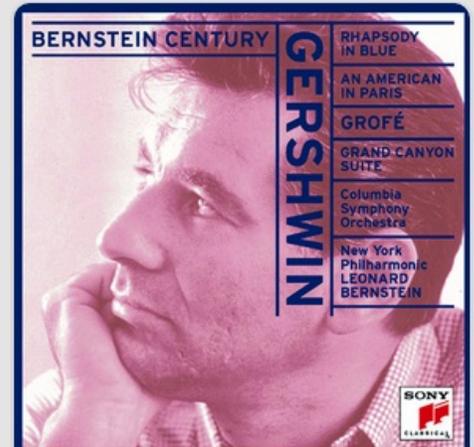
George Gershwin
Rhapsody In Blue

Gershwin: Rhapsody in Blue/An American in Paris



George Gershwin
Rhapsody In Blue

Gershwin Plays Gershwin: The Piano Rolls



George Gershwin
Rhapsody in Blue

Gershwin: Rhapsody in Blue / An American in Paris



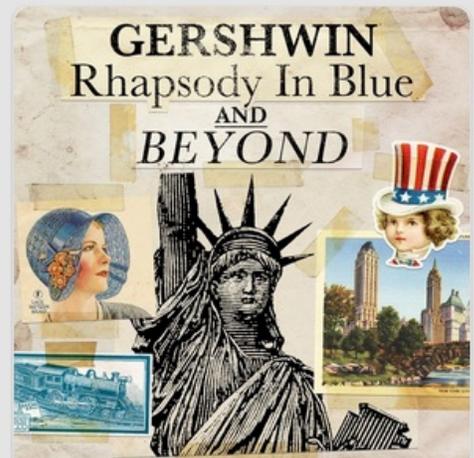
George Gershwin
Rhapsody in Blue

Gershwin: Piano Concerto in F, Rhapsody in



George Gershwin
Rhapsody in Blue

Gershwin: Rhapsody in Blue; Piano Concerto in



George Gershwin
Rhapsody in Blue

Gershwin - Rhapsody in Blue and Beyond

The Search View of our Music App

The routes we will need for this application will be:

- /search - search form and results

- /artists/:id - artist info, represented by a Spotify ID
- /albums/:id - album info, with a list of tracks using the Spotify ID
- /tracks/:id - track info and preview, also using the Spotify ID



Sample Code The complete code for the examples in this section can be found in the `routes/music` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

We will use the [Spotify API](#) to get information about tracks, artists and albums.

First Steps

The first file we need work on is `app.ts`. Let's start by importing classes we'll use from Angular:

`code/routes/music/app/ts/app.ts`

```

1  /*
2  *  Angular Imports
3  */
4  import {
5    Component,
6    provide
7  } from '@angular/core';
8  import {bootstrap} from '@angular/platform-browser-dynamic';
9  import {HTTP_PROVIDERS} from '@angular/http';
10 import {
11   ROUTER_DIRECTIVES,
12   provideRouter,
13   RouterConfig
14 } from '@angular/router';
15 import {
16   LocationStrategy,
17   HashLocationStrategy,
18   APP_BASE_HREF
19 } from '@angular/common';
20
21 /*
22 *  Components
23 */

```

Now that we have the imports there, let's think about the components we'll use for each route.

- For the Search route, we'll create a `SearchComponent`. This component will talk to the Spotify API to perform the search and then display the results on a grid.
- For the Artists route, we'll create an `ArtistComponent` which will show the artist's information
- For the Albums route, we'll create an `AlbumComponent` which will show the list of tracks in the album
- For the Tracks route, we'll create a `TrackComponent` which will show the track and let us play a preview of the song

Since this new component will need to interact with the Spotify API, it seems like we need to build a service that uses the `http` module to call out to the API server.

Everything in our app depends on the data, so let's build the `SpotifyService` first.

The `spotifyService`



You can find the full code for the SpotifyService in the routes/music/app/ts/services folder of the sample code.

The first method we'll implement is `searchByTrack` which will search for track, given a search term.

One of the endpoints documented on Spotify API docs is [the Search endpoint](#).

This endpoint does exactly what we want: it takes a query (using the `q` parameter) and a type parameter.

Query in this case is the search term. And since we're searching for songs, we should use `type=track`.

Here's what a first version of the service could look like:

```
1 class SpotifyService {
2   constructor(public http: Http) {
3   }
4
5   searchByTrack(query: string) {
6     let params: string = [
7       `q=${query}`,
8       `type=track`
9     ].join("&");
10    let queryURL: `https://api.spotify.com/v1/search?${params}`;
11    return this.http.request(queryURL).map(res => res.json());
12  }
13 }
```

This code performs an HTTP GET request to the URL <https://api.spotify.com/v1/search>, passing our query as the search term and type hardcoded to track.

This `http` call returns an `Observable`. We are going one step further and using the RxJS function `map` to transform the result we would get (which is an `http` module's `Response` object) and parsing it as JSON, resulting on an object.

Any function that calls `searchByQuery` then will have to use the `Observable` API to subscribe to the response like this:

```
1 service
2   .searchTrack('query')
3   .subscribe((res: any) => console.log('Got object', res))
```

The searchComponent

Now that we have a service that will perform track searches, we can start coding the `SearchComponent`.

Again, we start with an import section:

code/routes/music/app/ts/components/SearchComponent.ts

```
1 /*
2  * Angular
3  */
4
5 import {Component, OnInit} from '@angular/core';
6 import {CORE_DIRECTIVES} from '@angular/common';
7 import {
8   Router,
9   ROUTER_DIRECTIVES,
```

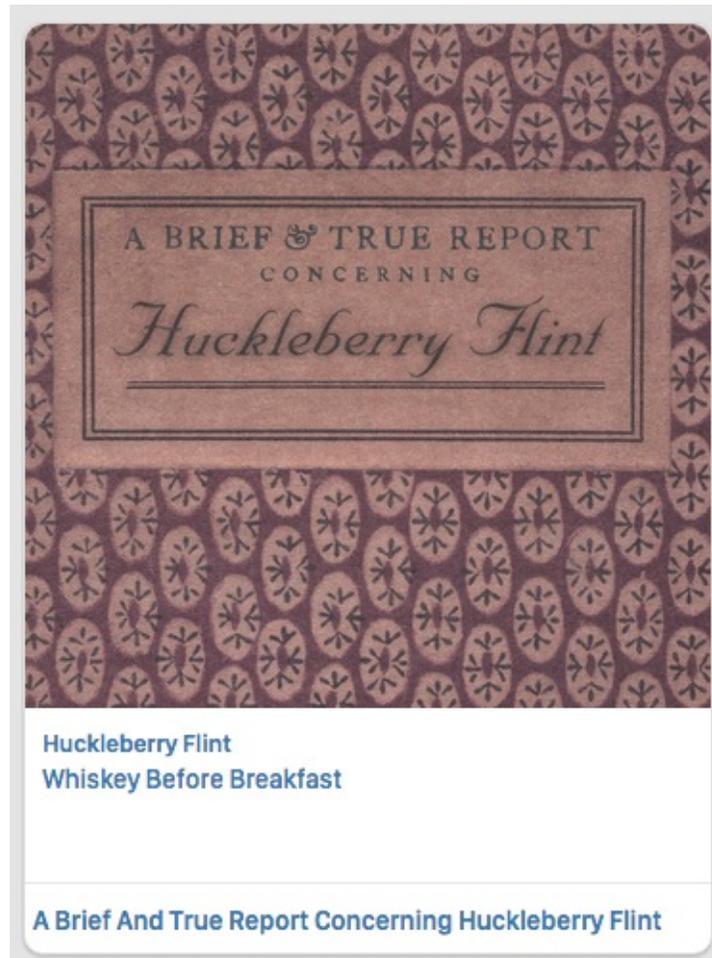
```

10   ActivatedRoute,
11 } from '@angular/router';
12
13 /*
14  * Services
15  */
16 import {SpotifyService} from 'services/SpotifyService';

```

Here we're importing, among other things, the SpotifyService class we just created.

The goal here is to render each resulting track side by side on a card like below:



Music App Card

We then start coding the component. We're using search as the selector, importing a few directives and using the following template. The template is a bit long because we're putting some reasonable styles on it, but it isn't particularly complicated, relative to what we've done so far:

code/routes/music/app/ts/components/SearchComponent.ts

```

18 @Component({
19   selector: 'search',
20   directives: [ROUTER_DIRECTIVES, CORE_DIRECTIVES],
21   template: `
22     <h1>Search</h1>
23
24     <p>
25       <input type="text" #newquery
26         [value]="query"
27         (keydown.enter)="submit(newquery.value)">
28       <button (click)="submit(newquery.value)">Search</button>
29     </p>
30
31     <div *ngIf="results">
32       <div *ngIf="!results.length">
33         No tracks were found with the term '{{ query }}'

```

```

34 </div>
35
36 <div *ngIf="results.length">
37   <h1>Results</h1>
38
39   <div class="row">
40     <div class="col-sm-6 col-md-4" *ngFor="let t of results">
41       <div class="thumbnail">
42         <div class="content">
43           
44           <div class="caption">
45             <h3>
46               <a [routerLink]="['/artists', t.artists[0].id]">
47                 {{ t.artists[0].name }}
48               </a>
49             </h3>
50             <br>
51             <p>
52               <a [routerLink]="['/tracks', t.id]">
53                 {{ t.name }}
54               </a>
55             </p>
56           </div>
57           <div class="attribution">
58             <h4>
59               <a [routerLink]="['/albums', t.album.id]">
60                 {{ t.album.name }}
61               </a>
62             </h4>
63           </div>
64         </div>
65       </div>
66     </div>
67   </div>
68 </div>
69 </div>
70 `
71 })

```

The Search Field

Let's break down the HTML template a bit.

This first section will have the search field:

code/routes/music/app/ts/components/SearchComponent.ts

```

24 <p>
25   <input type="text" #newquery
26     [value]="query"
27     (keydown.enter)="submit(newquery.value)">
28   <button (click)="submit(newquery.value)">Search</button>
29 </p>

```

Here we have the input field and we're binding its DOM element value property to the query property of our component.

We also give this element a template variable named #newquery. We can now access the value of this input within our template code by using newquery.value.

The button will trigger the submit method of the component, passing the value of the input field as a parameter.

We also want to trigger submit when the user hits "Enter" so we bind to the keydown.enter event on the input.

Search Results and Links

The next section displays the results. We're relying on the NgFor directive to iterate through each track from our results object:

```
code/routes/music/app/ts/components/SearchComponent.ts
```

```
39     <div class="row">
40       <div class="col-sm-6 col-md-4" *ngFor="let t of results">
41         <div class="thumbnail">
```

For each track, we display the artist name:

```
code/routes/music/app/ts/components/SearchComponent.ts
```

```
45         <h3>
46           <a [routerLink]="['/artists', t.artists[0].id]">
47             {{ t.artists[0].name }}
48           </a>
49         </h3>
```

Notice how we're using the RouterLink directive to redirect to `['/artists', t.artists[0].id]`.

This is how we set *route parameters* for a given route. Say we have an artist with an id `abc123`. When this link is clicked, the app would then navigate to `/artist/abc123` (where `abc123` is the `:id` parameter).

Further down we'll show how we can retrieve this value inside the component that handles this route.

Now we display the track:

```
code/routes/music/app/ts/components/SearchComponent.ts
```

```
51         <p>
52           <a [routerLink]="['/tracks', t.id]">
53             {{ t.name }}
54           </a>
55         </p>
```

And the album:

```
code/routes/music/app/ts/components/SearchComponent.ts
```

```
58         <h4>
59           <a [routerLink]="['/albums', t.album.id]">
60             {{ t.album.name }}
61           </a>
62         </h4>
```

SearchComponent Class

Let's take a look at the constructor first:

```
code/routes/music/app/ts/components/SearchComponent.ts
```

```
72 export class SearchComponent implements OnInit {
73   query: string;
74   results: Object;
75
76   constructor(private spotify: SpotifyService, private router: Router,
77               private route: ActivatedRoute) {
78     router.routerState.queryParams
79       .subscribe(params => { this.query = params['query'] || ''; });
80   }
```

Here we're declaring two properties:

- query for current search term and
- results for the search results

On the constructor we're injecting the `SpotifyService` (that we created above), `Router`, and the `ActivatedRoute` and making them properties of our class.

In our constructor we subscribe to the `queryParams` property - this lets us access *query parameters*, such as the search term (`params['query']`).

In a URL like: `http://localhost/#/search?query=cats&order=ascending`, `queryParams` gives us the parameters in an object. This means we could access the order with `params['order']` (in this case, `ascending`).

Also note that `queryParams` are different than `route.params`. Whereas `route.params` match parameters in the *route* `queryParams` match parameters in the query string.

In this case, if there is no query param, we set `this.query` to the empty string.

search

In our `SearchComponent` we will call out to the `SpotifyService` and render the results. There are two cases when we want to run a search:

We want to run a search when the user:

- enters a search query and submits the form
- navigates to this page with a given URL in the query parameters (e.g. someone shared a link or bookmarked the page)

To perform the actual search for both cases, we create the search method:

`code/routes/music/app/ts/components/SearchComponent.ts`

```
91 search(): void {
92   if (!this.query) {
93     return;
94   }
95
96   this.spotify
97     .searchTrack(this.query)
98     .subscribe((res: any) => this.renderResults(res));
99 }
```

The search function uses the current value of `this.query` to know what to search for. Because we subscribed to the `queryParams` in the constructor, we can be sure that `this.query` will always have the most up-to-date value.

We then subscribe to the `searchTrack Observable` and whenever new results are emitted we call `renderResults`.

`code/routes/music/app/ts/components/SearchComponent.ts`

```
101 renderResults(res: any): void {
102   this.results = null;
103   if (res && res.tracks && res.tracks.items) {
104     this.results = res.tracks.items;
105   }
106 }
```

We declared `results` as a component property. Whenever its value is changed, the view will be automatically updated by Angular.

Searching on Page Load

As we pointed out above, we want to be able to jump straight into the results if the URL includes a search query.

To do that, we are going to implement a hook Angular router provides for us to run whenever our component is initialized.



But isn't that what constructors are for? Well, yes and no. Yes, constructors are used to initialize values, but if you want to write good, testable code, you want to minimize the side effects of *constructing* an object. So keep in mind that you should put your component's initialization logic always on a hook like below.

Here's the implementation of the `ngOnInit` method:

code/routes/music/app/ts/components/SearchComponent.ts

```
82 ngOnInit(): void {  
83   this.search();  
84 }
```

To use `ngOnInit` we imported the `OnInit` class and declared that our component implements `OnInit`.

As you can see, we're just performing the search here. Since the term we're searching for comes from the URL, we're good.

submit

Now let's see what we do when the user submits the form.

code/routes/music/app/ts/components/SearchComponent.ts

```
86 submit(query: string): void {  
87   this.router.navigate(['search'], { queryParams: { query: query } })  
88   .then(_ => this.search() );  
89 }
```

We're manually telling the router to navigate to the search route, and providing a query parameter, then performing the actual search.

Doing things this way gives us a great benefit: if we reload the browser, we're going to see the same search result rendered. We can say that we're **persisting the search term on the URL**.

Putting it all together

Here's the full listing for the `SearchComponent` class:

code/routes/music/app/ts/components/SearchComponent.ts

```
1 /*  
2  * Angular
```

```

3  */
4
5  import {Component, OnInit} from '@angular/core';
6  import {CORE_DIRECTIVES} from '@angular/common';
7  import {
8    Router,
9    ROUTER_DIRECTIVES,
10   ActivatedRoute,
11 } from '@angular/router';
12
13 /*
14  * Services
15  */
16 import {SpotifyService} from 'services/SpotifyService';
17
18 @Component({
19   selector: 'search',
20   directives: [ROUTER_DIRECTIVES, CORE_DIRECTIVES],
21   template: `
22     <h1>Search</h1>
23
24     <p>
25       <input type="text" #newquery
26         [value]="query"
27         (keydown.enter)="submit(newquery.value)">
28       <button (click)="submit(newquery.value)">Search</button>
29     </p>
30
31     <div *ngIf="results">
32       <div *ngIf="!results.length">
33         No tracks were found with the term '{{ query }}'
34       </div>
35
36       <div *ngIf="results.length">
37         <h1>Results</h1>
38
39         <div class="row">
40           <div class="col-sm-6 col-md-4" *ngFor="let t of results">
41             <div class="thumbnail">
42               <div class="content">
43                 
44                 <div class="caption">
45                   <h3>
46                     <a [routerLink]="['/artists', t.artists[0].id]">
47                       {{ t.artists[0].name }}
48                   </a>
49                 </h3>
50                 <br>
51                 <p>
52                   <a [routerLink]="['/tracks', t.id]">
53                     {{ t.name }}
54                   </a>
55                 </p>
56               </div>
57               <div class="attribution">
58                 <h4>
59                   <a [routerLink]="['/albums', t.album.id]">
60                     {{ t.album.name }}
61                   </a>
62                 </h4>
63               </div>
64             </div>
65           </div>
66         </div>
67       </div>
68     </div>
69   `
70 })
71
72 export class SearchComponent implements OnInit {
73   query: string;
74   results: Object;
75
76   constructor(private spotify: SpotifyService, private router: Router,
77               private route: ActivatedRoute) {
78     router.routerState.queryParams
79       .subscribe(params => { this.query = params['query'] || ''; });

```

```
80 }
81
82 ngOnInit(): void {
83   this.search();
84 }
85
86 submit(query: string): void {
87   this.router.navigate(['search'], { queryParams: { query: query } })
88     .then(_ => this.search() );
89 }
90
91 search(): void {
92   if (!this.query) {
93     return;
94   }
95
96   this.spotify
97     .searchTrack(this.query)
98     .subscribe((res: any) => this.renderResults(res));
99 }
100
101 renderResults(res: any): void {
102   this.results = null;
103   if (res && res.tracks && res.tracks.items) {
104     this.results = res.tracks.items;
105   }
106 }
107 }
```

Trying the search

Now that we have completed the code for the search, let's try it out:

Search

andre de sapato novo

Results



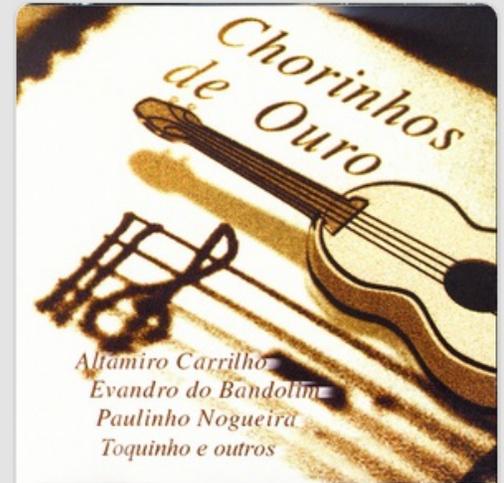
Bando De Macambira
André do Sapato Novo

Chorinho



Ordinarius
André de Sapato Novo / Tico Tico no Fubá

Rio de Choro



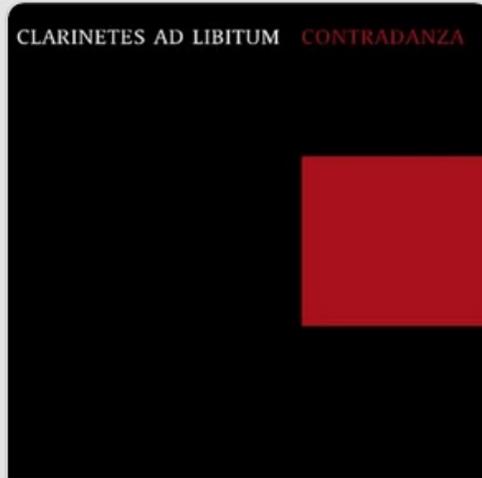
Evandro Do Bandolim
André De Sapato Novo

Chorinhos De Ouro



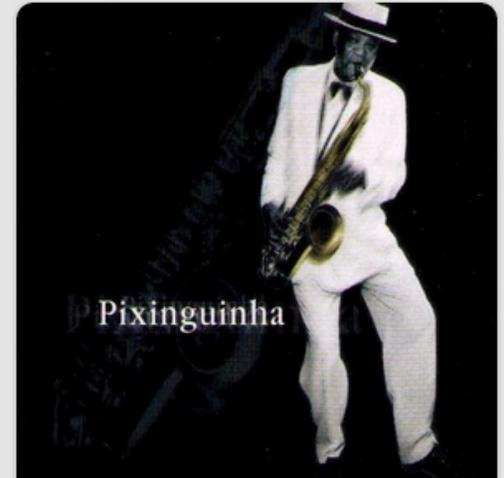
Pixinguinha
André de Sapato Novo

Benedito Lacerda E Pixinguinha



Clarinetes Ad Libitum
André de Sapato Novo

Contradanza



Pixinguinha
Andre De Sapato Novo

Latin Jazz Roots

Trying out Search

We can click the artist, track or album links to navigate to the proper route.

TrackComponent

For the track route, we use the TrackComponent. It basically displays the track name, the album cover image and allow the user to play a preview using an HTML5 audio tag:

code/routes/music/app/ts/components/TrackComponent.ts

```
22  template: `
23  <div *ngIf="track">
24    <h1>{{ track.name }}</h1>
25
26    <p>
27      
28    </p>
29
30    <p>
31      <audio controls src="{{ track.preview_url }}"></audio>
32    </p>
33
34    <p><a href (click)="back()">Back</a></p>
35  </div>
36  `
```

Like we did for the search before, we're going to use the Spotify API. Let's refactor the method searchTrack and extract two other useful methods we can reuse:

code/routes/music/app/ts/services/SpotifyService.ts

```
13  export class SpotifyService {
14    static BASE_URL: string = 'https://api.spotify.com/v1';
15
16    constructor(public http: Http) {
17    }
18
19    query(URL: string, params?: Array<string>): Observable<any[]> {
20      let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;
21      if (params) {
22        queryURL = `${queryURL}?${params.join('&')}`;
23      }
24
25      return this.http.request(queryURL).map((res: any) => res.json());
26    }
27
28    search(query: string, type: string): Observable<any[]> {
29      return this.query(`/search`, [
30        `q=${query}`,
31        `type=${type}`
32      ]);
33    }
34  }
```

Now that we've extracted those methods into the SpotifyService, notice how much simpler searchTrack becomes:

code/routes/music/app/ts/services/SpotifyService.ts

```
35  searchTrack(query: string): Observable<any[]> {
36    return this.search(query, 'track');
37  }
```

Now let's create a method to allow the component we're building retrieve track information, based in the track ID:

code/routes/music/app/ts/services/SpotifyService.ts

```
39  getTrack(id: string): Observable<any[]> {
40    return this.query(`/tracks/${id}`);
41  }
```

And now we can now use getTrack from a new ngOnInit method on the TrackComponent:

```
47 ngOnInit(): void {  
48   this.spotify  
49     .getTrack(this.id)  
50     .subscribe((res: any) => this.renderTrack(res));  
51 }
```

The other components work in a similar way and use `get*` methods from the `SpotifyService` to retrieve information about either an Artist or a Track based on their ID.

Wrapping up music search

Now we have a pretty functional music search and preview app. Try searching for a few of your favorite tunes and try it out!



It Had to Route You

Router Hooks

There are times that we may want to do some action when changing routes. A classical example of that is authentication. Let's say we have a **login** route and a **protected** route.

We want to only allow the app to go to the protected route if the correct username and password were provided on the login page.

In order to do that, we need to hook into the lifecycle of the router and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

In order to check if a component can be activated we add a *guard class* to the key `canActivate` in our router configuration.

Let's revisit our initial application, adding login and password input fields and a new protected route that only works if we provide a certain username and password combination.

 **Sample Code** The complete code for the examples in this section can be found in the `routes/auth` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

AuthService

Let's create a very simple and minimal implementation of a service, responsible for authentication and authorization of resources:

`code/routes/auth/app/ts/services/AuthService.ts`

```
1 import {Injectable, provide} from '@angular/core';
2
3 @Injectable()
4 export class AuthService {
5   login(user: string, password: string): boolean {
6     if (user === 'user' && password === 'password') {
7       localStorage.setItem('username', user);
8       return true;
9     }
10
11     return false;
12   }
```

The `login` method will return `true` if the provided user/password pair equals `'user'` and `'password'`, respectively. Also, when it is matched, it's going to use `localStorage` to save the username. This will also serve as a flag to indicate whether or not there is an active logged user.

 If you're not familiar, `localStorage` is an HTML5 provided key/value pair that allows you to persist information on the browser. The API is very simple, and basically allows the setting, retrieval and deletion of items. For more information, see the [Storage interface documents on MDN](#)

The `logout` method just clears the username value:

`code/routes/auth/app/ts/services/AuthService.ts`

```
14 logout(): any {
15   localStorage.removeItem('username');
16 }
```

And the final two methods:

- `getUser` returns the username or null
- `isLoggedIn` uses `getUser()` to return `true` if we have a user

Here's the code for those methods:

```
code/routes/auth/app/ts/services/AuthService.ts
```

```
18  getUser(): any {
19    return localStorage.getItem('username');
20  }
21
22  isLoggedIn(): boolean {
23    return this.getUser() !== null;
24  }
```

The last thing we do is export an AUTH_PROVIDERS, so it can be injected into our app:

```
code/routes/auth/app/ts/services/AuthService.ts
```

```
27 export var AUTH_PROVIDERS: Array<any> = [
28   provide(AuthService, {useClass: AuthService})
29 ];
```

Now that we have the AuthService we can inject it in our components to log the user in, check for the currently logged in user, log the user out, etc.

In a little bit, we'll also use it in our router to protect the ProtectedComponent. But first, let's create the component that we use to log in.

LoginComponent

This component will either show a login form, for the case when there is no logged user, or display a little banner with user information along with a logout link.

The relevant code here is the login and logout methods:

```
code/routes/auth/app/ts/components/LoginComponent.ts
```

```
40 export class LoginComponent {
41   message: string;
42
43   constructor(public authService: AuthService) {
44     this.message = '';
45   }
46
47   login(username: string, password: string): boolean {
48     this.message = '';
49     if (!this.authService.login(username, password)) {
50       this.message = 'Incorrect credentials.';
51       setTimeout(function() {
52         this.message = '';
53       }.bind(this), 2500);
54     }
55     return false;
56   }
57
58   logout(): boolean {
59     this.authService.logout();
60     return false;
61   }
```

Once our service validates the credentials, we log the user in.

The component template has two snippets that are displayed based on whether the user is logged in or not.

The first is a login form, protected by `*ngIf="!authService.getUser()"`:

```
code/routes/auth/app/ts/components/LoginComponent.ts
```

```

18 <form class="form-inline" *ngIf="!authService.getUser()">
19   <div class="form-group">
20     <label for="username">User:</label>
21     <input class="form-control" name="username" #username>
22   </div>
23
24   <div class="form-group">
25     <label for="password">Password:</label>
26     <input class="form-control" type="password" name="password" #password>
27   </div>
28
29   <a class="btn btn-default" (click)="login(username.value, password.value)">
30     Submit
31   </a>
32 </form>

```

And the information banner, containing the logout link, protected by the inverse - `*ngIf="authService.getUser()"`:

code/routes/auth/app/ts/components/LoginComponent.ts

```

34 <div class="well" *ngIf="authService.getUser()">
35   Logged in as <b>{{ authService.getUser() }}</b>
36   <a href (click)="logout()">Log out</a>
37 </div>

```

There's another snippet of code that is displayed when we have an authentication error:

code/routes/auth/app/ts/components/LoginComponent.ts

```

14 <div class="alert alert-danger" role="alert" *ngIf="message">
15   {{ message }}
16 </div>

```

Now that we can handle the user login, let's create a resource that we are going to protect behind a user login.

ProtectedComponent and Route Guards

The ProtectedComponent

Before we can protect the component, it needs to exist. Our ProtectedComponent is straightforward:

code/routes/auth/app/ts/components/ProtectedComponent.ts

```

1 /*
2  * Angular
3  */
4 import {Component} from '@angular/core';
5
6 @Component({
7   selector: 'protected',
8   template: `<h1>Protected content</h1>`
9 })
10 export class ProtectedComponent {
11 }

```

We want this component to only be accessible to logged in users. But how can we do that?

The answer is to use the router hook `canActivate` with a *guard class* that implements `CanActivate`.

The LoggedInGuard

We create a new folder called `guards` and create `loggedIn.guard.ts`:

code/routes/auth/app/ts/guards/loggedIn.guard.ts

```

1 import { Injectable } from '@angular/core';
2 import { CanActivate } from '@angular/router';
3 import { AuthService } from 'services/AuthService';
4
5 @Injectable()
6 export class LoggedInGuard implements CanActivate {
7   constructor(private authService: AuthService) {}
8
9   canActivate(): boolean {
10    return this.authService.isLoggedIn();
11  }
12 }

```

Our guard states that it implements the `CanActivate` interface. This is satisfied by implementing a method `canActive`.

We inject the `AuthService` into this class in the constructor and save it as a private variable `authService`.

In our `canActivate` function we check `this.authService` to see if the user `isLoggedIn`.

Configuring the Router

To configure the router to use this guard we need to do the following:

1. import the `LoggedInGuard`
2. Use the `LoggedInGuard` in a route configuration
3. Include `LoggedInGuard` in the list of providers (so that it can be injected)

We do all of these steps in our `app.ts`.

We import the `LoggedInGuard`:

code/routes/auth/app/ts/app.ts

```

27 import {AUTH_PROVIDERS} from 'services/AuthService';
28 import {LoggedInGuard} from 'guards/loggedIn.guard';

```

We add `canActivate` with our guard to the protected route:

code/routes/auth/app/ts/app.ts

```

68 const routes: RouterConfig = [
69   { path: '', redirectTo: 'home', terminal: true },
70   { path: 'home', component: HomeComponent },
71   { path: 'about', component: AboutComponent },
72   { path: 'contact', component: ContactComponent },
73   { path: 'protected', component: ProtectedComponent,
74     canActivate: [LoggedInGuard]}
75 ];

```

We add `LoggedInGuard` to our list of providers:

code/routes/auth/app/ts/app.ts

```

77 bootstrap(RoutesDemoApp, [
78   provideRouter(routes),
79   AUTH_PROVIDERS,
80   LoggedInGuard,
81   provide(LocationStrategy, {useClass: HashLocationStrategy}),
82   provideForms()
83 ]);

```

Logging in

We now have to add:

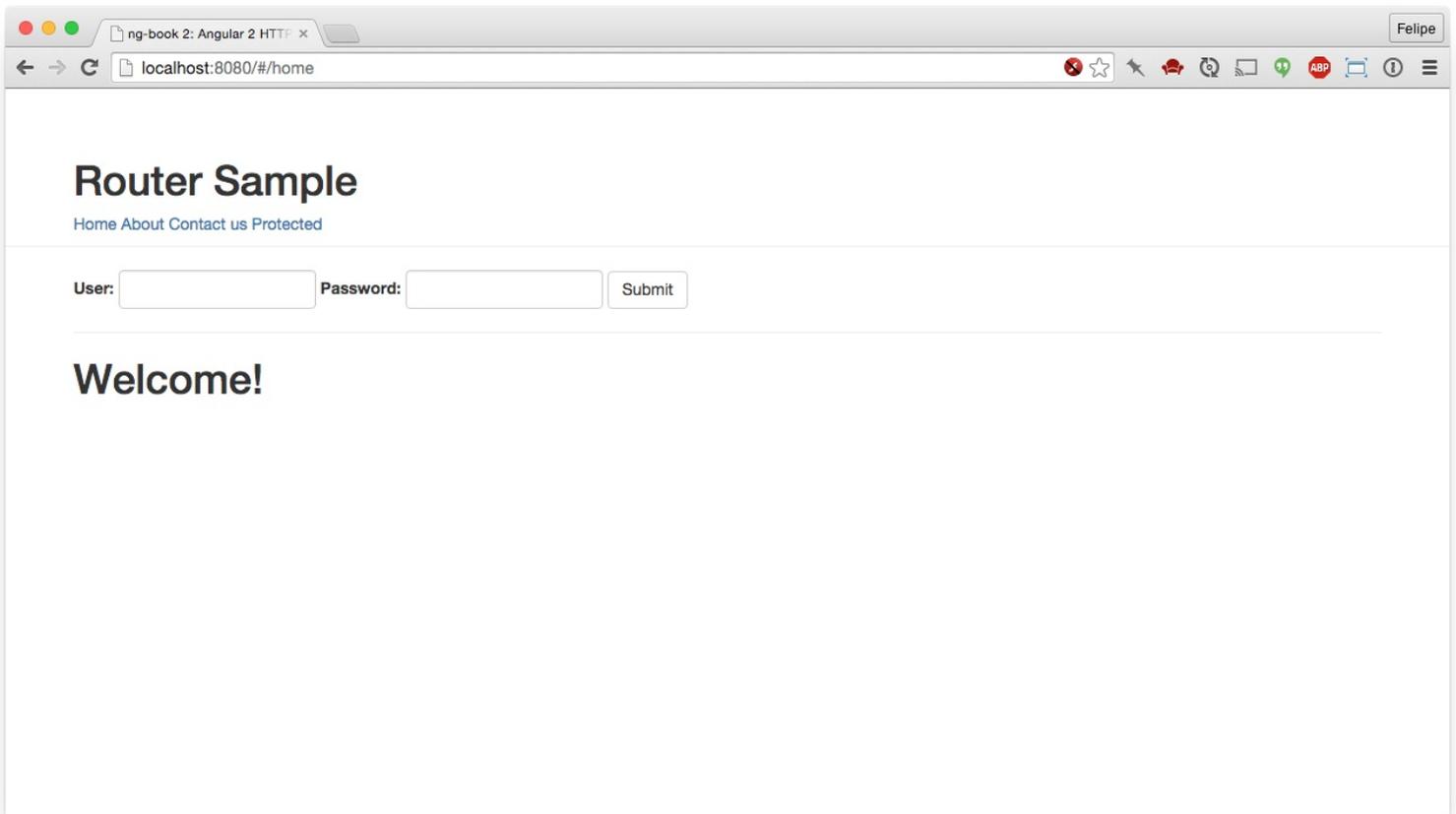
1. LoginComponent to the list of directives
2. a new link to the protected route
3. a `<login>` tag to the template, to render the new component

Here's what it should look like:

code/routes/auth/app/ts/app.ts

```
35 @Component({
36   selector: 'router-app',
37   directives: [ROUTER_DIRECTIVES, LoginComponent],
38   template: `
39     <div class="page-header">
40       <div class="container">
41         <h1>Router Sample</h1>
42         <div class="navLinks">
43           <a [routerLink]="['/home']">Home</a>
44           <a [routerLink]="['/about']">About</a>
45           <a [routerLink]="['/contact']">Contact us</a>
46           <a [routerLink]="['/protected']">Protected</a>
47         </div>
48       </div>
49     </div>
50
51     <div id="content">
52       <div class="container">
53
54         <login></login>
55
56         <hr>
57
58         <router-outlet></router-outlet>
59       </div>
60     </div>
61   `
62 })
63 class RoutesDemoApp {
64   constructor(public router: Router) {
65   }
66 }
```

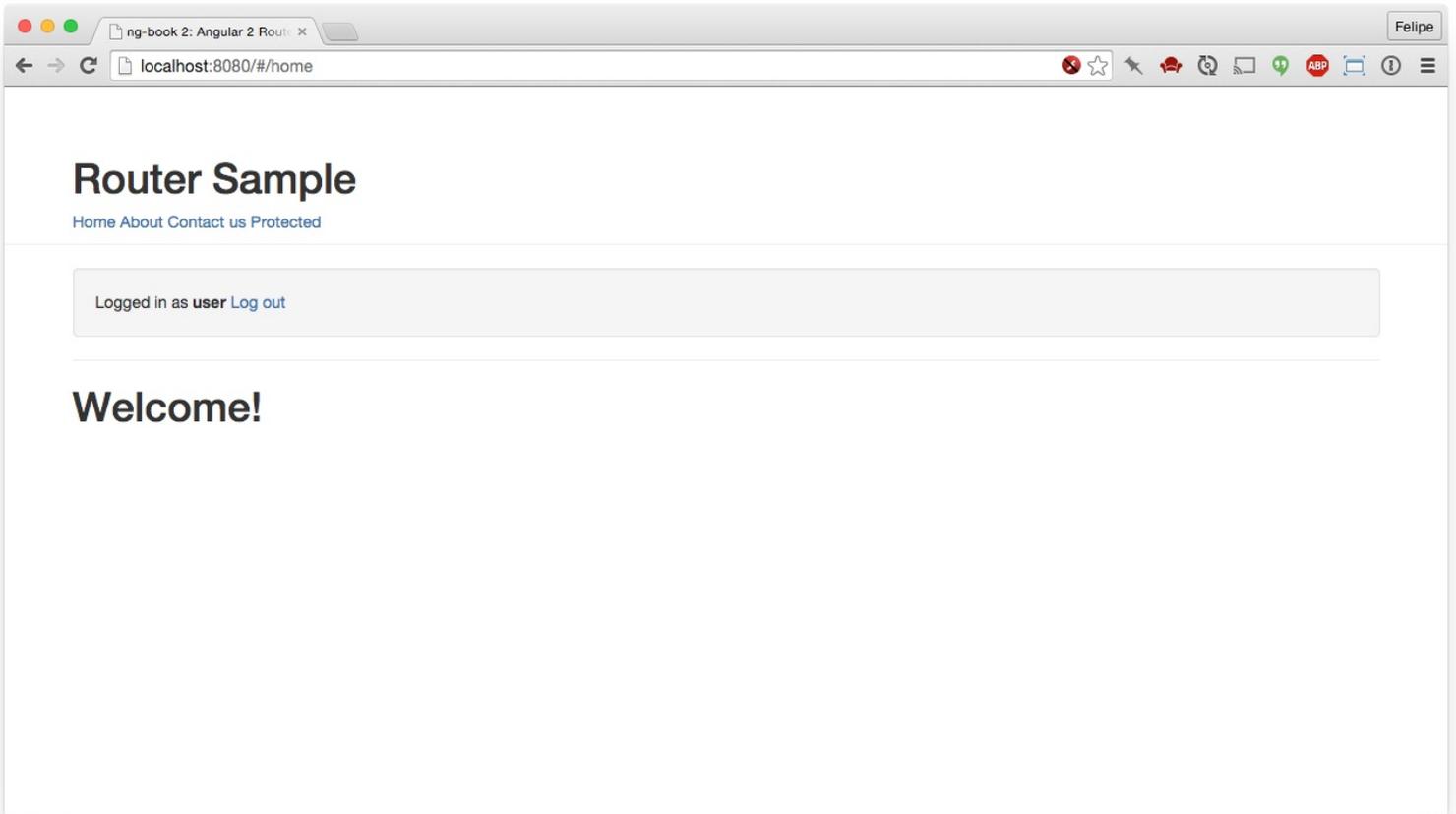
Now when we open the application on the browser, we can see the new login form and the new protected link:



Auth App - Initial Page

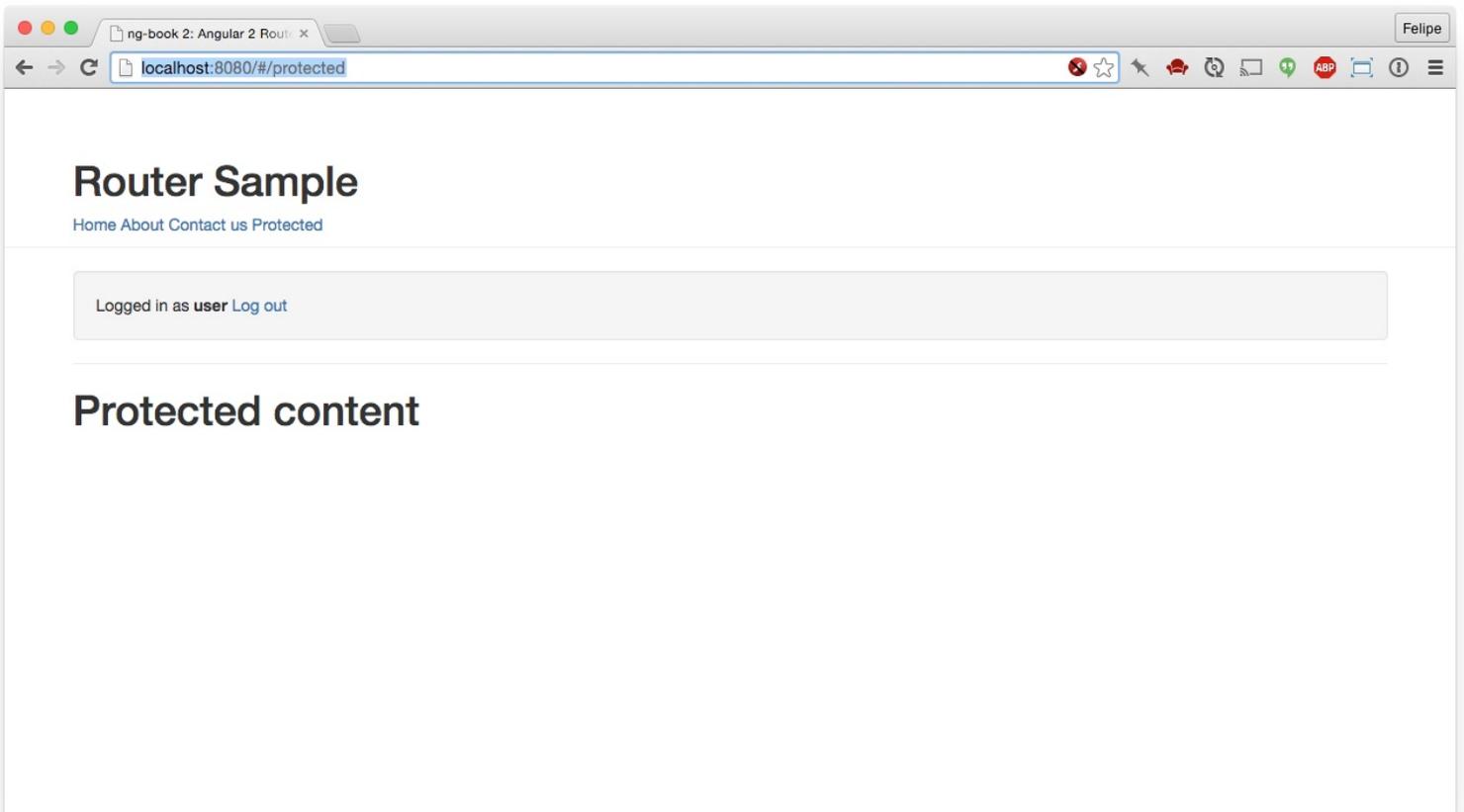
If you click the Protected link, you'll see nothing happens. The same happens if you try to manually visit <http://localhost:8080/#/protected>.

Now enter *user* and *password* on the form and click **Submit**. You'll see that we now get the current user displayed on a banner:



Auth App - Logged In

And, sure enough, if we click the Protected link, it gets redirected and the component is rendered:





A Note on Security: It's important to know how client-side route protection is working before you rely too heavily on it for security. That is, you should consider client-side route protection a form of *user-experience* and not one of security.

Ultimately all of the javascript in your app that gets served to the client can be inspected, whether the user is logged in or not.

So if you have sensitive data that needs to be protected, you must protect it with **server-side authentication**. That is, require an API key (or auth token) from the user which is validated by the server on every request for data.

Writing a full-stack authentication system is beyond the scope of this book. The important thing to know is that protecting routes on the client-side don't necessarily keep anyone from viewing the javascript pages behind those routes.

Nested Routes

Nested routes is the concept of containing routes within other routes. With nested routes we're able to encapsulate the functionality of parent routes and have that functionality apply to the child routes.

Let's say we have a website with one area to allow users to know our team, called **Who we are?** and another one for our **Products**.

We could think that the perfect route for **Who we are?** would be `/about` and for products `/products`.

And we're happily displaying all our team and all our products when visiting this areas.

What happens when the website grows and we now need to display individual information about each person in our team and also for each product we sell?

In order to support scenarios like these, the router allows the user to define nested routes.

To do that, you can have multiple, nested `router-outlet`. So each area of our application can have their own child components, that also have their own `router-outlets`.

Let's work on an example to clear things up.

In this example, we'll have a products section where the user will be able to view two highlighted products by visiting a nice URL. For all the other products, the routes will use the product ID.



Sample Code The complete code for the examples in this section can be found in the `routes/nested` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

Configuring Routes

We will start by describing two top-level routes on the `app.ts` file:

```
code/routes/nested/app/ts/app.ts
```

The home route looks very familiar, notice that products has a children parameter. Where does this come from? We've defined the childRoutes alongside the ProductsComponent. Let's take a look:

ProductsComponent

This component will have its own route configuration:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
49 export const routes: RouterConfig = [
50   { path: '', redirectTo: 'main' },
51   { path: 'main', component: MainComponent },
52   { path: ':id', component: ByIdComponent },
53   { path: 'interest', component: InterestComponent },
54   { path: 'sportify', component: SportifyComponent },
55 ];
```

Notice here that we have an empty path on the first object. We do this so that when we visit /products, we'll be redirected to the main route.

The other route we need to look at is :id. In this case, when the user visits something *that doesn't match any other route*, it will fallback to this route. Everything that is passed after / will be extracted to a parameter of the route, called id.

Now on the component template, we'll have a link to each of those static child routes:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
27 <a [routerLink]="['./main']">Main</a> |
28 <a [routerLink]="['./interest']">Interest</a> |
29 <a [routerLink]="['./sportify']">Sportify</a> |
```

You can see that the route links are all in the format ['./main'], with a preceding ./ . This indicates that you want to navigate the Main route *relative to the current route context*.

You could also declare the routes with the ['products', 'main'] notation. The downside is that by doing it this way, the child route is aware of the parent route and if you were to move this component around or reuse it, you would have to rewrite your route links.

After the links, we'll add an input where the user will be able to enter a product id, along with a button to navigate to it, and lastly add our router-outlet:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
23 template: `
24   <h2>Products</h2>
25
26   <div class="navLinks">
27     <a [routerLink]="['./main']">Main</a> |
28     <a [routerLink]="['./interest']">Interest</a> |
29     <a [routerLink]="['./sportify']">Sportify</a> |
30     Enter id: <input #id size="6">
31     <button (click)="goToProduct(id.value)">Go</button>
32   </div>
33
34   <div class="products-area">
35     <router-outlet></router-outlet>
36   </div>
37 `
```

Let's look at the ProductsComponent definition:

code/routes/nested/app/ts/components/ProductsComponent.ts

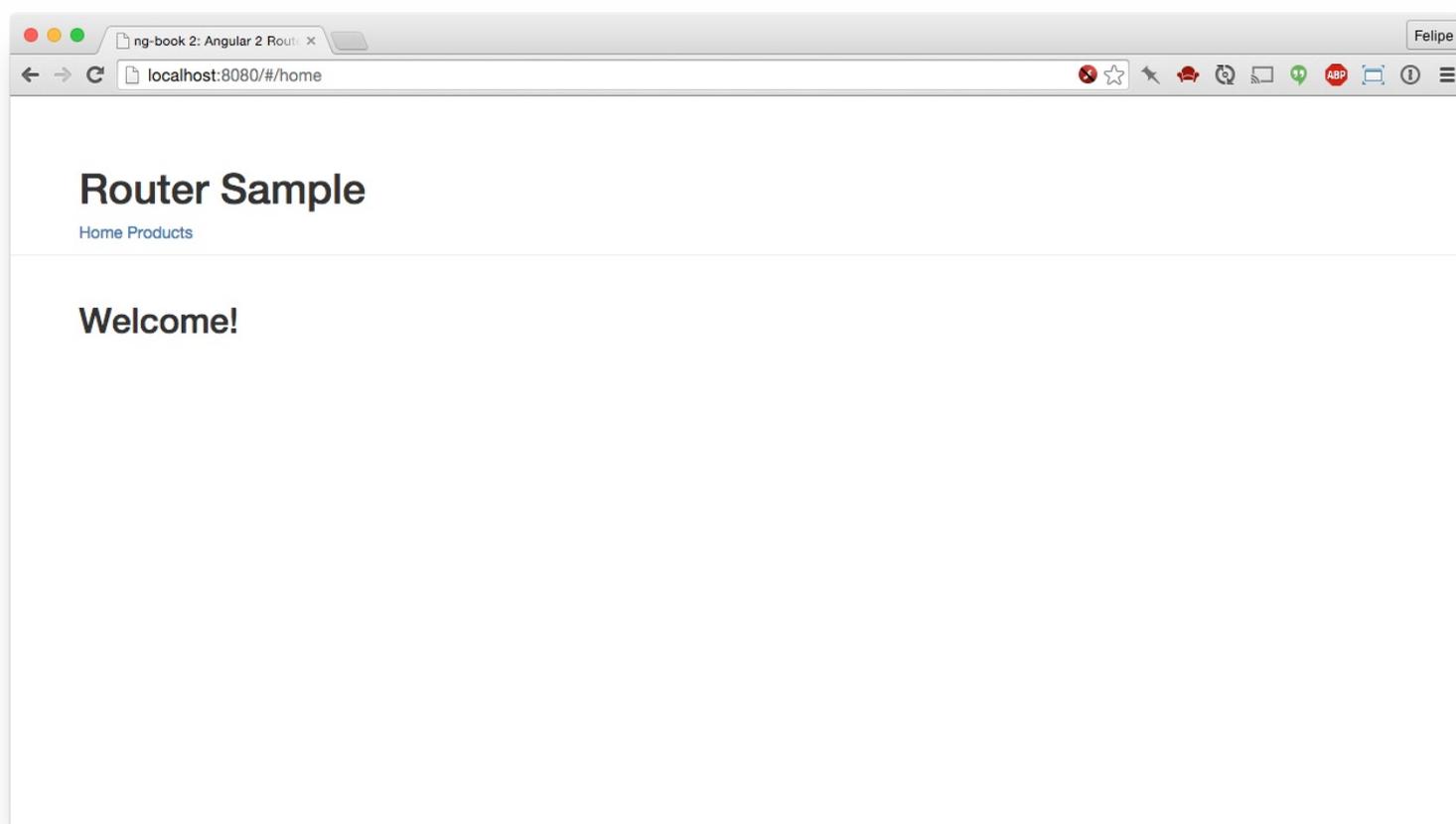
```
40 export class ProductsComponent {
41   constructor(private router: Router, private route: ActivatedRoute) {
42   }
43
44   goToProduct(id:string): void {
45     this.router.navigate(['./', id], {relativeTo: this.route});
46   }
47 }
```

First on the constructor we're declaring an instance variable for the Router, since we're going to use that instance to navigate to the product by id.

When we want to go to a particular product we use the goToProduct method. In goToProduct we call the router's navigate method and providing the route name and an object with route parameters. In our case we're just passing the id.

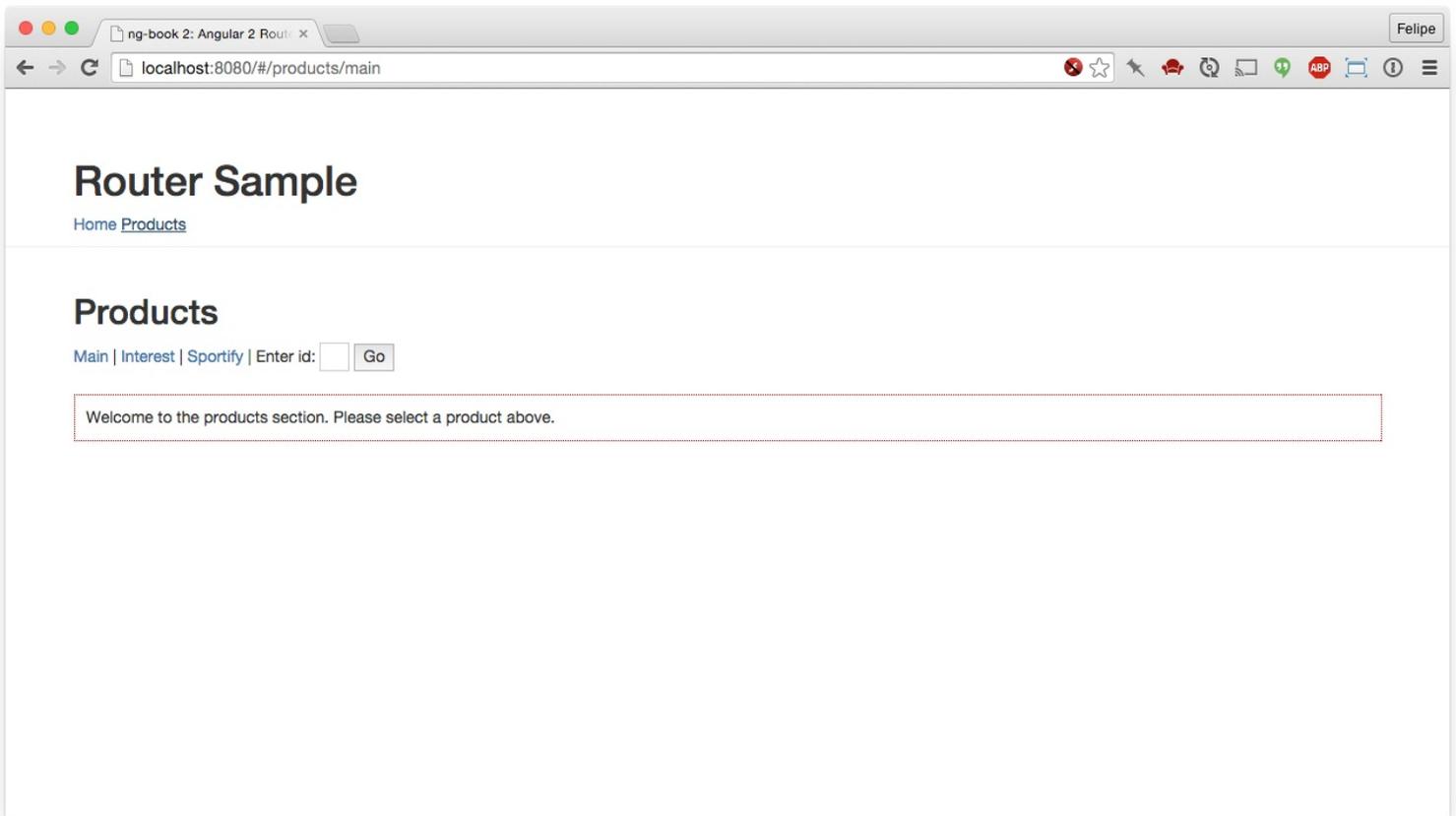
Notice that we use the relative ./ path in the navigate function. In order to use this we also pass the relativeTo object to the options, which tells the router what that route is relative to.

Now, if we run the application we will see the main page:



Nested Routes App

If you click on the Products link, you'll be redirected to /products/main that will render as follows:

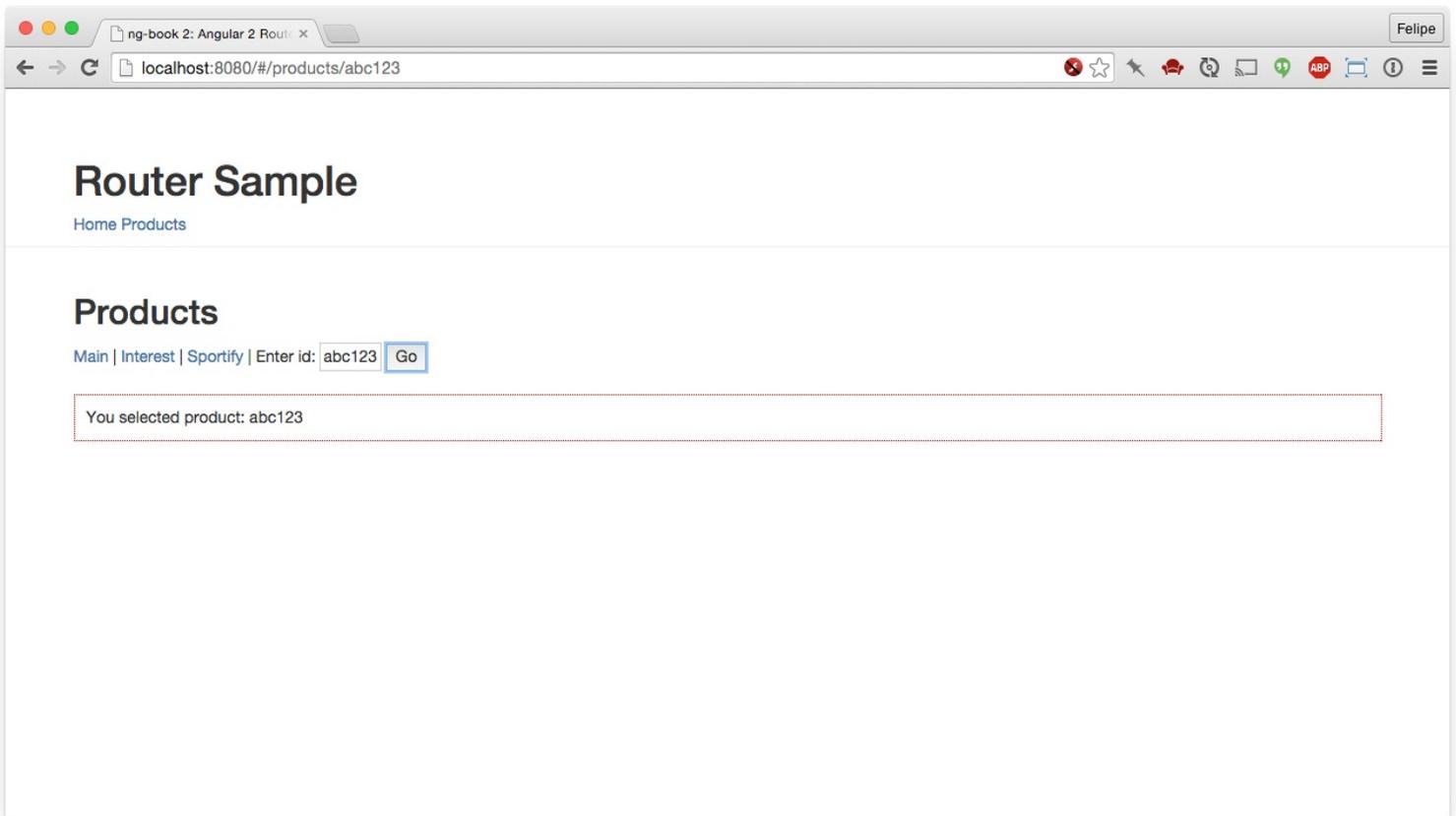


Nested Routes App - Products Section

Everything below that thin grey line is being rendered using the main application's router-outlet.

And the contents of the dotted red line is being rendered inside the ProductComponent's router-outlet. That's how you indicate how the parent and child routes will be rendered.

When we visit one of the product links, or if we an ID on the textbox and click Go, the new content is rendered inside the ProductComponent's outlet:



Nested Routes App - Product By Id

It's also worth noting that the Angular router is smart enough to prioritize concrete routes first (like `/products/sportify`) over the parameterized ones (like `/products/123`). This way `/products/sportify` will never be handled by the more generic, catch-all route `/products/:id`.

Redirecting and linking nested routes

Just to recap, if we want to go to a route named `MyRoute` on your top-level routing context, you use `['myRoute']`. This will only work if you're in that same top-level context.

If you are on a child component, and you try to link or redirect to `['myRoute']`, it will try to find a sibling route, and error out. In this case, you need to use `['/myRoute']` with a leading slash.

In a similar way, if we are on the top-level context and we want to link or redirect to a child route, we have to need to use multiple elements on the route definition array.

Let's say we want to visit the `Show` route, which is a child of the `Product` route. In this case, we use `['product', 'show']` as the route definition.

Summary

As we can see, the new Angular router is very powerful and flexible. Now go out and route your apps!

Advanced Components

Throughout this book, we've learned [how to use Angular's built-in components](#) and [how to create components of our own](#). In this chapter we'll take a deep dive into **advanced** features we can use to make components.

In this chapter we'll learn the following concepts:

- Styling components (with encapsulation)
- Modifying host DOM elements
- Modifying templates with *transclusion*
- Accessing neighbor directives
- Using lifecycle hooks
- Detecting changes

Styling

Angular provides a mechanism for specifying component-specific styles. CSS stands for *cascading style sheet*, but sometimes we **don't** want the cascade. Instead we want to provide styles for a component that won't leak out into the rest of our page.

Angular provides two attributes that allow us to define CSS classes for our component.

To define the style for our component, we use the View attribute `styles` to define in-line styles, or `styleUrls`, to use external CSS files. We can also declare those attributes directly on the Component annotation.

Let's write a component that uses inline styles:

code/advanced_components/app/ts/styling/styling.ts

```
4 @Component({
5   selector: 'inline-style',
6   styles: [`
7     .highlight {
8       border: 2px solid red;
9       background-color: yellow;
10      text-align: center;
11      margin-bottom: 20px;
12    }
13  `],
14   template: `
15    <h4 class="ui horizontal divider header">
16      Inline style example
17    </h4>
18
19    <div class="highlight">
20      This uses component <code>styles</code>
21      property
22    </div>
23  `
24 })
25 class InlineStyle {
26 }
```

In this example we defined the styles we want to use by declaring the `.highlight` class as an item on the array on the `styles` parameter.

Further on in the template we reference that class on the div using `<div class="highlight">`.

And the result is exactly what we expect - a div with a red border and yellow background:

Inline style example



This uses component `styles` property

Example of component using styles

Another way to declare CSS classes is to use the `styleUrls` property. This allows us to declare our CSS on an external file and just reference them from the component.

Let's write another component that uses this, but first let's create a file called `external.css` with the following class:

code/advanced_components/app/ts/styling/external.css

```
1 .highlight {
2   border: 2px dotted red;
3   text-align: center;
4   margin-bottom: 20px;
5 }
```

Then we can write the code that references it:

code/advanced_components/app/ts/styling/styling.ts

```
28 @Component({
29   selector: 'external-style',
30   styleUrls: [externalCSSUrl],
31   template: `
32     <h4 class="ui horizontal divider header">
33       External style example
34     </h4>
35
36     <div class="highlight">
37       This uses component <code>styleUrls</code>
38       property
39     </div>
40   `
41 })
42 class ExternalStyle {
43 }
```

And when we load the page, we see our div with a dotted border:

External style example



This uses component `styleUrls` property

Example of component using styleUrls

View (Style) Encapsulation

One interesting thing about this example is that both components define a class called `highlight` with different properties, but the attributes of one didn't leak into the other.

This happens because Angular styles are **encapsulated by the component context** by default. If we inspect the page and expand the `<head>`, we'll notice that Angular injected a `<style>` tag with our style:

The screenshot shows a browser window with the URL `localhost:8080`. The page title is "Angular 2 component styling demo". It features two examples of styling:

- Inline style example:** A yellow box with a red border and centered text: "This uses component styles property".
- External style example:** A dashed red border box with centered text: "This uses component styleUrls property".

The browser's developer tools are open, showing the `<head>` section of the page. The injected style tag is visible:

```
<style>
  .highlight[_ngcontent-hve-2] {
    border: 2px solid red;
    background-color: yellow;
    text-align: center;
    margin-bottom: 20px;
  }
</style>
```

The right-hand pane shows the "Styles" panel for the selected element, displaying the computed styles and the user agent styles.

Injected style

You'll also notice that the CSS class has been scoped with `_ngcontent-hve-2`:

```
1 .highlight[_ngcontent-hve-2] {
2   border: 2px solid red;
3   background-color: yellow;
4   text-align: center;
5   margin-bottom: 20px;
6 }
```

And if we check how our `<div>` is rendered, you'll find that `_ng-content-hve-2` was added:

Angular 2 - ngStyle demo

localhost:8080

ng-book 2

Angular 2 component styling demo

Inline style example

This uses component styles property

External style example

This uses component styleUrls property

Elements Console Sources Network Timeline Profiles Resources Security Audits

```
<div class="ui menu">...</div>
<div class="ui main text container">
  <style-sample-app>
    <inline-style _ngghost-hve-2>
      <h4 class="ui horizontal divider header" _ngcontent-hve-2>
        ::before
        "
        Inline style example
        "
        ::after
      </h4>
      <div class="highlight" _ngcontent-hve-2>
        "
        This uses component "
        <code _ngcontent-hve-2>styles</code>
        "
        property
        "
      </div>
    </inline-style>
    <external-style _ngghost-hve-3>...</external-style>
  </style-sample-app>
<!-- Our app loads here -->
</div>
```

Styles Computed >>

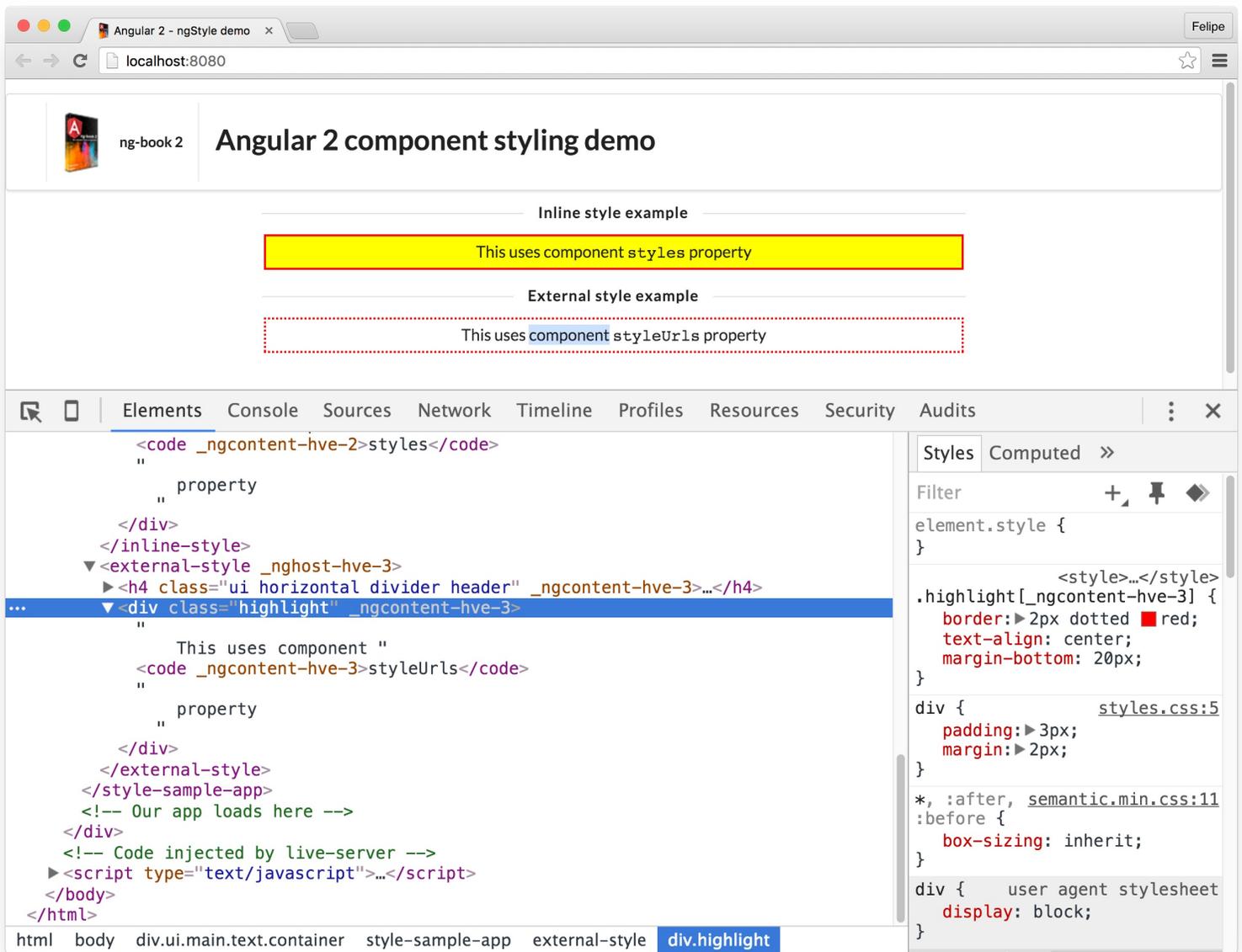
Filter + - ◆

```
element.style {
}
<style>...</style>
.highlight[_ngcontent-hve-2] {
  border: 2px solid red;
  background-color: yellow;
  text-align: center;
  margin-bottom: 20px;
}
div {
  padding: 3px;
  margin: 2px;
}
*, :after, semantic.min.css:11
:before {
  box-sizing: inherit;
}
div {
  user agent stylesheet
  display: block;
```

html body div.ui.main.text.container style-sample-app inline-style div.highlight

Injected style

The same thing happens for our external style:



External style

Angular allows us to change this behavior, by using the encapsulation property.

This property can have the following values, defined by the ViewEncapsulation enum:

- **Emulated** - this is the default option and it will encapsulate the styles using the technique we just explained above
- **Native** - with this option, Angular will use the Shadow DOM (more on this below)
- **None** - with this option set, Angular won't encapsulate the styles at all, allowing them to leak to other elements on the page

Shadow DOM Encapsulation

You might be wondering: what is the point of using the Shadow DOM? By using the Shadow DOM the component we **uses a unique DOM tree that is hidden from the other elements on the page**. This allows styles defined within that element to be invisible to the rest of the page.



For a deep dive into Shadow DOM, please check this [guide by Eric Bidelman](#).

Let's create another component that uses the **Native** encapsulation (Shadow DOM) to understand how this works:

code/advanced_components/app/ts/styling/styling.ts

```
45 @Component({
46   selector: `native-encapsulation`,
47   styles: [`
48     .highlight {
49       text-align: center;
50       border: 2px solid black;
51       border-radius: 3px;
52       margin-bottom: 20px;
53     }`],
54   template: `
55     <h4 class="ui horizontal divider header">
56       Native encapsulation example
57     </h4>
58
59     <div class="highlight">
60       This component uses <code>ViewEncapsulation.Native</code>
61     </div>
62   `,
63   encapsulation: ViewEncapsulation.Native
64 })
65 class NativeEncapsulation {
66 }
```

In this case, if we inspect the source code, we'll see:

The screenshot shows a browser window at localhost:8080 displaying an Angular 2 component styling demo. The demo includes three examples of styling:

- Inline style example:** A yellow box with a red border containing the text "This uses component styles property".
- External style example:** A red dashed border box containing the text "This uses component styleUrls property".
- Native encapsulation example:** A black solid border box containing the text "This component uses ViewEncapsulation.Native".

The Chrome DevTools 'Elements' panel shows the component's shadow root structure:

```

<external-style _nghost-jev-3>...</external-style>
...
<native-encapsulation>
  #shadow-root (open)
    <style>
      .highlight {
        text-align: center;
        border: 2px solid black;
        border-radius: 3px;
      }
    </style>
    <h4 class="ui horizontal divider header">
      Native encapsulation example
    </h4>
    <div class="highlight">
      "
      This component uses "
      <code>ViewEncapsulation.Native</code>
    </div>
    <style>...</style>
    <style>...</style>
  </native-encapsulation>
  <style-sample-app>

```

The 'Styles' panel shows the computed styles for the selected element:

```

element.style {
}
*, :after, semantic.min.css:11
:before {
  box-sizing: inherit;
}
Inherited from div.ui.main.te...
  semantic.min.css:11
  .ui.text.container {
    font-family:
      Lato, 'Helvetica
      Neue', Arial, Helvetica, ...
      serif;
    max-width:
      700px !important;
    line-height: 1.5;
    font-size: 1.14285714rem;
  }

```

Native encapsulation

Everything inside the #shadow-root element has been encapsulated and isolated from the rest of the page.

No Encapsulation

Finally, if we create a component that specifies `ViewEncapsulation.None`, no style encapsulation will be added:

code/advanced_components/app/ts/styling/styling.ts

```

68 @Component({
69   selector: `no-encapsulation`,
70   styles: [
71     .highlight {
72       border: 2px dashed red;
73       text-align: center;
74       margin-bottom: 20px;
75     }
76   ],
77   template: `
78 <h4 class="ui horizontal divider header">
79   No encapsulation example
80 </h4>
81
82 <div class="highlight">

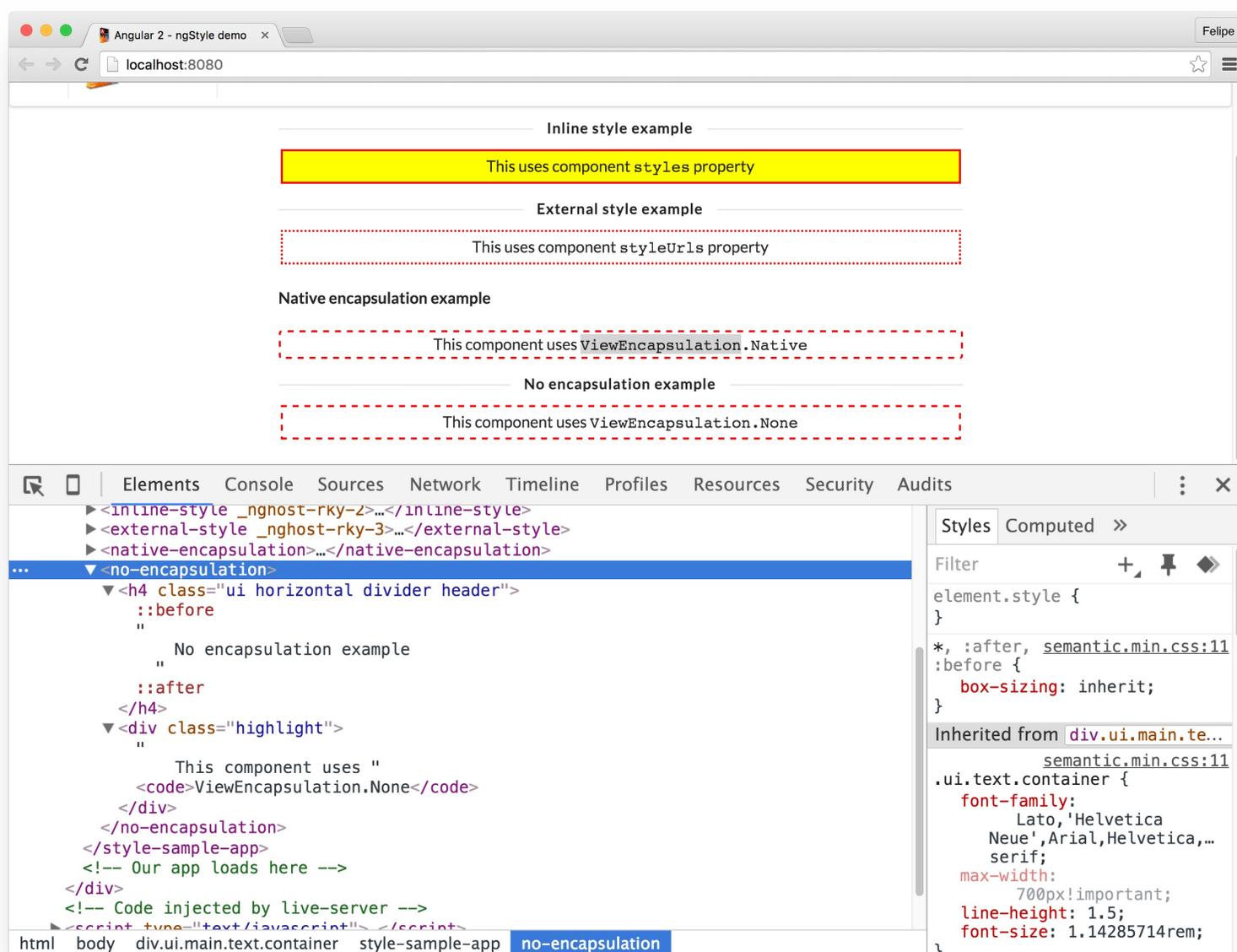
```

```

83 This component uses <code>ViewEncapsulation.None</code>
84 </div>
85 ,
86 encapsulation: ViewEncapsulation.None
87 })

```

When we inspect the element:



No encapsulation

We can see that nothing was injected on the HTML. Also on the header we can find that the `<style>` tag was also injected exactly like we defined on the `styles` parameter:

```

1 .highlight {
2   border: 2px dashed red;
3   text-align: center;
4   margin-bottom: 20px;
5 }

```

One side-effect of using `ViewEncapsulation.None` is that, since we don't have any encapsulation, this style "leaks" into other components. If we check the picture above, the `ViewEncapsulation.Native` component style was affected by this new component's style. But sometimes this can be exactly what you want.

You can comment out the `<no-encapsulation></no-encapsulation>` code on the `StyleSampleApp` template to see the difference.

Creating a Popup - Referencing and Modifying Host Elements

The *host element* is the element to which the directive or component is bound to. Sometimes we have component that needs to attach markup or behavior to its host element.

In this example, we're going to create a `Popup` directive that will attach behavior to its host element which will display a message when clicked.

Components vs. Directives - What's the difference?

Components and directives are closely related, but they are slightly different.

You may have heard that “components are directives with a view”. This isn't exactly true. Components come with functionality that makes it easy to add views, but directives can have views too. In fact, **components are implemented with directives**.

One great example of a directive that renders a conditional view is `NgIf`.

But we can attach behaviors to an element **without a template** by using a *directive*.

Think of it this way: Components are Directives and Components always have a view. Directives may or may not have a view.

If you choose to render a view (a template) in your Directive, you can have more control over how that template is rendered. We'll talk more about how to use that control later in this chapter.

Popup Structure

Let's now write our first directive. We want this directive to **show an alert when we click a DOM element** that includes the attribute `popup`. The message displayed will be identified by the element's `message` attribute.

Here's what we want it to look like:

```
1 <element popup message="Some message"></element>
```

In order to make this directive work, there are a couple of things we need to do:

- receive the message attribute *from* the host
- be notified when the host element is clicked

Let's start coding our directive:

code/advanced_components/app/ts/host/steps/host_01.ts

```
4 @Directive({
5   selector: '[popup]'
6 })
7 class Popup {
8   constructor() {
9     console.log('Directive bound');
10  }
11 }
```

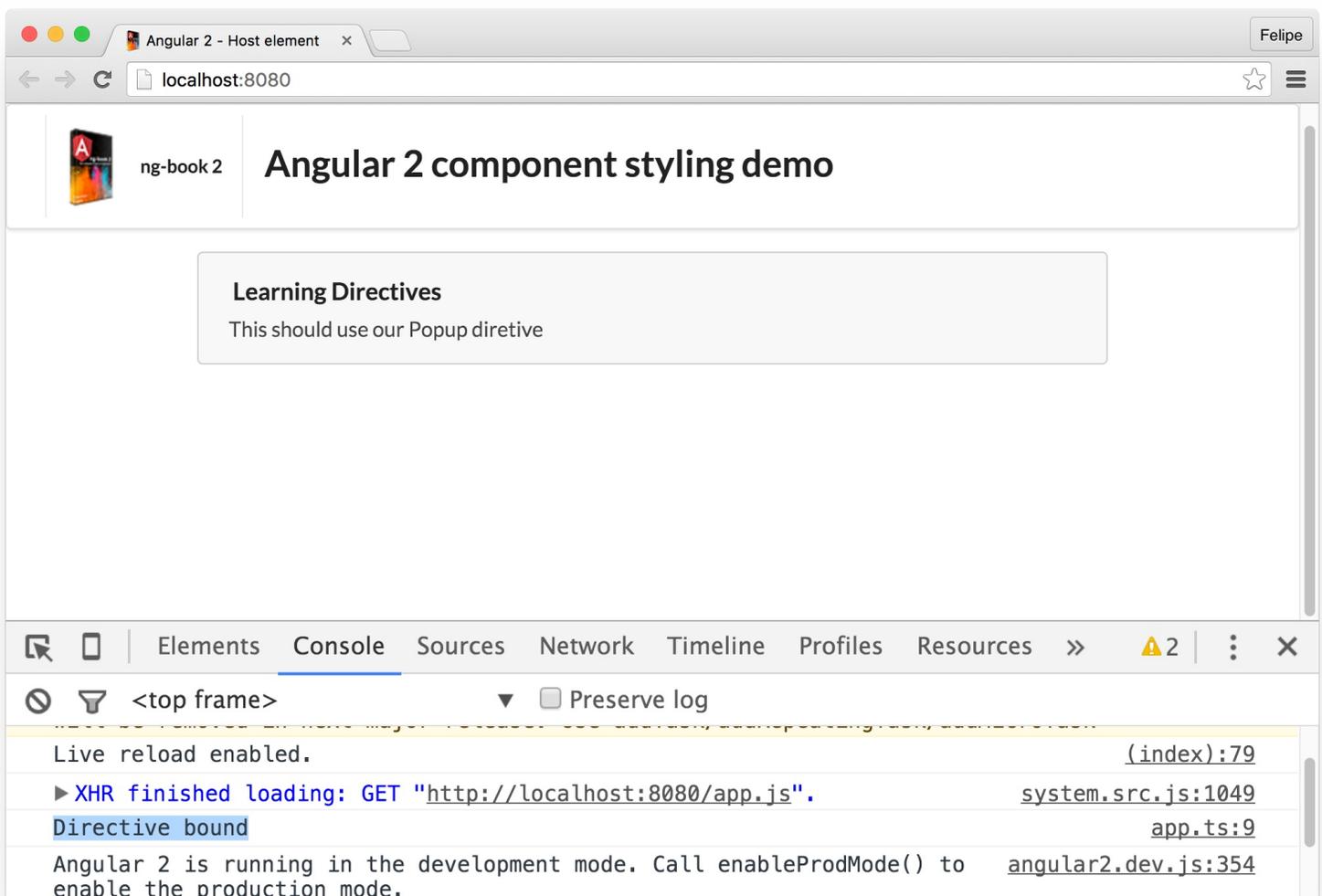
We use the Directive annotation and set the selector1 parameter to [popup]. This will make this directive bind to any elements that define the popup attribute.

Let's now create an app that has an element that has the popup attribute:

code/advanced_components/app/ts/host/steps/host_01.ts

```
13 @Component({
14   selector: 'host-sample-app',
15   directives: [Popup],
16   template: `
17 <div class="ui message" popup>
18   <div class="header">
19     Learning Directives
20   </div>
21
22   <p>
23     This should use our Popup directive
24   </p>
25 </div>
26 `
27 })
28 export class HostSampleApp1 {
29 }
```

When we run this application, we expect the message Directive bound to be logged on the console, indicating we have successfully bound to the first <div> in our template:



The screenshot shows a web browser window with the URL localhost:8080. The page title is "Angular 2 component styling demo" and the page content displays a message box with the text "Learning Directives" and "This should use our Popup directive". The browser's developer console is open, showing the following logs:

- Live reload enabled. (index):79
- XHR finished loading: GET "http://localhost:8080/app.js". system.src.js:1049
- Directive bound app.ts:9
- Angular 2 is running in the development mode. Call enableProdMode() to enable the production mode. angular2.dev.js:354

Using ElementRef

If we want to learn more about the host element a directive is bound to, we can use the built in ElementRef class.

This class holds the information about a given Angular element, including the native DOM element using the nativeElement property.

In order to see the elements our directive is binding to, we can change our directive constructor to receive the ElementRef and log it to the console:

code/advanced_components/app/ts/host/steps/host_02.ts

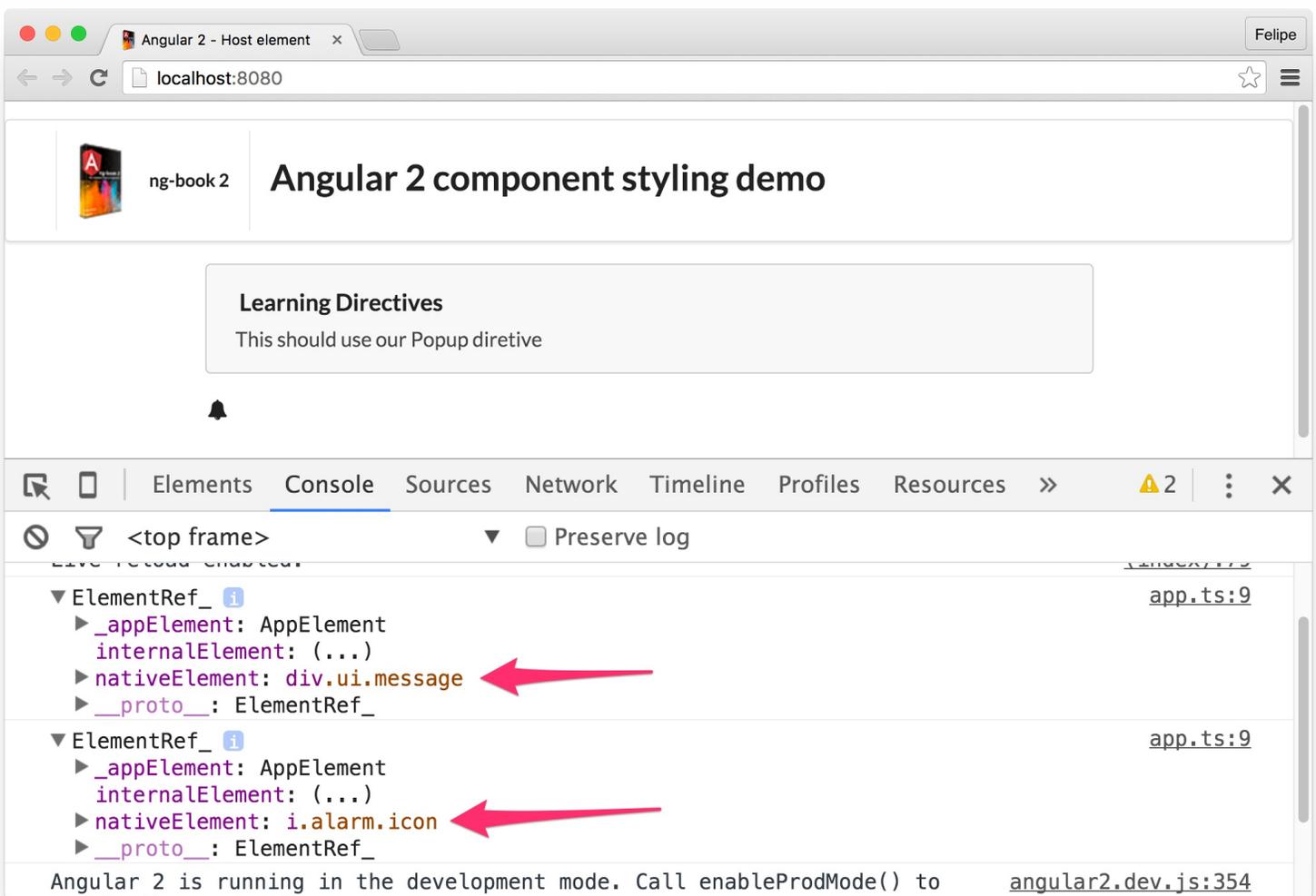
```
4 @Directive({
5   selector: '[popup]'
6 })
7 class Popup {
8   constructor(_elementRef: ElementRef) {
9     console.log(_elementRef);
10  }
11 }
```

We can also add a second element to the page that uses our directive, so we can see two different ElementRef logs to the console:

code/advanced_components/app/ts/host/steps/host_02.ts

```
13 @Component({
14   selector: 'host-sample-app',
15   directives: [Popup],
16   template: `
17     <div class="ui message" popup>
18       <div class="header">
19         Learning Directives
20       </div>
21
22       <p>
23         This should use our Popup directive
24       </p>
25     </div>
26
27     <i class="alarm icon" popup></i>
28 `
29 })
30 export class HostSampleApp2 {
31 }
```

When we run our app now, we can see two different ElementRef logs: one with `div.ui.message` and the other with `i.alarm.icon`. This means that the directive was successfully bound to two different host elements:



ElementRefs

Binding to the host

Moving on, our next goal is to do something when the host element is clicked.

We learned before that the way we bind events in elements in Angular is using the (event) syntax.

In order to bind events of the host element, we must do something very similar, but using the host attribute of the directive. **The host attribute allows a directive to change attributes and behaviors its host element.**

We also want the host element to define what message we will pop up when the element is clicked, using the message attribute.

In order to do that we use something we've used many times before: we add an inputs attribute to the directive.

Here's how our directive annotation looks like with those additions:

code/advanced_components/app/ts/host/steps/host_03.ts

```
4 @Directive({
5   selector: '[popup]',
6   inputs: ['message'],
```

```
7 host: {
8   '(click)': 'displayMessage()'
9 }
10 })
```

We're saying that we expect to receive an input called `message` and that when the host element is clicked we'll call the directive's `displayMessage` method.

We need to change our `Popup` class code by:

1. Adding a new `message` field to receive the input and
2. Creating the `displayMessage` function which will display the message the host element defines

Here's how we do it:

code/advanced_components/app/ts/host/steps/host_03.ts

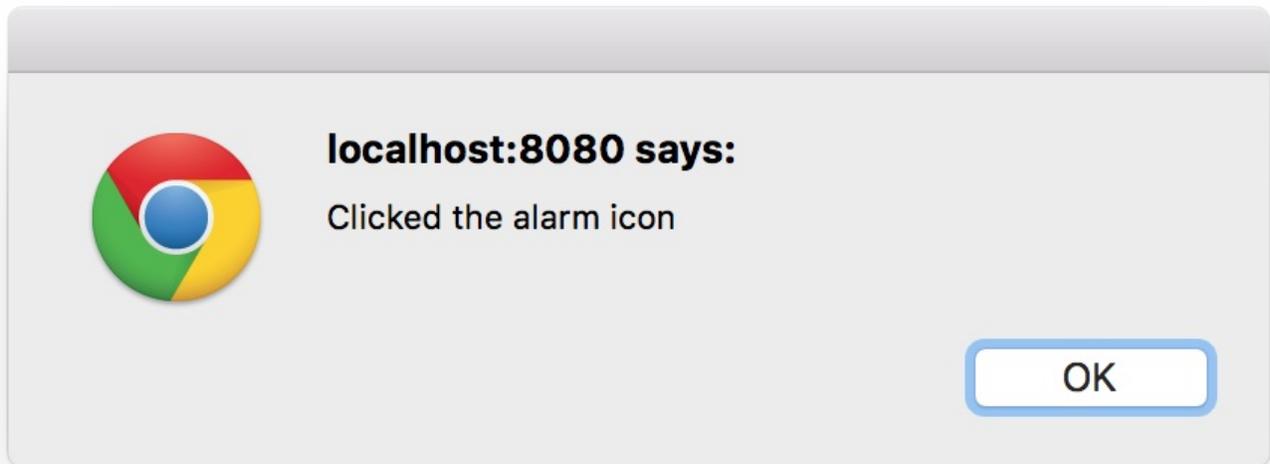
```
11 class Popup {
12   message: String;
13
14   constructor(_elementRef: ElementRef) {
15     console.log(_elementRef);
16   }
17
18   displayMessage(): void {
19     alert(this.message);
20   }
21 }
```

And finally, we need to change our app template a bit to add the message we want displayed for each element:

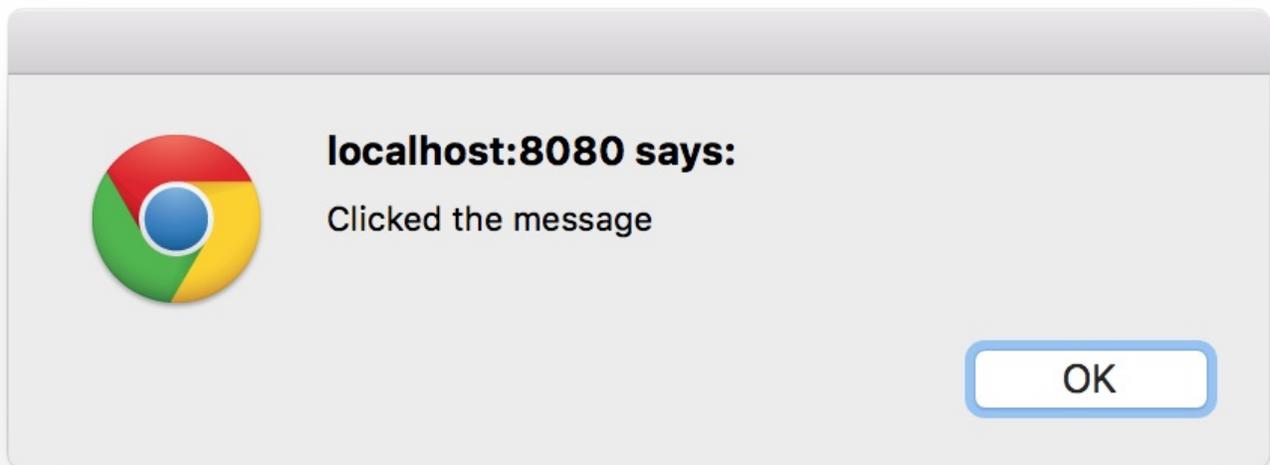
code/advanced_components/app/ts/host/steps/host_03.ts

```
23 @Component({
24   selector: 'host-sample-app',
25   directives: [Popup],
26   template: `
27     <div class="ui message" popup
28       message="Clicked the message">
29       <div class="header">
30         Learning Directives
31       </div>
32
33       <p>
34         This should use our Popup directive
35       </p>
36     </div>
37
38     <i class="alarm icon" popup
39       message="Clicked the alarm icon"></i>
40   `
41 })
42 export class HostSampleApp3 {
43 }
```

Notice that we use the `popup` directive twice, and we pass a different message each time we use it. This means when we run the app, we're able to click either on the message or on the alarm icon, and we'll see different messages:



Popup 1



Popup 2

Adding a Button using `exportAs`

Now let's say we have a new requirement: we want to trigger the alert manually by clicking a button. How could we trigger the popup message from **outside** the host element?

In order to achieve this, we need to make the **directive available from elsewhere in the template**. As we discussed in previous chapters, the way to reference a component is by using **template variables**. We can reference directives the same way.

In order to give the templates a reference to a directive we use the `exportAs` attribute. This will allow the host element (or a child of the host element) to define a template variable that references the directive using the `#var="exportName"` syntax.

Let's add the `exportAs` attribute to our directive:

code/advanced_components/app/ts/host/steps/host_04.ts

```
4 @Directive({
5   selector: '[popup]',
6   inputs: ['message'],
7   exportAs: 'popup',
8   host: {
9     '(click)': 'displayMessage()'
10  }
11 })
12 class Popup {
13   message: String;
14
15   constructor(_elementRef: ElementRef) {
16     console.log(_elementRef);
17   }
18
19   displayMessage(): void {
20     alert(this.message);
21   }
22 }
```

And now we need to change the two elements to export the template variable:

code/advanced_components/app/ts/host/steps/host_04.ts

```
28 <div class="ui message" popup #p1="popup"
29   message="Clicked the message">
30   <div class="header">
31     Learning Directives
32   </div>
33
34   <p>
35     This should use our Popup directive
36   </p>
37 </div>
38
39 <i class="alarm icon" popup #p2="popup"
40   message="Clicked the alarm icon"></i>
```

See that we used the template var `#p1` for the `div.message` and `#p2` for the icon.

Now let's add two buttons, one to trigger each popup:

code/advanced_components/app/ts/host/steps/host_04.ts

```
42 <div style="margin-top: 20px;">
43   <button (click)="p1.displayMessage()" class="ui button">
44     Display popup for message element
45   </button>
46
47   <button (click)="p2.displayMessage()" class="ui button">
48     Display popup for alarm icon
49   </button>
50 </div>
```

Now reload the page and click each of the buttons and each message will appear as expected.

Creating a Message Pane with Transclusion

Sometimes when we are creating components we want to pass inner markup as an argument to the component. This technique is called *transclusion*. The idea is that it lets us specify a bit of markup that will be expanded into a bigger template. Let's create a new directive that will render a nicely styled message like this:

Learning Directives

This should use our Popup directive

Popup 1

Our goal is to write markup like this:

```
1 <div message header="My Message">
2   This is the content of the message
3 </div>
```

Which will render into the more complicated HTML like:

```
1 <div class="ui message">
2   <div class="header">
3     My Message
4   </div>
5
6   <p>
7     This is the content of the message
8   </p>
9 </div>
```

We have two challenges here: we need to change the host element `<div>` to add the `ui` and `message` CSS classes, and we need to add the `div`'s contents to a specific place in our markup.

Changing the host CSS

To add attributes to the host element, we use the same attribute we used to add events to the host element: the `host` attributes. But now, instead of using the `(event)` notation, we define attribute names and attribute values. In our case using:

```
1 host: { 'class': 'ui message' }
```

Modified the host element, adding those to classes to the `class` attribute.

Using `ng-content`

Our next challenge is to include the original host element children in a specific part of a view. To do that, we use the `ng-content` directive.

Since this directive needs a template, let's use a component instead and write the following code:

code/advanced_components/app/ts/transclusion/transclusion.ts

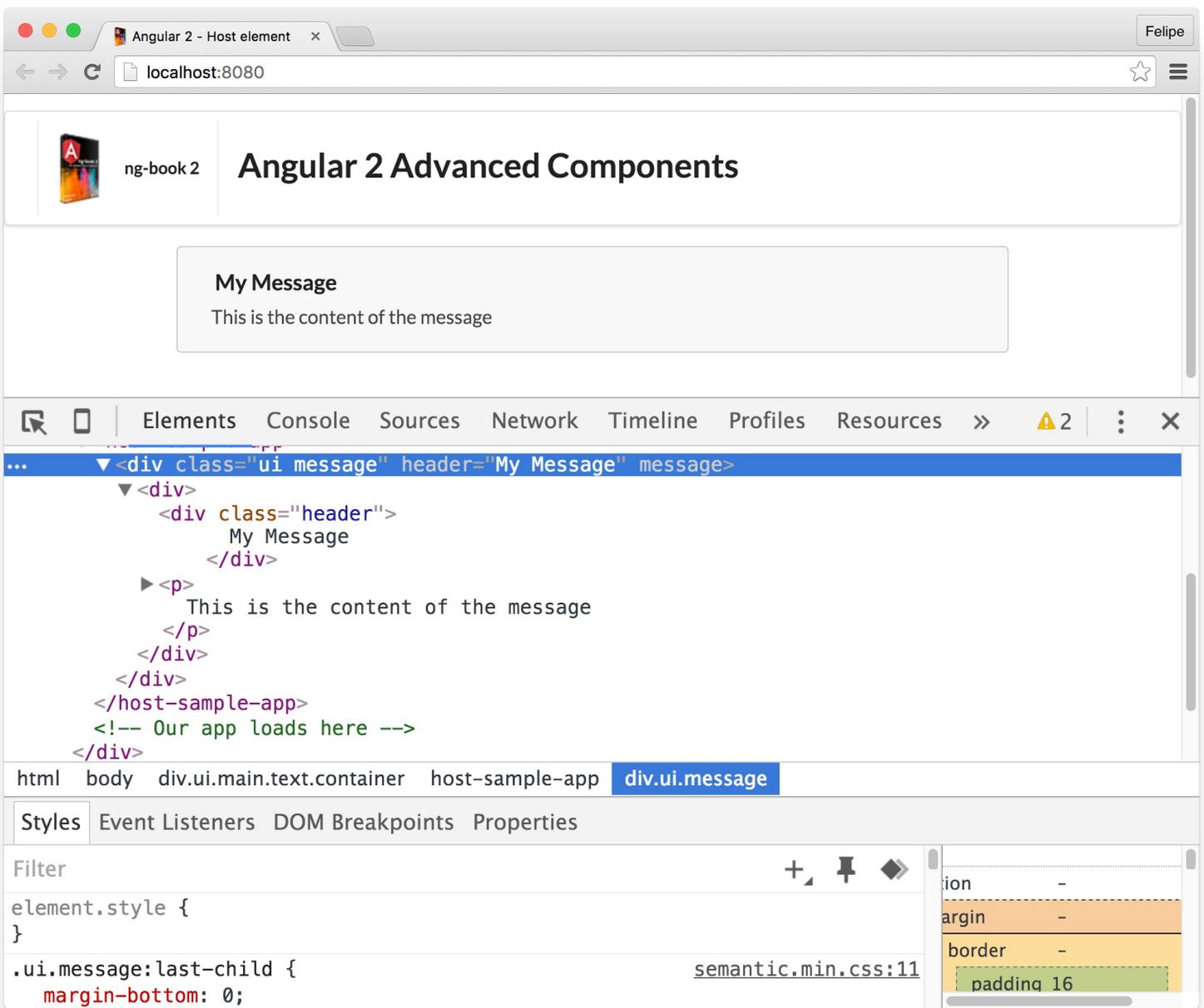
```
4 @Component({
5   selector: '[message]',
6   inputs: ['header'],
7   host: {
8     'class': 'ui message'
```

```
9   },
10  template: `
11  <div>
12    <div class="header">
13      {{ header }}
14    </div>
15    <p>
16      <ng-content></ng-content>
17    </p>
18  </div>
19  `
20 })
21 class Message {
22   header: string;
```

A few highlights:

- We use the `inputs` attribute to indicate we want to receive a message attribute, set on the host element
- We set the host element's `class` attribute to `ui message` using the `host` attribute of our component
- We use `<ng-content></ng-content>` to transclude the host element's children into a specific location of our template

When we open the app in the browser and inspect the message div, we see it worked exactly like we planned:



Transcluded content

Querying Neighbor Directives - Writing Tabs

It's great when you can create a component that fully encapsulates its own behavior.

However, as a component grows in features, it might make sense to split it up into several smaller components that work together.

A great example of components that work together is a tab pane that has multiple tabs. The tab panel or *tab set*, as it's usually called, is composed of multiple *tabs*. In this scenario we have a parent component (the *tabset*) and multiple child components (the *tabs*). The *tabset* and the *tabs* don't make sense separately, but putting all of the logic in one component is cumbersome. So in this example, we're going to cover how to make separate components that work together.

Let's start writing those components with goal that following markup can be used:

```
1 <tabset>
2   <tab title="Tab 1">Tab 1</tab>
3   <tab title="Tab 2">Tab 2</tab>
4   ...
5 </tabset>
```

We're going to use [Semantic UI Tab component](#) to render the tabs.

Tab Component

Let's start by writing the Tab component:

code/advanced_components/app/ts/tabs/tabs.ts

```
13 @Component({
14   selector: 'tab',
15   inputs: ['title'],
16   template: `
17     <div class="ui bottom attached tab segment"
18         [class.active]="active">
19
20       <ng-content></ng-content>
21
22     </div>
23 `
24 })
25 class Tab {
26   @Input('title') title: string;
27   active: boolean = false;
28   name: string;
29 }
```

There are not many new concepts here. We're declaring a component that will use the tab selector, and it will allow a `title` input to be set.

Then we're rendering a `<div>` and using the transclusion concept we learned on the previous section to inline the contents of the `<tab>` directive inside the `div`.

Next we declare 3 properties on our components: *title*, *active* and *name*. One thing to notice is the `@Input('title')` annotation we added to the `title` property. This annotation is a way to ask Angular to automatically bind the value of the *input* `title` into the *property* `title`.

Tabset Component

Now let's move on to the Tabset component that will be used to wrap the tabs:

code/advanced_components/app/ts/tabs/tabs.ts

```
31 @Component({
32   selector: 'tabset',
33   template: `
34     <div class="ui top attached tabular menu">
35       <a *ngFor="let tab of tabs"
36         class="item"
37         [class.active]="tab.active"
38         (click)="setActive(tab)">
39
40         {{ tab.title }}
41
42       </a>
43     </div>
44     <ng-content></ng-content>
45 `
46 })
47 class Tabset implements AfterContentInit {
48   tabs: QueryList<Tab>;
49 }
```

```
50 constructor(@Query(Tab) tabs:QueryList<Tab>) {
51   this.tabs = tabs;
52 }
53
54 ngAfterContentInit() {
55   this.tabs.toArray()[0].active = true;
56 }
57
58 setActive(tab: Tab) {
59   this.tabs.toArray().forEach((t) => t.active = false);
60   tab.active = true;
61 }
62 }
```

Let's break down the implementation so we can learn about the new concepts it introduces.

Tabset @Component Annotation

The @Component section doesn't have many new ideas. We're using the <tabset> tab as our selector.

The template itself uses ngFor to iterate through the tabs and if the tab has the *active* flag set to true, it will add the *active* CSS class to the <a> element that renders the tab.

We also specify that we are rendering the tabs themselves after the initial *div*, right where **ng-content** is.

Tabset class

Now let's turn our attention to the Tabset class. The first new idea we see here is that the Tabset class is implementing AfterContentInit. This *lifecycle hook* will tell Angular to call a method of our class (ngAfterContentInit) once the contents of the child directives has been initialized.

Tabset Query and QueryList

Next thing we do is declare the tabs property that will hold every Tab component we declare inside the **tabset**. Notice that instead of declaring this list as an array of Tabs, instead we use the class QueryList, passing a generic of Tab. Why is this?

QueryList is a class provided by Angular and when we use QueryList with a Query (as we do in the constructor) Angular populate this with the **components that match the query** and then **keep the items up to date** if the state of the application changes.

However, QueryList requires a Query to populate it, so let's take a look at that now:

On the constructor we use the @Query(Tab). This annotation will tell Angular to inject all the direct child directives (of the Tab type) into the tabs parameter. We then assign it to the tabs property of our component. With this **we now have access to all the child Tab components**.

Initializing the Tabset

When this component is initialized, we want to make the first tab active. To do this we use the ngAfterContentInit function (that is described by the AfterContentInit hook). Notice that we use this.tabs.toArray() to cast the Angular's QueryList into a native TypeScript array.

Tabset setActive

Finally we define a setActive method. This method is used when we click a tab on our template e.g. using (click)="setActive(tab)". This function will iterate through all the tabs, setting their active

properties to false. Then we set the tab we clicked active.

Using the Tabset

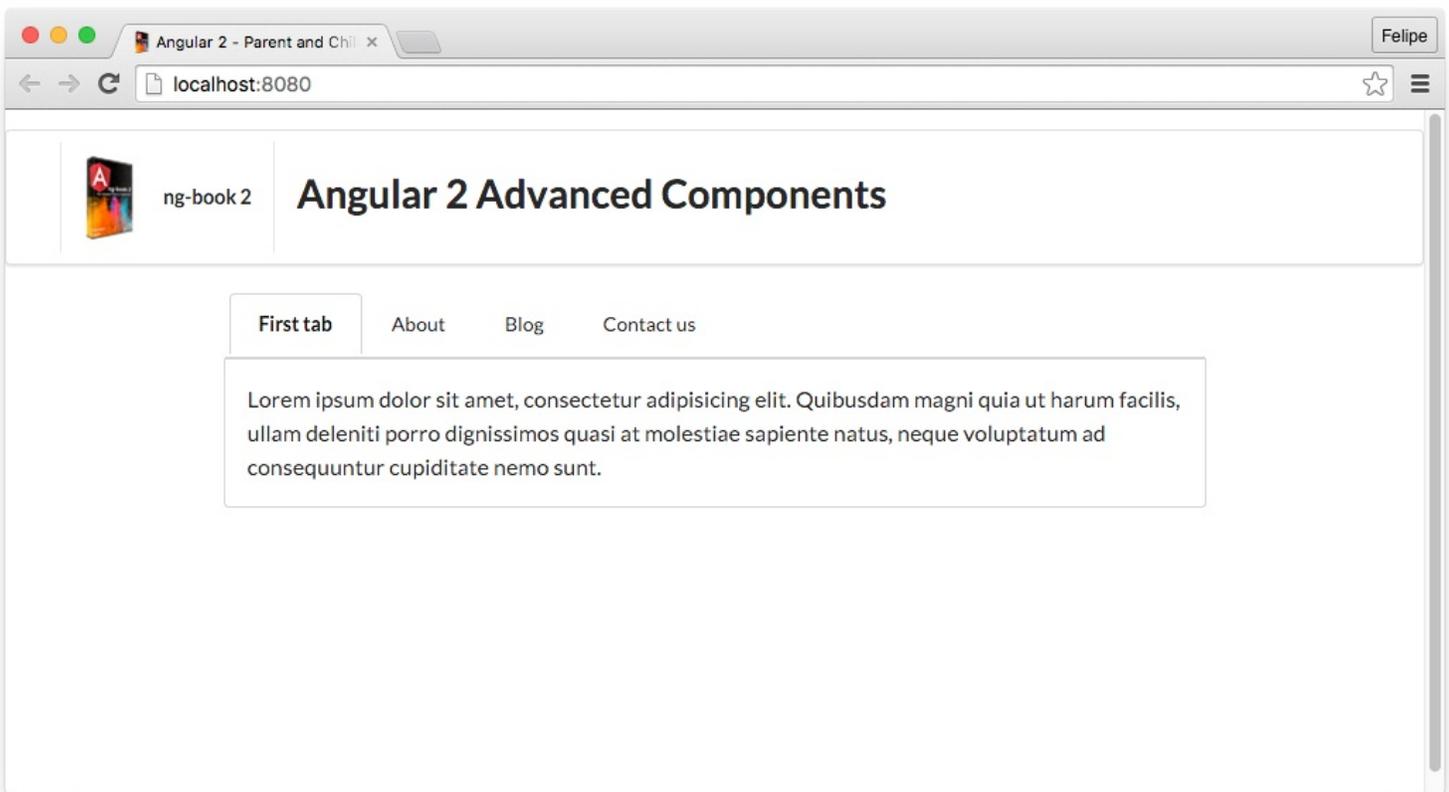
Now the next step is to code the application component that makes use of both the components we created. Here's how we do it:

code/advanced_components/app/ts/tabs/tabs.ts

```
64 @Component({
65   selector: 'tabs-sample-app',
66   directives: [Tabset, Tab],
67   template: `
68     <tabset>
69       <tab title="First tab">
70         Lorem ipsum dolor sit amet, consectetur adipiscing elit.
71         Quibusdam magni quia ut harum facilis, ullam deleniti porro
72         dignissimos quasi at molestiae sapiente natus, neque voluptatum
73         ad consequuntur cupiditate nemo sunt.
74       </tab>
75       <tab *ngFor="let tab of tabs" [title]="tab.title">
76         {{ tab.content }}
77       </tab>
78     </tabset>
79   `
80 })
81 export class TabsSampleApp {
82   tabs: any;
83
84   constructor() {
85     this.tabs = [
86       { title: 'About', content: 'This is the About tab' },
87       { title: 'Blog', content: 'This is our blog' },
88       { title: 'Contact us', content: 'Contact us here' },
89     ];
90   }
91 }
```

We're declaring that we're using **tabs-sample-app** as our component's selector and using the **Tabset** and **Tab** components.

On the template we then create a **tabset** and we add first a static tab (First tab) and we add a few more tabs from the **tabs** property of the component controller class, to illustrate how we can render tabs dynamically.



Tabset application

Lifecycle Hooks

Lifecycle hooks are the way Angular allows you to add code that runs before or after each step of the directive lifecycle.

The list of hooks Angular offers are:

- OnInit
- OnDestroy
- DoCheck
- OnChanges
- AfterContentInit
- AfterContentChecked
- AfterViewInit
- AfterViewChecked

Using these hooks each follow a similar pattern:

In order to be notified about those events you

1. declare that your directive class implements the interface and then
2. declare the ng method of the hook (e.g. ngOnInit)

Every method name is ng plus the name of the hook. For example, for onInit we declare the method ngOnInit, for AfterContentInit we declare ngAfterContentInit and so on.

When Angular knows that a component implements these functions, it will invoke them at the appropriate time.

Let's take a look at each hook individually and when we would use each of them.

onInit and onDestroy

The `onInit` hook is called when your directive properties have been initialized, and before any of the child directive properties are initialized.

Similarly, the `onDestroy` hook is called when the directive instance is destroyed. This is typically used if we need to do some cleanup every time our directive is destroyed.

In order to illustrate let's write a component that implements both `onInit` and `onDestroy`:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_01.ts

```
16 @Component({
17   selector: 'on-init',
18   template: `
19     <div class="ui label">
20       <i class="cubes icon"></i> Init/Destroy
21     </div>
22   `
23 })
24 class OnInitCmp implements OnInit, OnDestroy {
25   ngOnInit(): void {
26     console.log('On init');
27   }
28
29   ngOnDestroy(): void {
30     console.log('On destroy');
31   }
32 }
```

For this component, we're just logging *On init* and *On destroy* to the console when the hooks are called.

Now in order to test those hooks let's use our component in our app component using `ngFor` to conditionally display it based on a boolean property. Let's also add a button that allows us to toggle that flag. This way, when the flag is false, our component will be *removed* from the page, causing the `onDestroy` hook to be called. Similarly when the flag is toggled to true, the `onInit` hook will be called.

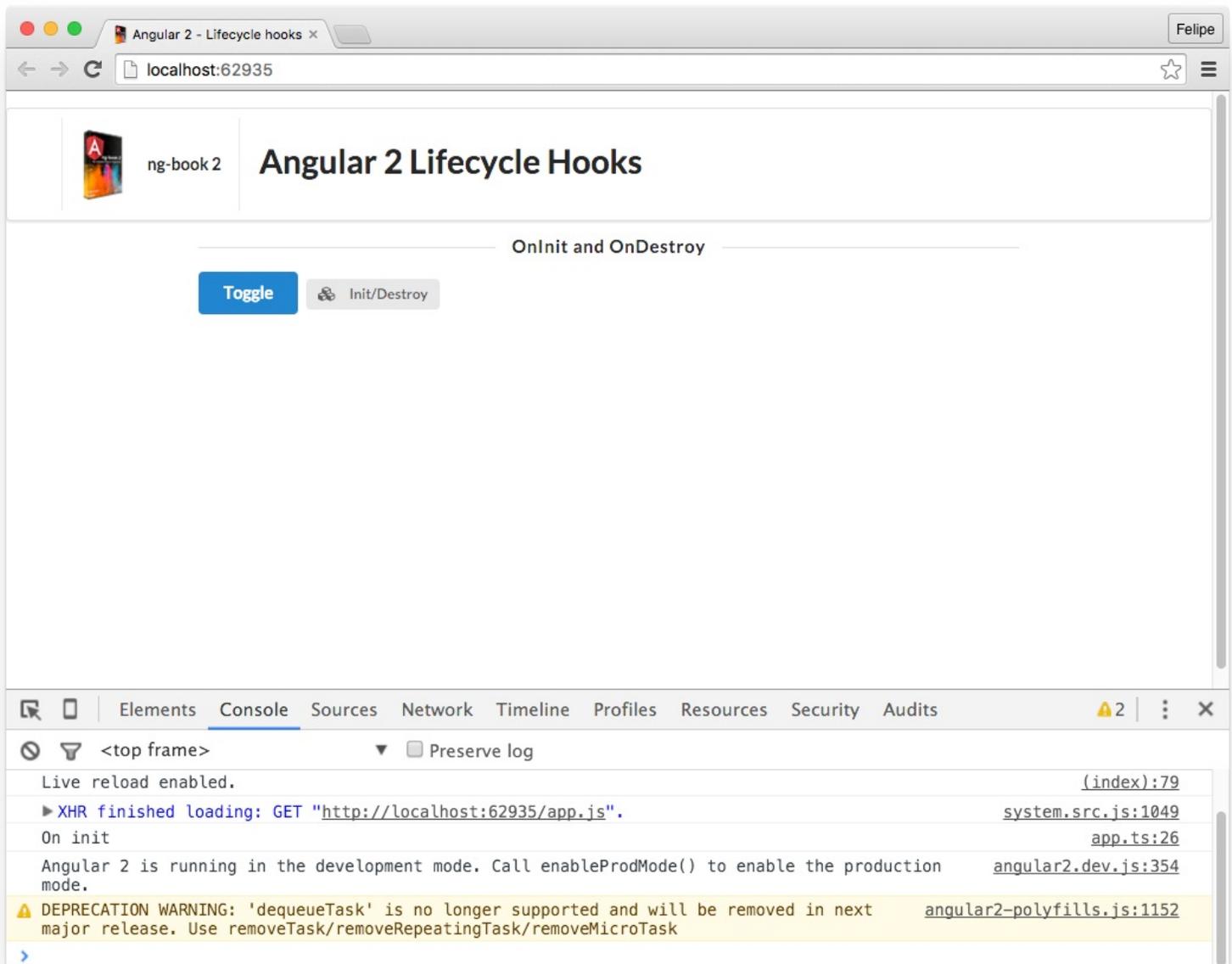
Here's how our app component will look:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_01.ts

```
34 @Component({
35   selector: 'lifecycle-sample-app',
36   directives: [OnInitCmp],
37   template: `
38     <h4 class="ui horizontal divider header">
39       OnInit and OnDestroy
40     </h4>
41
42     <button class="ui primary button" (click)="toggle()">
43       Toggle
44     </button>
45     <on-init *ngIf="display"></on-init>
46   `
47 })
48 export class LifecycleSampleApp1 {
49   display: boolean;
50
51   constructor() {
```

```
52   this.display = true;
53 }
```

When we first run the application, we can see that the `OnInit` hook was called when the component was first instantiated:



Initial state of our component

When I click the **Toggle** button for the first time, the component is destroyed and the hook is called as expected:

Angular 2 - Lifecycle hooks x Felipe

localhost:62935

ng-book 2 **Angular 2 Lifecycle Hooks**

OnInit and OnDestroy

Toggle Init/Destroy

Elements Console Sources Network Timeline Profiles Resources Security Audits 2

<top frame> Preserve log

Live reload enabled. (index):79

► XHR finished loading: GET "http://localhost:62935/app.js". system.src.js:1049

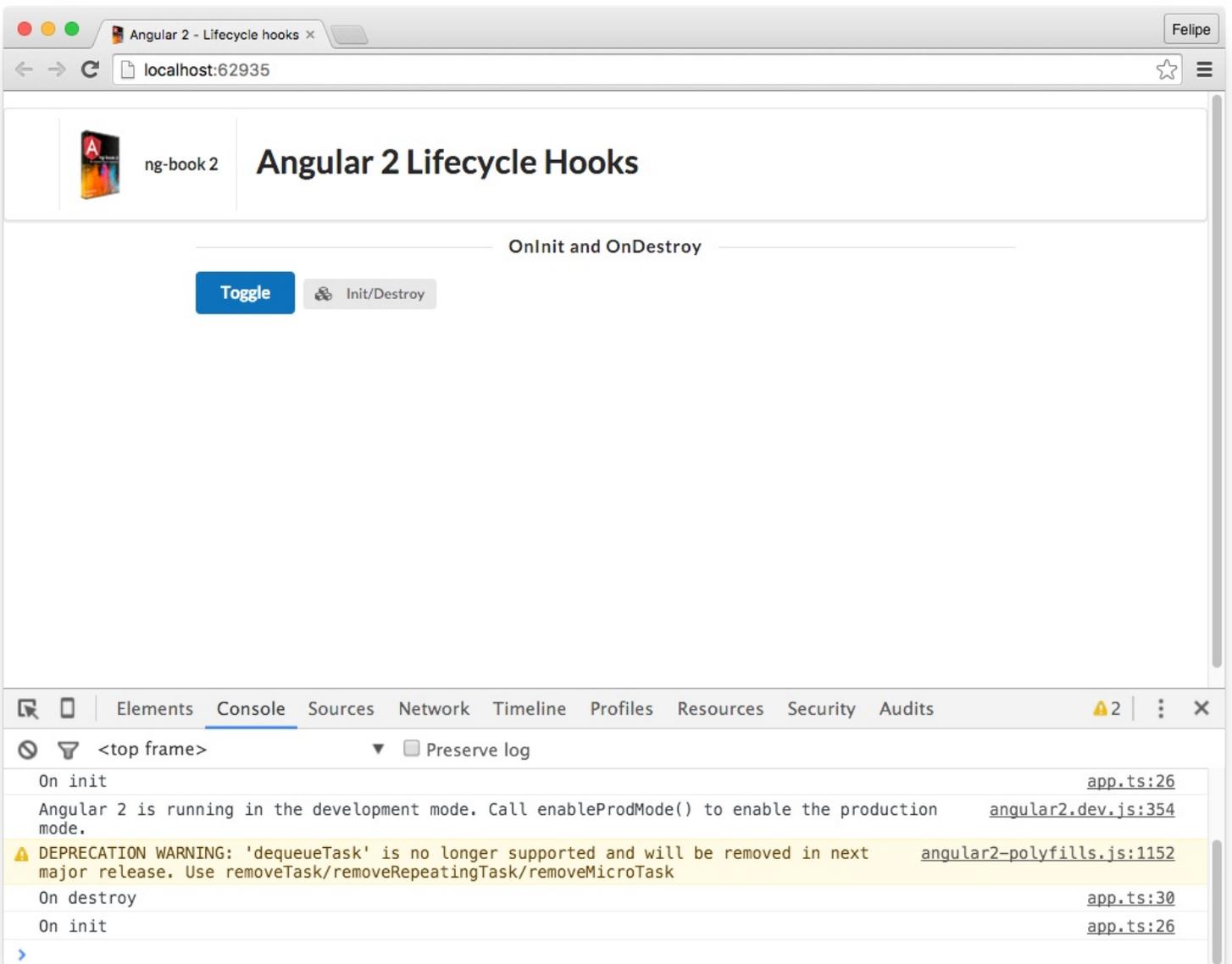
On init app.ts:26

Angular 2 is running in the development mode. Call enableProdMode() to enable the production mode. angular2.dev.js:354

⚠ DEPRECATION WARNING: 'dequeueTask' is no longer supported and will be removed in next major release. Use removeTask/removeRepeatingTask/removeMicroTask angular2-polyfills.js:1152

OnDestroy hook

And if we click it another time:



OnDestroy hook

OnChanges

The `onChanges` hook is called after one or more of our component properties have been changed. The `ngOnChanges` method receives a parameter which tells which properties have changed.

To understand this better, let's write a comment block component that have two inputs: *name* and *comment*:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
36 @Component({
37   selector: 'on-change',
38   inputs: ['name', 'comment'],
39   template: `
40   <div class="ui comments">
41     <div class="comment">
42       <a class="avatar">
43         
44       </a>
45       <div class="content">
46         <a class="author">{{name}}</a>
47         <div class="text">
48           {{comment}}
```

```

49     </div>
50   </div>
51 </div>
52 </div>
53 `
54 })
55 class OnChangeCmp implements OnChanges {
56   @Input('name') name: string;
57   @Input('comment') comment: string;
58
59   ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
60     console.log('Changes', changes);
61   }
62 }

```

The important thing about this component is that it implements the `OnChanges` interface, and it declares the `ngOnChanges` method with this signature:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```

59   ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
60     console.log('Changes', changes);
61   }

```

This method will be triggered whenever the values of either the *name* or *comment* properties change. When that happens, we receive an object that maps changed fields to `SimpleChange` objects.

Each `SimpleClass` instance has two fields: `currentValue` and `previousValue`. If both `name` and `comment` properties change for our component, we expect the value of `changes` in our method to be something like:

```

1 {
2   name: {
3     currentValue: 'new name value',
4     previousValue: 'old name value'
5   },
6   comment: {
7     currentValue: 'new comment value',
8     previousValue: 'old comment value'
9   }
10 }

```

Now, let's change the app component to use our component and also add a little form where we can play with the `name` and `comment` properties of our component:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```

64 @Component({
65   selector: 'lifecycle-sample-app',
66   directives: [OnInitCmp, OnChangeCmp],
67   template: `
68     <h4 class="ui horizontal divider header">
69       OnInit and OnDestroy
70     </h4>
71
72     <button class="ui primary button" (click)="toggle()">
73       Toggle
74     </button>
75     <on-init *ngIf="display"></on-init>
76
77     <h4 class="ui horizontal divider header">
78       OnChange
79     </h4>
80
81     <div class="ui form">
82       <div class="field">
83         <label>Name</label>
84         <input type="text" #namefld value="{{name}}"
85           (keyup)="setValues(namefld, commentfld)">

```

```

86     </div>
87
88     <div class="field">
89       <label>Comment</label>
90       <textarea (keyup)="setValues(namefld, commentfld)"
91         rows="2" #commentfld>{{comment}}</textarea>
92     </div>
93 </div>
94
95 <on-change [name]="name" [comment]="comment"></on-change>
96
97 })
98 export class LifecycleSampleApp2 {
99   display: boolean;
100   name: string;
101   comment: string;
102
103   constructor() {
104     this.display = true;
105     this.name = 'Felipe Coury';
106     this.comment = 'I am learning so much!';
107   }
108
109   setValues(namefld, commentfld): void {
110     this.name = namefld.value;
111     this.comment = commentfld.value;
112   }
113
114   toggle(): void {
115     this.display = !this.display;
116   }
117 }

```

The important pieces that we added here were the template areas where we declare a new form with name and comment fields:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```

81 <div class="ui form">
82   <div class="field">
83     <label>Name</label>
84     <input type="text" #namefld value="{{name}}"
85       (keyup)="setValues(namefld, commentfld)">
86   </div>
87
88   <div class="field">
89     <label>Comment</label>
90     <textarea (keyup)="setValues(namefld, commentfld)"
91       rows="2" #commentfld>{{comment}}</textarea>
92   </div>
93 </div>

```

Here, when the *keyup* event is fired for both the name or comment fields, we are calling `setValues` with the template vars `namefld` and `commentfld` that represent the input and textarea.

This method just takes the value from those fields and updates the name and comment properties accordingly:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```

109 setValues(namefld, commentfld): void {
110   this.name = namefld.value;
111   this.comment = commentfld.value;
112 }

```

So now, the first time we open the app, we can see that our `onChanges` hook is called:

Angular 2 - Lifecycle hooks x Felipe

localhost:8080

ng-book 2 **Angular 2 Lifecycle Hooks**

OnInit and OnDestroy

Toggle Init/Destroy

OnChange

Name
Felipe Coury

Comment
I am learning so much!

Felipe Coury
I am learning so much!

Elements Console Sources Network Timeline Profiles Resources Security Audits 2

<top frame> Preserve log

```
Changes ▼ Object {name: SimpleChange, comment: SimpleChange} app.ts:60
  ▼ comment: SimpleChange
    currentValue: "I am learning so much!"
    ▶ previousValue: Object
    ▶ __proto__: SimpleChange
  ▼ name: SimpleChange
    currentValue: "Felipe Coury"
    ▶ previousValue: Object
    ▶ __proto__: SimpleChange
  ▶ __proto__: Object
```

OnChange

This happens when the initial values are set, on the constructor of the `LifecycleSampleApp` component.

Now if we play with the name, we can see that the hook is called repeatedly. In the case below, we pasted the name *Nate Murray* on top of the previous name, and the values for the changes are displayed as expected:

Angular 2 - Lifecycle hooks x Felipe

localhost:8080

ng-book 2 **Angular 2 Lifecycle Hooks**

OnInit and OnDestroy

Toggle Init/Destroy

OnChange

Name

Nate Murray

Comment

I am learning so much!

Nate Murray
I am learning so much!

Elements Console Sources Network Timeline Profiles Resources Security Audits

<top frame> Preserve log

```
Changes ▾ Object {name: SimpleChange} ⓘ app.ts:60
  ▾ name: SimpleChange
    currentValue: "Nate Murray"
    previousValue: "Felipe Coury"
    ▶ __proto__: SimpleChange
    ▶ __proto__: Object
```

OnChange

DoCheck

The default notification system implemented by `OnChange` is triggered every time the Angular change detection mechanism notices there was a change on any of the directive properties.

However, there may be times when the overhead added by this change notification may be too much, specially if performance is a concern.

There may be times when we just want to do something in case an item was removed or added, or if only a particular property changed, for instance.

If we run into one of these scenarios, we can use the `DoCheck` hook.



It's important to note that the `OnChange` hook gets overridden by `DoCheck` so if we implement both, `OnChange` will be ignored.

Checking for changes

In order to evaluate what changed, Angular provides *differs*. **Differs will evaluate a given property of your directive to determine *what* changed.**

There are two types of built-in differs: *iterable differs* and *key-value differs*.

Iterable differs

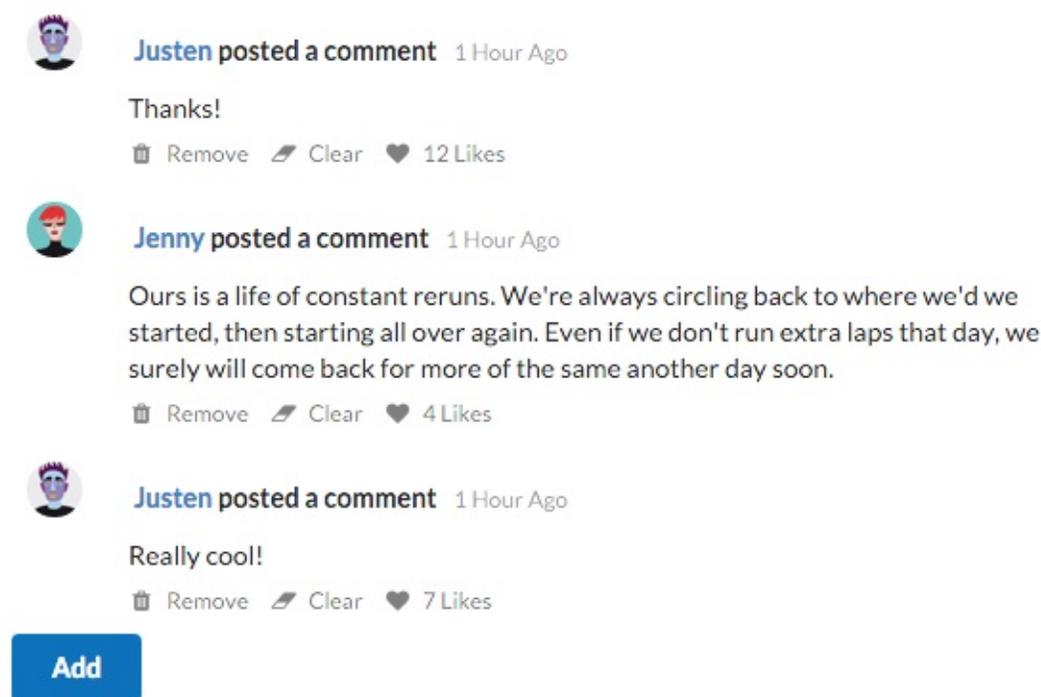
Iterable differs should be used when we have a list-like structure and we're only interested on knowing things that were added or removed from that list.

Key-value differs

Key-value differs should be used for dictionary-like structures, and it works at the key level. This differ will identify changes when a new key is added, when a key removed and when the value of a key changed.

Rendering a comment with `do-check-item`

To illustrate these concepts, let's build a component that renders a stream of comments, like below:



The screenshot shows a vertical list of three comments. Each comment has a circular profile picture on the left, followed by the author's name and 'posted a comment' in blue, and '1 Hour Ago' in grey. The first comment is from 'Justen' and says 'Thanks!'. Below it are icons for 'Remove', 'Clear', and '12 Likes'. The second comment is from 'Jenny' and says 'Ours is a life of constant reruns. We're always circling back to where we'd we started, then starting all over again. Even if we don't run extra laps that day, we surely will come back for more of the same another day soon.' Below it are icons for 'Remove', 'Clear', and '4 Likes'. The third comment is from 'Justen' and says 'Really cool!'. Below it are icons for 'Remove', 'Clear', and '7 Likes'. At the bottom left of the feed is a blue button with the text 'Add' in white.

DoCheck example

First, let's write a component that will render one individual comment:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
67 @Component({
68   selector: 'do-check-item',
69   inputs: ['comment'],
70   outputs: ['onRemove'],
71   template: `
72     <div class="ui feed">
73       <div class="event">
74         <div class="label" *ngIf="comment.author">
75           
76         </div>
77         <div class="content">
78           <div class="summary">
79             <a class="user">
80               {{comment.author}}
```

```

81     </a> posted a comment
82     <div class="date">
83       1 Hour Ago
84     </div>
85   </div>
86   <div class="extra text">
87     {{comment.comment}}
88   </div>
89   <div class="meta">
90     <a class="trash" (click)="remove()">
91       <i class="trash icon"></i> Remove
92     </a>
93     <a class="trash" (click)="clear()">
94       <i class="eraser icon"></i> Clear
95     </a>
96     <a class="like" (click)="like()">
97       <i class="like icon"></i> {{comment.likes}} Likes
98     </a>
99   </div>
100 </div>
101 </div>
102 </div>
103 `
104 })

```

Here we are declaring the component metadata. Our component will receive the comment that should be rendered and it will emit an event with the remove button icon clicked.

Moving on to the implementation:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```

105 class DoCheckItem implements DoCheck {
106   @Input('comment') comment: any;
107   onRemove: EventEmitter<any>;
108   differ: any;

```

On the class declaration we indicate we're implementing the `DoCheck` interface. We then declare the input property `comment`, and the output event `onRemove`. We also declare a `differ` property.

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```

110   constructor(differs: KeyValueDiffers) {
111     this.differ = differs.find([]).create(null);
112     this.onRemove = new EventEmitter();
113   }

```

On the constructor we're receiving a `KeyValueDiffers` instance on the `differs` variable. We then use this variable to create an instance of the key value differ using this syntax `differs.find([]).create(null)`. We're also initializing our event emitter `onRemove`.

Next, let's implement the `ngDoCheck` method, required by the interface:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```

115   ngDoCheck(): void {
116     var changes = this.differ.diff(this.comment);
117
118     if (changes) {
119       changes.forEachAddedItem(r => this.logChange('added', r));
120       changes.forEachRemovedItem(r => this.logChange('removed', r));
121       changes.forEachChangedItem(r => this.logChange('changed', r));
122     }
123   }

```

This is how you check for changes, if you're using a key-value differ. You call the `diff` method, providing the property you want to check. In our case, we want to know if there were changes to the

comment property.

When no changes are detected, the returned value will be `null`. Now, if there are changes, we can call three different iterable methods on the differ:

- `forEachAddedItem`, for *keys* that were added
- `forEachRemovedItem`, for *keys* that were removed
- `forEachChangedItem`, for *keys* that were changed

Each method will call the provided callback with a *record*. For the key-value differ, this record will be an instance of the `KVChangeRecord` class.

```
▼ KVChangeRecord {key: "likes", previousValue: null, currentValue: 10, _nextPrevious: null, _next: null...} ⓘ  
  _next: null  
  _nextAdded: null  
  _nextChanged: null  
  _nextPrevious: null  
  _nextRemoved: null  
  _prevRemoved: null  
  currentValue: 10  
  key: "likes"  
  previousValue: 10
```

Example of a `KVChangeRecord` instance

The important fields for understanding what changed are *key*, *previousValue* and *currentValue*.

Next, let's write a method that will log to the console a nice sentence about what changed:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
125 logChange(action, r) {  
126   if (action === 'changed') {  
127     console.log(r.key, action, 'from', r.previousValue, 'to', r.currentValue);  
128   }  
129   if (action === 'added') {  
130     console.log(action, r.key, 'with', r.currentValue);  
131   }  
132   if (action === 'removed') {  
133     console.log(action, r.key, '(was ' + r.previousValue + ')');  
134   }  
135 }
```

Finally, let's write the methods that will help us change things on our component, to trigger our `DoCheck` hook:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
137 remove(): void {  
138   this.onRemove.emit(this.comment);  
139 }  
140  
141 clear(): void {  
142   delete this.comment.comment;  
143 }  
144  
145 like(): void {  
146   this.comment.likes += 1;  
147 }
```

The `remove()` method will emit the event indicating that the user asked for this comment to be removed, the `clear()` method will remove the comment text from the comment object, and the `like()` method will increase to the like counter for the comment.

Rendering a list of comments with do-check

Now that we have written a component for one individual comment, let's write a second component that will be responsible for rendering the list of comments:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
150 @Component({
151   selector: 'do-check',
152   directives: [DoCheckItem],
153   template: `
154     <do-check-item [comment]="comment"
155       *ngFor="let comment of comments" (onRemove)="removeComment($event)">
156     </do-check-item>
157
158     <button class="ui primary button" (click)="addComment()">
159       Add
160     </button>
161   `
162 })
```

The component metadata is pretty straightforward: we're using the component we created above, and then using `ngFor` to iterate through a list of comments, rendering them. We also have a button that will allow the user to add more comments to the list.

Now let's implement our comment list class `DoCheckCmp`:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
163 class DoCheckCmp implements DoCheck {
164   comments: any[];
165   iterable: boolean;
166   authors: string[];
167   texts: string[];
```

Here we declare the variables we'll use: `comments`, `iterable`, `authors`, and `texts`.

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
170 constructor(differ: IterableDiffer) {
171   this.differ = differ.find([]).create(null);
172   this.comments = [];
173
174   this.authors = ['Elliot', 'Helen', 'Jenny', 'Joe', 'Justen', 'Matt'];
175   this.texts = [
176     "Ours is a life of constant reruns. We're always circling back to where we\
177 'd we started, then starting all over again. Even if we don't run extra laps tha\
178 t day, we surely will come back for more of the same another day soon.",
179     'Really cool!',
180     'Thanks!'
181   ];
182
183   this.addComment();
184 }
```

For this component, we'll be using an `iterable differ`. We can see that the class we're using to create the differ is now `IterableDiffer`. However, the way we create a differ remains the same.

On the constructor we also initialize a list of authors and a list of comment texts to be used when adding new comments.

Finally, we call the `addComment()` method so we don't initialize the app with an empty list of comments.

The next three methods are used to add a new comment:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
184  getRandomInt(max: number): number {
185      return Math.floor(Math.random() * (max + 1));
186  }
187
188  getRandomItem(array: string[]): string {
189      let pos: number = this.getRandomInt(array.length - 1);
190      return array[pos];
191  }
192
193  addComment(): void {
194      this.comments.push({
195          author: this.getRandomItem(this.authors),
196          comment: this.getRandomItem(this.texts),
197          likes: this.getRandomInt(20)
198      });
199  }
200
201  removeComment(comment) {
202      let pos = this.comments.indexOf(comment);
203      this.comments.splice(pos, 1);
204  }
```

We are declaring two methods that will return a random integer and a random item from an array, respectively.

Finally, the `addComment()` method will push a new comment to the list, with a random author, random text and a random number of likes.

Next, we have the `removeComment()` method, that will be used to remove one comment from the list:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
201  removeComment(comment) {
202      let pos = this.comments.indexOf(comment);
203      this.comments.splice(pos, 1);
204  }
```

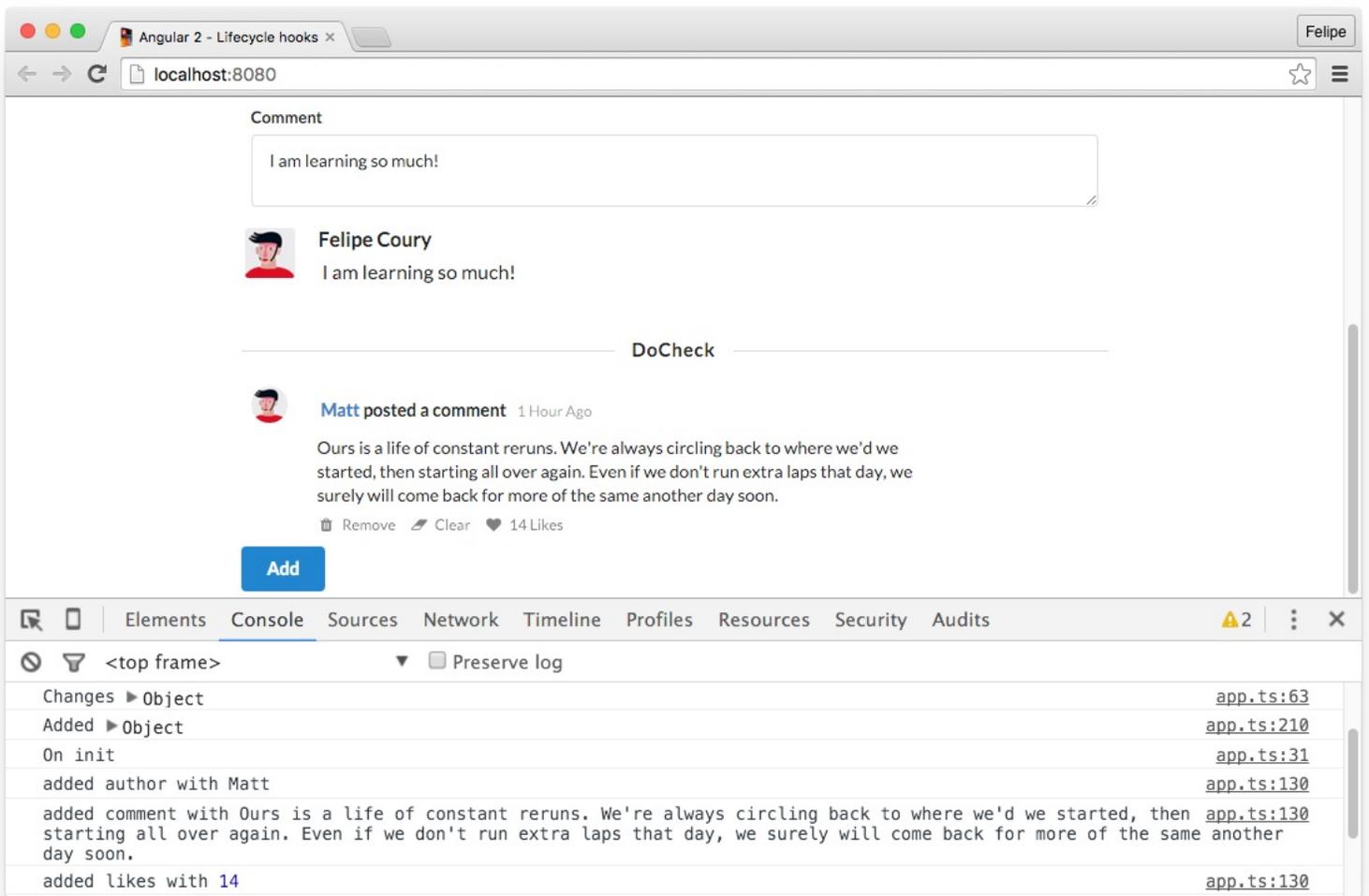
And finally we declare our change detection method `ngDoCheck()`:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
206  ngDoCheck(): void {
207      var changes = this.differ.diff(this.comments);
208
209      if (changes) {
210          changes.forEachAddedItem(r => console.log('Added', r.item));
211          changes.forEachRemovedItem(r => console.log('Removed', r.item));
212      }
213  }
214 }
```

The iterable differ behaves the same way the key-value differ but it only provide methods for items that were added or removed.

When we run the app now, we get the list of comments with one comment:



Initial state

We can also see that a few things were logged to the console, like:

- 1 added author with Matt
- 2 ...
- 3 added likes with 14

Let's see what happens when we add a new comment to the list by clicking the Add button:

Angular 2 - Lifecycle hooks x Felipe

localhost:8080

 **Felipe Coury**
I am learning so much!

DoCheck

 **Matt posted a comment** 1 Hour Ago

Ours is a life of constant reruns. We're always circling back to where we'd we started, then starting all over again. Even if we don't run extra laps that day, we surely will come back for more of the same another day soon.

 Remove  Clear  14 Likes

 **Helen posted a comment** 1 Hour Ago

Thanks!

 Remove  Clear  17 Likes

Add

Elements Console Sources Network Timeline Profiles Resources Security Audits

<top frame> Preserve log

Added Object {author: "Helen", comment: "Thanks!", likes: 17}	app.ts:210
added author with Helen	app.ts:130
added comment with Thanks!	app.ts:130
added likes with 17	app.ts:130

Angular 2 - Lifecycle hooks x Felipe

localhost:8080

 **Felipe Coury**
I am learning so much!

DoCheck

 **Matt posted a comment** 1 Hour Ago

Ours is a life of constant reruns. We're always circling back to where we'd we started, then starting all over again. Even if we don't run extra laps that day, we surely will come back for more of the same another day soon.

 Remove  Clear  14 Likes

 **Helen posted a comment** 1 Hour Ago

Thanks!

 Remove  Clear  17 Likes

Add

Elements Console Sources Network Timeline Profiles Resources Security Audits

<top frame> Preserve log

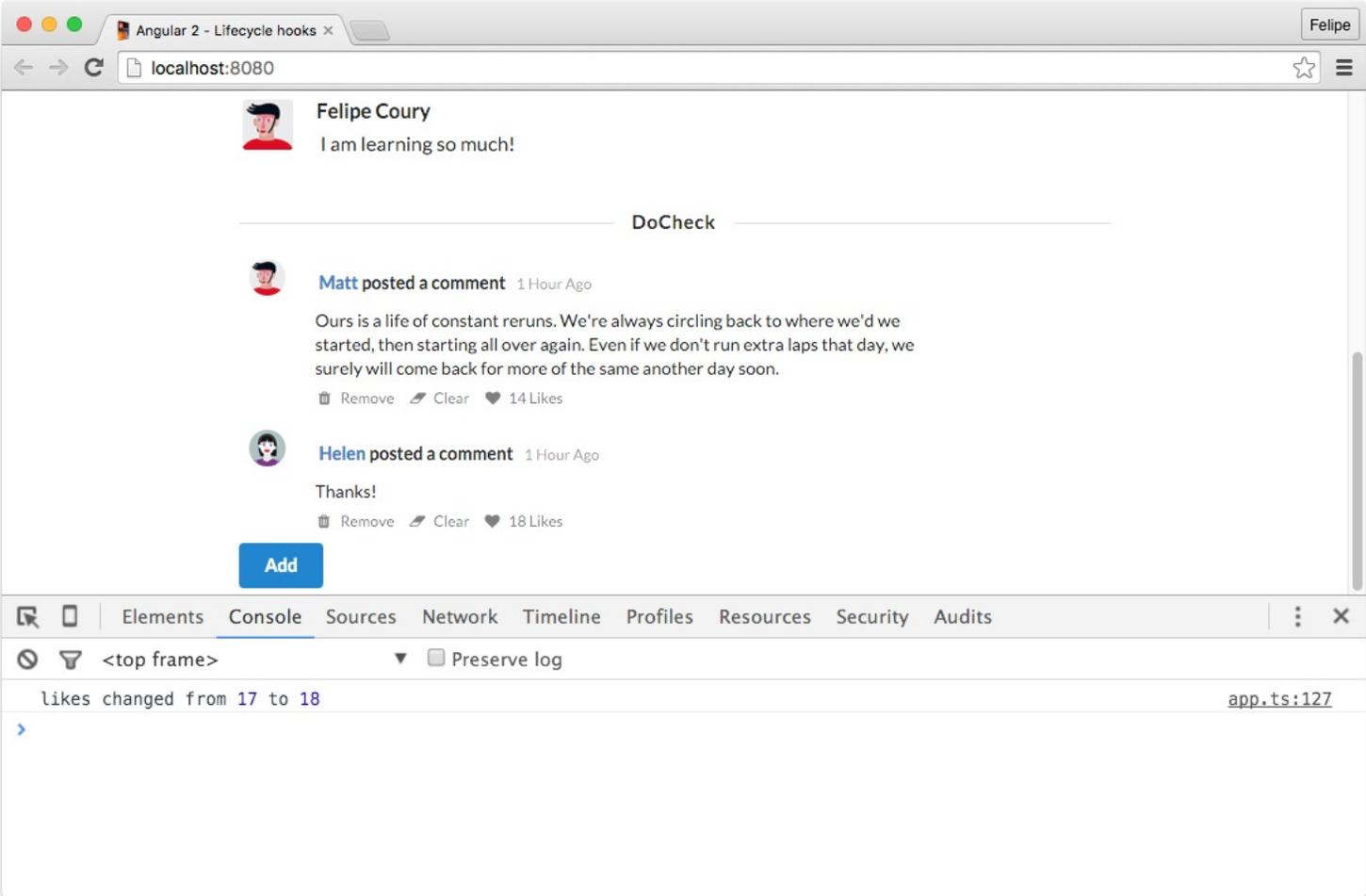
Added Object {author: "Helen", comment: "Thanks!", likes: 17}	app.ts:210
added author with Helen	app.ts:130
added comment with Thanks!	app.ts:130
added likes with 17	app.ts:130

We can see that the iterable differs identified that we added a new object to the list {author: "Hellen", comment: "Thanks!", likes: 17}.

We also got individual changes to the comment object logged, as detected by the key-value differ:

- 1 added author with Helen
- 2 added comment with Thanks!
- 3 added likes with 17

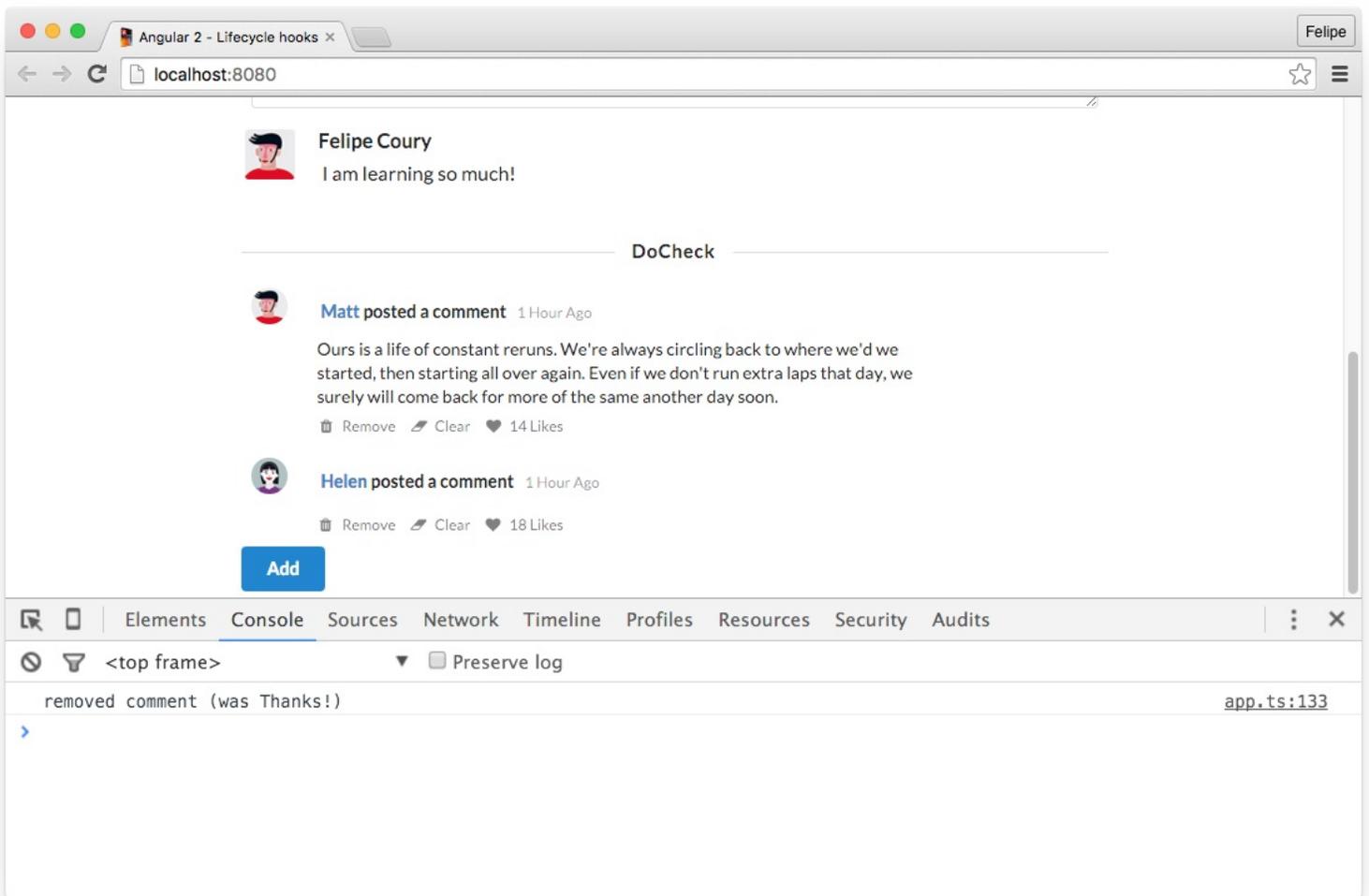
Now we can click the like button for this new comment:



Number of likes changed

And now only the like change was detected.

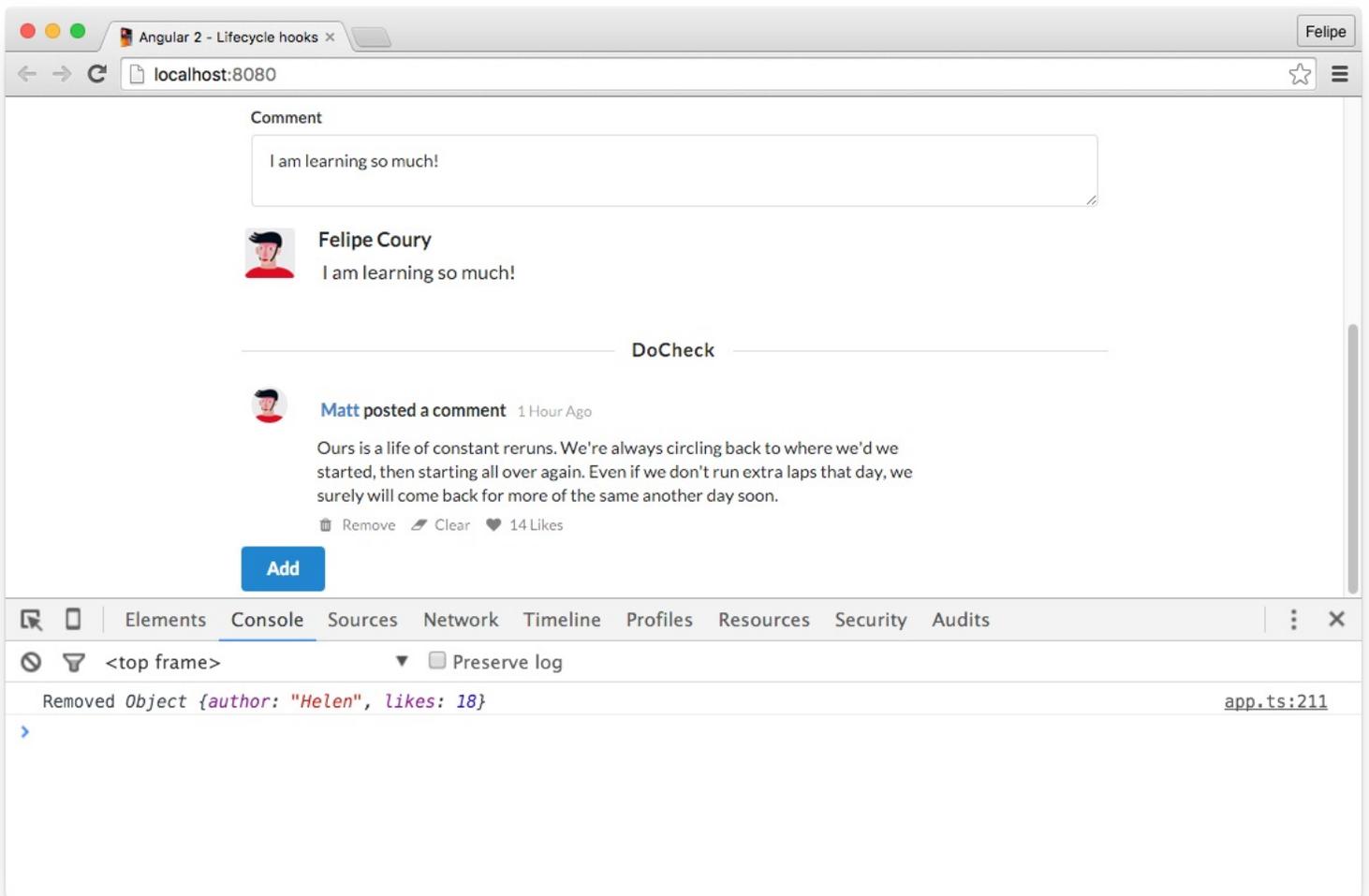
If we click the *Clear* icon, it will remove the comment key from the comment object:



Comment text cleared

And the log confirms that we removed that key.

Finally, let's remove the last comment, by clicking the *Remove* icon:



Comment removed

And as expected, we get a removed object log.

AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked

The `AfterContentInit` hook is called after `OnInit`, right after the initialization of the content of the component or directive has finished.

The `AfterContentChecked` works similarly, but it's called after the directive check has finished. The check, in this context, is the change detection system check.

The other two hooks: `AfterViewInit` and `AfterViewChecked` are triggered right after the content ones above, right after the view has been fully initialized. Those two hooks are only applicable to components, and not to directives.

Also, the `AfterXXXInit` hooks are only called once during the directive lifecycle, while the `AfterXXXChecked` hooks are called after every change detection cycle.

To better understand this, let's write another component that logs to the console during each lifecycle hook. It will also have a counter that we can increment by clicking a button:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```

216 @Component({
217   selector: 'afters',
218   template: `
219     <div class="ui label">
220       <i class="list icon"></i> Counter: {{ counter }}
221     </div>
222
223     <button class="ui primary button" (click)="inc()">
224       Increment
225     </button>
226   `
227 })
228 class AftersCmp implements OnInit, OnDestroy, DoCheck,
229   OnChanges, AfterContentInit,
230   AfterContentChecked, AfterViewInit,
231   AfterViewChecked {
232   counter: number;
233
234   constructor() {
235     console.log('AfterCmd ----- [constructor]');
236     this.counter = 1;
237   }
238   inc() {
239     console.log('AfterCmd ----- [counter]');
240     this.counter += 1;
241   }
242   ngOnInit() {
243     console.log('AfterCmd - OnInit');
244   }
245   ngOnDestroy() {
246     console.log('AfterCmp - OnDestroy');
247   }
248   ngDoCheck() {
249     console.log('AfterCmp - DoCheck');
250   }
251   ngOnChanges() {
252     console.log('AfterCmp - OnChanges');
253   }
254   ngAfterContentInit() {
255     console.log('AfterCmp - AfterContentInit');
256   }
257   ngAfterContentChecked() {
258     console.log('AfterCmp - AfterContentChecked');
259   }
260   ngAfterViewInit() {
261     console.log('AfterCmp - AfterViewInit');
262   }
263   ngAfterViewChecked() {
264     console.log('AfterCmp - AfterViewChecked');
265   }
266 }

```

Now let's add it to the app component, along with a Toggle button, like the one we used for the OnDestroy hook:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```

311 <afters *ngIf="displayAfters"></afters>
312 <button class="ui primary button" (click)="toggleAfters()">
313   Toggle
314 </button>
315 `

```

The final implementation for the app component now should look like this:

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```

268 @Component({
269   selector: 'lifecycle-sample-app',
270   directives: [OnInitCmp, OnChangeCmp, DoCheckCmp, AftersCmp],
271   template: `
272     <h4 class="ui horizontal divider header">
273       OnInit and OnDestroy
274     </h4>

```

```

275 <button class="ui primary button" (click)="toggle()">
276     Toggle
277 </button>
278 <on-init *ngIf="display"></on-init>
279
280
281 <h4 class="ui horizontal divider header">
282     OnChange
283 </h4>
284
285 <div class="ui form">
286     <div class="field">
287         <label>Name</label>
288         <input type="text" #namefld value="{{name}}"
289             (keyup)="setValues(namefld, commentfld)">
290     </div>
291
292     <div class="field">
293         <label>Comment</label>
294         <textarea (keyup)="setValues(namefld, commentfld)"
295             rows="2" #commentfld>{{comment}}</textarea>
296     </div>
297 </div>
298
299 <on-change [name]="name" [comment]="comment"></on-change>
300
301 <h4 class="ui horizontal divider header">
302     DoCheck
303 </h4>
304
305 <do-check></do-check>
306
307 <h4 class="ui horizontal divider header">
308     AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked
309 </h4>
310
311 <afters *ngIf="displayAfters"></afters>
312 <button class="ui primary button" (click)="toggleAfters()">
313     Toggle
314 </button>
315
316 })
317 export class LifecycleSampleApp4 {
318     display: boolean;
319     displayAfters: boolean;
320     name: string;
321     comment: string;
322
323     constructor() {
324         // OnInit and OnDestroy
325         this.display = true;
326
327         // OnChange
328         this.name = 'Felipe Coury';
329         this.comment = 'I am learning so much!';
330
331         // AfterXXX
332         this.displayAfters = true;
333     }
334
335     setValues(namefld, commentfld) {
336         this.name = namefld.value;
337         this.comment = commentfld.value;
338     }
339
340     toggle(): void {
341         this.display = !this.display;
342     }
343
344     toggleAfters(): void {
345         this.displayAfters = !this.displayAfters;
346     }
347 }

```

When the application starts, we can see each hook is logged:

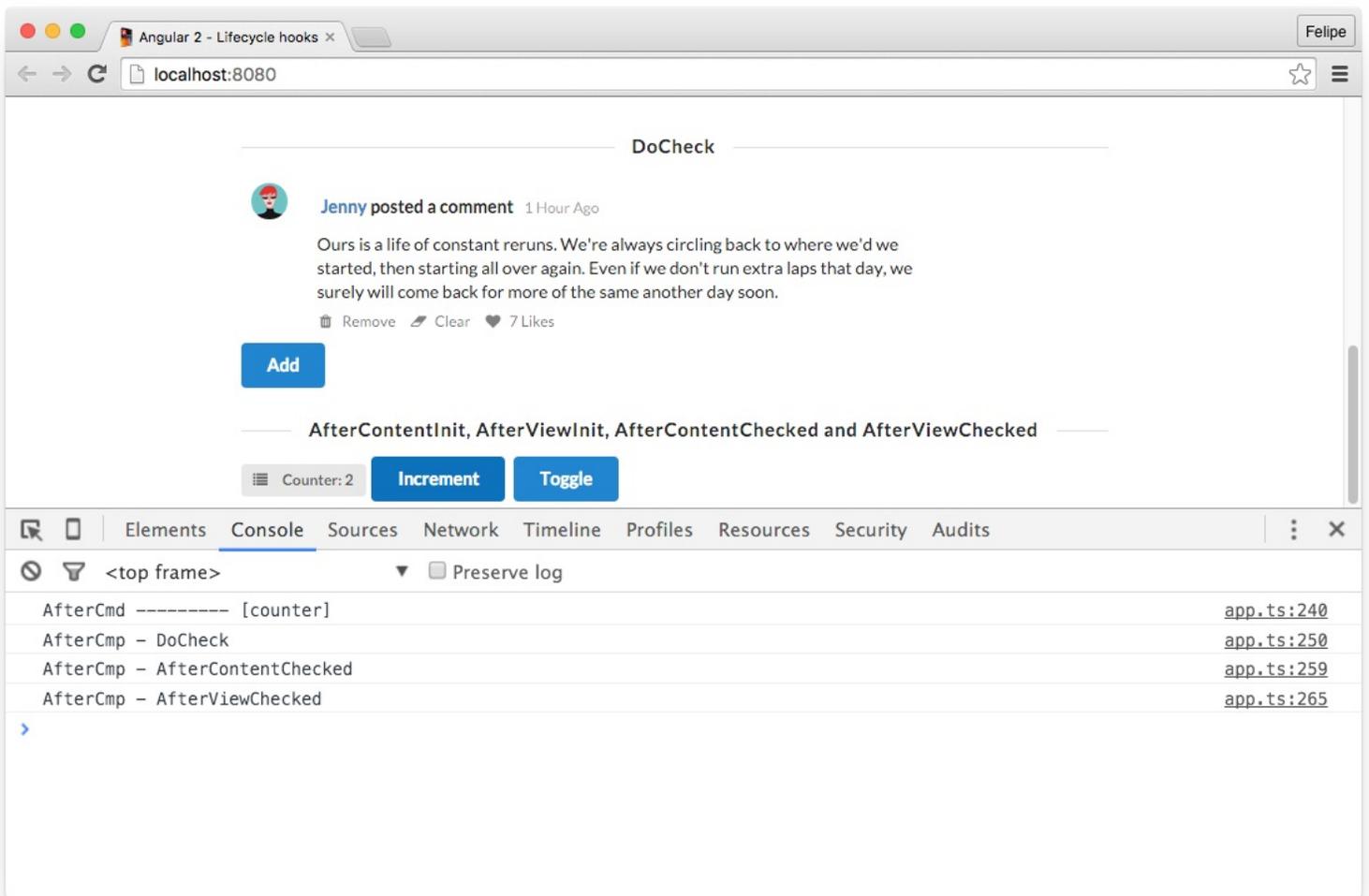
The screenshot shows a web browser window with the URL `localhost:8080`. The page content includes a header `DoCheck`, a comment by Jenny posted 1 hour ago with the text "Ours is a life of constant reruns. We're always circling back to where we'd we started, then starting all over again. Even if we don't run extra laps that day, we surely will come back for more of the same another day soon." and 7 likes. Below the comment is an `Add` button. Further down, there is a section titled `AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked` with a `Counter: 1` display and `Increment` and `Toggle` buttons.

The Chrome DevTools console is open, showing the following log entries:

Log Entry	File/Line
<code>AfterCmd ----- [constructor]</code>	<code>app.ts:236</code>
<code>On init</code>	<code>app.ts:31</code>
<code>AfterCmd - OnInit</code>	<code>app.ts:244</code>
<code>AfterCmp - DoCheck</code>	<code>app.ts:250</code>
<code>AfterCmp - AfterContentInit</code>	<code>app.ts:256</code>
<code>AfterCmp - AfterContentChecked</code>	<code>app.ts:259</code>
<code>AfterCmp - AfterViewInit</code>	<code>app.ts:262</code>
<code>AfterCmp - AfterViewChecked</code>	<code>app.ts:265</code>
<code>added author with Jenny</code>	<code>app.ts:130</code>
<code>added comment with Ours is a life of constant reruns. We're always circling back to where we'd we started,</code>	<code>app.ts:130</code>

App started

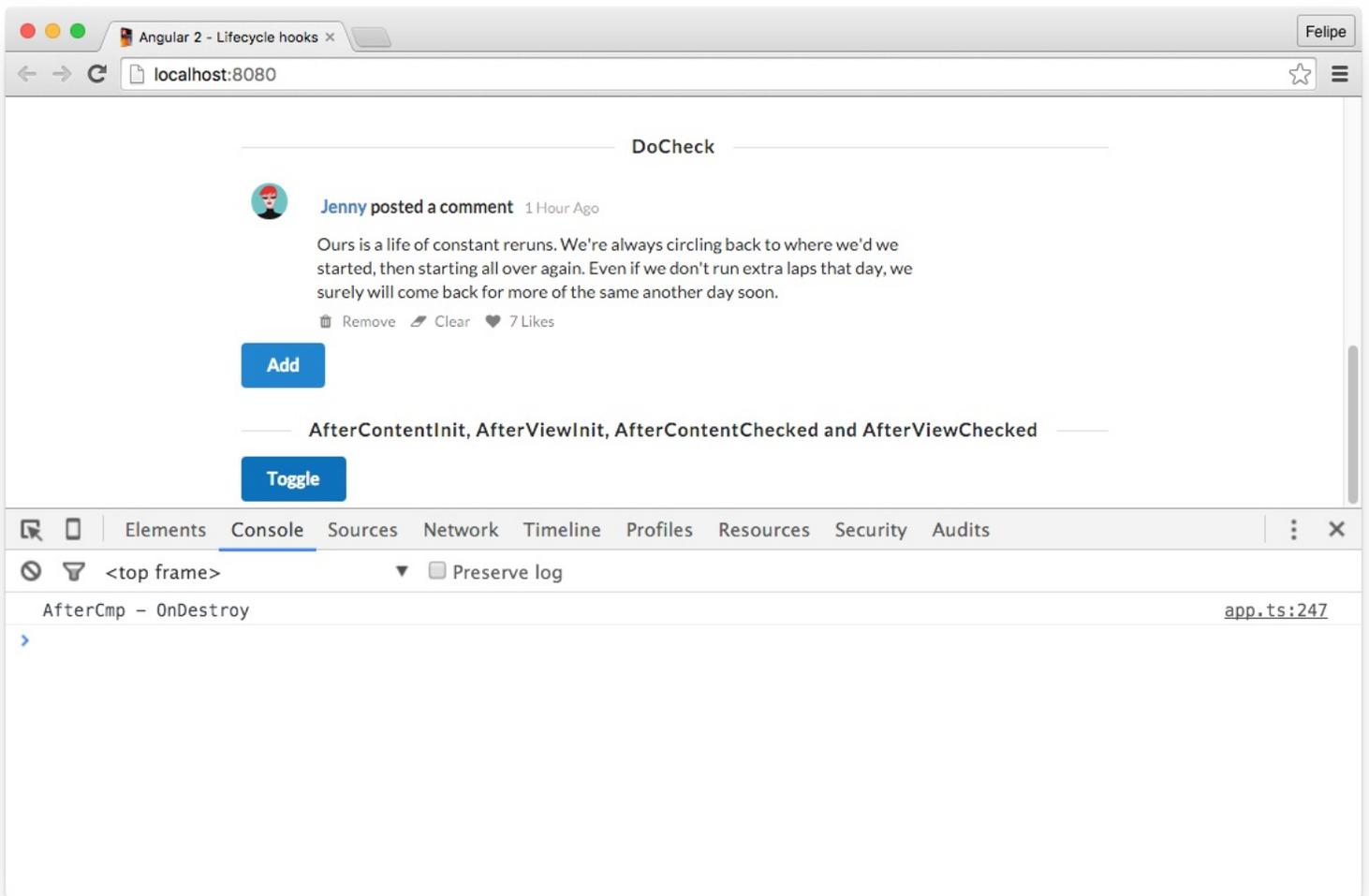
Now let's clear the console and click the Increment button:



After counter increment

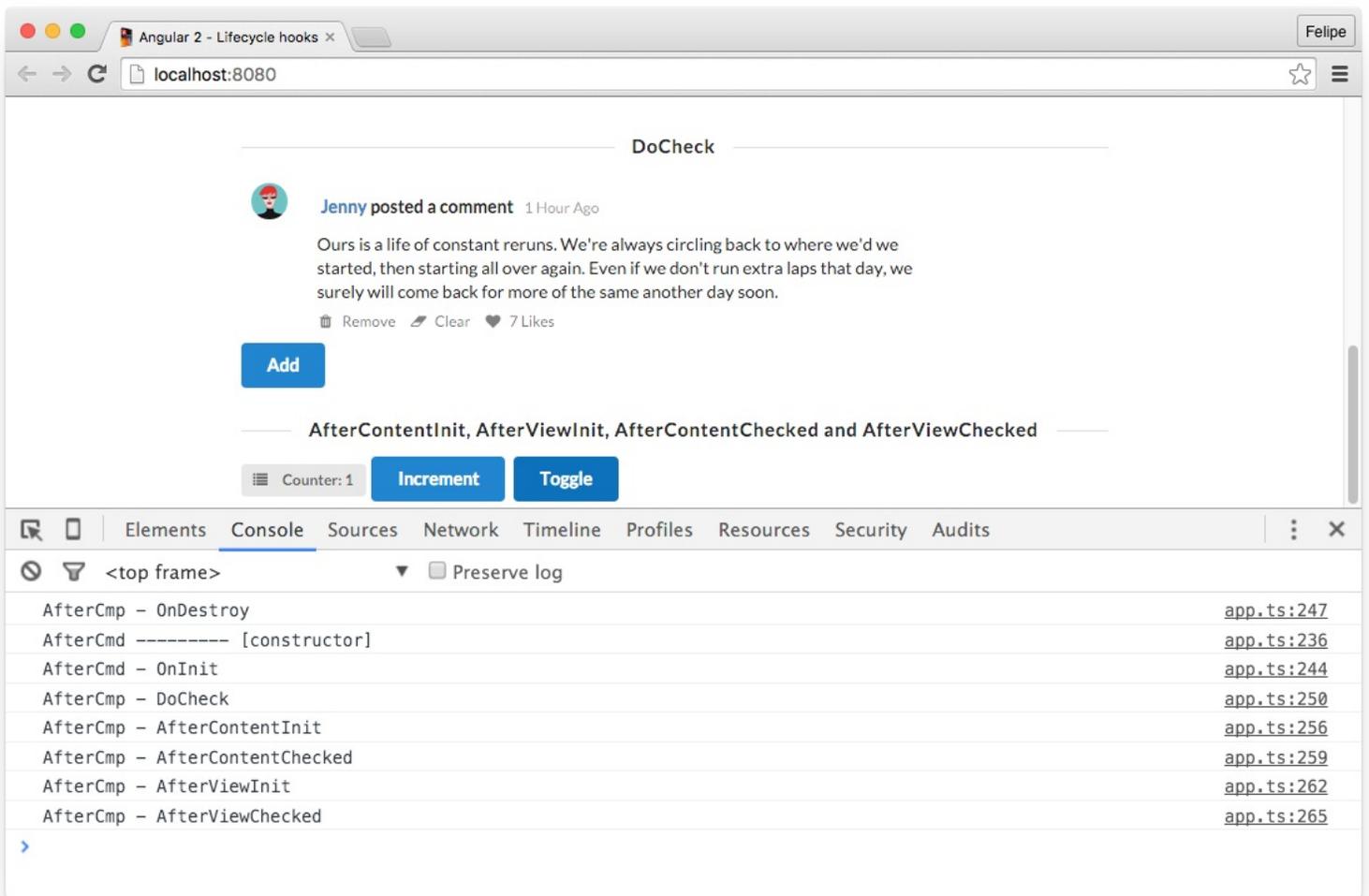
You can see that now only the DoCheck, AfterContentCheck and AfterViewCheck hooks were triggered.

Sure enough, if we click the Toggle button:



App started

And click it again:



App started

All the hooks are triggered.

Advanced Templates

Templates elements are special elements used to create views that can be dynamically manipulated.

In order to make working with templates simpler, Angular provides some syntactic sugar to create templates, so we often don't create them by hand.

For instance, when we write:

```

1 <do-check-item
2   *ngFor="let comment of comments"
3   [comment]="comment"
4   (onRemove)="removeComment($event)">
5 </do-check-item>

```

This gets converted into:

```

1 <do-check-item
2   template="ngFor let comment of comments; #i=index"
3   [comment]="comment"
4   (onRemove)="removeComment($event)">
5 </do-check-item>

```

Which then gets converted into:

```
1 <template
2   ngFor
3   [ngForOf]="comments"
4   let-comment="$implicit"
5   let-index="i">
6   <do-check-item
7     [comment]="comment"
8     (onRemove)="removeComment($event)">
9   </do-check-item>
10 </template>
```

It's important that we understand this underlying concept so we can build our own directives.

Rewriting ngIf - ngBookIf

Let's create a directive that does exactly what ngIf does. Let's call it ngBookIf.

ngBookIf @Directive

We start by declaring the @Directive annotation for our class:

```
1 @Directive({
2   selector: '[ngBookIf]',
3   inputs: ['ngBookIf']
4 })
```

We're using [ngBookIf] as the selector because, as we learned above, when we use *ngBookIf="condition", it will be converted to:

```
1 <template ngBookIf [ngBookIf]="condition">
```

Since ngBookIf is also an attribute we need to indicate that we're expecting to receive it as an input.

The behavior of this directive should be to add the directive template contents when the condition is true and remove it when it's false.

So when the condition is true, we will use a *view container*. The view container is used to attach one or more views to the directive.

We will use the view container to either:

- create a new view with our directive template embedded or
- clear the view container contents.

Before we do that, we need to inject the ViewContainerRef and the TemplateRef. They will be injected with the directive's view container and template.

Here's the code we'll need:

code/advanced_components/app/ts/templates/if.ts

```
19 constructor(private viewContainer: ViewContainerRef,
20              private template: TemplateRef<any>) {}
```

Now that we have references to both the view container and the template, we will use a TypeScript property setter construct:

code/advanced_components/app/ts/templates/if.ts

```
22 set ngBookIf(condition) {
23   if (condition) {
24     this.viewContainer.createEmbeddedView(this.template);
25   }
26   else {
27     this.viewContainer.clear();
28   }
29 }
```

This method will be called every time we set a value on the `ngBookIf` property of our class. That is, this method will be called anytime the `condition` in `ngBookIf="condition"` changes.

Now we use the view container's `createEmbeddedView` method to attach the directive's template if the condition is true, or the `clear` method to remove everything from the view container.

Using `ngBookIf`

In order to use our directive, we can write the following component:

code/advanced_components/app/ts/templates/if.ts

```
32 @Component({
33   selector: 'template-sample-app',
34   directives: [NgBookIf],
35   template: `
36     <button class="ui primary button" (click)="toggle()">
37       Toggle
38     </button>
39
40     <div *ngBookIf="display">
41       The message is displayed
42     </div>
43 `
44 })
45 export class IfTemplateSampleApp {
46   display: boolean;
47
48   constructor() {
49     this.display = true;
50   }
51
52   toggle() {
53     this.display = !this.display;
54   }
55 }
56 }
```

When we run the application, we can see that the directive works as expected: when we click the **Toggle** button the message *This message is displayed* is toggled on and off the page.

Rewriting `ngFor` - `ngBookRepeat`

Now let's write a simplified version of the `ngFor` directive, that Angular provides to handle repetition of templates for a given collection.

`ngBookRepeat` template deconstruction

This directive will be used with the `*ngBookRepeat="let var of collection"` notation.

Like we did for the previous directive, we need to declare the selector as being `[ngBookRepeat]`. However the input parameter in this case won't be `ngBookRepeat` only.

If we look back at how Angular converts the `*something="let var in collection"` notation, we can see that the final form of the element is the equivalent of:

```
1 <template something [somethingOf]="collection" let-var="$implicit">
2   <!-- ... -->
3 </template>
```

As we can see, the attribute that's being passed isn't `something` but `somethingOf` instead. That's where our directive receives the collection we're iterating on.

For template that is generated, we're going to have a local view variable `#var`, that will receive the value from the `$implicit` local variable. That's the name of the local variable that Angular uses when "de-sugaring" the syntax into a template.

ngBookRepeat @Directive

Time to write the directive. First we have to write the directive annotation:

code/advanced_components/app/ts/templates/for.ts

```
14 @Directive({
15   selector: '[ngBookRepeat]',
16   inputs: ['ngBookRepeatOf']
17 })
```

ngBookRepeat class

Then we start writing the class:

code/advanced_components/app/ts/templates/for.ts

```
18 class NgBookRepeat implements DoCheck {
19   private items: any;
20   private differ: IterableDiffer;
21   private views: Map<any, ViewRef> = new Map<any, ViewRef>();
22
23
24   constructor(private viewContainer: ViewContainerRef,
25               private template: TemplateRef<any>,
26               private changeDetector: ChangeDetectorRef,
27               private differs: IterableDiffers) {}
```

We are declaring some properties for our class:

- `items` holds the collection we're iterating on
- `differ` is an `IterableDiffer` (which we learned about in the [Lifecycle Hooks section above](#)) that will be used for change detection purposes
- `views` is a `Map` that will link a given item on the collection with the view that contains it

The constructor will receive the `viewContainer`, the `template` and an `IterableDiffers` instance (we discussed each of these things earlier in this chapter above).

Now, the next thing that's being injected is a change detector. We will have a deep dive of change detection on the next section. For now, let's say that this is the class that Angular creates to trigger the detection when properties of our directive changes.

The next step is to write code that will trigger when we set the `ngBookRepeatOf` input:

```
code/advanced_components/app/ts/templates/for.ts
```

```
28 set ngBookRepeatOf(items) {
29   this.items = items;
30   if (this.items && !this.differ) {
31     this.differ = this.differs.find(items).create(this.changeDetector);
32   }
}
```

When we set this attribute, we're keeping the collection on the directive's `item` property and if the collection is valid and we don't have a differ yet, we create one.

To do that, we're creating an instance of `IterableDiffer` that reuses the directive's change detector (the one we injected on the constructor).

Now it's time to write the code that will react to a change on the collection. For this, we're going to use the **DoCheck** lifecycle hook by implementing the `ngDoCheck` method as follows:

```
code/advanced_components/app/ts/templates/for.ts
```

```
35 ngDoCheck(): void {
36   if (this.differ) {
37     let changes = this.differ.diff(this.items);
38     if (changes) {
39
40       changes.forEachAddedItem((change) => {
41         let view = this.viewContainer.createEmbeddedView(this.template,
42           {'$implicit': change.item});
43         this.views.set(change.item, view);
44       });
45       changes.forEachRemovedItem((change) => {
46         let view = this.views.get(change.item);
47         let idx = this.viewContainer.indexOf(view);
48         this.viewContainer.remove(idx);
49         this.views.delete(change.item);
50       });
51     }
}
```

Let's break this down a bit. First thing we do in this method is make sure we already instantiated the differ. If not, we do nothing.

Next, we ask the differ what changed. If there are changes, we first iterate through the times that were added using `changes.forEachAddedItem`. This method will receive a `CollectionChangeRecord` object for every element that was added.

Then for each element, we create a new embedded view using the view container's `createEmbeddedView` method.

```
1 let view = this.viewContainer.createEmbeddedView(this.template, {'$implicit': ch\
2 ange.item});
```

The second argument to `createEmbeddedView` is the *view context*. In this case, we're setting the `$implicit` local variable to `change.item`. This will allow we to reference the variable we declared back on the `*ngBookRepeat="let var of collection"` as `var` on that view. That is, the `var` in `let var` is the `$implicit` variable. We use `$implicit` because we don't know what name the user will assign to it when we're writing this component.

The final thing we need to do is to correlate the item of the collection to its view. The reason behind this is that, if an item gets removed from the collection, we need to get rid of the correct view, as we do next.

Now for each item that was removed from the collection, we use the item to view map we keep to find the view. Then we ask the view container for the index of that view. We need that because the view container's remove method needs an index. Finally, we also clean up the view from the item to view map.

Trying out our directive

To test our new directive, let's write the following component:

code/advanced_components/app/ts/templates/for.ts

```
57 @Component({
58   selector: 'template-sample-app',
59   directives: [NgBookRepeat],
60   template: `
61     <ul>
62       <li *ngBookRepeat="let p of people">
63         {{ p.name }} is {{ p.age }}
64         <a href (click)="remove(p)">Remove</a>
65       </li>
66     </ul>
67
68     <div class="ui form">
69       <div class="fields">
70         <div class="field">
71           <label>Name</label>
72           <input type="text" #name placeholder="Name">
73         </div>
74         <div class="field">
75           <label>Age</label>
76           <input type="text" #age placeholder="Age">
77         </div>
78       </div>
79     </div>
80     <div class="ui submit button"
81       (click)="add(name, age)">
82       Add
83     </div>
84 `
85 })
86 export class ForTemplateSampleApp {
87   people: any[];
88
89   constructor() {
90     this.people = [
91       {name: 'Joe', age: 10},
92       {name: 'Patrick', age: 21},
93       {name: 'Melissa', age: 12},
94       {name: 'Kate', age: 19}
95     ];
96   }
97
98   remove(p) {
99     let idx: number = this.people.indexOf(p);
100     this.people.splice(idx, 1);
101     return false;
102   }
103
104   add(name, age) {
105     this.people.push({name: name.value, age: age.value});
106     name.value = '';
107     age.value = '';
108   }
109 }
```

We're using our directive to iterate through a list of people:

code/advanced_components/app/ts/templates/for.ts

```
61 <ul>
62   <li *ngBookRepeat="let p of people">
63     {{ p.name }} is {{ p.age }}
64     <a href (click)="remove(p)">Remove</a>
65   </li>
66 </ul>
```

When we click **Remove** we remove the item from the collection, triggering the change detection.

We also provide a form that allows adding items to the collection:

code/advanced_components/app/ts/templates/for.ts

```
68 <div class="ui form">
69   <div class="fields">
70     <div class="field">
71       <label>Name</label>
72       <input type="text" #name placeholder="Name">
73     </div>
74     <div class="field">
75       <label>Age</label>
76       <input type="text" #age placeholder="Age">
77     </div>
78   </div>
79 </div>
80 <div class="ui submit button"
81   (click)="add(name, age)">
82   Add
83 </div>
```

Change Detection

As a user interacts with our app, data (state) changes and our app needs to respond accordingly.

One of the big problems any modern JavaScript framework needs to solve is how to figure out when changes have happened and re-render components accordingly.

In order to make the view react to changes on components state, Angular uses *change detection*.

What are the things that can trigger changes in a component's state? The most obvious thing is user interaction. For instance, if we have a component:

```
1 @Component({
2   selector: 'my-component',
3   template: `
4     Name: {{name}}
5     <button (click)="changeName()">Change!</button>
6   `
7 })
8 class MyComponent {
9   name: string;
10  constructor() {
11    this.name = 'Felipe';
12  }
13
14  changeName() {
15    this.name = 'Nate';
16  }
17 }
```

We can see that when the user *clicks* on the **Change!** button, the component's *name* property will change.

Another source of change could be, for instance, a HTTP request:

```

1 @Component({
2   selector: 'my-component',
3   template: `
4     Name: {{name}}
5   `
6 })
7 class MyComponent {
8   name: string;
9   constructor(private http: Http) {
10    this.http.get('/names/1')
11      .map(res => res.json())
12      .subscribe(data => this.name = data.name);
13  }
14 }

```

And finally, we could have a timer that would trigger the change:

```

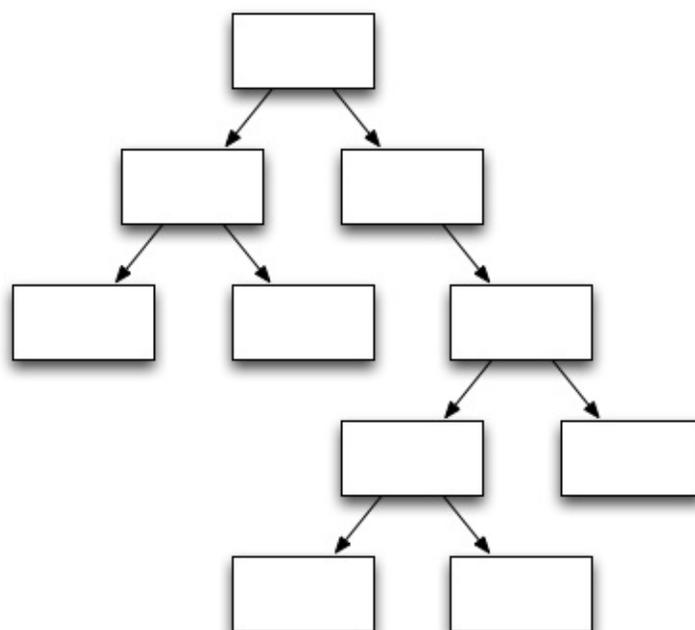
1 @Component({
2   selector: 'my-component',
3   template: `
4     Name: {{name}}
5   `
6 })
7 class MyComponent {
8   name: string;
9   constructor() {
10    setTimeout(() => this.name = 'Felipe', 2000);
11  }
12 }

```

But how does Angular become aware of these changes?

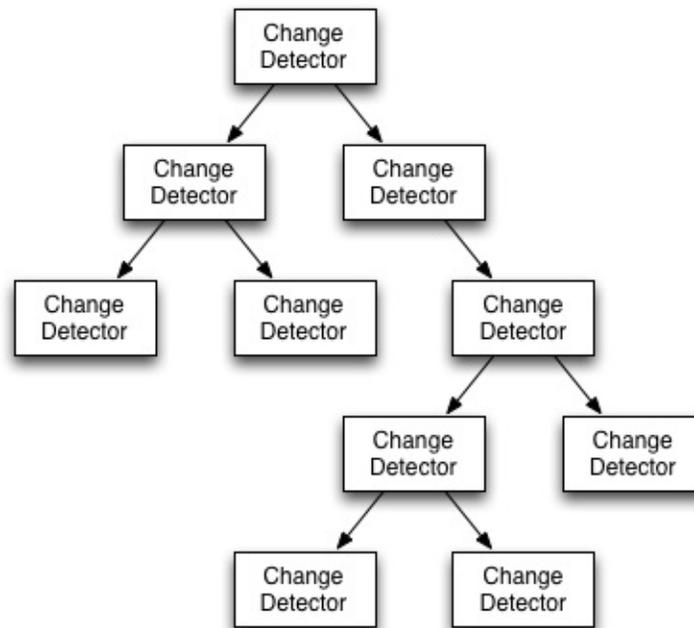
The first thing to know is that each component gets a change detector.

Like we've seen before, a typical application will have a number of components that will interact with each other, creating a dependency tree like below:



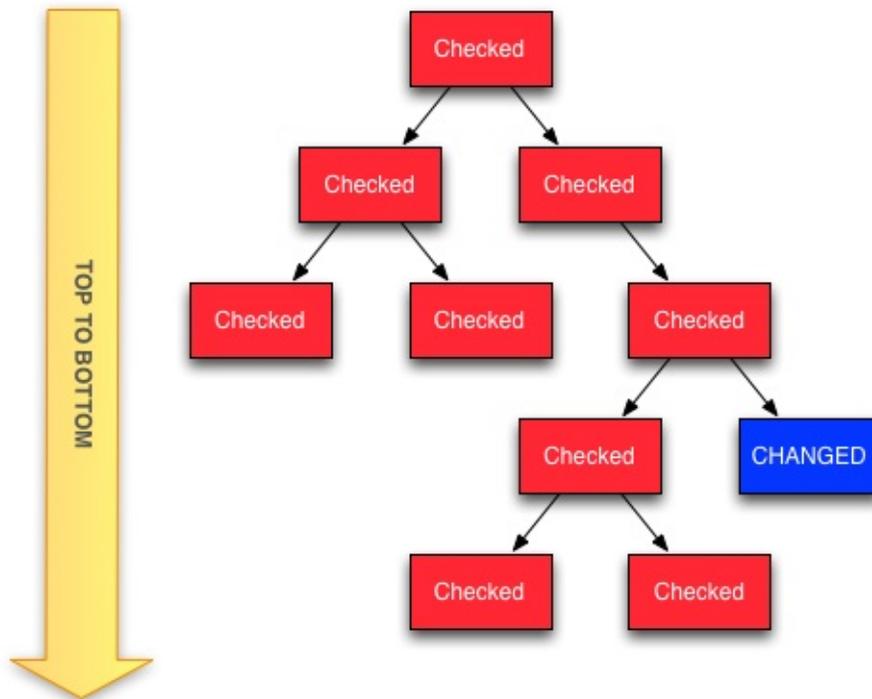
Component tree

For each component on our tree, a change detector is created and so we end up with a tree of change detectors:



Change detector tree

When one of the the components change, no matter where in the tree it is, a change detection pass is triggered for the whole tree. This happens because Angular scans for changes from the top component node, all the way to the bottom leaves of the tree.



Default change detection

In our diagram above, the component in blue changed, but as we can see, it triggered checks for the whole component tree. Objects that were checked are indicated in red (note that the component itself was also checked).

It is natural to think that this check may be a very expensive operation. However, due to a number of optimizations (that make Angular code eligible for further optimization by the JavaScript engine), it's actually surprisingly fast.

Customizing Change Detection

There are times that the built-in or default change detection mechanism may be overkill. One example is if you're using immutable objects or if your application architecture relies on observables. In these cases, Angular provides mechanisms for configuring the change detection system such that you'll get very fast performance.

The first way to change the change detector behavior is by telling a component that it only should be checked if one of its *input values* change.

To recap, an input value is the attributes your component receive from the outside world. For instance, on this code:

```
1 class Person {
2   constructor(public name: string, public age: string) {}
3 }
4
5 @Component({
6   selector: 'mycomp',
7   inputs: ['person'],
8   template: `
9     <div>
10      <span class="name">{person.name}</span>
11      is {person.age} years old.
12    </div>
13 `
14 })
15 class MyComp {
16 }
```

We have person as an input attribute. Now, if we want to make this component change only when its input attribute changes, we just need to change the change detection strategy, by setting its `changeDetection` attribute to `ChangeDetectionStrategy.OnPush`.

 By the way, if you're curious, the default value for `changeDetection` is `ChangeDetectionStrategy.Default`.

Let's write a small experiment with two components. The first one will use the default change detection behavior and the other will use the `OnPush` strategy:

[code/advanced_components/app/ts/change-detection/onpush.ts](#)

```
1 import { Component,
2   Directive,
3   Input,
4   ChangeDetectorRef,
5   ChangeDetectionStrategy,
6 } from '@angular/core';
7
8
9 class Profile {
10  constructor(private first: string, private last: string) {}
11
12  lastChanged() {
13    return new Date();
14  }
15 }
```

So we start with some imports and we declare a Person class that will be used as the input of both our components. Notice that we also created a method called `lastChange()` to the Person class. This will be very useful to determine when the change detection is triggered. When a given component is marked as needing to be checked, this method will be called, since it's present on the template. So this method will reliably indicate the last time the component was checked for changes.

Next, we declare the `DefaultCmp` that will use the default change detection strategy:

code/advanced_components/app/ts/change-detection/onpush.ts

```
17 @Component({
18   selector: 'default',
19   inputs: ['profile'],
20   template: `
21 <h4 class="ui horizontal divider header">
22   Default Strategy
23 </h4>
24
25 <form class="ui form">
26   <div class="field">
27     <label>First Name</label>
28     <input
29       type="text"
30       [(ngModel)]="profile.first"
31       placeholder="First Name">
32   </div>
33   <div class="field">
34     <label>Last Name</label>
35     <input
36       type="text"
37       [(ngModel)]="profile.last"
38       placeholder="Last Name">
39   </div>
40 </form>
41 <div>
42   {{profile.lastChanged() | date:'medium'}}
43 </div>
44 `
45 })
46 class DefaultCmp {
47   @Input() profile: Profile;
48 }
```

And a second component using `OnPush` strategy:

code/advanced_components/app/ts/change-detection/onpush.ts

```
50 @Component({
51   selector: 'on-push',
52   inputs: ['profile'],
53   changeDetection: ChangeDetectionStrategy.OnPush,
54   template: `
55 <h4 class="ui horizontal divider header">
56   OnPush Strategy
57 </h4>
58
59 <form class="ui form">
60   <div class="field">
61     <label>First Name</label>
62     <input
63       type="text"
64       [(ngModel)]="profile.first"
65       placeholder="First Name">
66   </div>
67   <div class="field">
68     <label>Last Name</label>
69     <input
70       type="text"
71       [(ngModel)]="profile.last"
72       placeholder="Last Name">
73   </div>
74 </form>

```

```
75 <div>
76   {{profile.lastChanged() | date:'medium'}}
77 </div>
78 `
79 })
80 class OnPushCmp {
81   @Input() profile: Profile;
82 }
```

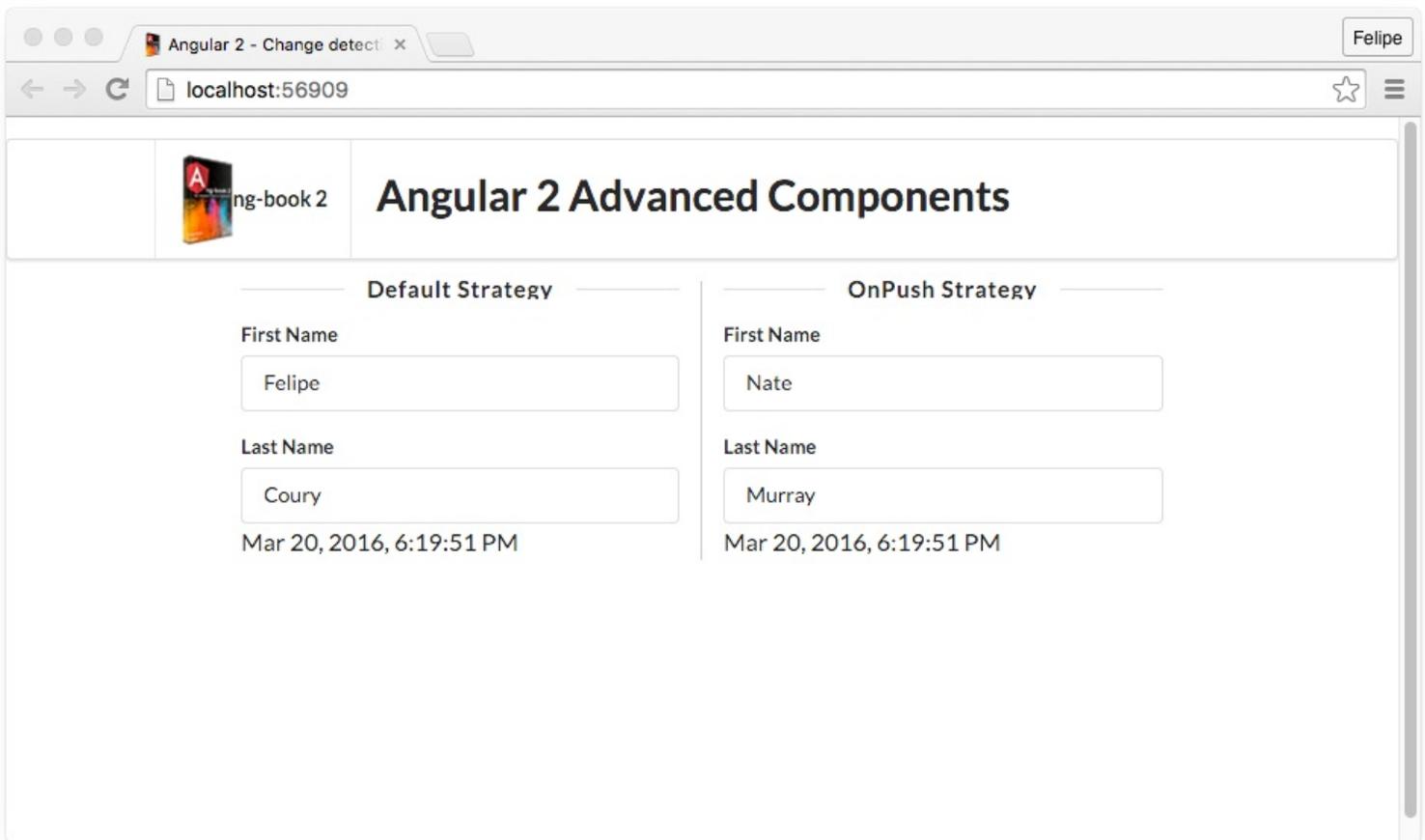
As we can see, both component use the same template. The only thing that is different is the header.

Finally, let's add the component that will render both components side by side:

code/advanced_components/app/ts/change-detection/onpush.ts

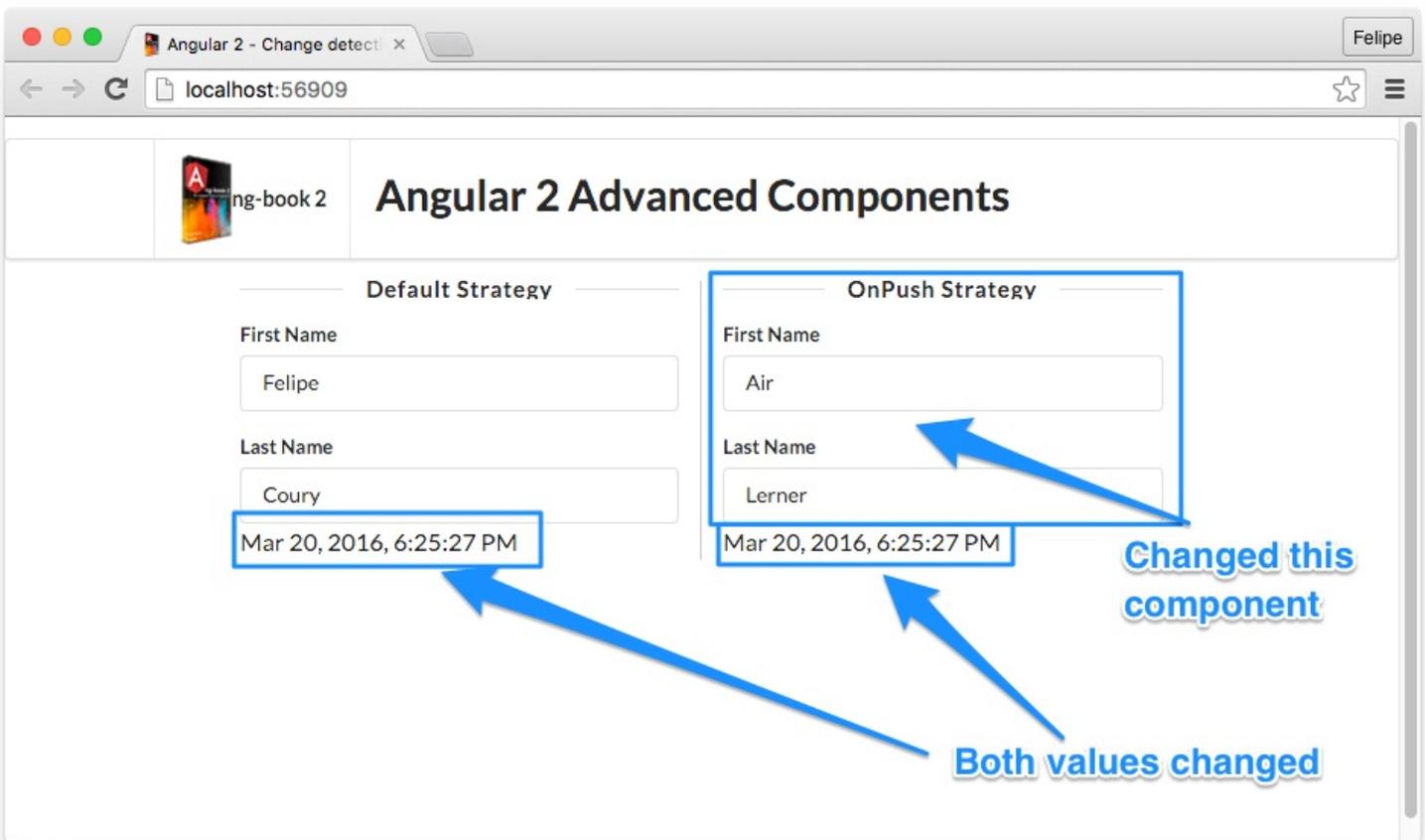
```
84 @Component({
85   selector: 'change-detection-sample-app',
86   directives: [DefaultCmp, OnPushCmp],
87   template: `
88     <div class="ui page grid">
89       <div class="two column row">
90         <div class="column area">
91           <default [profile]="profile1"></default>
92         </div>
93         <div class="column area">
94           <on-push [profile]="profile2"></on-push>
95         </div>
96       </div>
97     </div>
98   `
99 })
100 export class OnPushChangeDetectionSampleApp {
101   profile1: Profile = new Profile('Felipe', 'Coury');
102   profile2: Profile = new Profile('Nate', 'Murray');
103 }
```

When we run this application, we should see both components rendered like below:



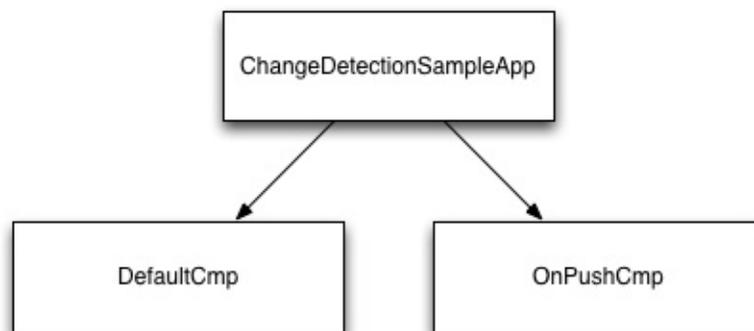
Default vs. OnPush strategies

When we change something on the component on the left, with the default strategy, we notice that the timestamp for the component on the right doesn't change:



OnPush changed, default got checked

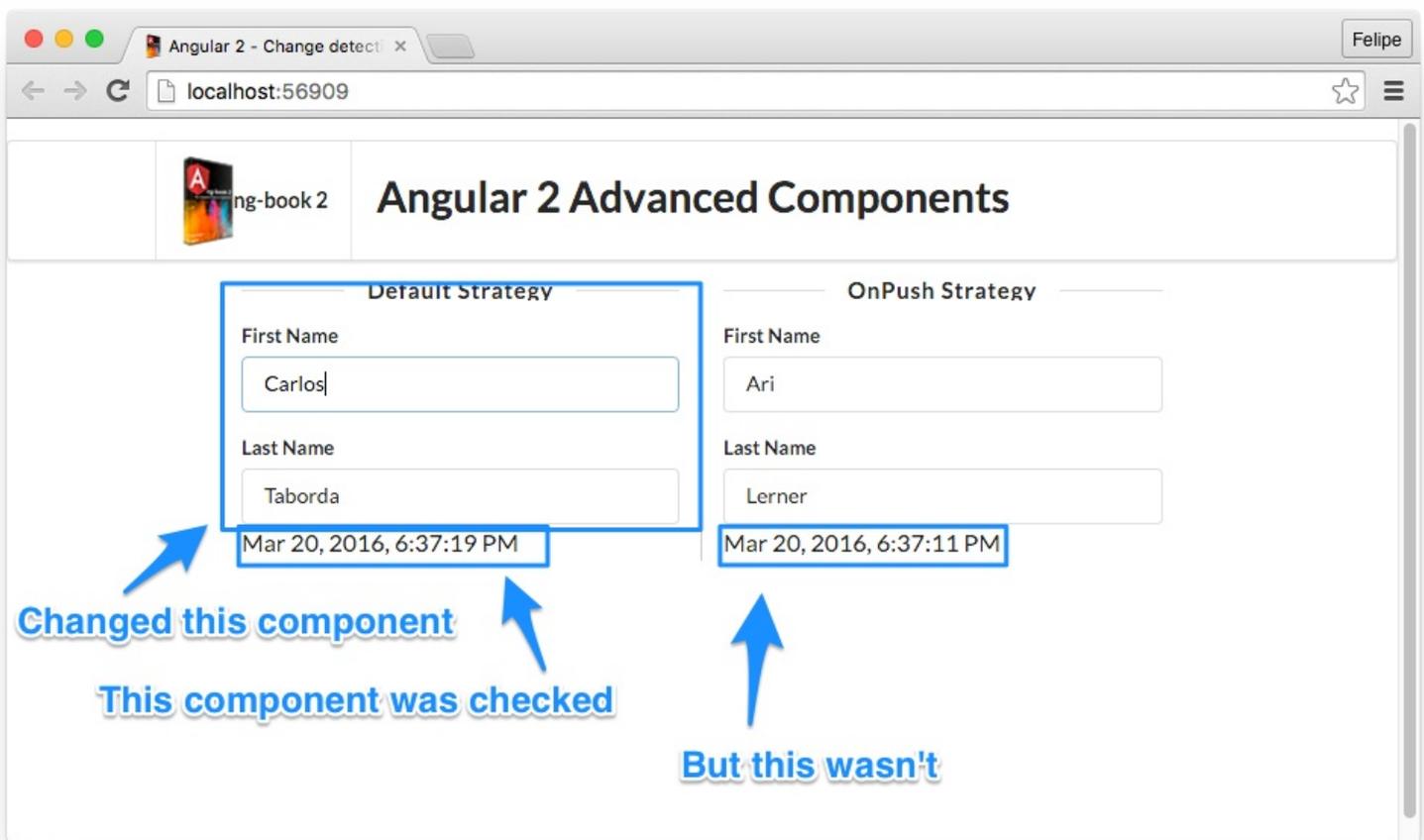
To understand why this happened, let's check this new tree of components:



Tree of components

Angular checks for changes from the top to the bottom, so it queried first `ChangeDetectionSampleApp`, then `DefaultCmp` and finally `OnPushCmp`. When it inferred that `OnPushCmp` changed, it updates all the components of the tree, from top to bottom, making the `DefaultCmp` to be rendered again.

Now when we change the value of the component on the right:



Default changed, OnPush didn't get checked

So now the change detection engine kicked in, the `DefaultCmp` component was checked but `OnPushCmp` wasn't. This happens because when we set the `OnPush` strategy for this component, it made the change detection to kick in for this component *only* when one of its input attributes change. Changing other components of the tree doesn't trigger this component's change detector.

Zones

Under the hood, Angular uses a library called Zones to automatically detect things changed and trigger the change detection mechanism. Zones will automatically tell Angular that something changed under the most common scenarios:

- when a DOM Event occurs (like *click*, *change*, etc.)
- when an HTTP request is resolved
- when a Timer is trigger (*setTimeout* or *setInterval*)

However, there are scenarios where Zones won't be able to automatically identify that something changed. That's another scenario where the **OnPush** strategy can be very useful.

A few examples of things that may fall off the Zones control, would be:

- using a third party library that runs asynchronously
- immutable data
- Observables

This is a perfect fit to using **OnPush** along with a technique to manually hint Angular that something changed.

Observables and OnPush

Let's write a component that receives an **Observable** as a parameter. Every time we receive a value from this observable, we will increment a counter that is a property of the component.

If we were using the regular change detection strategy, any time we incremented the counter, we would get change detection triggered by Angular. However, we will declare this component to use the **OnPush** strategy and, instead of letting the change detector kick in for each increment, we'll only kick it when the number is a multiple of 5 or when the observable completes.

In order to do that, let's write our component:

code/advanced_components/app/ts/change-detection/observables.ts

```
1 import {
2   Component,
3   Input,
4   ChangeDetectorRef,
5   ChangeDetectionStrategy
6 } from '@angular/core';
7
8 import { Observable } from 'rxjs/Rx';
9
10 @Component({
11   selector: 'observable',
12   inputs: ['items'],
13   changeDetection: ChangeDetectionStrategy.OnPush,
14   template: `
15     <div>
16       <div>Total items: {{counter}}</div>
17     </div>
18   `
19 })
20 class ObservableCmp {
21   @Input() items: Observable<number>;
22   counter = 0;
23
24   constructor(private changeDetector: ChangeDetectorRef) {
25   }
26
27   ngOnInit() {
28     this.items.subscribe((v) => {
29       console.log('got value', v);
30       this.counter++;
31       if (this.counter % 5 == 0) {
32         this.changeDetector.markForCheck();
33       }
34     }, null, () => {
35       this.changeDetector.markForCheck();
36     });
37   }
}
```

Let's break down the code a bit so we can make sure we understand. First, we're declaring the component to take `items` as the input attribute and using the `OnPush` detection strategy:

code/advanced_components/app/ts/change-detection/observables.ts

```
10 @Component({
11   selector: 'observable',
12   inputs: ['items'],
13   changeDetection: ChangeDetectionStrategy.OnPush,
14   template: `
15     <div>
16       <div>Total items: {{counter}}</div>
17     </div>

```

```
18 `
19 `})
```

Next, we're storing our input attribute on the `items` property of the component class, and setting another property, called `counter` to `0`.

```
code/advanced_components/app/ts/change-detection/observables.ts
```

```
20 class ObservableCmp {
21   @Input() items: Observable<number>;
22   counter = 0;
```

We then use the constructor to get a hold of the component's change detector:

```
code/advanced_components/app/ts/change-detection/observables.ts
```

```
24 constructor(private changeDetector: ChangeDetectorRef) {
25 }
```

Then, during the component initialization, on the `ngOnInit` hook:

```
code/advanced_components/app/ts/change-detection/observables.ts
```

```
27 ngOnInit() {
28   this.items.subscribe((v) => {
29     console.log('got value', v);
30     this.counter++;
31     if (this.counter % 5 == 0) {
32       this.changeDetector.markForCheck();
33     }
34   }, null, () => {
35     this.changeDetector.markForCheck();
36   });
37 }
```

We're subscribing to the Observable. The `subscribe` method takes three callbacks: **onNext**, **onError** and **onCompleted**.

Our `onNext` callback will print out the value we got, then increment the counter. Finally, if the current counter value is a multiple of 5, we call the change detector's `markForCheck` method. That's the method we use whenever we want to tell Angular that a change was made, so the change detector should kick in.

Then for the **onError** callback, we're using `null`, indicating we don't want to handle this scenario.

Finally, for the **onComplete** callback, we're also triggering the change detector, so the final counter can be displayed.

Now, on to the application component code, that will create the subscriber:

```
code/advanced_components/app/ts/change-detection/observables.ts
```

```
40 @Component({
41   selector: 'change-detection-sample-app',
42   directives: [ObservableCmp],
43   template: `
44     <observable [items]="itemObservable"></observable>
45 `
46 })
47 export class ObservableChangeDetectionSampleApp {
48   itemObservable: Observable<number>;
49
50   constructor() {
51     this.itemObservable = Observable.timer(100, 100).take(101);
```

```
52 }  
53 }
```

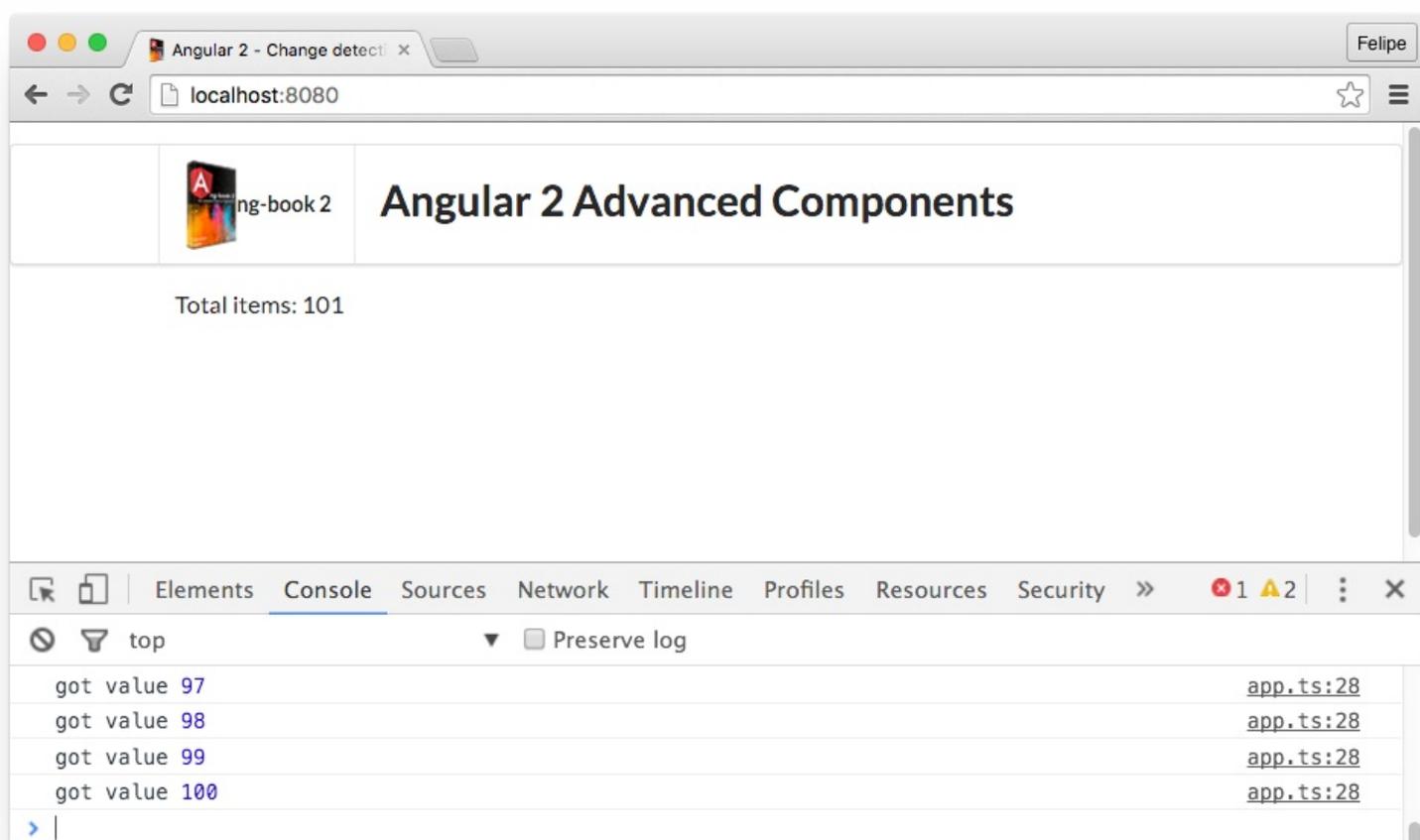
The important line here is the following:

```
1 this.itemObservable = Observable.timer(100, 100).take(101);
```

This line creates the Observable we're passing to the component on the `items` input attribute. We're passing two parameters to the `timer` method: the first is the number of milliseconds to wait before producing the first value and the second is the milliseconds to wait between values. So this observable will generate sequential values every 100 values forever.

Since we don't want the observable to run forever, we use the `take` method, to take only the first 101 values.

When we run this code, we'll see that the counter will only be updated for each 5 values obtained from the observer and also when the observable completes, generating a final value of 101:



Manually triggering change detection

Summary

Angular 2 provides us with many tools we can use for writing advanced components. Using the techniques in this chapter you will be able to write nearly any component functionality you wish.

However there's one important concept that you'll use in many advanced components that we haven't talked about yet: Dependency Injection.

With dependency injection we can hook our components into many other parts of the system. In the next chapter we'll talk about what DI is, how you can use it in your apps, and common patterns for injecting services.

Converting an Angular 1 App to Angular 2

If you've been using Angular for a while, then you probably already have production Angular 1 apps. Angular 2 is great, but there's no way we can drop everything and rewrite our entire production apps in Angular 2. What we need is a way to *incrementally* upgrade our Angular 1 app. Thankfully, Angular 2 has a fantastic way to do that.

The interoperability of Angular 1 (ng1) and Angular 2 (ng2) works really well. In this chapter, we're going to talk about how to upgrade your ng1 app to ng2 by writing a *hybrid* app. A hybrid app is running ng1 and ng2 simultaneously (and we can exchange data between them).

Peripheral Concepts

When we talk about interoperability between Angular 1 and Angular 2, there's a lot of peripheral concepts. For instance:

Mapping Angular 1 Concepts to Angular 2: At a high level, ng2 Components are ng1 directives. We also use Services in both. However, this chapter is about using both ng1 and ng2, so we're going to assume you have basic knowledge of both. If you haven't used ng2 much, checkout the chapter on [How Angular Works](#) before reading this chapter.

Preparing ng1 apps for ng2: Angular 1.5 provides a new `.component` method to make "component-directives". `.component` is a great way to start preparing your ng1 app for ng2. Furthermore, creating thin controllers (or [banning them altogether](#)) is a great way to refactor your ng1 app such that it's easier to integrate with ng2.

Another way to prepare your ng1 app is to reduce or eliminate your use of two-way data-binding in favor of a one-way data flow. In-part, you'd do this by reducing `$scope` changes that pass data between directives and instead use services to pass your data around.

These ideas are important and warrant further exploration. However, we're not going to extensively cover best-practices for pre-upgrade refactoring in this chapter.

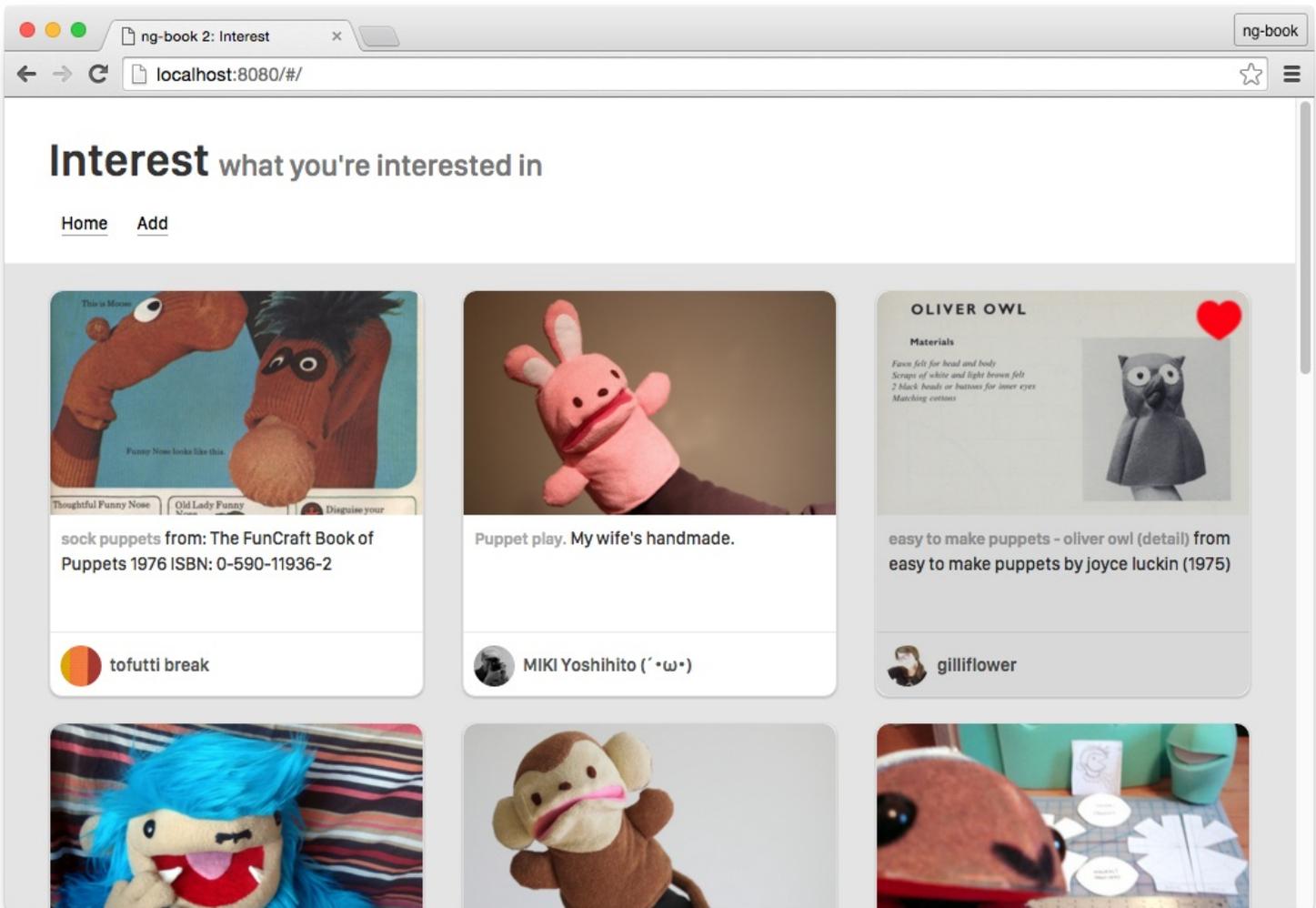
Instead, here's what we **are** going to talk about:

Writing hybrid ng1/ng2 apps: ng2 provides a way to bootstrap your ng1 app and then write ng2 components and services. You can write ng2 components that will mix with ng1 components and it "just works". Furthermore, the dependency injection system supports passing between ng1 and ng2 (both directions), so you can write services which will run in either ng1 or ng2.

The best part? Change detection runs within Zones, so you don't need to call `$scope.apply` or worry much about change-detection at all.

What We're Building

In this chapter, we're going to be converting an app called "Interest" - it's a Pinterest-like clone. The idea is that you can save a "Pin" which is a link with an image. The Pins will be shown in a list and you can "fav" (or unfav) a pin.



Our completed Pinterest-like app

 You can find the completed code for both the ng1 version and the completed hybrid version in the sample code download under code/conversion/ng1 and code/conversion/hybrid

Before we dive in, let's set the stage for interoperability between ng1 and ng2

Mapping Angular 1 to Angular 2

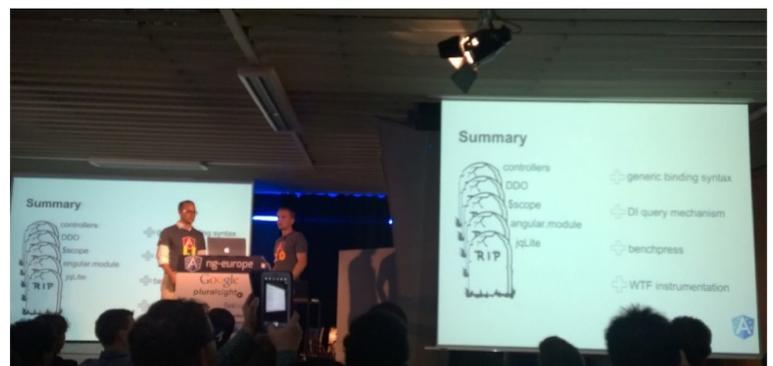
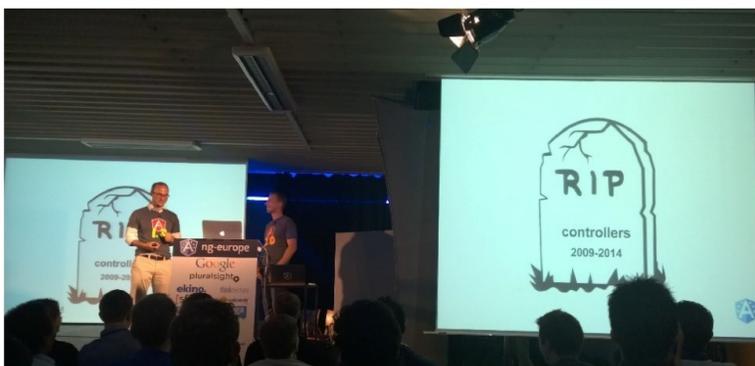
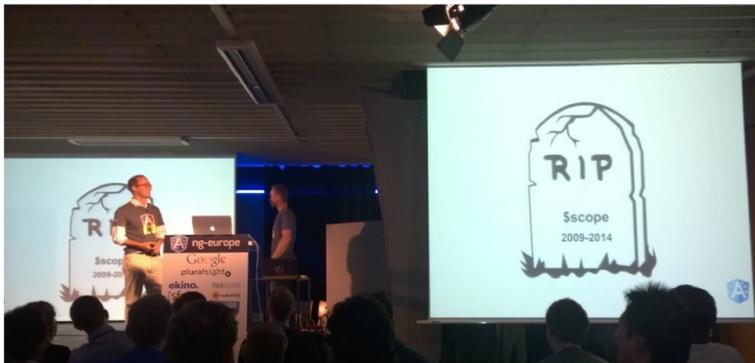
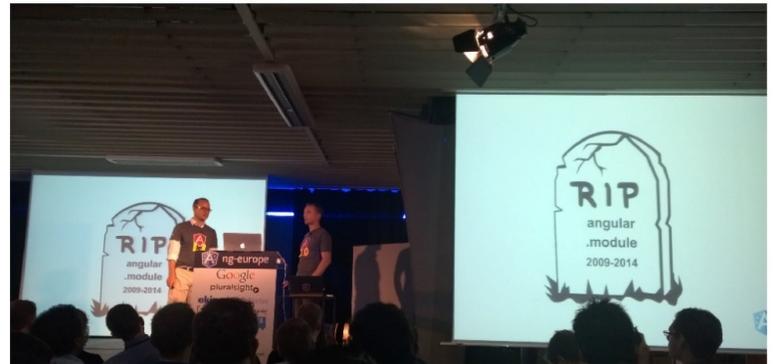
From a high level, the five main parts of Angular 1 are:

- Directives
- Controllers
- Scopes
- Services

- Dependency Injection

Angular 2 changes this list significantly. You might have heard that at ngEurope 2014 Igor and Tobias from the Angular core team announced that they were killing off several “core” ideas in Angular 1 ([video here](#)). Specifically, they announced that Angular 2 was killing off:

- \$scope (& two-way binding by default)
- Directive Definition Objects
- Controllers
- `angular.module`



1

As someone who's built Angular 1 apps and is used to thinking in ng1, we might ask: if we take those things away, what is left? How can you build Angular apps without Controllers and \$scope?

Well, as much as people like to dramatize how **different** Angular 2 is, it turns out, a lot of the same ideas are still with us and, in fact, Angular 2 provides just as much functionality but with **a much simpler model**.

At a high-level Angular 2 core is made up of:

- Components (think “directives”) and
- Services

Of course there’s tons of infrastructure required to make those things work. For instance, you need Dependency Injection to manage your Services. And you need a strong change detection library to efficiently propagate data changes to your app. And you need an efficient rendering layer to handle rendering the DOM at the right time.

Requirements for Interoperability

So given these two different systems, what features do we need for easy interoperability?

- **Use Angular 2 Components in Angular 1:** The first thing that comes to mind is that we need to be able to write new ng2 components, but use them within our ng1 app.
- **Use Angular 1 Components in Angular 2:** It’s likely that we won’t replace a whole branch of our component-tree with all ng2 components. We want to be able to re-use any ng1 components we have *within* a ng2 component.
- **Service Sharing:** If we have, say, a UserService we want to share that service between both ng1 and ng2. Services are normally plain Javascript objects so, more generally, what we need is an interoperable **dependency injection** system.
- **Change Detection:** If we make changes in one side, we want those changes to propagate to the other.

Angular 2 provides solutions for all of these situations and we’ll cover them in this chapter.

In this chapter we’re going to do the following:

- Describe the ng1 app we’ll be converting
- Explain how to setup your hybrid app by using ng2’s UpgradeAdapter
- Explain step-by-step how to share components (directives) and services between ng1 and ng2 by converting the ng1 app to a hybrid app

The Angular 1 App

To set the stage, let’s go over the Angular 1 version of our app.



This chapter assumes some knowledge of Angular 1 and [ui-router](#). If you’re not comfortable with Angular 1 yet, checkout [ng-book 1](#).

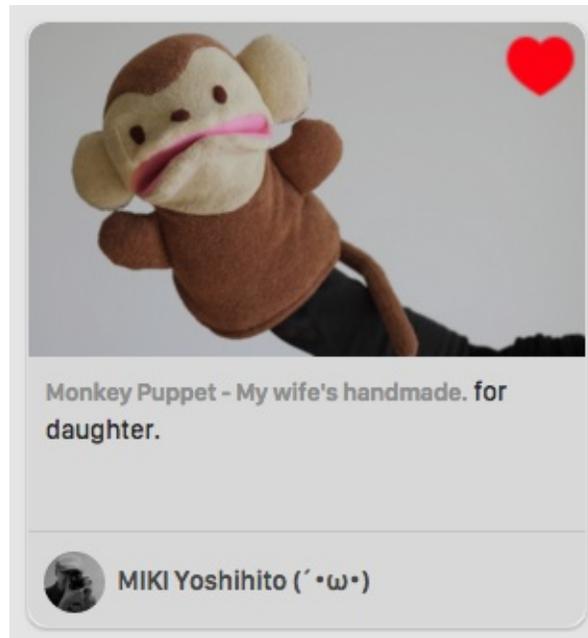
We won’t be diving too deeply into explaining each Angular 1 concept. Instead, we’re going to review the structure of the app to prepare for our upgrade to a ng2/hybrid app.

To run the ng1 app, cd into `conversion/ng1` in the code samples, install the dependencies, and run the app.

```
1 cd code/conversion/ng1 # change directories
2 npm install             # install dependencies
3 npm run go              # run the app
```

If your browser doesn't open automatically, open the url: <http://localhost:8080>.

In this app, you can see that our user is collecting puppets. We can hover over an item and click the heart to "fav" a pin.



Red heart indicates a faved pin

We can also go to the /add page and add a new pin. Try submitting the default form.



Handling image uploads is more complex than we want to handle in this demo. For now, just paste the full URL to an image if you want to try a different image.

The ng1-app HTML

The index.html in our ng1 app uses a common structure:

code/conversion/ng1/index.html

```
1 <!DOCTYPE html>
2 <html ng-app='interestApp'>
3 <head>
4   <meta charset="utf-8">
5   <title>Interest</title>
6   <link rel="stylesheet" href="css/bootstrap.min.css">
7   <link rel="stylesheet" href="css/sf.css">
8   <link rel="stylesheet" href="css/interest.css">
9 </head>
10 <body class="container-fullwidth">
11
12   <div class="page-header">
13     <div class="container">
14       <h1>Interest <small>what you're interested in</small></h1>
15
16       <div class="navLinks">
17         <a ui-sref='home' id="navLinkHome">Home</a>
18         <a ui-sref='add' id="navLinkAdd">Add</a>
19       </div>
```

```

20 </div>
21 </div>
22
23 <div id="content">
24   <div ui-view=''></div>
25 </div>
26
27 <script src="js/vendor/lodash.js"></script>
28 <script src="js/vendor/angular.js"></script>
29 <script src="js/vendor/angular-ui-router.js"></script>
30 <script src="js/app.js"></script>
31 </body>
32 </html>

```

- Notice that we're using ng-app in the html tag to specify that this app uses the module interestApp.
- We load our javascript with script tags at the bottom of the body.
- The template contains a page-header which stores our navigation
- We're using ui-router which means we:
 - Use ui-sref for our links (Home and Add) and
 - We use ui-view where we want the router to populate our content.

Code Overview

We'll look at each section in code, but first, let's briefly describe the moving parts.

In our app, we have two routes:

- / uses the HomeController
- /add uses the AddController

We use a PinsService to hold an array of all of the current pins. HomeController renders the list of pins and AddController adds a new element to that list.

Our root-level route uses our HomeController to render pins. We have a pin directive that renders each pin.

The PinsService stores the data in our app, so let's look at the PinsService first.

ng1: PinsService

code/conversion/ng1/js/app.js

```

1 angular.module('interestApp', ['ui.router'])
2 .service('PinsService', function($http, $q) {
3   this._pins = null;
4
5   this.pins = function() {
6     var self = this;
7     if(self._pins == null) {
8       // initialize with sample data
9       return $http.get("/js/data/sample-data.json").then(
10        function(response) {
11          self._pins = response.data;
12          return self._pins;
13        })
14     } else {
15       return $q.when(self._pins);
16     }
17   }
18
19   this.addPin = function(newPin) {

```

```
20 // adding would normally be an API request so lets mock async
21 return $q.when(
22     this._pins.unshift(newPin)
23 );
24 }
25 })
```

The `PinsService` is a `.service` that stores an array of pins in the property `_.pins`.

The method `.pins` returns a promise that resolves to the list of pins. If `_.pins` is `null` (i.e. the first time), then we will load sample data from `/js/data/sample-data.json`.

code/conversion/ng1/js/data/sample-data.json

```
1 [
2   {
3     "title": "sock puppets",
4     "description": "from:\nThe FunCraft Book of Puppets\n1976\nISBN: 0-590-11936\
5 -2",
6     "user_name": "tofutti break",
7     "avatar_src": "images/avatars/42826303@N00.jpg",
8     "src": "images/pins/106033588_167d811702_o.jpg",
9     "url": "https://www.flickr.com/photos/tofuttibreak/106033588/",
10    "faved": false,
11    "id": "106033588"
12  },
13  {
14    "title": "Puppet play.",
15    "description": "My wife's handmade.",
16    "user_name": "MIKI Yoshihito (´ω´)",
17    "avatar_src": "images/avatars/7940758@N07.jpg",
18    "src": "images/pins/4422575066_7d5c4c41e7_o.jpg",
19    "url": "https://www.flickr.com/photos/mujitra/4422575066/",
20    "faved": false,
21    "id": "4422575066"
22  },
23  {
24    "title": "easy to make puppets - oliver owl (detail)",
25    "description": "from easy to make puppets by joyce luckin (1975)",
26    "user_name": "gilliflower",
27    "avatar_src": "images/avatars/26265986@N00.jpg",
28    "src": "images/pins/6819859061_25d05ef2e1_o.jpg",
29    "url": "https://www.flickr.com/photos/gilliflower/6819859061/",
30    "faved": false,
31    "id": "6819859061"
32  },
33 ]
```

Snippet from Sample Data

The method `.addPin` simply adds the new pin to the array of pins. We use `$q.when` here to return a promise, which is likely what would happen if we were doing a real `async` call to a server.

ng1: Configuring Routes

We're going to configure our routes with `ui-router`.

 If you're unfamiliar with `ui-router` you can [read the docs here](#).

As we mentioned, we're going to have two routes:

code/conversion/ng1/js/app.js

```

26 .config(function($stateProvider, $urlRouterProvider) {
27   $stateProvider
28     .state('home', {
29       templateUrl: '/templates/home.html',
30       controller: 'HomeController as ctrl',
31       url: '/',
32       resolve: {
33         'pins': function(PinsService) {
34           return PinsService.pins();
35         }
36       }
37     })
38     .state('add', {
39       templateUrl: '/templates/add.html',
40       controller: 'AddController as ctrl',
41       url: '/add',
42       resolve: {
43         'pins': function(PinsService) {
44           return PinsService.pins();
45         }
46       }
47     })
48   $urlRouterProvider.when('', '/') ;
49 })
50 })

```

The first route / maps to the HomeController. It has a template, which we'll look at in a minute. Notice that we also are using the resolve functionality of ui-router. This says that before we load this route for the user, we want to call `PinsService.pins()` and inject the result (the list of pins) into the controller (HomeController).

The /add route as similarly, except that it has a different template and a different controller.

Let's first look at our HomeController.

ng1: HomeController

Our HomeController is straightforward. We save pins, which is injected because of our resolve, to `$scope.pins`.

code/conversion/ng1/js/app.js

```

60 .controller('HomeController', function(pins) {
61   this.pins = pins;
62 })

```

ng1: / HomeController template

Our home template is small: we use an `ng-repeat` to repeat over the pins in `$scope.pins`. Then we render each pin with the `pin` directive.

code/conversion/ng1/templates/home.html

```

1 <div class="container">
2   <div class="row">
3     <pin item="pin" ng-repeat="pin in ctrl.pins">
4     </pin>
5   </div>
6 </div>

```

Let's dive deeper and look at this `pin` directive.

ng1: pin Directive

The `pin` directive is restricted to matching an element (E) and has a template.

We can input our `pin` via the `item` attribute, as we did in the `home.html` template.

Our `link` function, defines a function on the scope called `toggleFav` which toggles the `pin`'s `faved` property.

code/conversion/ng1/js/app.js

```
92 })
93 .directive('pin', function() {
94   return {
95     restrict: 'E',
96     templateUrl: '/templates/pin.html',
97     scope: {
98       'pin': "=item"
99     },
100    link: function(scope, elem, attrs) {
101      scope.toggleFav = function() {
102        scope.pin.faved = !scope.pin.faved;
103      }
104    }
105  }
106 })
```



This directive shouldn't be taken as an example of directive best-practices in 2016. For instance, if I was writing this component anew (in ng1) I would probably use the new `.component` directive in Angular 1.5. At the very least, I'd probably use `controllerAs` instead of `link` here.

But this section is less about how to write ng1 code, as much as how to work with the ng1 code you already have.

ng1: pin Directive template

The template `templates/pin.html` renders an individual pin on our page.

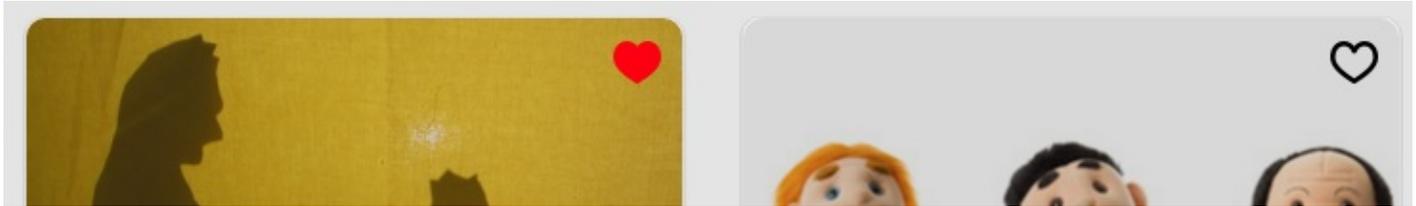
code/conversion/ng1/templates/pin.html

```
1 <div class="col-sm-6 col-md-4">
2   <div class="thumbnail">
3     <div class="content">
4       
5       <div class="caption">
6         <h3>{{pin.title}}</h3>
7         <p>{{pin.description | truncate:100}}</p>
8       </div>
9       <div class="attribution">
10        
11        <h4>{{pin.user_name}}</h4>
12      </div>
13    </div>
14    <div class="overlay">
15      <div class="controls">
16        <div class="heart">
17          <a ng-click="toggleFav()">
18            </img>
19            </img>
20          </a>
21        </div>
22      </div>
23    </div>
24  </div>
25 </div>
```

The directives we use here are ng1 built-ins:

- We use `ng-src` to render the `img`.
- Next we show the `pin.title` and `pin.description`.
- We use `ng-if` to show either the red or empty heart

The most interesting thing here is the `ng-click` that will call `toggleFav`. `toggleFav` changes the `pin.faved` property and thus the red or empty heart will be shown accordingly.



Red vs. Black Heart

Now let's turn our attention to the `AddController`.

ng1: AddController

Our `AddController` has a bit more code than the `HomeController`. We open by defining the controller and specifying the services it will inject:

code/conversion/ng1/js/app.js

```
63 .controller('AddController', function($state, PinsService, $timeout) {
64   var ctrl = this;
65   ctrl.saving = false;
```

We're using `controllerAs` syntax in our router and template, which means we set properties on this instead of on `$scope`. Scoping this in ES5 Javascript can be tricky, so we assign `var ctrl = this;` which helps disambiguate when we're referencing the controller in nested functions.

code/conversion/ng1/js/app.js

```
67   var makeNewPin = function() {
68     return {
69       "title": "Steampunk Cat",
70       "description": "A cat wearing goggles",
71       "user_name": "me",
72       "avatar_src": "images/avatars/me.jpg",
73       "src": "/images/pins/cat.jpg",
74       "url": "http://cats.com",
75       "faved": false,
76       "id": Math.floor(Math.random() * 10000).toString()
77     }
78   }
79
80   ctrl.newPin = makeNewPin();
```

We create a function `makeNewPin` that contains the default structure and data for a pin.

We also initialize this controller by setting `ctrl.newPin` to the value of calling this function.

The last thing we need to do is define the function to submit a new pin:

code/conversion/ng1/js/app.js

```
82   ctrl.submitPin = function() {
83     ctrl.saving = true;
```

```

84     $timeout(function() {
85         PinsService.addPin(ctrl.newPin).then(function() {
86             ctrl.newPin = makeNewPin();
87             ctrl.saving = false;
88             $state.go('home');
89         });
90     }, 2000);
91 }
92 })

```

Essentially, this article is calling out to `PinService.addPin` and creating a new pin. But there's a few other things going on here.

In a real application, this would almost certainly call back to a server. We're mimicking that effect by using `$timeout`. (That is, you could remove the `$timeout` function and this would still work. It's just here to deliberately slow down the app to give us a chance to see the "Saving" indicator.)

We want to give some indication to the user that their pin is saving, so we set the `ctrl.saving = true`.

We call `PinService.addPin` giving it our `ctrl.newPin`. `addPin` returns a promise, so in our promise function we

1. revert `ctrl.newPin` to the original value
2. we set `ctrl.saving` to `false`, because we're done saving the pin
3. we use the `$state` service to redirect the user to the homepage where we can see our new pin

Here's the whole code of the `AddController`:

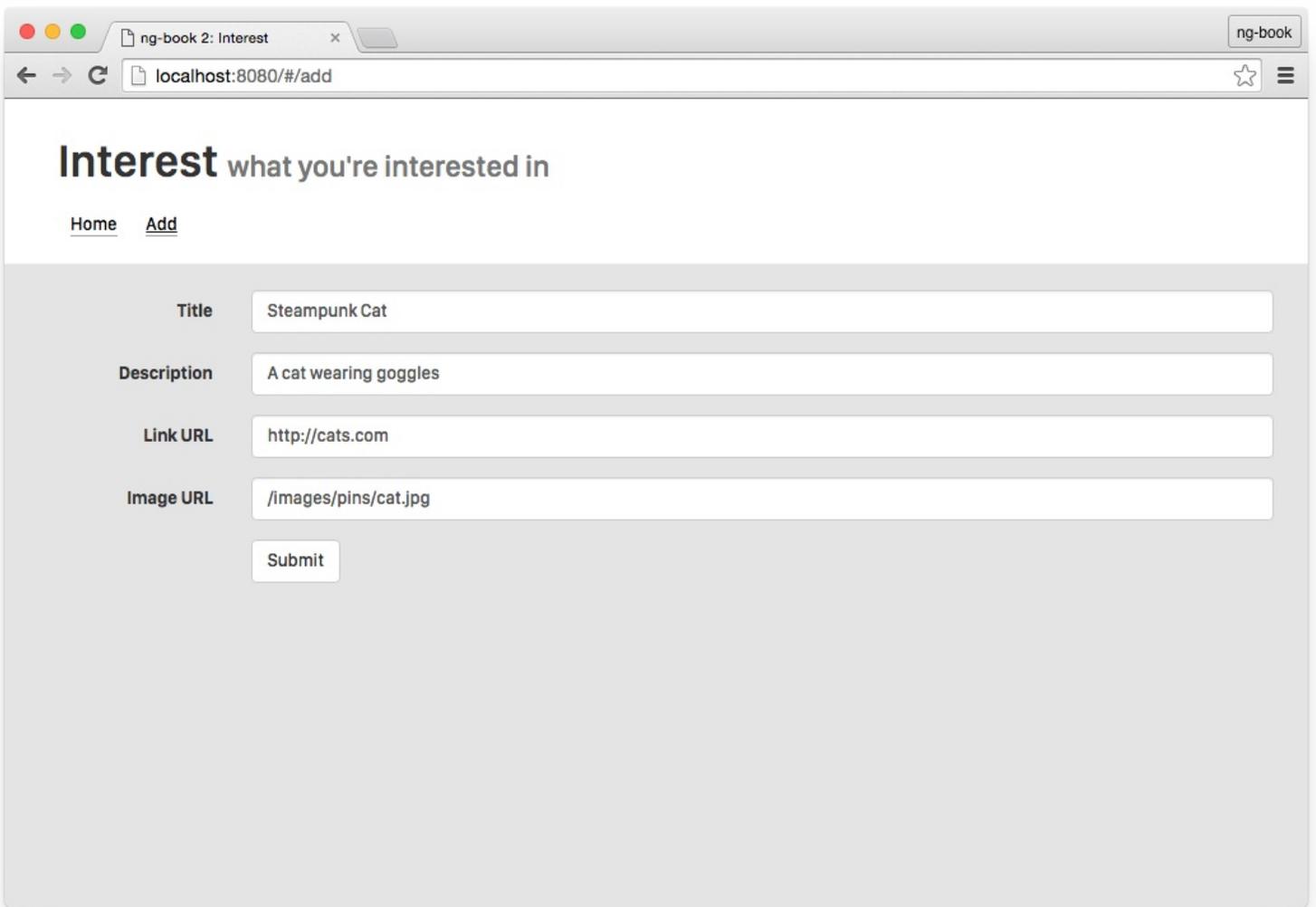
```

code/conversion/ng1/js/app.js
63 .controller('AddController', function($state, PinsService, $timeout) {
64     var ctrl = this;
65     ctrl.saving = false;
66
67     var makeNewPin = function() {
68         return {
69             "title": "Steampunk Cat",
70             "description": "A cat wearing goggles",
71             "user_name": "me",
72             "avatar_src": "images/avatars/me.jpg",
73             "src": "/images/pins/cat.jpg",
74             "url": "http://cats.com",
75             "faved": false,
76             "id": Math.floor(Math.random() * 10000).toString()
77         }
78     }
79
80     ctrl.newPin = makeNewPin();
81
82     ctrl.submitPin = function() {
83         ctrl.saving = true;
84         $timeout(function() {
85             PinsService.addPin(ctrl.newPin).then(function() {
86                 ctrl.newPin = makeNewPin();
87                 ctrl.saving = false;
88                 $state.go('home');
89             });
90         }, 2000);
91     }
92 })

```

ng1: AddController template

Our `/add` route renders the `add.html` template.



Adding a New Pin Form

The template uses `ng-model` to bind the input tags to the properties of the `newPin` on the controller.

The interesting things here are that:

- We use `ng-click` on the submit button to call `ctrl.submitPin` and
- We show a “Saving...” message if `ctrl.saving` is truthy

code/conversion/ng1/templates/add.html

```
1 <div class="container">
2   <div class="row">
3
4     <form class="form-horizontal">
5
6       <div class="form-group">
7         <label for="title"
8           class="col-sm-2 control-label">Title</label>
9         <div class="col-sm-10">
10          <input type="text"
11            class="form-control"
12            id="title"
13            placeholder="Title"
14            ng-model="ctrl.newPin.title">
15          </div>
16        </div>
17
18       <div class="form-group">
19         <label for="description"
20           class="col-sm-2 control-label">Description</label>
```

```

21     <div class="col-sm-10">
22         <input type="text"
23             class="form-control"
24             id="description"
25             placeholder="Description"
26             ng-model="ctrl.newPin.description">
27     </div>
28 </div>
29
30 <div class="form-group">
31     <label for="url"
32         class="col-sm-2 control-label">Link URL</label>
33     <div class="col-sm-10">
34         <input type="text"
35             class="form-control"
36             id="url"
37             placeholder="Link URL"
38             ng-model="ctrl.newPin.url">
39     </div>
40 </div>
41
42 <div class="form-group">
43     <label for="url"
44         class="col-sm-2 control-label">Image URL</label>
45     <div class="col-sm-10">
46         <input type="text"
47             class="form-control"
48             id="url"
49             placeholder="Image URL"
50             ng-model="ctrl.newPin.src">
51     </div>
52 </div>
53
54 <div class="form-group">
55     <div class="col-sm-offset-2 col-sm-10">
56         <button type="submit"
57             class="btn btn-default"
58             ng-click="ctrl.submitPin()">Submit</button>
59     </div>
60 </div>
61 <div ng-if="ctrl.saving">
62     Saving...
63 </div>
64 </form>
65
66 </div>
67 </div>

```

ng1: Summary

There we have it. This app has just the right amount of complexity that we can start porting it to Angular 2.

Building A Hybrid

Now we're ready to start putting some Angular 2 in our Angular 1 app.

Before we start using Angular 2 in our browser, we're going to need to make some modifications to our project structure.

 You can find the code for this example in `code/conversion/hybrid`.

Hybrid Project Structure

The first step to creating a hybrid app is to make sure you have both ng1 and ng2 loaded as dependencies. Everyone's situation is going to be slightly different.

In this example we've **vendored** the Angular 1 libraries (in js/vendor) and we're loading the Angular 2 libraries from npm.

In your project, you might want to vendor them both, use [bower](#), etc. However, using npm is very convenient for Angular 2, and so we suggest using npm to install Angular 2.

Dependencies with package.json

You install dependencies with npm using the package.json file. Here's our package.json for the hybrid example:

code/conversion/hybrid/package.json

```
1 {
2   "name": "ng-hybrid-pinterest",
3   "version": "0.0.1",
4   "description": "toy pinterest clone in ng1/ng2 hybrid",
5   "contributors": [
6     "Nate Murray <nate@fullstack.io>",
7     "Felipe Coury <felipe@ng-book.com>"
8   ],
9   "main": "index.js",
10  "private": true,
11  "scripts": {
12    "clean": "rm -f ts/*.js ts/*.js.map ts/components/*.js ts/components/*.js.ma\
13 p ts/services/*.js ts/services.js.map",
14    "tsc": "./node_modules/.bin/tsc",
15    "tsc:w": "./node_modules/.bin/tsc -w",
16    "serve": "./node_modules/.bin/live-server --host=localhost --port=8080 .",
17    "go": "concurrent \"npm run tsc:w\" \"npm run serve\" "
18  },
19  "dependencies": {
20    "@angular/common": "2.0.0-rc.4",
21    "@angular/compiler": "2.0.0-rc.4",
22    "@angular/core": "2.0.0-rc.4",
23    "@angular/forms": "0.2.0",
24    "@angular/http": "2.0.0-rc.4",
25    "@angular/platform-browser": "2.0.0-rc.4",
26    "@angular/platform-browser-dynamic": "2.0.0-rc.4",
27    "@angular/router": "3.0.0-beta.1",
28    "@angular/router-deprecated": "2.0.0-rc.2",
29    "@angular/upgrade": "2.0.0-rc.2",
30    "core-js": "2.2.2",
31    "es6-shim": "0.35.0",
32    "reflect-metadata": "0.1.3",
33    "rxjs": "5.0.0-beta.6",
34    "systemjs": "0.19.6",
35    "ts-helpers": "1.1.1",
36    "tslint": "3.7.0-dev.2",
37    "typescript": "1.9.0-dev.20160409",
38    "typings": "0.8.1",
39    "zone.js": "0.6.12"
40  },
41  "devDependencies": {
42    "concurrently": "1.0.0",
43    "karma": "0.12.22",
44    "karma-chrome-launcher": "0.1.4",
45    "karma-jasmine": "0.1.5",
46    "live-server": "0.9.0",
47    "typescript": "1.7.3"
48  }
49 }
```



If you're unfamiliar with what one of these packages does, it's a good idea to find out. `rxjs`, for example, is the library that provides our observables. `systemjs` provides the module loader that we're going to use in this chapter.

Once you've added the Angular 2 dependencies, run the command `npm install` to install them.

Compiling our code

You'll notice that in the `package.json` "scripts" key we have another key that specifies "tsc". This means we can run the command `npm run tsc` and it will call out to the TypeScript compiler and compile our code.

We're going to be using TypeScript in this example alongside our Javascript Angular 1 code.

To do this, we're going to put all of our TypeScript code in the folder `ts/` and our Javascript code in the folder `js/`.

We configure the TypeScript compiler by using the `tsconfig.json` file. The important thing to know right now about that file is that in the `filesGlob` key we're specifying a glob of: `./ts/**/*.ts` which means "when we run the TypeScript compiler, we want to compile all files ending in `.ts` in the `ts/` directory".

In this project **our browser will only load Javascript**. We're going to use the TypeScript compiler (`tsc`) to compile our code to Javascript and then we will load our `ng1` and `ng2` *JavaScript* in our browser.

Loading `index.html` dependencies

Now that we have our dependencies and our compiler setup, we need to load these Javascript files into our browser. We do that by adding `script` tags:

`code/conversion/ng1/hybrid/index.html`

```
23 <div id="content">
24   <div ui-view=''></div>
25 </div>
26
27 <!-- Libraries -->
28 <script src="node_modules/es6-shim/es6-shim.js"></script>
29 <script src="node_modules/zone.js/dist/zone.js"></script>
30 <script src="node_modules/reflect-metadata/Reflect.js"></script>
31 <script src="node_modules/systemjs/dist/system.src.js"></script>
32
33 <script src="js/vendor/angular.js"></script>
34 <script src="js/vendor/angular-ui-router.js"></script>
```

The files we loaded from `node_modules/` are Angular 2 and its dependencies. Similarly, the files we loaded from `js/vendor/` are Angular 1 and its dependencies.

But you'll notice here we didn't load any of *our* code in these tags. To load our code we're going to use `System.js`.

Configuring `System.js`

We're going to use `System.js` as the module loader for this example.



We could use webpack (as we do in other examples in this book) or a variety of other loaders (requirejs etc.). However System.js is a wonderful and flexible loader that is often used with Angular 2. This chapter will provide a nice example of how you can use Angular 2 with System.js

To configure System.js we do the following in a `<script>` tag in our `index.html`:

```
1 <script src="resources/systemjs.config.js"></script>
2 System.import('ts/app.js')
3   .then(null, console.error.bind(console));
```

`System.import('ts/app.js')` says that the entry point of our app will be the file `ts/app.js`. When we write hybrid ng2 apps **the Angular 2 code becomes the entry point**. This makes sense because it's Angular 2 that's providing the backwards compatibility with Angular 1. We'll talk more about how to bootstrap the app in a minute.

Another thing to notice here is that we're loading a `.js` file from the `ts/` directory. Why? Because our TypeScript compiler will have compiled this file down to Javascript by the time this page loads.

We have configured System.js in `resources/systemjs.config.js`. That file contains a mostly-standard configuration, but since we have to be able to load our ng1 app in our ng2 code we've added a special key `interestAppNg1` that points to our ng1 app. This option lets us do the following in our TypeScript code:

```
1 import 'interestAppNg1'; // "bare import" for side-effects
```

The module loader will see the string `'interestAppNg1'` and load our Angular 1 app at `./js/app.js`.

The `packages` key specifies that files in the `ts` "packages" will have the extension `.js` and use the System.js register module format.



There are a bunch of module formats your TypeScript compiler can output. The System.js format needs to match the module format you're compiling to. So in this case, the register module format will work with our TypeScript because we specified `compilerOptions.module` as `"system"` in our `tsconfig.json`



Configuring System.js is fairly advanced and there are a lot of potential options here.

This isn't a book on module loaders and, in-fact, it would probably take a whole book to explore in-depth how to configure System.js and other Javascript module loaders.

For now, we're not going to talk much more about module loading, but you can read up more on [System.js here](#)



Would you like to read a book on Javascript module loaders? We're considering writing one. If you'd like to be notified when it's ready, [put in your email here](#)

Bootstrapping our Hybrid App

Now that we have our project structure in place, let's bootstrap the app.

If you recall, with Angular 1 you can bootstrap the app in 1 of two ways:

1. You can use the ng-app directive, such as ng-app='interestApp', in your HTML or
2. You can use angular.bootstrap in Javascript

In hybrid apps we use a **new bootstrap** method that comes from an UpgradeAdapter.

Since we'll be bootstrapping the app in code, **make sure you remove the ng-app from your index.html.**

Here's what a minimal bootstrapping of our code would look like:

```
1 // code/conversion/hybrid/ts/app.ts
2
3 import { UpgradeAdapter } from '@angular/upgrade';
4 import * as angular from 'angular2/src/upgrade/angular_js';
5 import 'interestAppNg1'; // "bare import" for side-effects
6
7 /*
8  * Create our upgradeAdapter
9  */
10 const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter();
11
12 /*
13  * Bootstrap the App
14  */
15 upgradeAdapter.bootstrap(document.body, ['interestApp']);
```

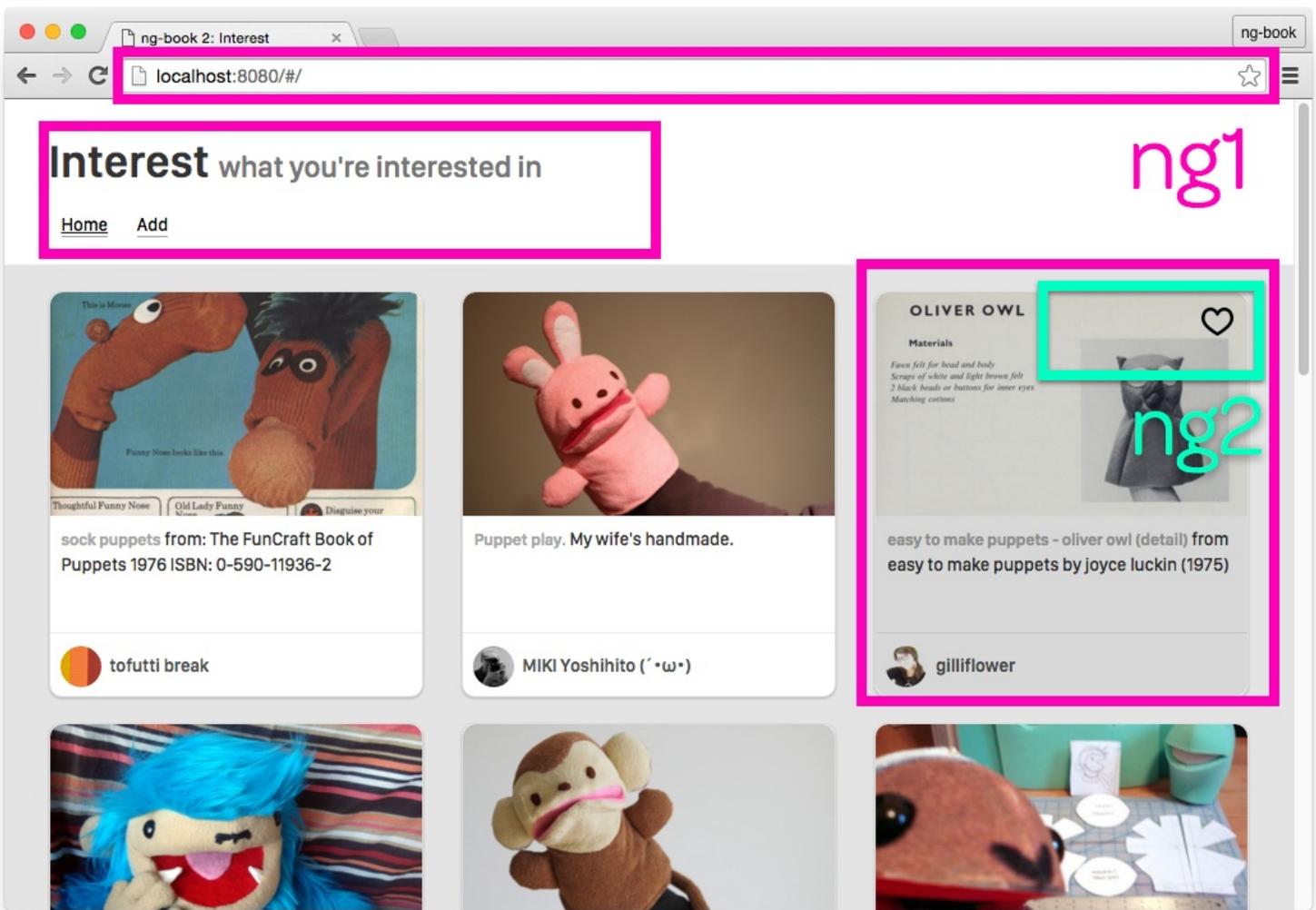
We start by importing the UpgradeAdapter and then we create an instance of it: upgradeAdapter. We tell the upgradeAdapter to bootstrap our app on the element document.body and we specify the module name of our **angular 1 app**.

This will bootstrap our Angular 1 app within our Angular 2 app! Now we can start replacing pieces with Angular 2.

What We'll Upgrade

Let's discuss what we're going to port to ng2 in this example and what will stay in ng1.

The Homepage



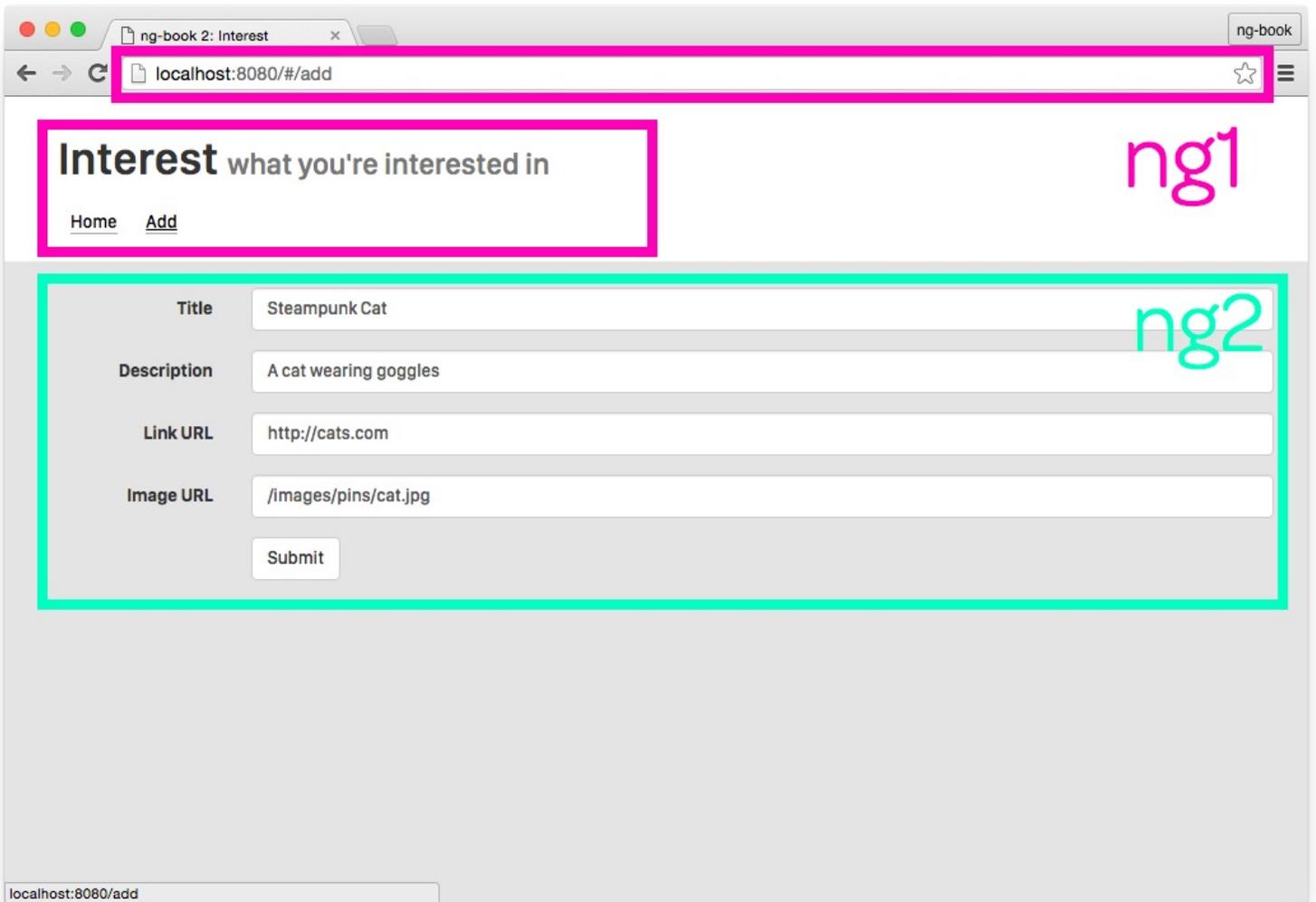
Homepage ng1 and ng2 Components

The first thing to notice is that we're going to continue to manage routing with ng1. Of course, Angular 2 has its own routing, which you can read about in [our routing chapter](#). But if you're building a hybrid app, you probably have lots of routes configured with Angular 1 and so in this example we'll continue to use ui-router for the routing.

On the homepage, we're going to nest a ng2 component within an ng1 directive. In this case, we're going to convert the "pin controls" to a ng2 component. That is, our ng1 pin directive, will call out to the ng2 pin-controls component and pin-controls will render the fav heart.

It's a small example that shows a powerful idea: how to seamlessly exchange data between ng versions.

The About Page



About Page ng1 and ng2 Components

We're going to use ng1 for the router and header on the about page as well. However on the about page, we're going to replace the whole form with a ng2 component: `AddPinComponent`.

If you recall, the form will add a new pin to the `PinsService`, and so in this example we're going to need to somehow make the (ng1) `PinsService` accessible to the (ng2) `AddPinComponent`.

Also, remember that when a new pin is added, the app should be redirected to the homepage. However, to change routes we need to use the ui-router `$state` service (ng1) in the `AddPinComponent` (ng2). So we also need to make sure the `$state` service can be used in `AddPinComponent` as well.

Services

So far we've talked about two ng1 services that will be *upgraded* to ng2:

- `PinsService` and
- `$state`

We also want to explore “downgrading” a ng2 service to be used by ng1. For this, later on in the chapter, we'll create an `AnalyticsService` in `TypeScript/ng2` that we share with ng1.

Taking Inventory

So to recap we're going to "cross-expose" the following:

- Downgrade the ng2 PinControlsComponent to ng1 (for the fav buttons)
- Downgrade the ng2 AddPinComponent to ng1 (for the add pin page)
- Downgrade the ng2 AnalyticsService to ng1 (for recording events)
- Upgrade the ng1 PinsService to ng2 (for adding new pins)
- Upgrade the ng1 \$state service to ng2 (for controlling routes)

A Minor Detour: Typing Files

One of the great things about TypeScript is the compile-time typing. However, if you're building a hybrid app, I suspect that you've got a lot of untyped Javascript code that you're going to be integrating into this project.

When you try to use your Javascript code from TypeScript you may get compiler errors because the compiler doesn't know the structure of your Javascript objects. You could try casting everything to `<any>` but that is ugly and error prone.

The better solution is to, instead, provide your TypeScript compiler with custom *type annotations*. Then the compiler will be able to enforce the types of your Javascript code.

For instance, remember how in our ng1 app we created a pin object in `makeNewPin`?

`code/conversion/ng1/js/app.js`

```
67 var makeNewPin = function() {
68   return {
69     "title": "Steampunk Cat",
70     "description": "A cat wearing goggles",
71     "user_name": "me",
72     "avatar_src": "images/avatars/me.jpg",
73     "src": "/images/pins/cat.jpg",
74     "url": "http://cats.com",
75     "faved": false,
76     "id": Math.floor(Math.random() * 10000).toString()
77   }
78 }
79
80 ctrl.newPin = makeNewPin();
```

It would be nice if we could tell the compiler about the structure of these objects and not resort to using `any` everywhere.

Furthermore, we're going to be using the `ui-router` `$state` service in Angular 2 / TypeScript, and we need to tell the compiler what functions are available there, too.

So while providing TypeScript custom type definitions is a TypeScript (and not an Angular-specific) chore, it's a chore we need to do nonetheless. And it's something that many people haven't done yet because TypeScript is, at time of publishing, relatively new.

So in this section I want to walk through how you deal with custom typings in TypeScript.



If you're already familiar with how to create and use TypeScript type definition files, you can safely skim this section.

Typing Files

In TypeScript we can describe the structure of our code by writing *typing definition files*. Typing definition files generally end in the extension `.d.ts`.

Generally, when you write TypeScript code, you don't need to write a `.d.ts` because your TypeScript code itself contains types. We write `.d.ts` files when we have some external Javascript code that we want to add typing to after the fact.

For instance, in describing our pin object, we could write an interface for it like so:

code/conversion/hybrid/js/app.d.ts

```
3  export interface Pin {
4    title: string;
5    description: string;
6    user_name: string;
7    avatar_src: string;
8    src: string;
9    url: string;
10   faved: boolean;
11   id: string;
12 }
```

Notice that we're not declaring a class, and we're not creating an instance. Instead, we're defining the shape (types) of an interface.

In order to use `.d.ts` files, you need to tell the TypeScript compiler where they are. The easiest way to do this is by modifying the `tsconfig.json` file. For instance, if we had a file `js/app.d.ts` we could add it like this:

```
1  // tsconfig.json
2  "compilerOptions": { ... },
3  "files": [
4    "ts/app.ts",
5    "js/app.d.ts"
6  ],
7  // more...
```

Look closely at the paths of the files in this case. We're loading our TypeScript `ts/app.ts`. And we're loading `app.d.ts` from `js/`. This is because the `js/app.d.ts` is the typing file for `js/app.js` (the ng1 Javascript file, not the ng2 TypeScript).

We'll write `app.d.ts` in a little bit. First, let's explore a tool that exists to help us with third-party TypeScript definition files: `typings`.

Third-party libraries with `typings`

`typings` is a tool for managing TypeScript type definition files for libraries that may not have them otherwise.

We're going to use `angular-ui-router` with our app, so let's install the typings for `angular-ui-router`. Here's how to get it setup.

You need to have `typings` installed, which you can do with `npm install -g typings`.

Next we configure a `typings.json` file, which you can create with `typings init` (or use the one provided).

Then we install the package we need by running: `typings install angular-ui-router --save`.

Notice that `typings` created a `typings` directory that contains a file `browser.d.ts`. This `browser.d.ts` is the entry point for the rest of the typings that are **managed by typings**. That is, if you write your own typings files, they're not going to be here, but any of the typings files you install via `typings` will be loaded via the reference tag in that file.



Don't modify the `typings/browser.d.ts` file directly! `typings` manages this file for you and if you change it your changes may be overwritten.

Now that we have the typings file `typings/browser.d.ts`, how do we use it? We have to tell our compiler about it, and we do that via the `tsconfig.json` file.

```
1 // tsconfig.json
2 "compilerOptions": { ... },
3 "files": [
4   "typings/browser.d.ts",
5   "ts/app.ts",
6   "js/app.d.ts"
7 ],
8 // more...
```

Notice that we added `typings/browser.d.ts` to the `files` array. This tells our compiler that we want to include our typings typings at compile time.



What if we were loading a different library, such as underscore and we needed to load it from System.js as well?

The idea is that you have to 1. make the typings available to the compiler at compile time and 2. make the code available at runtime

One way is like this:

1. `typings install underscore` - installs the typings file
2. `npm install underscore` - installs the javascript file in `node_modules`
3. In your `index.html` where you call `System.config`, add a new entry to the `paths` key like: `underscore: './node_modules/underscore/underscore.js'`
4. Then you can import underscore in your TypeScript using: `import * as _ from 'underscore';`
5. Use underscore like so: `let foo = _.map([1,2,3], (x) => x + 1);`



We've already done a `typings install` for you for this application so you don't need to install the dependencies yourself.

In fact, if you do run `typings install` you may find that you get the error:

```
1 node_modules/angular2/typings/angular-protractor/angular-protractor.d.ts(1679,13\
2 ): error TS2403: Subsequent variable declarations must have the same type. Vari
3 able '$' must be of type 'JQueryStatic', but here has type 'cssSelectorHelper'.
```

This is due to a bug between the `jquery` and the `angular` typings both trying to assign a type to the dollar sign `$`. At time of publishing, the hacky workaround is to open `typings/jquery/jquery.d.ts` and comment out this line:

```
1 // declare var $: JQueryStatic; // - ng-book told me to comment this
```

Of course, this will cause problems if you're trying to use `jQuery`-specific typings via `$` in TypeScript (but we aren't for this example).

Custom Typing Files

Being able to use third-party typing files is great, but there are going to be situations where typing files don't already exist: especially in the case of our own code.

Generally, when we write custom typing files we co-locate the file alongside its respective Javascript code. So let's create the file `js/app.d.ts`:

`code/conversion/hybrid/js/app.d.ts`

```
1 declare module interestAppNg1 {
2
3   export interface Pin {
4     title: string;
5     description: string;
6     user_name: string;
7     avatar_src: string;
8     src: string;
9     url: string;
10    faved: boolean;
11    id: string;
12  }
13
14  export interface PinsService {
15    pins(): Promise<Pin[]>;
16    addPin(pin: Pin): Promise<any>;
17  }
18
19 }
```

```
20
21 declare module 'interestAppNg1' {
22   export = interestAppNg1;
23 }
```

When we use the `declare` keyword, that is called making an “ambient declaration” and the idea is that we’re defining a variable that didn’t originate from a TypeScript file. In this case, we’re defining two interfaces:

1. `Pin`
2. `PinsService`

The `Pin` interface describes the keys and value-types of a pin object.

The `PinsService` interface describes the types of our two methods on our `PinsService`.

- `pins()` returns a `Promise` of an array of `Pins`
- `addPin()` takes a `Pin` as an argument and returns a `Promise`



Learn More about Writing Type Definition Files

If you’d like to learn more about writing `.d.ts` files, checkout these helpful links:

- [TypeScript Handbook: Working with other Javascript Libraries](#)
- [TypeScript Handbook: Writing definition files](#)
- [Quick tip: Typescript declare keyword](#)

You might have noticed that we don’t declare the token `interestAppNg1` anywhere in our `ng1` Javascript code. `interestAppNg1` is just an identifier we use on the TypeScript side to specify this javascript code.

Now that we have this file setup, we can import these types like so:

```
1 import { Pin, PinsService } from 'interestAppNg1';
```

Writing `ng2 PinControlsComponent`

Now that we have the typings figured out, let’s turn our attention back to the hybrid app.

The first thing we’re going to do is write the `ng2 PinControlsComponent`. This will be an `ng2` component nested within an `ng1` directive. The `PinControlsComponent` displays the fav hearts and toggles fav’ing a pin.

Let’s start by importing our `Pin` type, along with a few other constants that we’ll need:

`code/conversion/hybrid/ts/components/PinControlsComponent.ts`

```
1 /*
2  * PinControls: a component that holds the controls for a particular pin
3  */
4 import {
5   Component,
6   Input,
```

```
7   Output,  
8   EventEmitter  
9 } from '@angular/core';  
10 import { NgIf } from '@angular/common';  
11 import { Pin } from 'interestAppNg1';
```

Next, let's write the @Component annotation:

code/conversion/hybrid/ts/components/PinControlsComponent.ts

```
13 @Component({  
14   selector: 'pin-controls',  
15   directives: [NgIf],  
16   template: `  
17 <div class="controls">  
18   <div class="heart">  
19     <a (click)="toggleFav()">  
20         
21         
22     </a>  
23   </div>  
24 </div>  
25 `,  
26 })
```

Notice here that we'll match the element `pin-controls`. We also specify that we're using the `NgIf` directive.

Our template looks very similar to the `ng1` version except we're using the `ng2` template syntax for `(click)` and `*ngIf`.

Now the component definition class:

code/conversion/hybrid/ts/components/PinControlsComponent.ts

```
27 export class PinControlsComponent {  
28   @Input() pin: Pin;  
29   @Output() faved: EventEmitter<Pin> = new EventEmitter<Pin>();  
30  
31   toggleFav(): void {  
32     this.faved.next(this.pin);  
33   }  
34 }
```

Notice that instead of specifying inputs and outputs in the @Component annotation, in this case we're annotating the properties on the class directly with the @Input and @Output annotations. This is a convenient way to us to provide typings to these properties.

This component will take an input of `pin`, which is the `Pin` object we're controlling.

This component specifies an output of `faved`. This is a little bit different than how we did it in the `ng1` app. If you look at `toggleFav` all we're doing is emitting (on the `EventEmitter`) the current `pin`.

The idea here is that we've already implemented how to change the `faved` state in `ng1` and we may not want to re-implement that functionality `ng2` (you may want to, it just depends on your team conventions).

Using ng2 PinControlsComponent

Now that we have an `ng2` `pin-controls` component, we can use it in a template. Here's what our `pin.html` template looks like now:

```
1 <div class="col-sm-6 col-md-4">
2   <div class="thumbnail">
3     <div class="content">
4       
5       <div class="caption">
6         <h3>{{pin.title}}</h3>
7         <p>{{pin.description | truncate:100}}</p>
8       </div>
9       <div class="attribution">
10        
11        <h4>{{pin.user_name}}</h4>
12      </div>
13    </div>
14    <div class="overlay">
15      <pin-controls [pin]="pin"
16        (faved)="toggleFav($event)"></pin-controls>
17    </div>
18  </div>
19 </div>
```

This template is for an ng1 directive, and we can use ng1 directives such as ng-src. However, notice the line where we use our ng2 pin-controls component:

```
1 <pin-controls [pin]="pin"
2   (faved)="toggleFav($event)"></pin-controls>
```

What's interesting here is that we're using the ng2 input bracket syntax [pin] and the ng2 output parenthesis syntax (faved).

In a hybrid app **when you use ng2 directives in ng1, you still use the ng2 syntax.**

With our input [pin] we're passing the pin which comes from the scope of the ng1 directive.

With our output (faved) we're calling the toggleFav function on the scope of the ng1 directive. Notice what we did here: we didn't modify the pin.faved state within the ng2 directive (although, we could have). Instead, we asked the ng2 PinControlsComponent to simply emit the pin when toggleFav is called there. (If this is confusing, take a second look at toggleFav of PinControlsComponent.)

Again, the reason we do this is because we're showing how you can keep your existing functionality (scope.toggleFav) in ng1, but start porting over components to ng2. In this case, the ng1 pin directive listens for the faved event on the ng2 PinControlsComponent.

If you refresh your page now, you'll notice that it doesn't work. That's because there's one more thing we need to do: downgrade PinControlsComponent to ng1.

Downgrading ng2 PinControlsComponent to ng1

The final step to using our components across ng2/ng1 borders is to use our UpgradeAdapter to downgrade our components (or upgrade, as we'll see in a bit).

We perform this downgrade in our app.ts file (where we called upgradeAdapter.bootstrap).

First we need to have angular imported:

```
code/conversion/hybrid/ts/app.ts
```

```
9 import { UpgradeAdapter } from '@angular/upgrader';
10 import * as angular from '@angular/upgrader/src/angular_js';
11 import 'interestAppNg1'; // "bare import" for side-effects
```

Then we create a `.directive` in (almost) the normal ng1 way:

code/conversion/hybrid/ts/app.ts

```
21 /*
22  * Expose our ng2 content to ng1
23  */
24 angular.module('interestApp')
25   .directive('pinControls',
26             upgradeAdapter.downgradeNg2Component(PinControlsComponent))
```

Above, remember that when we import `'interestAppNg1'` this will load up our ng1 app, which calls `angular.module('interestApp', [])`. That is, our ng1 app has already registered the `interestApp` module with angular.

Now we want to look up that module by calling `angular.module('interestApp')` and then add directives to it, just like we do in ng1 normally.

`angular.module` getter and setter syntax

If you recall, when we pass an array as the second argument to `angular.module`, we are *creating* a module. That is, `angular.module('foo', [])` will *create* the module `foo`. Informally, we call this the “setter” syntax.

Similarly, if we omit the array we are *getting* a module (that is assumed to already exist). That is, `angular.module('foo')` will *get* the module `foo`. We call this the “getter” syntax.

 In this example, if you forget this distinction and call `angular.module('interestApp', [])` in `app.ts` (ng2) then you will accidentally overwrite your existing `interestApp` module and your app won't work. Careful!

We're calling `.directive` and creating a directive called `'pinControls'`. This is standard ng1 practice. For the second argument, the directive definition object (DDO), we don't create the DDO manually. Instead, we call `upgradeAdapter.downgradeNg2Component`.

`downgradeNg2Component` will convert our `PinControlsComponent` into an ng1-compatible directive. Pretty neat.

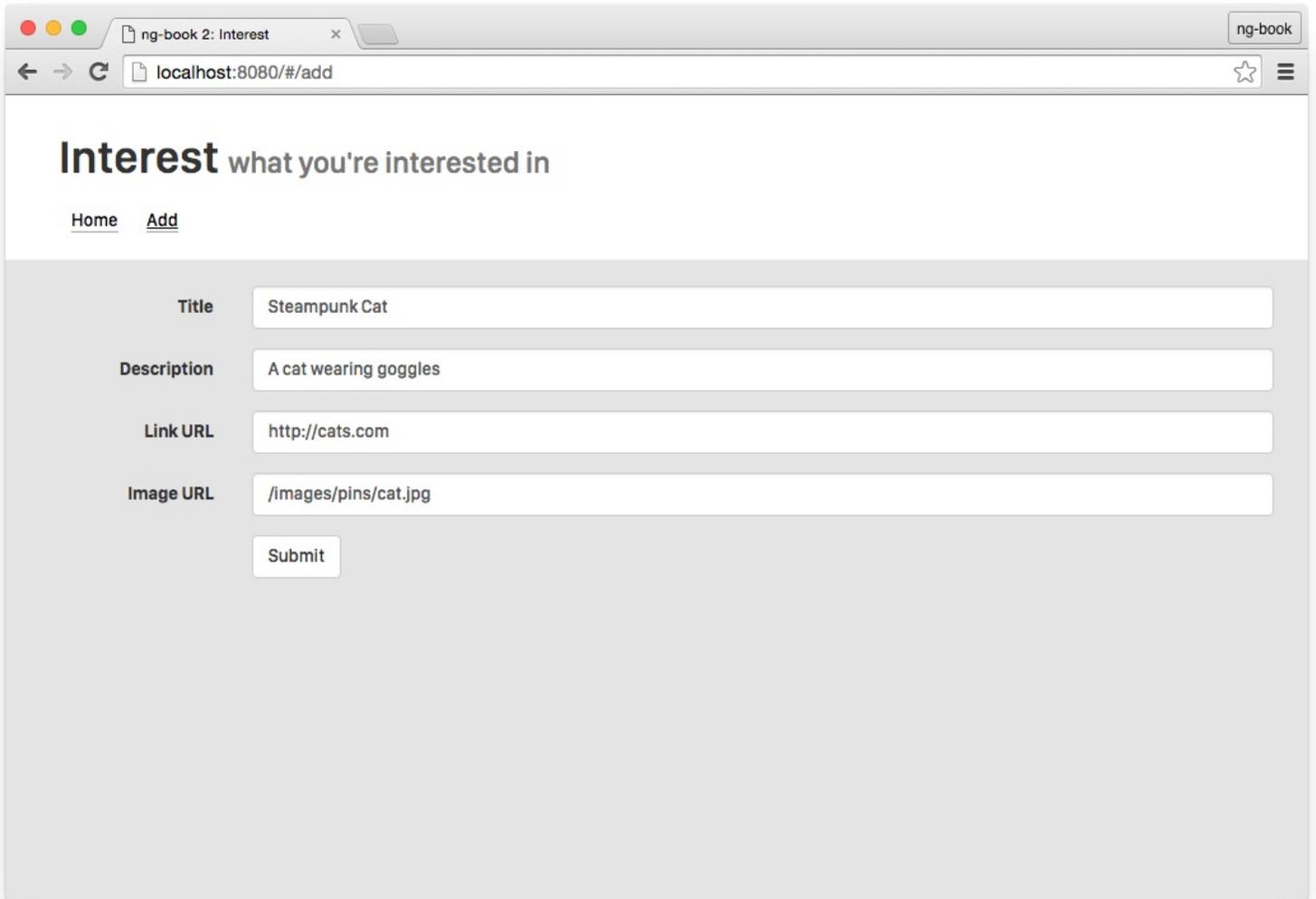
Now if you try refreshing, you'll notice that our faving works just like before, only now we're using ng2 embedded in ng1!



Faving works like a charm

Adding Pins with ng2

The next thing we want to do is upgrade the add pins page with an ng2 component.



The screenshot shows a web browser window with the following details:

- Tab: ng-book 2: Interest
- Address bar: localhost:8080/#/add
- Page title: Interest what you're interested in
- Navigation: Home, Add
- Form fields:
 - Title: Steampunk Cat
 - Description: A cat wearing goggles
 - Link URL: http://cats.com
 - Image URL: /images/pins/cat.jpg
- Submit button

Adding a New Pin Form

If you recall, this page does three things:

1. Present a form to the user for describing the pin
2. Use the `PinsService` to add the new pin to the list of pins
3. Redirect the user to the homepage

Let's think through how we're going to do these things from ng2.

Angular 2 provides a robust forms library. So there's no complication here. We're going to write a straight ng2 form.

However the `PinsService` comes from ng1. Often we have many existing services in ng1 and we don't have time to upgrade them all. So for this example, we're going to keep `PinsService` as an ng1 object, and *inject it into ng2*.

Similarly, we're using `ui-router` in `ng1` for our routing. To change pages in `ui-router` we have to use the `$state` service, which is an `ng1` service.

So what we're going to do is **upgrade** the `PinsService` and the `$state` service from `ng1` to `ng2`. And this couldn't be any easier.

Upgrading `ng1` `PinsService` and `$state` to `ng2`

To upgrade `ng1` services we call `upgradeAdapter.upgradeNg1Provider`:

`code/conversion/hybrid/ts/app.ts`

```
35 /*
36  * Expose our ng1 content to ng2
37  */
38 upgradeAdapter.upgradeNg1Provider('PinsService');
39 upgradeAdapter.upgradeNg1Provider('$state');
```

And that's it. Now we can `@Inject` our `ng1` services into `ng2` components like so:

```
1 class AddPinComponent {
2   constructor(@Inject('PinsService') public pinsService: PinsService,
3               @Inject('$state') public uiState: IStateService) {
4   }
5   // ...
6   // now you can use this.pinsService
7   // or this.uiState
8   // ...
9 }
```

In this constructor, there's a few things to look at:

The `@Inject` annotation, says that we want the next variable to be assigned the value of what the injection will resolve to. In the first case, that would be our `ng1` `PinsService`.

In TypeScript, in a constructor when you use the `public` keyword, it is a shorthand for assigning that variable to `this`. That is, here when we say `public pinsService` what we're saying is, 1. declare a property `pinsService` on instances of this class and 2. assign the constructor argument `pinsService` to `this.pinsService`.

The result is that we can access `this.pinsService` throughout our class.

Lastly we define the type of both services we're injecting: `PinsService` and `IStateService`.

`PinsService` comes from the `app.d.ts` we defined previously:

`code/conversion/hybrid/js/app.d.ts`

```
14 export interface PinsService {
15   pins(): Promise<Pin[]>;
16   addPin(pin: Pin): Promise<any>;
17 }
```

And `IStateService` comes from the typings for `ui-router`, which we installed with typings.

By telling TypeScript the types of these services we can enjoy type-checking as we write our code.

Let's write the rest of our `AddPinComponent`.

Writing ng2 AddPinComponent

We start by importing the types we need:

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
1 /*
2  * AddPinComponent: a component that controls the "add pin" page
3  */
4 import {
5   Component,
6   Inject
7 } from '@angular/core';
8 import { Pin, PinsService } from 'interestAppNg1';
9 import { IStateService } from 'angular-ui-router';
```

Again, notice that we're importing our custom types Pin and PinsService. And we're also importing IStateService from angular-ui-router.

AddPinComponent @Component

Our @Component annotation is straightforward:

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
11 @Component({
12   selector: 'add-pin',
13   templateUrl: '/templates/add-ng2.html'
14 })
```

AddPinComponent template

We're loading our template using a templateUrl. In that template, we setup our form much like the ng1 form, only we're using ng2 form directives.



We're not going to describe ngModel / ngSubmit deeply here. If you'd like to know more about how Angular 2 forms work, checkout [the forms chapter](#), where we describe forms in depth.

code/conversion/hybrid/templates/add-ng2.html

```
1 <div class="container">
2   <div class="row">
3
4     <form (ngSubmit)="onSubmit()"
5         class="form-horizontal">
6
7       <div class="form-group">
8         <label for="title"
9           class="col-sm-2 control-label">Title</label>
10        <div class="col-sm-10">
11          <input type="text"
12            class="form-control"
13            id="title"
14            placeholder="Title"
15            [(ngModel)]="newPin.title">
16        </div>
17      </div>
```

We're using two directives here: ngSubmit and ngModel.

We use (ngSubmit) on the form to call the onSubmit function when the form is submitted. (We'll define onSubmit on the AddPinComponent controller below.)

We use `[(ngModel)]` to bind the value of the `title` input tag to the value of `newPin.title` on the controller.

Here's the full listing of the template:

code/conversion/hybrid/templates/add-ng2.html

```
1 <div class="container">
2   <div class="row">
3
4     <form (ngSubmit)="onSubmit()"
5         class="form-horizontal">
6
7       <div class="form-group">
8         <label for="title"
9           class="col-sm-2 control-label">Title</label>
10        <div class="col-sm-10">
11          <input type="text"
12            class="form-control"
13            id="title"
14            placeholder="Title"
15            [(ngModel)]="newPin.title">
16        </div>
17      </div>
18
19      <div class="form-group">
20        <label for="description"
21          class="col-sm-2 control-label">Description</label>
22        <div class="col-sm-10">
23          <input type="text"
24            class="form-control"
25            id="description"
26            placeholder="Description"
27            [(ngModel)]="newPin.description">
28        </div>
29      </div>
30
31      <div class="form-group">
32        <label for="url"
33          class="col-sm-2 control-label">Link URL</label>
34        <div class="col-sm-10">
35          <input type="text"
36            class="form-control"
37            id="url"
38            placeholder="Link URL"
39            [(ngModel)]="newPin.url">
40        </div>
41      </div>
42
43      <div class="form-group">
44        <label for="url"
45          class="col-sm-2 control-label">Image URL</label>
46        <div class="col-sm-10">
47          <input type="text"
48            class="form-control"
49            id="url"
50            placeholder="Image URL"
51            [(ngModel)]="newPin.src">
52        </div>
53      </div>
54
55      <div class="form-group">
56        <div class="col-sm-offset-2 col-sm-10">
57          <button type="submit"
58            class="btn btn-default"
59            >Submit</button>
60        </div>
61      </div>
62      <div *ngIf="saving">
63        Saving...
64      </div>
65    </form>
66
67
```

```
68 </div>
69 </div>
```

AddPinComponent Controller

Now we can define AddPinComponent. We start by setting up two instance variables:

```
code/conversion/hybrid/ts/components/AddPinComponent.ts
```

```
15 export class AddPinComponent {
16   saving: boolean = false;
17   newPin: Pin;
```

We use saving to indicate to the user that the save is in progress and we use newPin to store the Pin we're working with.

```
code/conversion/hybrid/ts/components/AddPinComponent.ts
```

```
19 constructor(@Inject('PinsService') public pinsService: PinsService,
20             @Inject('$state') public uiState: IStateService) {
21   this.newPin = this.makeNewPin();
22 }
```

In our constructor we Inject the services, as we discussed above. We also set this.newPin to the value of makeNewPin, which we'll define now:

```
code/conversion/hybrid/ts/components/AddPinComponent.ts
```

```
24 makeNewPin(): Pin {
25   return {
26     title: 'Steampunk Cat',
27     description: 'A cat wearing goggles',
28     user_name: 'me',
29     avatar_src: 'images/avatars/me.jpg',
30     src: '/images/pins/cat.jpg',
31     url: 'http://cats.com',
32     faved: false,
33     id: Math.floor(Math.random() * 10000).toString()
34   };
35 }
```

This looks a lot like how we defined it in ng1, only now we have the benefit of it being typed.

When the form is submitted, we call onSubmit. Let's define that:

```
code/conversion/hybrid/ts/components/AddPinComponent.ts
```

```
37 onSubmit(): void {
38   this.saving = true;
39   console.log('submitted', this.newPin);
40   setTimeout(() => {
41     this.pinsService.addPin(this.newPin).then(() => {
42       this.newPin = this.makeNewPin();
43       this.saving = false;
44       this.uiState.go('home');
45     });
46   }, 2000);
47 }
```

Again, we're using a timeout to *simulate* the effect of what would happen if we had to call out to a server to save this pin. Here, we're using setTimeout. Compare that to how we defined this function in ng1:

```
code/conversion/ng1/js/app.js
```

```
82 ctrl.submitPin = function() {
83   ctrl.saving = true;
84   $timeout(function() {
```

```
85     PinsService.addPin(ctrl.newPin).then(function() {
86         ctrl.newPin = makeNewPin();
87         ctrl.saving = false;
88         $state.go('home');
89     });
90 }, 2000);
91 }
```

Notice that in ng1 we had to use the `$timeout` service. Why is that? Because ng1 is based around the digest loop. If you use `setTimeout` in ng1, then when the callback function is called, it's "outside" of angular and so your changes aren't propagated unless something kicks off a digest loop (e.g. using `$scope.apply`).

However in ng2, we can use `setTimeout` directly because change detection in ng2 uses Zones and is therefore, more or less automatic. We don't need to worry about the digest loop in the same way, which is really nice.

In `onSubmit` we're calling out to the `PinsService` by:

```
1 this.pinsService.addPin(this.newPin).then(() => {
2 // ...
```

Again, the `PinsService` is accessible via `this.pinsService` because of how we defined the constructor. The compiler doesn't complain because we said that `addPin` takes a `Pin` as the first argument in our `app.d.ts`:

```
code/conversion/hybrid/js/app.d.ts
14 export interface PinsService {
15     pins(): Promise<Pin[]>;
16     addPin(pin: Pin): Promise<any>;
17 }
```

And we defined `this.newPin` to be a `Pin`.

After `addPin` resolves, we reset the pin using `makeNewPin` and set `this.saving = false`.

To go back to the homepage, we use the `ui-router $state` service, which we stored as `this.uiState`. So we can change states by calling `this.uiState.go('home')`.

Using AddPinComponent

Downgrade ng2 AddPinComponent

To use `AddPinComponent` we need to downgrade it:

```
code/conversion/hybrid/ts/app.ts
24 angular.module('interestApp')
25     .directive('pinControls',
26         upgradeAdapter.downgradeNg2Component(PinControlsComponent))
27     .directive('addPin',
28         upgradeAdapter.downgradeNg2Component(AddPinComponent));
```

This will create the `addPin` directive in ng1, which will match the tag `<add-pin>`.

Routing to add-pin

In order to use our new AddPinComponent page, we need to place it somewhere within our ng1 app. What we're going to do is take the add state in our router and just set the <add-pin> directive to be the template:

code/conversion/hybrid/js/app.js

```
39     .state('add', {
40       template: "<add-pin></add-pin>",
41       url: '/add',
42       resolve: {
43         'pins': function(PinsService) {
44           return PinsService.pins();
45         }
46       }
47     })
```

Exposing an ng2 service to ng1

So far we've downgraded ng2 components to use in ng2, and upgraded ng1 services to be used in ng2. But as our application start converting over to ng2, we'll probably start writing services in Typescript/ng2 that we'll want to expose to our ng1 code.

Let's create a simple service in ng2: an "analytics" service that will record events.

The idea is that we have an AnalyticsService in our app that we use to recordEvents. In reality, we're just going to console.log the event and store it in an array. But it gives us a chance to focus on what's important: describing how we share a ng2 service with ng1.

Writing the AnalyticsService

Let's take a look at the AnalyticsService implementation:

code/conversion/hybrid/ts/services/AnalyticsService.ts

```
1 import {Injectable, bind} from '@angular/core';
2
3 /**
4  * Analytics Service records metrics about what the user is doing
5  */
6 @Injectable()
7 export class AnalyticsService {
8   events: string[] = [];
9
10  public recordEvent(event: string): void {
11    console.log(`Event: ${event}`);
12    this.events.push(event);
13  }
14 }
15
16 export var analyticsServiceInjectables: Array<any> = [
17   bind(AnalyticsService).toClass(AnalyticsService)
18 ];
```

There are two things to note here: 1. recordEvent and 2. being Injectable

recordEvent is straightforward: we take an event: string, log it, and store it in events. In your application you would probably send the event to an external service like Google Analytics or Mixpanel.

To make this service injectable, we do two things: 1. Annotate the class with @Injectable and 2. bind the token AnalyticsService to this class.

Now Angular will manage a singleton of this service and we will be able to inject it where we need it.

Downgrade ng2 AnalyticsService to ng1

Before we can use the AnalyticsService in ng1, we need to downgrade it.

The process of downgrading an ng2 service to ng1 is similar to the process of downgrading a directive, but there is one extra step: we need to call `upgradeAdapter.addProvider` before we configure the `angular.factory`:

code/conversion/hybrid/ts/app.ts

```
30 upgradeAdapter.addProvider(AnalyticsService);
31 angular.module('interestApp')
32   .factory('AnalyticsService',
33     upgradeAdapter.downgradeNg2Provider(AnalyticsService));
```

Here we first call `upgradeAdapter.addProvider(AnalyticsService);`, which essentially “registers” the AnalyticsService with the upgradeAdapter.

Next we call `angular.module('interestApp')` to get our ng1 module and then call `.factory` like we would in ng1. To downgrade the service, we call `upgradeAdapter.downgradeNg2Provider(AnalyticsService)`, which wraps our AnalyticsService in a function that adapts it to an ng1 factory.

Using AnalyticsService in ng1

Now we can inject our ng2 AnalyticsService into ng1. Let’s say we want to record whenever the HomeController is visited. We could record this event like so:

code/conversion/hybrid/js/app.js

```
60 .controller('HomeController', function(pins, AnalyticsService) {
61   AnalyticsService.recordEvent('HomeControllerVisited');
62   this.pins = pins;
63 })
```

Here we inject AnalyticsService as if it was a normal ng1 service we call `recordEvent`. Fantastic!

We can use this service anywhere we would use injection in ng1. For instance, we can also inject the AnalyticsService into our ng1 pin directive:

code/conversion/hybrid/js/app.js

```
64 .directive('pin', function(AnalyticsService) {
65   return {
66     restrict: 'E',
67     templateUrl: '/templates/pin.html',
68     scope: {
69       'pin': "=item"
70     },
71     link: function(scope, elem, attrs) {
72       scope.toggleFav = function() {
73         AnalyticsService.recordEvent('PinFaved');
74         scope.pin.faved = !scope.pin.faved;
75       }
76     }
77   }
78 })
```

Summary

Now you have all the tools you need to start upgrading your ng1 app to a hybrid ng1/ng2 app. The interoperability between ng1 and ng2 works very well and we owe a lot to the Angular team for making this so easy.

Being able to exchange directives and services between ng1 and ng2 make it super easy to start upgrading your apps. We can't always upgrade our apps to ng2 overnight, but the UpgradeAdapter lets us start using ng2 - without having to throw our old code away.

References

If you're looking to learn more about hybrid Angular apps, here are a few resources:

- [The Official Angular Upgrade Guide](#)
- [The Angular2 Upgrade Spec Test](#)
- [The Angular2 Source for DowngradeNg2ComponentAdapter](#)

Testing

After spending hours, days, months on a web app you're finally ready to release it to the world. Plenty of hard work and time has been poured into it and now it's time for it to pay off... and then boom: a blocking bug shows up that prevents anyone from signing up.

Test driven?

Testing can help reveal bugs before they appear, instill confidence in your web application, and makes it easy to onboard new developers into the application. There is little doubt about the power of testing amongst the world of software development. However, there is debate about how to go about it.

Is it better to write the tests first and then write the implementation to make those tests pass or would it be better to validate that code that we've already written is correct? It's pretty odd to think that this is a source of contention across the development community, but there is a debate that can get pretty heated as to which is the *right* way to handle testing.

In our experience, particularly when coming from a prototype-heavy background, we focus on building test-able code. Although your experience may differ, we have found that while we are prototyping applications, testing individual pieces of code that are likely to change can double or triple the amount of work it takes to keep them up. In contrast, we focus on building our applications in small components, keeping large amounts of functionality broken into several methods which allows us to test the functionality of a part of the larger picture. This is what we mean when we say *testable* code.



An alternative methodology to prototyping (and then testing after) is called “Red-Green-Refactor”. The idea is that you **write your tests first** and they fail (red) because you haven't written any code yet. Only after you have failing tests do you go on to write your implementation code until it all passes (green).

Of course, the decision of *what* to test is up to you and your team, however we'll focus on *how* to test your applications in this chapter.

End-to-end vs. Unit Testing

There are two major ways to test you applications: *end-to-end testing* or *unit testing*.

If you take a top-down approach on testing you write tests that see the application as a “black box” and you interact with the application like a user would and evaluate if the app seems to work from the “outside”. This top-down technique of testing is called *End to End testing*.



In the Angular world, the tool that is mostly used is called [Protractor](#). Protractor is a tool that opens a browser and interacts with the application, collecting results, to check whether the testing expectations were met.

The second testing approach commonly used is to isolate each part of the application and test it in isolation. This form of testing is called *Unit Testing*.

In Unit Testing we write tests that provide a given input to a given aspect of that unit and evaluate the output to make sure it matches our expectations.

In this chapter we're going to be covering how to **unit test** your Angular apps.

Testing Tools

In order to test our apps, we'll use two tools: Jasmine and Karma.

Jasmine

[Jasmine](#) is a behavior-driven development framework for testing JavaScript code.

Using Jasmine, you can set expectations about what your code should do when invoked.

For instance, let's assume we have a `sum` function on a `Calculator` object. We want to make sure that adding 1 and 1 results in 2. We could express that test (also called a `_spec`), by writing the following code:

```
1 describe('Calculator', () => {
2   it('sums 1 and 1 to 2', () => {
3     var calc = new Calculator();
4     expect(calc.sub(1, 1)).toEqual(2);
5   });
6 });
```

One of the nice things about Jasmine is how readable the tests are. You can see here that we expect the `calc.sub` operation to equal 2.

We organize our tests with `describe` blocks and `it` blocks.

Normally we use `describe` for each logical unit we're testing and inside that each we use one `it` for each expectation you want to assert. However, this isn't a hard and fast rule. You'll often see an `it` block contain several expectations.

On the `calculator` example above we have a very simple object. For that reason, we used one `describe` block for the whole class and one `it` block for each method.

This is not the case most of the times. For example, methods that produce different outcomes depending on the input will probably have more than one `it` block associated. On those cases, it's perfectly fine to have nested `describes`: one for the object and one for each method, and then different assertions inside individual `it` blocks.

We'll be looking at a lot of describe and it blocks throughout this chapter, so don't worry if it isn't clear when to use one vs. the other. We'll be showing lots of examples.

For more information about Jasmine and all its syntax, check out the [Jasmine documentation page](#).

Karma

With Jasmine we can describe our tests and their expectations. Now, in order to actually run the tests we need to have a browser environment.

That's where Karma comes in. Karma allows us to run JavaScript code within a browser like Chrome or Firefox, or on a **headless** browser (or a browser that doesn't expose a user interface) like PhantomJS.

Writing Unit Tests

Our main focus on this section will be to understand how we write unit tests against different parts of our Angular apps.

We're going to learn to test **Services, Components, HTTP requests** and more. Along the way we're also going to see a couple of different techniques to make our code more testable.

Angular Unit testing framework

Angular provides its own set of classes that build upon the Jasmine framework to help writing unit testing for the framework.

The main testing framework can be found on the `@angular/core/testing` package. (Although, for testing components we'll use the `@angular/compiler/testing` package and `@angular/platform-browser/testing` for some other helpers. But more on that later.)



If this is your first time testing Angular I want to prepare you for something: When you write tests for Angular, there is a bit of setup.

For instance, when we have dependencies to inject, we often manually configure them. When we want to test a component, we have to use testing-helpers to initialize them. And when we want to test routing, there are quite a few dependencies we need to structure.

If it feels like there is a lot of setup, don't worry: you'll get the hang of it and find that the setup doesn't change that much from project to project. Besides, we'll walk you through each step in this chapter.

As always, you can find all of the sample code for this chapter in the code download. Looking over the code directly in your favorite editor can provide a good overview of the details we cover in this chapter. We'd encourage you to keep the code open as you go through this chapter.

Setting Up Testing

Earlier in the [Routing Chapter](#) we created an application for searching for music. In this chapter, let's write tests for that application.

Karma requires a configuration in order to run. So the first thing we need to do to setup Karma is to create a `karma.conf.js` file.

Let's `karma.conf.js` file on the root path of our project, like so:

`code/routes/music/karma.conf.js`

```
1 // Karma configuration
2 var path = require('path');
3 var cwd = process.cwd();
4
5 module.exports = function(config) {
6   config.set({
7     // base path that will be used to resolve all patterns (eg. files, exclude)
8     basePath: '',
9
10    // frameworks to use
11    // available frameworks: https://npmjs.org/browse/keyword/karma-adapter
12    frameworks: ['jasmine'],
13
14    // list of files / patterns to load in the browser
15    files: [
16      { pattern: 'test.bundle.js', watched: false }
17    ],
18
19    // list of files to exclude
20    exclude: [
21    ],
22
23    // preprocess matching files before serving them to the browser
24    // available preprocessors: https://npmjs.org/browse/keyword/karma-preprocessor
25    preprocessors: {
26      'test.bundle.js': ['webpack', 'sourcemap']
27    },
28
29    webpack: {
30      devtool: 'inline-source-map',
31      resolve: {
32        root: [path.resolve(cwd)],
33        modulesDirectories: ['node_modules', 'app', 'app/ts', 'test', '.'],
34        extensions: ['', '.ts', '.js', '.css'],
35        alias: {
36          'app': 'app'
37        }
38      },
39    },
40    module: {
41      loaders: [
42        { test: /\.ts$/, loader: 'ts-loader', exclude: [/node_modules/]}
43      ]
44    },
45    stats: {
46      colors: true,
47      reasons: true
48    },
49    watch: true,
50    debug: true
51  },
52
53  webpackServer: {
54    noInfo: true
55  },
56
57  // test results reporter to use
58  // possible values: 'dots', 'progress'
59  // available reporters: https://npmjs.org/browse/keyword/karma-reporter
60  reporters: ['spec'],
61
62
63
64  // web server port
65  port: 9876,
66
67
68  // enable / disable colors in the output (reporters and logs)
```

```

69 colors: true,
70
71
72 // level of logging
73 // possible values: config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WARN\
74 N || config.LOG_INFO || config.LOG_DEBUG
75 logLevel: config.LOG_INFO,
76
77
78 // enable / disable watching file and executing tests whenever any file chan\
79 ges
80 autoWatch: true,
81
82
83 // start these browsers
84 // available browser launchers: https://npmjs.org/browse/keyword/karma-launc\
85 her
86 browsers: ['PhantomJS'],
87
88
89 // Continuous Integration mode
90 // if true, Karma captures browsers, runs the tests and exits
91 singleRun: false
92 })
93 }

```

Don't worry too much about this file's contents right now, just keep in mind a few things about it:

- sets PhantomJS as the target testing browser;
- uses Jasmine karma framework for testing;
- uses a WebPack bundle called `test.bundle.js` that basically wraps all our testing and app code;

The next step is to create a new `test` folder to hold our test files.

```
1 mkdir test
```

Testing Services and HTTP

Services in Angular start out their life as plain classes. In one sense, this makes our services easy to test because we can sometimes test them directly without using Angular.

With Karma configuration done, let's start testing the `SpotifyService` class. If we remember, this service works by interacting with the Spotify API to retrieve album, track and artist information.

Inside the `test` folder, let's create a service subfolder where all our service tests will go. Finally, let's create our first test file inside it, called `SpotifyService.spec.ts`.

Now we can start putting this test file together. The first thing we need to do is import the test helpers from the `@angular/core/testing` package:

```
code/routes/music/test/services/SpotifyService.spec.ts
```

```

1 import {
2   it,
3   describe,
4   expect,
5   inject,
6   fakeAsync,
7   tick,
8   addProviders
9 } from '@angular/core/testing';

```

Next, we'll import a couple more classes:

```
10 import {MockBackend} from '@angular/http/testing';
11 import {provide} from '@angular/core';
12 import {
13   Http,
14   ConnectionBackend,
15   BaseRequestOptions,
16   Response,
17   RequestOptions
18 } from '@angular/http';
```

Since our service uses HTTP requests, we'll import the `MockBackend` class from `@angular/http/testing` package. This class will help us set expectations and verify HTTP requests.

The last thing we need to import is the class we're testing:

code/routes/music/test/services/SpotifyService.spec.ts

```
19 import {SpotifyService} from '../../../app/ts/services/SpotifyService';
```

HTTP Considerations

We could start writing our tests right now, but during each test execution we would be calling out and hitting the Spotify server. This is far from ideal for two reasons:

1. HTTP requests are relatively slow and as our test suite grows, we'd notice it takes longer and longer to run all of the tests.
2. Spotify's API has a quota, and if our whole team is running the tests, we might use up our API call resources needlessly
3. If we are offline or if Spotify is down or inaccessible our tests would start breaking, even though our code might technically be correct

This is a good hint when writing unit tests: isolate everything that you don't control before testing.

In our case, this piece is the Spotify service. The solution is that we will replace the HTTP request with something that would behave like it, but will **not hit the real Spotify server**.

Doing this in the testing world is called *mocking* a dependency. They are sometimes also called *stubbing* a dependency.



You can read more about the difference between Mocks and Stubs in this article [Mocks are not Stubs](#)

Let's pretend we're writing code that depends on a given `Car` class.

This class has a bunch of methods: you can start a car instance, stop it, park it and getSpeed of that car.

Let's see how we could use stubs and mocks to write tests that depend on this class.

Stubs

Stubs are objects we create on the fly, with a subset of the behaviors our dependency has.

Let's write a test that just interacts with the `start` method of the class.

You could create a *stub* of that car class on-the-fly and inject that into the class you're testing:

```
1 describe('Speedtrap', function() {
2   it('tickets a car at more than 60mph', function() {
3     var stubCar = { getSpeed: function() { return 61; } };
4     var speedTrap = new SpeedTrap(stubCar);
5     speedTrap.ticketCount = 0;
6     speedTrap.checkSpeed();
7     expect(speedTrap.ticketCount).toEqual(1);
8   });
9 });
```

This would be a typical case for using a stub and we'd probably only use it locally to that test.

Mocks

Mocks in our case will be a more complete representation of objects, that overrides parts or all of the behavior of the dependency. Mocks can, and most of the time will be reused by more than one test across our suite.

They will also be used sometimes to assert that given methods were called the way they were supposed to be called.

One example of a mock version of our car class would be:

```
1 class MockCar {
2   startCallCount: number = 0;
3
4   start() {
5     this.startCallCount++;
6   }
7 }
```

And it would be used to write another test like this:

```
1 describe('CarRemote', function() {
2   it('starts the car when the start key is held', function() {
3     var car = new MockCar();
4     var remote = new CarRemote();
5     remote.holdButton('start');
6     expect(car.startCallCount).toEqual(1);
7   });
8 });
```

The biggest difference between a mock and a stub is that:

- a stub provides a subset of functionality with “manual” behavior overrides whereas
- a mock generally sets expectations and verifies that certain methods were called

Http MockBackend

Now that we have this background in mind, let's go back to writing our service test code.

Interacting with the live Spotify service every time we run our tests is a poor idea but thankfully Angular provides us with a way to create fake HTTP calls with MockBackend.

This class can be injected into a `Http` instance and gives us control of how we want the HTTP interaction to act. We can interfere and assert in a variety of different ways: we can manually set a response, simulate an HTTP error, and add expectations, like asserting the URL being requested matches what we want, if the provided request parameters are correct and a lot more.

So the idea here is that we're going to provide our code with a "fake" `Http` library. This "fake" library will appear to our code to be the real `Http` library: all of the methods will match, it will return responses and so on. However, we're not *actually* going to make the requests.

In fact, beyond not making the requests, our `MockBackend` will actually allow us to setup *expectations* and watch for behaviors we expect.

addProviders Hooks

One of the things that is complicated at first when testing Angular code is that we do a lot more manual *injection* that we would have to do otherwise.

When testing Angular code, we have to *manually setup injections*. This is good because it gives us more control over what we're actually testing.

So in the case of testing `Http` requests, we don't want to inject the "real" `Http` class, but instead we want to inject something that looks like `Http`, but really intercepts the requests and returns the responses we configure.

To do that, we create a version of the `Http` class that uses `MockBackend` internally.

To do this, we use the `addProviders` in the `beforeEach` hook. This hook takes a callback function that will be called before each test is run, giving us a great opportunity to configure alternative class implementations.

code/routes/music/test/services/SpotifyService.spec.ts

```
22 describe('SpotifyService', () => {
23   beforeEach(() => {
24     addProviders([
25       BaseRequestOptions,
26       MockBackend,
27       SpotifyService,
28       provide(Http, {
29         useFactory: (backend: ConnectionBackend,
30                   defaultOptions: BaseRequestOptions) => {
31           return new Http(backend, defaultOptions);
32         }, deps: [MockBackend, BaseRequestOptions]}),
33     ]);
34   });
```

Notice that `addProviders` accepts an **array of providers** to be used by the test injector.

`BaseRequestOptions` and `SpotifyService` are just the default implementation of those classes. But the last provider is a little more complicated :

code/routes/music/test/services/SpotifyService.spec.ts

```
28   provide(Http, {
```

```
29     useFactory: (backend: ConnectionBackend,  
30                 defaultOptions: BaseRequestOptions) => {  
31     return new Http(backend, defaultOptions);  
32 }, deps: [MockBackend, BaseRequestOptions]]),
```

This code uses `provide` to create a version of the `Http` class, using a factory (that's what `useFactory` does).

That factory has a signature that expects `ConnectionBackend` and a `BaseRequestOptions` instances. The second key on that object is `deps: [MockBackend, BaseRequestOptions]`. That indicates that we'll be using `MockBackend` as the first parameter of the factory and `BaseRequestOptions` (the default implementation) as the second.

Finally, we return our customized `Http` class with the `MockBackend` as a result of that function.

What benefit do we get from this? Well now every time (in our test) that our code requests `Http` as an injection, it will instead receive our customized `Http` instance.

This is a powerful idea that we'll use a lot in testing: use dependency injection to customize dependencies and isolate the functionality you're trying to test.

Testing `getTrack`

Now, when writing tests for the service, we want to verify that we're calling the correct URL.



If you haven't looked at the Routing chapter music example in a while, you can find the [code for this example here](#)

Let's write a test for the `getTrack` method:

code/routes/music/app/ts/services/SpotifyService.ts

```
39 getTrack(id: string): Observable<any[]> {  
40     return this.query(`/tracks/${id}`);  
41 }
```

If you remember how that method works, it uses the `query` method, that builds the URL based on the parameters it receives:

code/routes/music/app/ts/services/SpotifyService.ts

```
19 query(URL: string, params?: Array<string>): Observable<any[]> {  
20     let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;  
21     if (params) {  
22         queryURL = `${queryURL}?${params.join('&' )}`;  
23     }  
24  
25     return this.http.request(queryURL).map((res: any) => res.json());  
26 }
```

Since we're passing `/tracks/${id}` we assume that when calling `getTrack('TRACK_ID')` the expected URL will be `https://api.spotify.com/v1/tracks/TRACK_ID`.

Here is how we write the test for this:

```

1 describe('getTrack', () => {
2   it('retrieves using the track ID',
3     inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend\
4 ) => {
5     var res;
6     mockBackend.connections.subscribe(c => {
7       expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
8       let response = new ResponseOptions({body: '{"name": "felipe"}'});
9       c.mockRespond(new Response(response));
10    });
11    spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
12      res = _res;
13    });
14    tick();
15    expect(res.name).toBe('felipe');
16  })
17 });
18 });

```

This seems like a lot to grasp at first, so let's break it down a bit:

Every time we write tests with dependencies, we need to ask Angular injector to provide us with the instances of those classes. To do that we use:

```

1 inject([Class1, ..., ClassN], (instance1, ..., instanceN) => {
2   ... testing code ...
3 })

```

When you are testing code that returns either a Promise or an RxJS Observable, you can use `fakeAsync` helper to test that code as if it were synchronous. This way every Promises are fulfilled and Observables are notified immediately after you call `tick()`.

So in this code:

```

1 inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend) =>\
2 {
3   ...
4 }));

```

We're getting two variables: `spotifyService` and `mockBackend`. The first one has a concrete instance of the `SpotifyService` and the second is an instance `MockBackend` class. Notice that the arguments to the inner function (`spotifyService`, `mockBackend`) are injections of the classes specified in the first argument array of the `inject` function (`SpotifyService` and `MockBackend`).

We're also running inside `fakeAsync` which means that async code will be run synchronously when `tick()` is called.

Now that we've setup the injections and context for our test, we can start writing our "actual" test. We start by declaring a `res` variable that will eventually get the HTTP call response. Next we subscribe to `mockBackend.connections`:

```

1 var res;
2 mockBackend.connections.subscribe(c => { ... });

```

Here we're saying that whenever a new connection comes in to `mockBackend` we want to be notified (e.g. call this function).

We want to verify that the `SpotifyService` is calling out to the correct URL given the track id `TRACK_ID`. So what we do is specify an *expectation* that the URL is as we would expect. We can get the

URL from the connection `c` via `c.request.url`. So we setup an expectation that `c.request.url` should be the string `'https://api.spotify.com/v1/tracks/TRACK_ID'`:

```
1 expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
```

When our test is run, if the request URL doesn't match, then the test will fail.

Now that we've received our request and verified that it is correct, we need to craft a response. We do this by creating a new `ResponseOptions` instance. Here we specify that it will return the JSON string: `{"name": "felipe"}` as the body of the response.

```
1 let response = new ResponseOptions({body: '{"name": "felipe"}'});
```

Finally, we tell the connection to replace the response with a `Response` object that wraps the `ResponseOptions` instance we created:

```
1 c.mockRespond(new Response(response));
```



An interesting thing to note here is that your callback function in `subscribe` can be as sophisticated as you wish it to be. You could have conditional logic based on the URL, query parameters, or anything you can read from the request object etc.

This allows us to write tests for nearly every possible scenario our code might encounter.

We have now everything setup to call the `getTrack` method with `TRACK_ID` as a parameter and tracking the response in our `res` variable:

```
1 spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
2   res = _res;
3 });
```

If we ended our test here, we would be waiting for the HTTP call to be made and the response to be fulfilled before the callback function would be triggered. It would also happen on a different execution path and we'd have to orchestrate our code to sync things up. Thankfully using `fakeAsync` takes that problem away. All we need to do is call `tick()` and, like magic, our async code will be executed:

```
1 tick();
```

We now perform one final check just to make sure our response we setup is the one we received:

```
1 expect(res.name).toBe('felipe');
```

If you think about it, the code for all the methods of this service are *very* similar. So let's extract the snippet we use to setup the URL expectation into a function called `expectURL`:

code/routes/music/test/services/SpotifyService.spec.ts

```
37 function expectURL(backend: MockBackend, url: string) {
38   backend.connections.subscribe(c => {
39     expect(c.request.url).toBe(url);
40     let response = new ResponseOptions({body: '{"name": "felipe"}'});
41     c.mockRespond(new Response(response));
42   });
43 }
```

Following the same lines, it should be very simple to create similar tests for `getArtist` and `getAlbum` methods:

code/routes/music/test/services/SpotifyService.spec.ts

```
59 describe('getArtist', () => {
60   it('retrieves using the artist ID',
61     inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
62       var res;
63       expectURL(backend, 'https://api.spotify.com/v1/artists/ARTIST_ID');
64       svc.getArtist('ARTIST_ID').subscribe((_res) => {
65         res = _res;
66       });
67       tick();
68       expect(res.name).toBe('felipe');
69     }))
70 );
71 });
72
73 describe('getAlbum', () => {
74   it('retrieves using the album ID',
75     inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
76       var res;
77       expectURL(backend, 'https://api.spotify.com/v1/albums/ALBUM_ID');
78       svc.getAlbum('ALBUM_ID').subscribe((_res) => {
79         res = _res;
80       });
81       tick();
82       expect(res.name).toBe('felipe');
83     }))
84 );
85 });
```

Now `searchTrack` is slightly different: instead of calling `query`, this method uses the `search` method:

code/routes/music/app/ts/services/SpotifyService.ts

```
35 searchTrack(query: string): Observable<any[]> {
36   return this.search(query, 'track');
37 }
```

And then `search` calls `query` with `/search` as the first argument and an Array containing `q=<query>` and `type=track` as the second argument:

code/routes/music/app/ts/services/SpotifyService.ts

```
28 search(query: string, type: string): Observable<any[]> {
29   return this.query(`/search`, [
30     `q=${query}`,
31     `type=${type}`
32   ]);
33 }
```

Finally, `query` will transform the parameters into a URL *path* with a *QueryString*. So now, the URL we expect to call ends with `/search?q=<query>&type=track`.

Let's now write the test for `searchTrack` that takes into consideration what we learned above:

code/routes/music/test/services/SpotifyService.spec.ts

```
87 describe('searchTrack', () => {
88   it('searches type and term',
89     inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
90       var res;
91       expectURL(backend, 'https://api.spotify.com/v1/search?q=TERM&type=track');
92     });
93     svc.searchTrack("TERM").subscribe((_res) => {
94       res = _res;
95     });
96     tick();
97     expect(res.name).toBe('felipe');
```

```
98     })))
99   });
100  });
```

The test ended up also being very similar to the ones we wrote so far. Let's review what this test does:

- it hooks into the HTTP lifecycle, by adding a callback when a new HTTP connection is initiated
- it sets an expectation for the URL we expect the connection to use including the query type and the search term
- it calls the method we're testing, `searchTrack`
- it then tells Angular to complete all the pending async calls
- it finally asserts that we have the expected response

In essence, when testing services our goals should be:

1. Isolate all the dependencies by using stubs or mocks
2. In case of async calls, use `fakeAsync` and `tick` to make sure they are fulfilled
3. Call the service method you're testing
4. Assert that the returning value from the method matches what we expect

Now let's move on to the classes that usually consume the services: components.

Testing Routing to Components

When testing components, we can either:

1. write tests that will interact with the component from the outside, passing attributes in and checking how the markup is affected or
2. test individual component methods and their output.

Those test strategies are known as **black box** and **white box** testing, respectively. During this section, we'll see a mix of both.

We'll begin by writing tests for the `ArtistComponent` class, which is one of the simpler components we have. This initial set of tests will test the component's internals, so it falls into the **white box** category of testing.

Before we jump into it, let's remember what `ArtistComponent` does:

The first thing we do on the class constructor is retrieve the **id** from the `routeParams` collection:

```
code/routes/music/app/ts/components/ArtistComponent.ts
33 constructor(public route: ActivatedRoute, public spotify: SpotifyService,
34             public location: Location) {
35   route.params.subscribe(params => { this.id = params['id']; });
36 }
```

And with that we have our first obstacle. How can we retrieve the ID of a route without an available running router?

Creating a Router for Testing

Remember that when we write tests in Angular we manually configure many of the classes that are injected. Routing (and testing components) has a daunting number of dependencies that we need to inject. That said, once it's configured, it isn't something we change very much and it's very easy to use.

When we test write tests it's often convenient to use `beforeEach` with `addProviders` to set the dependencies that can be injected. In the case of testing our `ArtistComponent` we're going to create a custom function that will create and configure our router for testing:

code/routes/music/test/components/ArtistComponent.spec.ts

```
20 describe('ArtistComponent', () => {
21   beforeEach(() => {
22     addProviders(musicTestProviders());
23   });
```

We define `musicTestProviders` in the helper file `MusicTestHelpers.ts`. Let's look at that now.

Hold your breath, here's the implementation of `musicTestProviders`. Don't worry, we'll explain each part:

code/routes/music/test/MusicTestHelpers.ts

```
72 export function musicTestProviders() {
73   const mockSpotifyService: MockSpotifyService = new MockSpotifyService();
74
75   return [
76     RouterOutletMap,
77     {provide: UrlSerializer, useClass: DefaultUrlSerializer},
78     {provide: Location, useClass: SpyLocation},
79     {provide: LocationStrategy, useClass: HashLocationStrategy},
80     {provide: PlatformLocation, useClass: BrowserPlatformLocation},
81     {
82       provide: Router,
83       useFactory: (resolver: ComponentResolver, urlSerializer: UrlSerializer,
84                 outletMap: RouterOutletMap, location: Location,
85                 injector: Injector) => {
86         return new Router(
87           RootCmp, resolver, urlSerializer, outletMap,
88           location, injector, routerConfig);
89       },
90       deps: [
91         ComponentResolver,
92         UrlSerializer,
93         RouterOutletMap,
94         Location,
95         Injector
96       ]
97     },
98     {
99       provide: ActivatedRoute,
100      useFactory: (r: Router) => r.routerState.root, deps: [Router]
101     },
102     mockSpotifyService.getProviders(),
103   ];
104 };
```

If you look closely, you'll see we're returning an array with **eight providers**:

- RouterOutletMap
- UrlSerializer
- Location
- LocationStrategy
- PlatformLocation
- The Router (an important one)

- `ActivatedRoute`
- `The MockSpotifyService` (via `mockSpotifyService.getProviders()`)

`RouterOutletMap`

The `RouterOutletMap` lets us know what names map to which `router-outlet` directive (this is especially useful for multiple, named routes).

`UrlSerializer`

The `UrlSerializer` will parse the URL string into an object, to make it easy to use in code. (`UrlSerializer` also does the reverse and converts objects/code into the URL string.) It's used by the Router to work with URLs.

`Location`

The next provider is `Location` but notice that we use the class `SpyLocation`. We'll talk more about spies, but essentially spies allow you to watch an object and ensure a value was called. We'll use this `SpyLocation` to make assertions about the current location (think roughly "the current URL") of the router.

It's useful to stop and take note of how these providers work. When we say `provide: Location`, the `Location` class is treated as a *token*. When we use `useClass` we're able to pass a substitute class (in this case `SpyLocation`), **even when the client asked for the injection of `Location`!**

`LocationStrategy`

The `LocationStrategy` manages reading and writing the route from the URL. Two common `LocationStrategies` are `HashLocationStrategy` and `PathLocationStrategy`.

The `HashLocationStrategy` will use "anchor-hash" routing like `http://ng-book.com/#/foo/bar`. Whereas the `PathLocationStrategy` is used for "HTML5"-style routing like `http://ng-book.com/foo/bar`.

`PlatformLocation`

We're digging into the weeds a bit by using `PlatformLocation`, but suffice it to say that the Router is designed to work across many platforms and not just the browser. By implementing a browser-specific platform location in `BrowserPlatformLocation` Angular leaves the door open to support other platforms in the future.

`Router`

The next provider is `Router`. Again the `provide: Router` option means "when something asks for an injection of `Router`" - and what do we give them? In this case we are using `useFactory`, which is a fancy way of saying "call this function and inject whatever I return."

Notice that the factory function can take injections itself! Of course, it is possible to encounter loading-order issues when your dependencies require dependencies of their own. To deal with this we use the `deps` key. There you can see we specify what dependencies our `Router` injection has.

One thing we haven't talked about yet is what routes we want to use when testing. There are many different ways of doing this. First we'll look at what we're doing here:

[code/routes/music/test/MusicTestHelpers.ts](#)

```

36 @Component({
37   selector: 'blank-cmp',
38   template: ``,
39   directives: [ROUTER_DIRECTIVES]
40 })
41 export class BlankCmp {
42 }
43
44 @Component({
45   selector: 'root-cmp',
46   template: `<router-outlet></router-outlet>`,
47   directives: [ROUTER_DIRECTIVES],
48   precompile: [BlankCmp, SearchComponent, ArtistComponent,
49               TrackComponent, AlbumComponent]
50 })
51 export class RootCmp {
52 }
53
54 export function createRoot(tcb: TestComponentBuilder,
55                             router: Router,
56                             type: any): ComponentFixture<any> {
57   const f = tcb.createFakeAsync(type);
58   advance(f);
59   (<any>router).initialNavigation();
60   advance(f);
61   return f;
62 }
63
64 export const routerConfig: RouterConfig = [
65   { path: '', component: BlankCmp },
66   { path: 'search', component: SearchComponent },
67   { path: 'artists/:id', component: ArtistComponent },
68   { path: 'tracks/:id', component: TrackComponent },
69   { path: 'albums/:id', component: AlbumComponent }
70 ];

```

Here instead of redirecting (like we do in the real router config) for the empty URL, we're just using BlankCmp.

Of course, if you want to use the same RouterConfig as in your top-level app then all you need to do is export it somewhere and import it here.

If you have a more complex scenario where you need to test lots of different route configurations, you could even accept a parameter to the musicTestProviders function where you use a new router configuration each time.

There are many possibilities here and you'll need to pick whichever fits best for your team. This configuration works for cases where your routes are relatively static and one configuration works for all of the tests.

Now that we have all of the dependencies, we create the new Router and call `r.initialNavigation()` on it.

ActivatedRoute

The ActivatedRoute service keeps track of the "current route". It requires the Router itself as a dependency so we put it in deps and inject it.

MockSpotifyService

Earlier we tested our SpotifyService by mocking out the HTTP library that backed it. Instead here, we're going to **mock out the whole service itself**. Let's look at how we can mock out this, or any, service.

Mocking dependencies

If you look inside `music/test` you'll find a `mocks/spotify.ts` file. Let's take a look:

`code/routes/music/test/mocks/spotify.ts`

```
1 import {provide} from '@angular/core';
2 import {SpyObject} from './helper';
3 import {SpotifyService} from '../../app/ts/services/SpotifyService';
4
5 export class MockSpotifyService extends SpyObject {
6   getAlbumSpy;
7   getArtistSpy;
8   getTrackSpy;
9   searchTrackSpy;
10  mockObservable;
11  fakeResponse;
```

Here we're declaring the `MockSpotifyService` class, which will be a mocked version of the real `SpotifyService`. These instance variables will act as *spies*.

Spies

A *spy* is a specific type of mock object that gives us two benefits:

1. we can simulate return values and
2. count how many times the method was called and with which parameters.

In order to use spies with Angular, we're using the internal `SpyObject` class (it's used by Angular to test itself).

You can either declare a class by creating a new `SpyObject` on the fly or you can make your mock class inherit from `SpyObject`, like we're doing in our code.

The great thing inheriting or using this class gives us is the `spy` method. The `spy` method lets us override a method and force a return value (as well as watch and ensure the method was called). We use `spy` on our class constructor:

`code/routes/music/test/mocks/spotify.ts`

```
13 constructor() {
14   super(SpotifyService);
15
16   this.fakeResponse = null;
17   this.getAlbumSpy = this.spy('getAlbum').andReturn(this);
18   this.getArtistSpy = this.spy('getArtist').andReturn(this);
19   this.getTrackSpy = this.spy('getTrack').andReturn(this);
20   this.searchTrackSpy = this.spy('searchTrack').andReturn(this);
21 }
```

The first line of the constructor call's the `SpyObject` constructor, passing the concrete class we're mocking. Calling `super(...)` is optional, but when you do the mock class will inherit all the concrete class methods, so you can override just the pieces you're testing.



If you're curious about how `SpyObject` is implemented you can check it on the [angular/angular repository, on the file /modules/angular2/src/testing/testing_internal.ts](#)

After calling `super`, we're initializing the `fakeResponse` field, that we'll use later to `null`.

Next we declare spies that will replace the concrete class methods. Having a reference to them will be helpful to set expectations and simulate responses while writing our tests.

When we use the `SpotifyService` within the `ArtistComponent`, the real `getArtist` method returns an `Observable` and the method we're calling from our components is the `subscribe` method:

```
code/routes/music/app/ts/components/ArtistComponent.ts
38  ngOnInit(): void {
39    this.spotify
40      .getArtist(this.id)
41      .subscribe((res: any) => this.renderArtist(res));
42  }
```

However, in our mock service, we're going to do something tricky: instead of returning an observable from `getArtist`, we're returning `this`, the `MockSpotifyService` itself. That means the return value of `this.spotify.getArtist(this.id)` above will be the `MockSpotifyService`.

There's one problem with doing this though: our `ArtistComponent` was expecting to call `subscribe` on an `Observable`. To account for this, we're going to define `subscribe` on our `MockSpotifyService`:

```
code/routes/music/test/mocks/spotify.ts
23  subscribe(callback) {
24    callback(this.fakeResponse);
25  }
```

Now when `subscribe` is called on our mock, we're immediately calling the callback, making the async call happen synchronously.

The other thing you'll notice is that we're calling the callback function with `this.fakeResponse`. This leads us to the next method:

```
code/routes/music/test/mocks/spotify.ts
27  setResponse(json: any): void {
28    this.fakeResponse = json;
29  }
```

This method doesn't replace anything on the concrete service, but is instead a helper method to allow the test code to set a given response (that would come from the service on the concrete class) and with that simulate different responses.

```
code/routes/music/test/mocks/spotify.ts
31  getProviders(): Array<any> {
32    return [provide(SpotifyService, {useValue: this})];
33  }
```

This last method is a helper method to be used in `addProviders` like we'll see later when we get back to writing component tests.

Here's what our `MockSpotifyService` looks like altogether:

```
code/routes/music/test/mocks/spotify.ts
1  import {provide} from '@angular/core';
2  import {SpyObject} from './helper';
3  import {SpotifyService} from '../../app/ts/services/SpotifyService';
```

```

4 export class MockSpotifyService extends SpyObject {
5   getAlbumSpy;
6   getArtistSpy;
7   getTrackSpy;
8   searchTrackSpy;
9   mockObservable;
10  fakeResponse;
11
12
13  constructor() {
14    super(SpotifyService);
15
16    this.fakeResponse = null;
17    this.getAlbumSpy = this.spy('getAlbum').andReturn(this);
18    this.getArtistSpy = this.spy('getArtist').andReturn(this);
19    this.getTrackSpy = this.spy('getTrack').andReturn(this);
20    this.searchTrackSpy = this.spy('searchTrack').andReturn(this);
21  }
22
23  subscribe(callback) {
24    callback(this.fakeResponse);
25  }
26
27  setResponse(json: any): void {
28    this.fakeResponse = json;
29  }
30
31  getProviders(): Array<any> {
32    return [provide(SpotifyService, {useValue: this})];
33  }
34 }

```

Back to Testing Code

Now that we have all our dependencies under control, it is easier to write our tests. Let's write our test for our ArtistComponent.

As usual, we start with imports:

code/routes/music/test/components/ArtistComponent.spec.ts

```

1 import {
2   it,
3   describe,
4   inject,
5   fakeAsync,
6   addProviders
7 } from '@angular/core/testing';
8 import { Router } from '@angular/router';
9 import { Location } from '@angular/common';
10 import { TestComponentBuilder } from '@angular/compiler/testing';
11 import { MockSpotifyService } from '../mocks/spotify';
12 import { SpotifyService } from '../../../app/ts/services/SpotifyService';
13 import {
14   musicTestProviders,
15   advance,
16   createRoot,
17   RootCmp
18 } from '../MusicTestHelpers';

```

Next we can start to describe our tests and we use addProviders to make sure we use our musicTestProviders in each test:

code/routes/music/test/components/ArtistComponent.spec.ts

```

20 describe('ArtistComponent', () => {
21   beforeEach(() => {
22     addProviders(musicTestProviders());
23   });

```

Next, we'll write a test for everything that happens during the initialization of the component. First, let's take a refresh look at what happens on initialization of our `ArtistComponent`:

code/routes/music/app/ts/components/ArtistComponent.ts

```
29 export class ArtistComponent implements OnInit {
30   id: string;
31   artist: Object;
32
33   constructor(public route: ActivatedRoute, public spotify: SpotifyService,
34               public location: Location) {
35     route.params.subscribe(params => { this.id = params['id']; });
36   }
37
38   ngOnInit(): void {
39     this.spotify
40       .getArtist(this.id)
41       .subscribe((res: any) => this.renderArtist(res));
42   }
```

Remember that during the creation of the component, we use `route.params` to retrieve the current route `id` param and store it on the `id` attribute of the class.

When the component is initialized `ngOnInit` is triggered by Angular (because we declared that this component implements `OnInit`). We then use the `SpotifyService` to retrieve the artist for the received `id`, and we subscribe to the returned observable. When the artist is finally retrieved, we call `renderArtist`, passing the artist data.

An important idea here is that we used dependency injection to get the `SpotifyService`, but remember, **we created a `MockSpotifyService`!**

So in order to test this behavior, let's:

1. Use our router to navigate to the `ArtistComponent`, which will initialize the component
2. Check our `MockSpotifyService` and ensure that the `ArtistComponent` did, indeed, try to get the artist with the appropriate `id`.

Here's the code for our test:

code/routes/music/test/components/ArtistComponent.spec.ts

```
25 describe('initialization', () => {
26   it('retrieves the artist', fakeAsync(
27     inject([Router, SpotifyService, TestComponentBuilder],
28       (router: Router,
29         mockSpotifyService: MockSpotifyService,
30         tcb: TestComponentBuilder) => {
31       const fixture = createRoot(tcb, router, RootCmp);
32
33       router.navigateByUrl('/artists/2');
34       advance(fixture);
35
36       expect(mockSpotifyService.getArtistSpy).toHaveBeenCalled('2');
37     })));
38 });
```

Let's take it step by step.

fakeAsync and advance

We start by wrapping the test in `fakeAsync`. Without getting too bogged down in the details, by using `fakeAsync` we're able to have more control over when change detection and asynchronous operations occur. A consequence of this is that we need to explicitly tell our components that they need to detect changes after we make changes in our tests.

Normally you don't need to worry about this when writing your apps, as zones tend to do the right thing, but during tests we manipulate the change detection process more carefully.

If you skip a few lines down you'll notice that we're using a function called `advance` that comes from our `MusicTestHelpers`. Let's take a look at that function:

```
code/routes/music/test/MusicTestHelpers.ts
31 export function advance(fixture: ComponentFixture<any>): void {
32   tick();
33   fixture.detectChanges();
34 }
```

So we see here that `advance` does two things:

1. It tells the component to detect changes and
2. Calls `tick()`

When we use `fakeAsync`, timers are actually synchronous and we use `tick()` to simulate the asynchronous passage of time.

Practically speaking, in our tests we'll call `advance` whenever we want Angular to "work its magic". So for instance, whenever we navigate to a new route, update a form element, make an HTTP request etc. we'll call `advance` to give Angular a chance to do its thing.

inject

In our test we need some dependencies. We use `inject` to get them. The `inject` function takes two arguments:

1. An array of *tokens* to inject
2. A function into which to provide the injections

And what classes will `inject` use? The providers we defined in `addProviders`.

Notice that we're injecting:

1. Router
2. SpotifyService
3. TestComponentBuilder

The Router that will be injected is the Router we configured in `musicTestProviders` above.

For `SpotifyService`, notice that we're requesting injection of the *token* `SpotifyService`, but we're receiving a `MockSpotifyService`. A little tricky, but hopefully it makes sense given what we've talked about so far.

The other thing we inject is the `TestComponentBuilder`. This aptly-named service helps you build components for testing. We're going to use it to create the root component (which has the router attached to it).

Testing ArtistComponent's Initialization

Let's review the contents of our actual test:

code/routes/music/test/components/ArtistComponent.spec.ts

```
31     const fixture = createRoot(tcb, router, RootCmp);
32
33     router.navigateByUrl('/artists/2');
34     advance(fixture);
35
36     expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
```

We start by creating an instance of our `RootCmp` by using `createRoot`. Let's look at the `createRoot` helper function:

code/routes/music/test/MusicTestHelpers.ts

```
54 export function createRoot(tcb: TestComponentBuilder,
55                             router: Router,
56                             type: any): ComponentFixture<any> {
57     const f = tcb.createFakeAsync(type);
58     advance(f);
59     (<any>router).initialNavigation();
60     advance(f);
61     return f;
62 }
```

Notice here that when we call `createRoot` we

1. Create an instance of the root component
2. advance it
3. Tell the router to setup its `initialNavigation`
4. advance again
5. return the new root component.

This is something we'll do a lot when we want to test a component that depends on routing, so it's handy to have this helper function around.



`RootCmp` is an empty component that we created in `MusicTestHelpers`. You definitely don't need to create an empty component for your root component, but I like to do it this way because it lets us test our child component (`ArtistComponent`) more-or-less in isolation. That is, we don't have to worry about the effects of the parent app component.

That said, maybe you *want* to make sure that the child component operates correctly in context. In that case instead of using `RootCmp` you'd probably want to use your app's normal parent component.

Next we use `router` to navigate to the url `/artists/2` and `advance`. When we navigate to that URL, `ArtistComponent` should be initialized, so we assert that the `getArtist` method of the `SpotifyService` was called with the proper value.

Testing ArtistComponent Methods

Recall that the `ArtistComponent` has an `href` which calls the `back()` function.

```
code/routes/music/app/ts/components/ArtistComponent.ts
```

```
44 back(): void {
45   this.location.back();
46 }
```

Let's test that when the `back` method is called, the router will redirect the user back to the previous location.

The current location state is controlled by the `Location` service. When we need to send the user back to the previous location, we use the `Location`'s `back` method.

Here is how we test the `back` method:

```
code/routes/music/test/components/ArtistComponent.spec.ts
```

```
40 describe('back', () => {
41   it('returns to the previous location', fakeAsync(
42     inject([Router, TestComponentBuilder, Location],
43       (router: Router, tcb: TestComponentBuilder, location: Location) => {
44         const fixture = createRoot(tcb, router, RootCmp);
45         expect(location.path()).toEqual('/');
46
47         router.navigateByUrl('/artists/2');
48         advance(fixture);
49         expect(location.path()).toEqual('/artists/2');
50
51         const artist = fixture.debugElement.children[1].componentInstance;
52         artist.back();
53         advance(fixture);
54
55         expect(location.path()).toEqual('/');
56       }));
57 });
```

The initial structure is similar: we inject our dependencies and create a new component.

We have a new expectation - we assert that the `location.path()` is equal to what we expect it to be.

We also have another new idea: we're accessing the methods on the `ArtistComponent` itself. We get a reference to our `ArtistComponent` instance through the line `fixture.debugElement.children[1].componentInstance`.

Now that we have the instance of the component, we're able to call methods on it directly, like `back()`.

After we call `back()` we advance and then verify that the `location.path()` is what we expected it to be.

Testing ArtistComponent DOM Template Values

The last thing we need to test on `ArtistComponent` is the template that renders the artist.

```
code/routes/music/app/ts/components/ArtistComponent.ts
```

```
17 template: `
18 <div *ngIf="artist">
19   <h1>{{ artist.name }}</h1>
20
21   <p>
```

```
22     
23 </p>
24
25 <p><a href (click)="back()">Back</a></p>
26 </div>
27 `
```

Remember that the instance variable `artist` is set by the result of the `SpotifyService` `getArtist` call. Since we're mocking the `SpotifyService` with `MockSpotifyService`, the data we should have in our template should be whatever the `mockSpotifyService` returns. Let's look at how we do this:

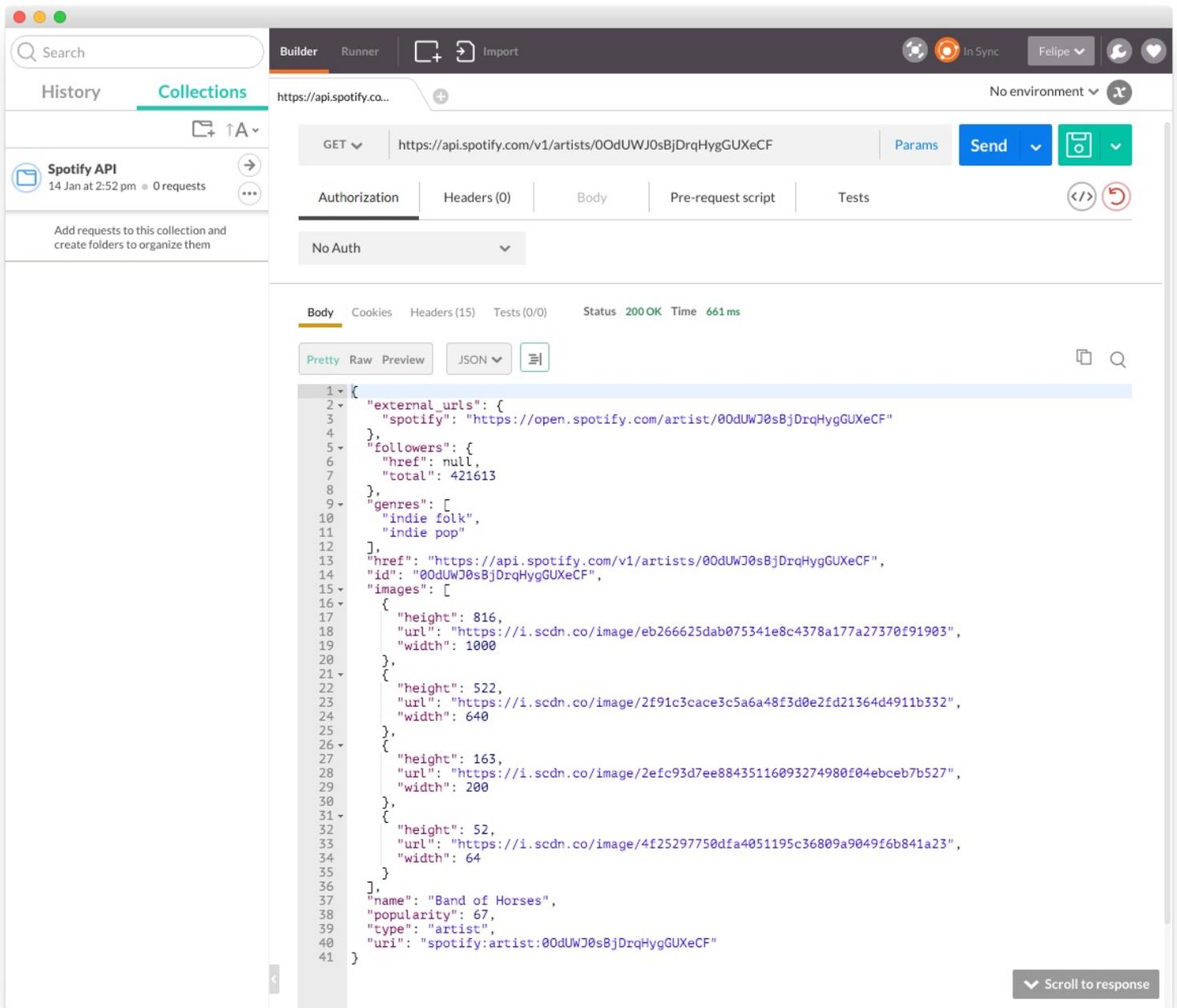
code/routes/music/test/components/ArtistComponent.spec.ts

```
59 describe('renderArtist', () => {
60   it('renders album info', fakeAsync(
61     inject([Router, TestComponentBuilder, SpotifyService],
62       (router: Router, tcb: TestComponentBuilder,
63         mockSpotifyService: MockSpotifyService) => {
64       const fixture = createRoot(tcb, router, RootCmp);
65
66       let artist = {name: 'ARTIST NAME', images: [{url: 'IMAGE_1'}]};
67       mockSpotifyService.setResponse(artist);
68
69       router.navigateByUrl('/artists/2');
70       advance(fixture);
71
72       const compiled = fixture.debugElement.nativeElement;
73
74       expect(compiled.querySelector('h1').innerHTML).toContain('ARTIST NAME');
75       expect(compiled.querySelector('img').src).toContain('IMAGE_1');
76     }));
77 });
```

The first thing that's new here is that we're *manually setting the response* of the `mockSpotifyService` with `setResponse`.

The `artist` variable is a *fixture* that represents what we get from the Spotify API when we call the `artists` endpoint at GET `https://api.spotify.com/v1/artists/{id}`.

Here's what the real JSON looks like:



Postman - Spotify Get Artist Endpoint

However, for this test we need only the name and images properties.

When we call the `setResponse` method, that response will be used for the next call we make to any of the service methods. In this case, we want the method `getArtist` to return this response.

Next we navigate with the router and advance. Now that the view is rendered, we can use the DOM representation of the component's view to check if the artist was properly rendered.

We do that by getting the `nativeElement` property of the `DebugElement` with the line `fixture.debugElement.nativeElement`.

In our assertions, we expect to see H1 tag containing the artist's name, in our case the string `ARTIST NAME` (because of our artist fixture above).

To check those conditions, we use the `NativeElement`'s `querySelector` method. This method will return the first element that matches the provided CSS selector.

For the H1 we check that the text is indeed `ARTIST NAME` and for the image, we check its `src` property is `IMAGE 1`.

With this, we are done testing the `ArtistComponent` class.

Testing Forms

To write form tests, let's use the `DemoFormNgModel` component we created [back in the Forms chapter](#). This example is a good candidate because it uses a few features of Angular's forms:

- it uses a `FormBuilder`
- has validations
- handles events

Here's the full code for that class:

code/forms/app/forms/demo_form_with_events.ts

```
1 import { Component } from '@angular/core';
2 import {
3   FORM_DIRECTIVES,
4   REACTIVE_FORM_DIRECTIVES,
5   FormBuilder,
6   FormGroup,
7   Validators,
8   AbstractControl
9 } from '@angular/forms';
10
11 @Component({
12   selector: 'demo-form-with-events',
13   directives: [FORM_DIRECTIVES, REACTIVE_FORM_DIRECTIVES],
14   template: `
15     <div class="ui raised segment">
16       <h2 class="ui header">Demo Form: with events</h2>
17       <form [formGroup]="myForm"
18         (ngSubmit)="onSubmit(myForm.value)"
19         class="ui form">
20
21         <div class="field"
22           [class.error]="!sku.valid && sku.touched">
23           <label for="skuInput">SKU</label>
24           <input type="text"
25             class="form-control"
26             id="skuInput"
27             placeholder="SKU"
28             [formControl]="sku">
29           <div *ngIf="!sku.valid"
30             class="ui error message">SKU is invalid</div>
31           <div *ngIf="sku.hasError('required')"
32             class="ui error message">SKU is required</div>
33         </div>
34
35         <div *ngIf="!myForm.valid"
36           class="ui error message">Form is invalid</div>
37
38         <button type="submit" class="ui button">Submit</button>
39       </form>
40     </div>
41 `
42 })
43 export class DemoFormWithEvents {
44   myForm: FormGroup;
45   sku: AbstractControl;
46
47   constructor(fb: FormBuilder) {
```

```

48   this.myForm = fb.group({
49     'sku': ['', Validators.required]
50   });
51
52   this.sku = this.myForm.controls['sku'];
53
54   this.sku.valueChanges.subscribe(
55     (value: string) => {
56       console.log('sku changed to:', value);
57     }
58   );
59
60   this.myForm.valueChanges.subscribe(
61     (form: any) => {
62       console.log('form changed to:', form);
63     }
64   );
65 }
66 }
67
68 onSubmit(form: any): void {
69   console.log('you submitted value:', form.sku);
70 }
71 }

```

Just to recap, this code will have the following behavior:

- when no value is present for the SKU field, two validation error will be displayed: *SKU is invalid* and *SKU is required*
- when the value of the SKU field changes, we are logging a message to the console
- when the form changes, we are also logging to the console
- when the form is submitted, we log yet another final message to the console

It seems that one obvious external dependency we have is the console. As we learned before, we need to somehow mock all external dependencies.

Creating a consoleSpy

This time, instead of using a SpyObject to create a mock, let's do something simpler, since all we're using from the console is the log method.

We will replace the original console instance, that is held on the window.console object and replace by an object we control: a ConsoleSpy.

code/forms/test/util.ts

```

10 export class ConsoleSpy {
11   public logs: string[] = [];
12   log(...args) {
13     this.logs.push(args.join(' '));
14   }
15   warn(...args) {
16     this.log(...args);
17   }
18 }

```

The ConsoleSpy is an object that will take whatever is logged, naively convert it to a string, and store it in an internal list of things that were logged.



To accept a variable number of arguments on our version of the `console.log` method, we are using ES6 and TypeScript's [Rest parameters](#).

This operator, represented by an ellipsis, like `...theArgs` as our function argument. In a nutshell using it indicates that we're going to capture all the remaining arguments from that point on. If we had something like `(a, b, ...theArgs)` and called `func(1, 2, 3, 4, 5)`, `a` would be 1, `b` would be 2 and `theArgs` would have `[3, 4, 5]`.

You can play with it yourself if you have a recent version of [Node.js](#) installed:

```
1 $ node --harmony
2 > var test = (a, b, ...theArgs) => console.log('a=', a, 'b=', b, 'theArgs=', theArgs);
3 undefined
4 > test(1, 2, 3, 4, 5);
5 a= 1 b= 2 theArgs= [ 3, 4, 5 ]
```

So instead of writing it to the console itself, we'll be storing them on an array. If the code under test calls `console.log` three times:

```
1 console.log('First message', 'is', 123);
2 console.log('Second message');
3 console.log('Third message');
```

We expect the `_logs` field to have an array of `['First message is 123', 'Second message', 'Third message']`.

Installing the `consoleSpy`

To use our spy in our test we start by declaring two variables: `originalConsole` will keep a reference to the original console instance, and `fakeConsole` that will hold the *mocked* version of the console. We also declare a few variables that will be helpful in testing our input and form elements.

code/forms/test/forms/demo_form_with_events.spec.ts

```
26 describe('DemoFormWithEvents', () => {
27   let originalConsole, fakeConsole;
28   let el, input, form;
```

And then we can install the fake console and specify our providers:

code/forms/test/forms/demo_form_with_events.spec.ts

```
30 beforeEach(() => {
31   // replace the real window.console with our spy
32   fakeConsole = new ConsoleSpy();
33   originalConsole = window.console;
34   (<any>window).console = fakeConsole;
35
36   addProviders([
37     disableDeprecatedForms(),
38     provideForms(),
39     FormBuilder
40   ]);
41 });
```

Back to the testing code, the next thing we need to do is replace the real console instance with ours, saving the original instance.

Finally, on the `afterAll` method, we restore the original console instance to make sure it doesn't *leak* into other tests.

```
code/forms/test/forms/demo_form_with_events.spec.ts
```

```
43 // restores the real console
44 afterAll(() => (<any>window).console = originalConsole);
```

Now that we have control of the console, let's begin testing our form.

Testing The Form

Now we need to test the validation errors and the events of the form.

The first thing we need to do is to get the references to the SKU input field and to the form elements:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
46 it('validates and triggers events', inject([TestComponentBuilder],
47     fakeAsync((tcb) => {
48         tcb.createAsync(DemoFormWithEvents)
49             .then((fixture) => {
50             let e1 = fixture.debugElement.nativeElement;
51             let input = fixture.debugElement.query(By.css('input')).nativeElement;
52             let form = fixture.debugElement.query(By.css('form')).nativeElement;
53             fixture.detectChanges();
```

The last line tells Angular to commit all the pending changes, similar to what we did in the routing section above. Next, we will set the SKU input value to the empty string:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
55     input.value = '';
56     dispatchEvent(input, 'input');
57     fixture.detectChanges();
58     tick();
```

Here we use `dispatchEvent` to notify Angular that the input element changed, and then we trigger the change detection a second time. Finally we use `tick()` to make sure all asynchronous code triggered up to this point gets executed.

The reason we are using `fakeAsync` and `tick` on this test, is to assure the form events are triggered. If we used `async` and `inject` instead, we would finish the code before the events were triggered.

Now that we have changed the input value, let's make sure the validation is working. We ask the component element (using the `e1` variable) for all child elements that are error messages and then making sure we have both error messages displayed:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
61     let msgs = e1.querySelectorAll('.ui.error.message');
62     expect(msgs[0].innerHTML).toContain('SKU is invalid');
63     expect(msgs[1].innerHTML).toContain('SKU is required');
```

Next, we will do something similar, but this time we set a value to the SKU field:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
67     dispatchEvent(input, 'input');
68     fixture.detectChanges();
69     tick();
```

And make sure all the error messages are gone:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
71     msgs = e1.querySelectorAll('.ui.error.message');
72     expect(msgs.length).toEqual(0);
```

Finally, we will trigger the submit event of the form:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
74     fixture.detectChanges();
75     dispatchEvent(form, 'submit');
76     tick();
```

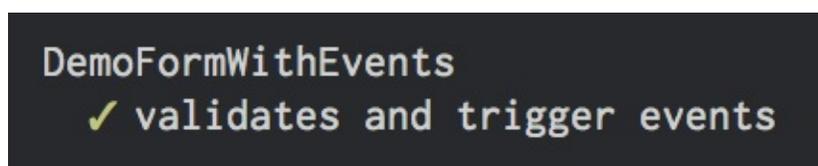
And finally we make sure the event was kicked by checking that the message we log to the console when the form is submitted is there:

```
code/forms/test/forms/demo_form_with_events_bad.spec.ts
```

```
78     // checks for the form submitted message
79     expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

We could continue and add new verifications for the other two events our form triggers: the SKU change and the form change events. However, our test is growing quite long.

When we run our tests, we see it passes:



DemoFormWithEvents test output

This test works, but stylistically we have some code smells:

- a really long it condition (more than 5-10 lines)
- more than one or two expects per it condition
- the word **and** on the test description

Refactoring Our Form Test

Let's fix that by first extracting the code that creates the component and gets the component element and also the elements for the input and for the form:

```
code/forms/test/forms/demo_form_with_events.spec.ts
```

```
46 function createComponent(tcb: TestComponentBuilder):
47     Promise<ComponentFixture<any>> {
48     return tcb.createAsync(DemoFormWithEvents)
49         .then((fixture) => {
50         e1 = fixture.debugElement.nativeElement;
51         input = fixture.debugElement.query(By.css('input')).nativeElement;
52         form = fixture.debugElement.query(By.css('form')).nativeElement;
53         fixture.detectChanges();
54
55         return fixture;
56     });
57 }
```

The `createComponent` code is pretty straightforward: it receives the `TestComponentBuilder` instance, creates the component, retrieves all the elements we need and calls `detectChanges`.

Since this code returns a `Promise`, it can be composed with another `Promise` to chain them together, making one run right after the previous one is resolved.

That's what our tests will do:

```
1 var promise = createComponent(tcb); // => this returns a Promise
2 promise.then((fixture) => ...);    // => when promise is resolved, we run our\
3 test code
```

Now the first thing we want to test is that given an empty SKU field, we should see two error messages:

```
code/forms/test/forms/demo_form_with_events.spec.ts
59 it('displays errors with no sku',
60   async(inject([TestComponentBuilder], (tcb) => {
61     return createComponent(tcb).then((fixture) => {
62       input.value = '';
63       dispatchEvent(input, 'input');
64       fixture.detectChanges();
65
66       // no value on sku field, all error messages are displayed
67       let msgs = el.querySelectorAll('.ui.error.message');
68       expect(msgs[0].innerHTML).toContain('SKU is invalid');
69       expect(msgs[1].innerHTML).toContain('SKU is required');
70     });
71   }));
```

See how much cleaner this is? Our test is focused and tests only one thing. Great job!

This new structure makes adding the second test easy. This time we want to test that, once we add a value to the SKU field, the error messages are gone:

```
code/forms/test/forms/demo_form_with_events.spec.ts
73 it('displays no errors when sku has a value',
74   async(inject([TestComponentBuilder], (tcb) => {
75     return createComponent(tcb).then((fixture) => {
76       input.value = 'XYZ';
77       dispatchEvent(input, 'input');
78       fixture.detectChanges();
79
80       let msgs = el.querySelectorAll('.ui.error.message');
81       expect(msgs.length).toEqual(0);
82     });
83   }));
```

One thing you may have noticed is that so far, our tests are not using `fakeAsync`, but `async` plus `inject` instead.

That's another bonus of this refactoring: we will only use `fakeAsync` and `tick()` when we want to check if something was added to the console, because that's all our form's event handlers do.

The next test will do exactly that - when the SKU value changes, we should have a message logged to the console:

```
code/forms/test/forms/demo_form_with_events.spec.ts
85 it('handles sku value changes', inject([TestComponentBuilder],
86   fakeAsync((tcb) => {
87     createComponent(tcb).then((fixture) => {
88       input.value = 'XYZ';
```

```
89     dispatchEvent(input, 'input');
90     tick();
91
92     expect(fakeConsole.logs).toContain('sku changed to: XYZ');
93   });
94 }
95 ));
```

We can write similar code for both the form change...

code/forms/test/forms/demo_form_with_events.spec.ts

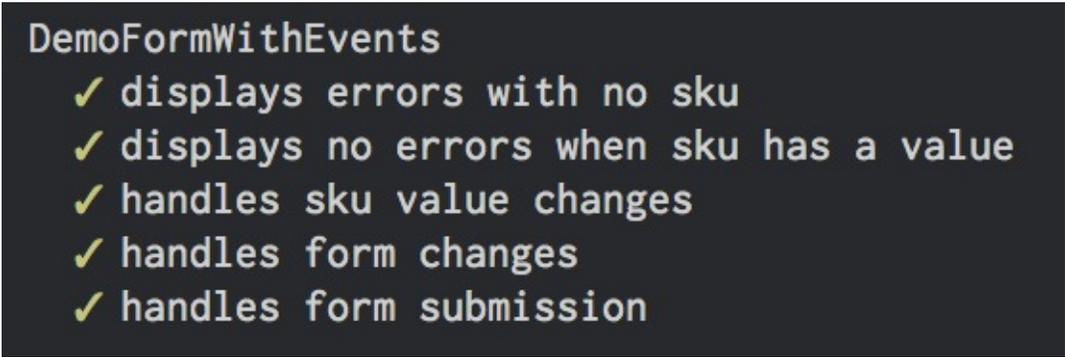
```
97   it('handles form changes', inject([TestComponentBuilder],
98     fakeAsync((tcb) => {
99     createComponent(tcb).then((fixture) => {
100       input.value = 'XYZ';
101       dispatchEvent(input, 'input');
102       tick();
103
104       expect(fakeConsole.logs).toContain('form changed to: [object Object]');
105     });
106   })
107 ));
```

... and the form submission events:

code/forms/test/forms/demo_form_with_events.spec.ts

```
109   it('handles form submission', inject([TestComponentBuilder],
110     fakeAsync((tcb) => {
111     createComponent(tcb).then((fixture) => {
112       input.value = 'ABC';
113       dispatchEvent(input, 'input');
114       tick();
115
116       fixture.detectChanges();
117       dispatchEvent(form, 'submit');
118       tick();
119
120       expect(fakeConsole.logs).toContain('you submitted value: ABC');
121     });
122   })
123 ));
```

When we run the tests now, we get a much nicer output:



```
DemoFormWithEvents
✓ displays errors with no sku
✓ displays no errors when sku has a value
✓ handles sku value changes
✓ handles form changes
✓ handles form submission
```

DemoFormWithEvents test output after refactoring

Another great benefit from this refactor can be seen when something goes wrong. Let's go back to the component code and change the message when the form gets submitted, in order to force one of our tests to fail:

```
1 onSubmit(form: any): void {
2   console.log('you have submitted the value:', form.sku);
3 }
```

If we ran the previous version of the test, here's what would happen:

```
DemoFormWithEvents
  ✗ validates and trigger events
    Expected [ 'sku changed to: ', 'form changed to: [object Object]', 'sku changed to: XYZ', 'form changed to: [object Object]', 'you have submitted the value: XYZ' ] to contain 'you submitted value: XYZ'.
      at /Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:41894
      at run (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
      at zoneBoundFn (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
      at lib$es6$promise$$internal$$tryCatch (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:485)
```

DemoFormWithEvents error output before refactoring

It's not immediately obvious what failed. We have to read the error code to realize it was the submission message that failed. We also can't be sure if that was the only thing that broke on the component code, since we may have other test conditions after the one that failed that never had a chance to be executed.

Now, compare that to the error we get from our refactored code:

```
DemoFormWithEvents
  ✓ displays errors with no sku
  ✓ displays no errors when sku has a value
  ✓ handles sku value changes
  ✓ handles form changes
  ✗ handles form submission
    Expected [ 'sku changed to: ABC', 'form changed to: [object Object]', 'you have submitted the value: ABC' ] to contain 'you submitted value: ABC'.
      at /Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:41673
      at run (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
      at zoneBoundFn (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
      at lib$es6$promise$$internal$$tryCatch (/Users/fcory/code/ng-book2/manuscript/code/forms/test.bundle.js:485)
```

DemoFormWithEvents error output after refactoring

This version makes it pretty obvious that the only thing that failed was the form submission event.

Testing HTTP requests

We could test the HTTP interaction in our apps using the same strategy as we used so far: write a mock version of the `Http` class, since it is an external dependency.

But since the vast majority of single page apps written using frameworks like Angular use HTTP interaction to talk to APIs, the Angular testing library already provides a built in alternative: `MockBackend`.

We have used this class before in this chapter when we were testing the `SpotifyService` class.

Let's dive a little deeper now and see some more testing scenarios and also some good practices. In order to do this, let's write tests for the examples from the *HTTP chapter*.

First, let's see how we test different HTTP methods, like `POST` or `DELETE` and how to test the correct HTTP headers are being sent.

Back on the HTTP chapter, we created this example that covered how to do those things using `Http`.

Testing a POST

The first test we'll write is to make sure we're doing a proper POST request on the `makePost` method:

`code/http/app/ts/components/MoreHttpRequests.ts`

```
33 makePost(): void {
34   this.loading = true;
35   this.http.post(
36     'http://jsonplaceholder.typicode.com/posts',
37     JSON.stringify({
38       body: 'bar',
39       title: 'foo',
40       userId: 1
41     }))
42   .subscribe((res: Response) => {
43     this.data = res.json();
44     this.loading = false;
45   });
46 }
```

When writing our test for this method, our goal is to test two things:

1. the request method (POST) is correct and that
2. the URL we're hitting is also correct.

Here's how we turn that into a test:

`code/http/test/MoreHttpRequests.spec.ts`

```
35 it('performs a POST',
36   async(inject([TestComponentBuilder, MockBackend], (tcb, backend) => {
37     return tcb.createAsync(MoreHttpRequests).then((fixture) => {
38       let comp = fixture.debugElement.componentInstance;
39
40       backend.connections.subscribe(c => {
41         expect(c.request.url)
42           .toBe('http://jsonplaceholder.typicode.com/posts');
43         expect(c.request.method).toBe(RequestMethod.Post);
44         c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
45       });
46
47       comp.makePost();
48       expect(comp.data).toEqual({'response': 'OK'});
49     });
50   }));
51 );
```

Notice how we have a `subscribe` call to `backend.connections`. This will trigger our code whenever a new HTTP connection is established, giving us an opportunity to peek into the request and also provide the response we want.

This place is where you can:

- add request assertions, like checking the correct URL or HTTP method was requested
- set a mocked response, to force your code to deal with different responses, given different test scenarios

Angular uses an enum called `RequestMethod` to identify HTTP methods. Here are the supported methods:

```
1 export enum RequestMethod {
2   Get,
3   Post,
```

```
4 Put,  
5 Delete,  
6 Options,  
7 Head,  
8 Patch  
9 }
```

Finally, after the call `makePost()` we're doing another check to make sure that the mock response we set was the one that was assigned to our component.

Now that we understand how this work, adding a second test for a DELETE method is easy.

Testing DELETE

Here's how the `makeDelete` method is implemented:

code/http/app/ts/components/MoreHttpRequests.ts

```
48 makeDelete(): void {  
49   this.loading = true;  
50   this.http.delete('http://jsonplaceholder.typicode.com/posts/1')  
51     .subscribe((res: Response) => {  
52       this.data = res.json();  
53       this.loading = false;  
54     });  
55 }
```

And this is the code we use to test it:

code/http/test/MoreHttpRequests.spec.ts

```
53 it('performs a DELETE',  
54   async(inject([TestComponentBuilder, MockBackend], (tcb, backend) => {  
55     return tcb.createAsync(MoreHttpRequests).then((fixture) => {  
56       let comp = fixture.debugElement.componentInstance;  
57  
58       backend.connections.subscribe(c => {  
59         expect(c.request.url)  
60           .toBe('http://jsonplaceholder.typicode.com/posts/1');  
61         expect(c.request.method).toBe(RequestMethod.Delete);  
62         c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));  
63       });  
64  
65       comp.makeDelete();  
66       expect(comp.data).toEqual({'response': 'OK'});  
67     });  
68   }));  
69 );
```

Everything here is the same, except for the URL that changes a bit and the HTTP method, which is now `RequestMethod.Delete`.

Testing HTTP Headers

The last method we have to test on this class is `makeHeaders`:

code/http/app/ts/components/MoreHttpRequests.ts

```
57 makeHeaders(): void {  
58   let headers: Headers = new Headers();  
59   headers.append('X-API-TOKEN', 'ng-book');  
60  
61   let opts: RequestOptions = new RequestOptions();  
62   opts.headers = headers;  
63  
64   this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)  
65     .subscribe((res: Response) => {  
66       this.data = res.json();  
67     });  
68 }
```

```
67   });
68 }
```

In this case, what our test should focus on is making sure the header `X-API-TOKEN` is being properly set to `ng-book`:

code/http/test/MoreHttpRequests.spec.ts

```
71 it('sends correct headers',
72     async(inject([TestComponentBuilder, MockBackend], (tcb, backend) => {
73       return tcb.createAsync(MoreHttpRequests).then((fixture) => {
74         let comp = fixture.debugElement.componentInstance;
75
76         backend.connections.subscribe(c => {
77           expect(c.request.url
78             .toBe('http://jsonplaceholder.typicode.com/posts/1'));
79           expect(c.request.headers.has('X-API-TOKEN')).toBeTruthy();
80           expect(c.request.headers.get('X-API-TOKEN')).toEqual('ng-book');
81           c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
82         });
83
84         comp.makeHeaders();
85         expect(comp.data).toEqual({'response': 'OK'});
86       });
87     }));
88 );
```

The connection's `request.headers` attribute returns a `Headers` class instance and we're using two methods to perform two different assertions:

- the `has` method to check whether a given header was set, ignoring its value
- the `get` method, that returns the value that was set

If having the header set is sufficient, use `has`. Otherwise, if you need to inspect the set value, use `get`.

And with that we finish the tests of different methods and headers on Angular. Time to move to a more complex example, that will be closer to what you will encounter when coding real world applications.

Testing YouTubeService

The other example we built back on the HTTP chapter was a YouTube video search. The HTTP interaction for that example takes place on a service called `YouTubeService`:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
48 /**
49  * YouTubeService connects to the YouTube API
50  * See: * https://developers.google.com/youtube/v3/docs/search/list
51  */
52 @Injectable()
53 export class YouTubeService {
54   constructor(public http: Http,
55               @Inject(YOUTUBE_API_KEY) private apiKey: string,
56               @Inject(YOUTUBE_API_URL) private apiUrl: string) {
57   }
58
59   search(query: string): Observable<SearchResult[]> {
60     let params: string = [
61       `q=${query}`,
62       `key=${this.apiKey}`,
63       `part=snippet`,
64       `type=video`,
65       `maxResults=10`
66     ].join('&');
67     let queryUrl: string = `${this.apiUrl}?${params}`;
68     return this.http.get(queryUrl)
69       .map((response: Response) => {
```

```

70     return (<any>response.json()).items.map(item => {
71         // console.log("raw item", item); // uncomment if you want to debug
72         return new SearchResult({
73             id: item.id.videoId,
74             title: item.snippet.title,
75             description: item.snippet.description,
76             thumbnailUrl: item.snippet.thumbnails.high.url
77         });
78     });
79 });
80 }
81 }

```

It uses the YouTube API to search for videos and parse the results into a `SearchResult` instance:

code/http/app/ts/components/YouTubeSearchComponent.ts

```

31 class SearchResult {
32     id: string;
33     title: string;
34     description: string;
35     thumbnailUrl: string;
36     videoUrl: string;
37
38     constructor(obj?: any) {
39         this.id = obj && obj.id || null;
40         this.title = obj && obj.title || null;
41         this.description = obj && obj.description || null;
42         this.thumbnailUrl = obj && obj.thumbnailUrl || null;
43         this.videoUrl = obj && obj.videoUrl ||
44             `https://www.youtube.com/watch?v=${this.id}`;
45     }
46 }

```

The important aspects of this service we need to test are that:

- given a JSON response, the service is able to parse the video id, title, description and thumbnail
- the URL we are requesting uses the provided search term
- the URL starts with what is set on the `YOUTUBE_API_URL` constant
- the API key used matches the `YOUTUBE_API_KEY` constant

With that in mind, let's start writing our test:

code/http/test/YouTubeSearchComponentBefore.spec.ts

```

25 describe('MoreHTTPRequests (before)', () => {
26     beforeEach(() => {
27         addProviders([
28             YouTubeService,
29             BaseRequestOptions,
30             MockBackend,
31             provide(YOUTUBE_API_KEY, {useValue: 'YOUTUBE_API_KEY'}),
32             provide(YOUTUBE_API_URL, {useValue: 'YOUTUBE_API_URL'}),
33             provide(Http, {
34                 useFactory: (backend: ConnectionBackend,
35                     defaultOptions: BaseRequestOptions) => {
36                     return new Http(backend, defaultOptions);
37                 },
38                 deps: [MockBackend, BaseRequestOptions]
39             }),
40         ]);
41     });

```

As we did for every test we wrote on this chapter, we start by declaring how we want to setup our dependencies: we're using the real `YouTubeService` instance, but setting fake values for `YOUTUBE_API_KEY` and `YOUTUBE_API_URL` constants. We also setting up the `Http` class to use a `MockBackend`.

Now, let's begin to write our first test case:

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
43 describe('search', () => {
44   it('parses YouTube response',
45     inject([YouTubeService, MockBackend], fakeAsync((service, backend) => {
46       let res;
47
48       backend.connections.subscribe(c => {
49         c.mockRespond(new Response(<any>{
50           body: `
51             {
52               "items": [
53                 {
54                   "id": { "videoId": "VIDEO_ID" },
55                   "snippet": {
56                     "title": "TITLE",
57                     "description": "DESCRIPTION",
58                     "thumbnails": {
59                       "high": { "url": "THUMBNAIL_URL" }
60                     }
61                   }
62                 }
63               ]
64             }
65           `);
66         });
67         service.search('hey').subscribe(_res => {
68           res = _res;
69         });
70         tick();
71
72         let video = res[0];
73         expect(video.id).toEqual('VIDEO_ID');
74         expect(video.title).toEqual('TITLE');
75         expect(video.description).toEqual('DESCRIPTION');
76         expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
77       });
78     }));
79 });
```

Here we are telling Http to return a fake response that will match the relevant fields what we expect the YouTube API to respond when we call the real URL. We do that by using the mockRespond method of the connection.

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
64 service.search('hey').subscribe(_res => {
65   res = _res;
66 });
67 tick();
```

Next, we're calling the method we're testing: search. We're calling it with the term *hey* and capturing the response on the res variable.

If you noticed before, we're using fakeAsync that requires us to manually sync asynchronous code by calling tick(). When we do that here, we expect that the search finished executing and our res variable to have a value.

Now is the time to evaluate that value:

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
69 let video = res[0];
70 expect(video.id).toEqual('VIDEO_ID');
71 expect(video.title).toEqual('TITLE');
72 expect(video.description).toEqual('DESCRIPTION');
73 expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
```

We are getting the first element from the list of responses. We know it's a `SearchResult`, so we're now checking that each attribute was set correctly, based on our provided response: the id, title, description and thumbnail URL should all match.

With this, we completed our first goal when writing this test. However, didn't we just say that having a huge `it` method and having too many `expects` are testing code smells?

We did, so before we continue let's refactor this code to make isolated assertions easier.

Add the following helper function inside our `describe('search', ...)`:

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
62 function search(term: string, response: any, callback) {
63   return inject([YouTubeService, MockBackend],
64     fakeAsync((service, backend) => {
65       var req;
66       var res;
67
68       backend.connections.subscribe(c => {
69         req = c.request;
70         c.mockRespond(new Response(<any>{body: response}));
71       });
72
73       service.search(term).subscribe(_res => {
74         res = _res;
75       });
76       tick();
77
78       callback(req, res);
79     })
80   )
81 }
```

Let's see what this function does: it uses `inject` and `fakeAsync` to perform the same thing we were doing before, but in a configurable way. We take a *search term*, a *response* and a *callback function*. We use those parameters to call the `search` method with the search term, set the fake response and call the callback function after the request is finished, providing the request and the response objects.

This way, all our test need to do is call the function and check one of the objects.

Let's break the test we had before into four tests, each testing one specific aspect of the response:

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
83 it('parses YouTube video id', search('hey', response, (req, res) => {
84   let video = res[0];
85   expect(video.id).toEqual('VIDEO_ID');
86 }));
87
88 it('parses YouTube video title', search('hey', response, (req, res) => {
89   let video = res[0];
90   expect(video.title).toEqual('TITLE');
91 }));
92
93 it('parses YouTube video description', search('hey', response, (req, res) => \
94 {
95   let video = res[0];
96   expect(video.description).toEqual('DESCRIPTION');
97 }));
98
99 it('parses YouTube video thumbnail', search('hey', response, (req, res) => {
100   let video = res[0];
101   expect(video.description).toEqual('DESCRIPTION');
102 }));
```

Doesn't it look good? Small, focused tests that test only one thing. Great!

Now it should be really easy to add tests for the remaining goals we had:

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
103     it('sends the query', search('term', response, (req, res) => {
104         expect(req.url).toContain('q=term');
105     }));
106
107     it('sends the API key', search('term', response, (req, res) => {
108         expect(req.url).toContain('key=YOUTUBE_API_KEY');
109     }));
110
111     it('uses the provided YouTube URL', search('term', response, (req, res) => {
112         expect(req.url).toMatch(/^YOUTUBE_API_URL\?/);
113     }));
```

Feel free to add more tests as you see fit. For example, you could add a test for when you have more than one item on the response, with different attributes. See if you can find other aspects of the code you'd like to test.

Conclusion

The Angular team has done a great job building testing right into Angular. It's easy to test all of the aspects of our application: from controllers, to services, forms and HTTP. Even testing asynchronous code that was a difficult to test is now a breeze.

Dependency Injection

As our programs grow in size, we often find that different parts of the app need to communicate with other modules. When module A requires module B to run, we say that B is a *dependency* of A.

One of the most common ways to get access to dependencies is to simply `import` a file. For instance, in this hypothetical module we might do the following:

```
1 // in A.ts
2 import {B} from 'B'; // a dependency!
3
4 B.foo(); // using B
```

In many cases, simply importing other code is sufficient. However there are times where we need to provide dependencies in a more sophisticated way. For instance:

- What if we wanted to substitute out the implementation of B for `MockB` during testing?
- What if we wanted to share a *single instance* of the B class across our whole app (e.g. the *Singleton* pattern)
- What if we wanted to create a *new instance* of the B class every time it was used? (e.g. the *Factory* pattern)

Dependency Injection can solve these problems.

Dependency Injection (DI) is a system to make parts of our program accessible to other parts of the program - and we can configure how that happens.



One way to think about an injector is as a replacement for the `new` operator

The term Dependency Injection is used to describe both a design pattern (that used in many different frameworks) and also the specific implementation DI library that is built-in to Angular.

The major benefit of using dependency injection is that the client component doesn't have to be aware of how to create the dependencies, all the component needs to know is how to *interact* with those dependencies.

Injections Example: PriceService

Let's imagine we have a `Product` class. Each product has a base price. In order to calculate the full price for this product, we rely on a service that takes as input

- the **base price** of the product and
- the **state** we're selling it to.

Here's how this would look without dependency injection:

```
1 class Product {
2   constructor(basePrice: number) {
3     this.service = new PriceService();
4     this.basePrice = basePrice;
5   }
6
7   price(state: string) {
8     return this.service.calculate(this.basePrice, state);
9   }
10 }
```

Now let's imagine we need to write a test for this Product class. Let's assume the PriceService class above uses a database lookup to retrieve taxes for a given state. If we write a test like:

```
1 let product;
2
3 beforeEach(() => {
4   product = new Product(11);
5 });
6
7 describe('price', () => {
8   it('is calculated based on the basePrice and the state', () => {
9     expect(product.price('FL')).toBe(11.66);
10  });
11 })
```

Even though the test may work, there are a few shortcomings to this approach. In order for the test to success a few preconditions have to be met:

1. The database must be running;
2. The tax entry for Florida must be what we're expecting;

Basically we're making our tests more brittle by adding an unexpected dependency between the Product class and the PriceService that, in turn, depends on a database.

What if we could write the Product class a little differently:

```
1 class Product {
2   constructor(service: PriceService, basePrice: number) {
3     this.service = service;
4     this.basePrice = basePrice;
5   }
6
7   price(state: string) {
8     return this.service.calculate(this.basePrice, state);
9   }
10 }
```

Now, when creating a Product the client class becomes responsible for deciding which concrete implementation of the PriceService is going to be given to the new instance.

With that, we can make our tests a lot simpler by creating a *mock* version of the PriceService class:

```
1 class MockPriceService {
2   calculate(basePrice: number, state: string) {
3     if (state === 'FL') {
4       return basePrice * 1.06;
5     }
6
7     return basePrice;
8   }
9 }
```

And with this small change, we can tweak our test slightly and get rid of the database dependency:

```
1 let product;
2
3 beforeEach(() => {
4   const service = new MockPriceService();
5   product = new Product(service, 11);
6 });
7
8 describe('price', () => {
9   it('is calculated based on the basePrice and the state', () => {
10     expect(product.price('FL')).toBe(11.66);
11   });
12 })
```

We also get the bonus of having confidence that we're testing the `Product` class *in isolation*. That is, we're making sure that our class works with a predictable dependency.

“Don't Call Us...”

This technique of injecting the dependencies relies on a principle called the inversion of control.

 The inversion of control (or IoC) principle is also called informally the “Hollywood principle”, that is a reference to the Hollywood motto “don't call us, we'll call you”.

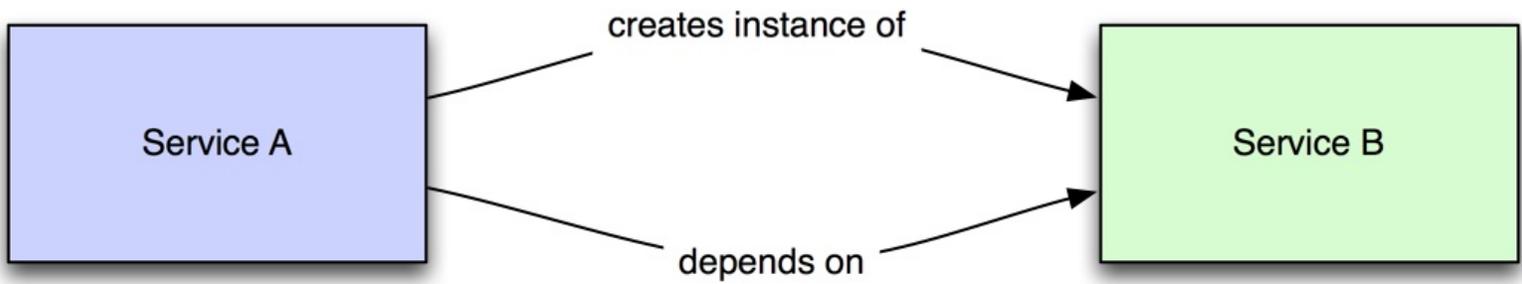
Over the years it was very common for every component to be aware of the complete application context and be responsible for creating and setting up the dependencies. This can be seen clearly on our example, where the `Product` class had to be aware of the `PriceService`.

The setback of doing things that way is that once a component becomes aware of the dependency, the component itself becomes more brittle and therefore harder to change. If we make change to a component on which many other components are dependent upon, we end up having to propagate the changes to a lot of different areas of our application and sometimes even outside the boundaries of it. In other words, we're making our components *tightly coupled*.

When we use DI we are moving towards a more *loosely coupled* architecture where changing bits and pieces of a single component affects the other areas of the application less. And, as long as the interface between those components don't change, we can even swap them altogether, without any other components even realizing.

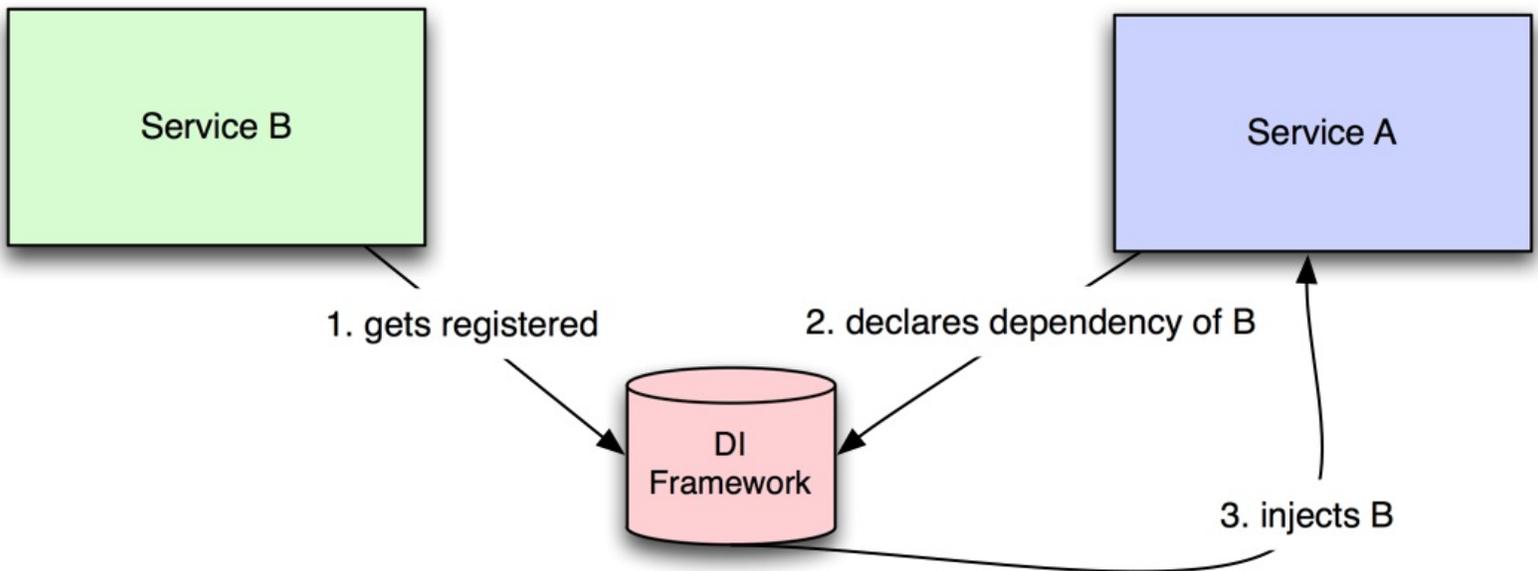
One of the great features that ng2 inherited from ng1 is that both frameworks uses this Inversion of Control pattern. Angular uses *dependency injection* to resolve dependencies out of the box.

Traditionally, if a component A depends on component B, what would happen is that an instance of B would be created inside A. This implies that now A depends on B.



Without a Dependency Injection Framework

Angular uses the Dependency Injection to change things around in a way that if we need component B inside component A, we expect that B will be *passed* to A.



With a Dependency Injection Framework

This brings many advantages over the traditional scenario. One example of an advantage is that, if we're testing A in isolation we can easily create a mocked version of B and inject it into A.

We have used services and therefore dependency injection a lot of times earlier in this book. For example, when we created the music application back on the Routing chapter. To interact with the Spotify API, we created the **SpotifyService** that was injected on a number of components as we can see on this snippet from the **AlbumComponent**:

code/routes/music/app/ts/components/AlbumComponent.ts

```
42 constructor(public route: ActivatedRoute, public spotify: SpotifyService,
43             public location: Location) {
44     route.params.subscribe(params => { this.id = params['id']; });
}
```

Now let's learn how to create our own services and the different forms we can inject them.

Dependency Injection Parts

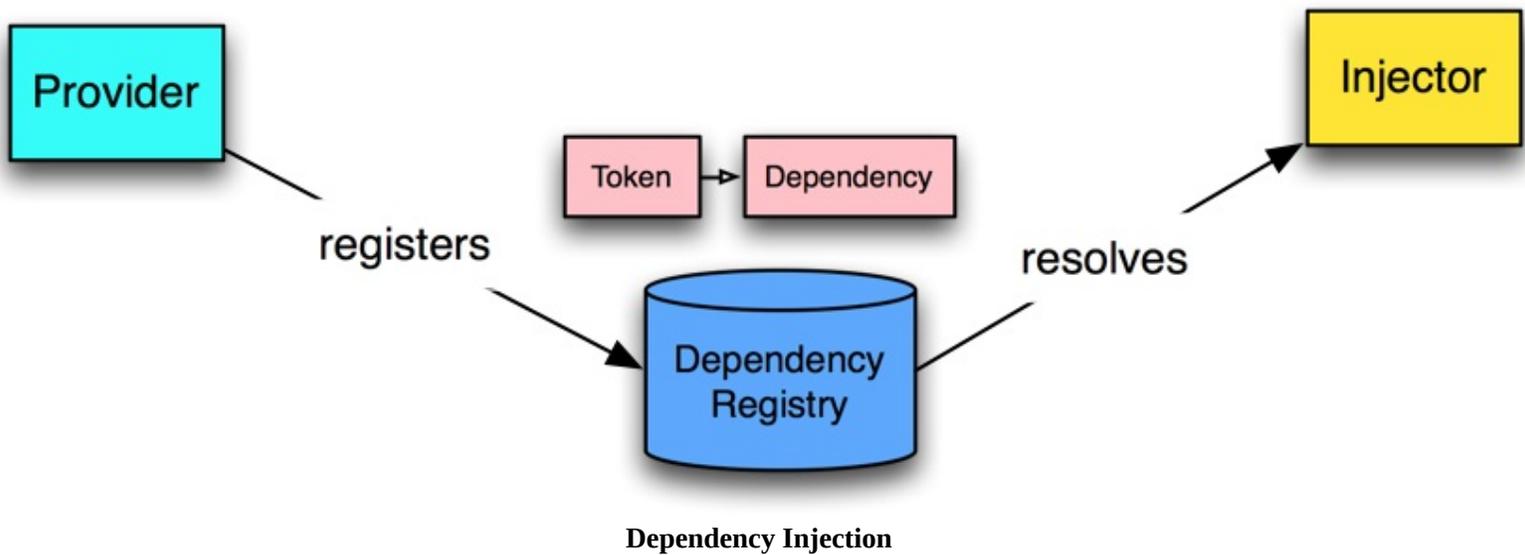
To register a dependency we have to bind it to something that will identify that dependency. This identification is called the dependency **token**. For instance, if we want to register the URL of an API,

we can use the string `API_URL` as the token. Similarly, if we're registering a class, we can use the class itself as its **token** as we'll see below.

Dependency injection in Angular has three pieces:

- the **Provider** (also often referred to as a binding) maps a *token* (that can be a string or a class) to a list of dependencies. It tells Angular how to create an object, given a token.
- the **Injector** that holds a set of bindings and is responsible for resolving dependencies and injecting them when creating objects
- the **Dependency** that is what's being injected

We can think of the role of each piece as illustrated below:



There are a lot of different options when dealing with DI, so let's see how each of them work.

One of the most common cases is providing a service or value that is the same across our whole application. This scenario would be what we would use 99% of the time in our apps.

If this is all we want to do, we'll cover how to write a basic service in the next section and that is going to be all we need for most of our apps most of the time.

Enough talk, let's code!

Playing with an Injector

As mentioned above, Angular is going to setup DI for us behind the scenes. But before we deal with annotations and the integrating injections into our components, let's first play with the injector by itself.

Let's create a sample service that only returns a string:

`code/dependency_injection/simple/app/ts/app.ts`

```
15 /*
16  * The injectable service
17  */
18 class MyService {
19   getValue(): string {
20     return 'a value';
21   }
22 }
```

```
21 }
22 }
```

Next, we want to create the app component:

code/dependency_injection/simple/app/ts/app.ts

```
24 @Component({
25   selector: 'di-sample-app',
26   template: `
27     <button (click)="invokeService()">Get Value</button>
28   `
29 })
30 class DiSampleApp {
31   myService: MyService;
32
33   constructor() {
34     let injector: any = ReflectiveInjector.resolveAndCreate([MyService]);
35     this.myService = injector.get(MyService);
36     console.log('Same instance?', this.myService === injector.get(MyService));
37   }
38
39   invokeService(): void {
40     console.log('MyService returned', this.myService.getValue());
41   }
42 }
```

Let's break things down a bit. We are declaring the `DiSampleApp` component that will render a button. When that button is clicked we call the `invokeService` method.

Focusing on the constructor of the component we can see that we are using a static method from `ReflectiveInjector` called `resolveAndCreate`. That method is responsible for creating a new injector. The parameter we pass in is an array with all the *injectable things* we want this new injector to *know*. In our case, we just wanted it to know about the `MyService` injectable.



The `ReflectiveInjector` is a concrete implementation of `Injector` that uses *reflection* to look up the proper parameter types. While there are other injectors that are possible `ReflectiveInjector` is the “normal” injector we'll be using in most apps.

One important thing to notice is that will inject a **singleton** instance of the class.

This can be verified by the last two lines of our constructor. We are first asking our newly created injector to give us the instance for the `MyService` class. We then store that into our component's `myService` field. Right after that, we have a `console.log` that asks the injector to give us the instance of `MyService` again. When the result of the comparison of the next line executes:

```
1 console.log('Same instance?', this.myService === injector.get(MyService));
```

We get the confirmation that both instances are actually the exact same object on the console:

```
1 Same instance? true
```

Notice that, since we're using our own injector, we didn't have to tell the application the injectables list as we're used to during bootstrapping:

code/dependency_injection/injector/app/ts/app.ts

```
44 // no need to add injectables here
45 bootstrap(DiSampleApp).catch((err: any) => console.error(err));
```

Now that we learned about how the injectors work behind the scenes, let's look at the parts of Angular dependency injection framework, then learn what other stuff we can inject and also how we can inject them.

Providers

One of the neat things about Angular's DI system is that there are several ways we can configure the injection. For instance we can:

- Inject a (singleton) instance of a class
- Call any function and inject the return value of that function
- Inject a value
- Create an alias

Let's look at how we could create each one:

Using a Class

Injecting a singleton instance of a class is probably the most common type of injection.

Here's how we configure it:

```
1 provide(MyComponent, {useClass: MyComponent})
```

What's interesting to note is that the `provide` method takes **two** arguments. The first is the *token* that we use to identify the injection and the second is how and what to inject.

So here we're mapping the `MyComponent` class to the `MyComponent` token. In this case, the name of the class and the token match. This is the common case, but know that the token and the injected thing *don't have to have the same name*.

As we've seen above, in this case the injector will create a **singleton** behind the scenes and return the same instance every time we inject it .

Of course, the first time it is injected, it hasn't been instantiated yet, so when creating the **MyComponent** instance for the first time, the DI system will trigger the class **constructor** method.

Now what happens if a service's **constructor** requires some parameter? Let's say we have this service:

```
code/dependency_injection/misc/app/ts/app.ts
```

```
25 class ParamService {
26   constructor(private phrase: string) {
27     console.log('ParamService is being created with phrase', phrase);
28   }
29
30   getValue(): string {
31     return this.phrase;
32   }
33 }
```

Notice how its constructor method takes a phrase as a parameter? If we try to use the regular injection mechanism:

```
1 bootstrap(MyApp, [ParameterService]);
```

We would see an error on the browser:

```
Cannot resolve all parameters for 'ParameterService'(?). Make sure that all the parameters are decorated with Inject or have valid type annotations and that 'ParameterService' is decorated with Injectable. lang.js:375
```

Injection error

This happens because we didn't provide the injector with enough information about the class we're trying to build. In order to resolve this problem, we need to tell the injector which parameter we want it to use when creating the service's instance.

If we need to pass in parameters when creating a service, we would need to use a factory instead.

Using a Factory

When we use a factory injection, we write a function that can return **any object**.

```
1 provide(MyComponent, useFactory: () => {
2   if (loggedIn) {
3     return new MyLoggedComponent();
4   }
5   return new MyComponent();
6 });
```

Notice in the case above, we inject on the token `MyComponent` but this will check the (out of scope) `loggedIn` variable. If `loggedIn` is truthy then the injection will give an instance of `MyLoggedComponent`, otherwise we will receive `MyComponent`.

Factories can also have dependencies:

```
1 provide(MyComponent, useFactory: (user) => {
2   if (user.loggedIn()) {
3     return new MyLoggedComponent(user);
4   }
5   return new MyComponent();
6 }, deps: [User])
```

So if we wanted to use our `ParamService` from above, we can wrap it with `useFactory` like so:

`code/dependency_injection/misc/app/ts/app.ts`

```
51 bootstrap(DiSampleApp, [
52   SimpleService,
53   provide(ParamService, {
54     useFactory: (): ParamService => new ParamService('YOLO')
55   })
56 ]).catch((err: any) => console.error(err));
```



In this bootstrap function, because **SimpleService** doesn't need any parameters, we can put it in the dependencies array without using `provide`, and it will get translated to:

```
1 provide(SimpleService, {useClass: SimpleService})
```

Using a factory is the most powerful way to create injectables, because we can do whatever we want within the factory function.

Using a Value

This is useful when we want to register a constant that can be redefined by another part of the application or even by environment (e.g. test or production).

```
1 provide('API_URL', {useValue: 'http://my.api.com/v1'})
```

We're going to do a more thorough example that uses values further down on the [Substituting Values section](#).

Using an alias

We can also make an alias to reference a previously registered token, like so:

```
1 provide(NewComponent, {useClass: MyComponent})
```

Dependency Injection in Apps

When writing our apps there are three steps we need to take in order to perform an injection:

1. Create the service class
2. Declare the dependencies on the receiving component and
3. Configure the injection (i.e. register the injection with Angular)

The first thing we do is create the service class, that is, the class that exposes some behavior we want to use. This will be called the *injectable* because it is the *thing* that our components will receive via the injection.

Here is how we would create a service:

```
code/dependency_injection/simple/app/ts/services/ApiService.ts
```

```
1 export class ApiService {
2   get(): void {
3     console.log('Getting resource...');
4   }
5 }
```

Now that we have the thing to be injected, we have to take the next step, which is declare the dependencies we want to receive when Angular creates our component.

Earlier we used the `Injector` class directly, but Angular provides two shortcuts for us we can use when writing our components.

The first and typical way of doing it, is by declaring the injectables we want in our component's constructor.

To do that, we require the service:

code/dependency_injection/simple/app/ts/app.ts

```
9 /*
10  * Services
11  */
12 import {ApiService} from 'services/ApiService';
```

And then we declare it on the constructor:

code/dependency_injection/simple/app/ts/app.ts

```
25 class DiSampleApp {
26   constructor(private apiService: ApiService) {
27   }
```

When we declare the injection in our component constructor, Angular will do some *reflection* to figure out what class to inject. That is, Angular will see that we are looking for an object of type `ApiService` in the constructor and check the DI system for an appropriate injection.

Sometimes we need to give Angular more hints about what we're trying to inject. In that case we use the second method by using the `@Inject` annotation:

```
1 class DiSampleApp {
2   private apiService: ApiService;
3   constructor(@Inject(ApiService) apiService) {
4     this.apiService = apiService;
5   }
```



If you want to play with the equivalent version, use the `app.long.ts` file provided alongside the `app.ts` file, just copy its contents over `app.ts`.

The final step for using dependency injection is to connect the things our components want injected with the injectables. In other words, we are telling Angular which *thing* to inject when a component declares its dependencies.

```
1 provide(ApiService, {useClass: ApiService})
```

In this case, we use the token `ApiService` to expose the singleton of the class `ApiService`.

Working with Injectors

We've played a little bit with injectors already, so let's talk a little more about when we would need to use them explicitly.

One case would be when we need to control the moment where the singleton instance of our dependency gets created.

To illustrate a scenario where that could happen, let's build another app that uses the **ApiService** we created above, along with a new service.

This service will be used to instantiate two other services, based on the size of the browser viewport. If it's less than 800 pixels, it will return a new instance of a service called **SmallService**. Otherwise, it will return an instance of **LargeService**.

Here's how **SmallService** look like:

code/dependency_injection/complex/app/ts/services/SmallService.ts

```
1 export class SmallService {
2   run(): void {
3     console.log('Small service...');
4   }
5 }
```

And **LargeService**:

code/dependency_injection/complex/app/ts/services/LargeService.ts

```
1 export class LargeService {
2   run(): void {
3     console.log('Large service...');
4   }
5 }
```

Then we'll write the **ViewPortService** that choses between the two:

code/dependency_injection/complex/app/ts/services/ViewPortService.ts

```
1 import {LargeService} from './LargeService';
2 import {SmallService} from './SmallService';
3
4 export class ViewPortService {
5   determineService(): any {
6     let w: number = Math.max(document.documentElement.clientWidth,
7                               window.innerWidth || 0);
8
9     if (w < 800) {
10      return new SmallService();
11    }
12    return new LargeService();
13  }
14 }
```

Now let's create an app that uses our services:

code/dependency_injection/complex/app/ts/app.ts

```
30 class DiSampleApp {
31   constructor(private apiService: ApiService,
32               @Inject('ApiServiceAlias') private aliasService: ApiService,
33               @Inject('SizeService') private sizeService: any) {
34   }
```

Here we are getting an instance of **ApiService** the way we saw ealier. But alongside we're also getting an instance of the same service, aliased as 'ApiServiceAlias'. Finally we're getting an instance of a 'SizeService' that is not yet defined.

To understand what each service represents, let's move to the bootstrap section:

code/dependency_injection/complex/app/ts/app.ts

```
54 bootstrap(DiSampleApp, [
55   ApiService,
```

```
56 ViewPortService,  
57 provide('ApiServiceAlias', {useExisting: ApiService}),  
58 provide('SizeService', {useFactory: (viewport: any) => {  
59     return viewport.determineService();  
60 }}, deps: [ViewPortService])  
61 ]).catch((err: any) => console.error(err));
```

Here we are saying that we want the app injector to be aware of the **ApiService** and **ViewPortService** injectables as they are.

We are then declaring that we want to use the existing **ApiService** with another token: the string `ApiServiceAlias`.

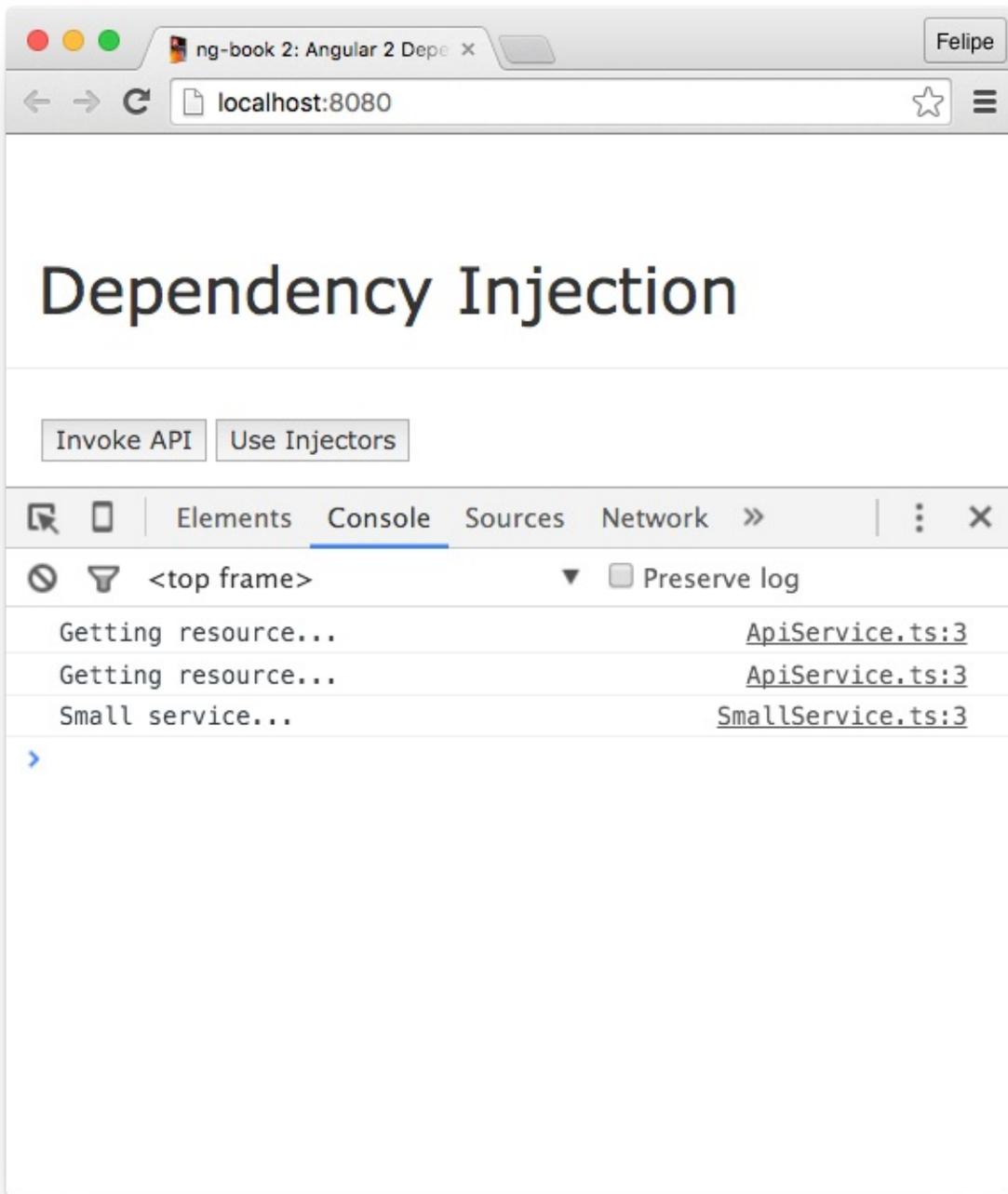
Next, we're declaring another injectable defined by the string token `SizeService`. This factory will receive an instance of the **ViewPortService** that is listed on its `deps` array. Then it will invoke the `determineService()` method on that class and that call will return either an instance of `SmallService` or `LargeService`, depending on our browser's width.

When we click a button in our template, we want to do three calls: one to the **ApiService**, one to its alias **ApiServiceAlias** and finally one to **SizeService**:

code/dependency_injection/complex/app/ts/app.ts

```
36 invokeApi(): void {  
37     this.apiService.get();  
38     this.aliasService.get();  
39     this.sizeService.run();  
40 }
```

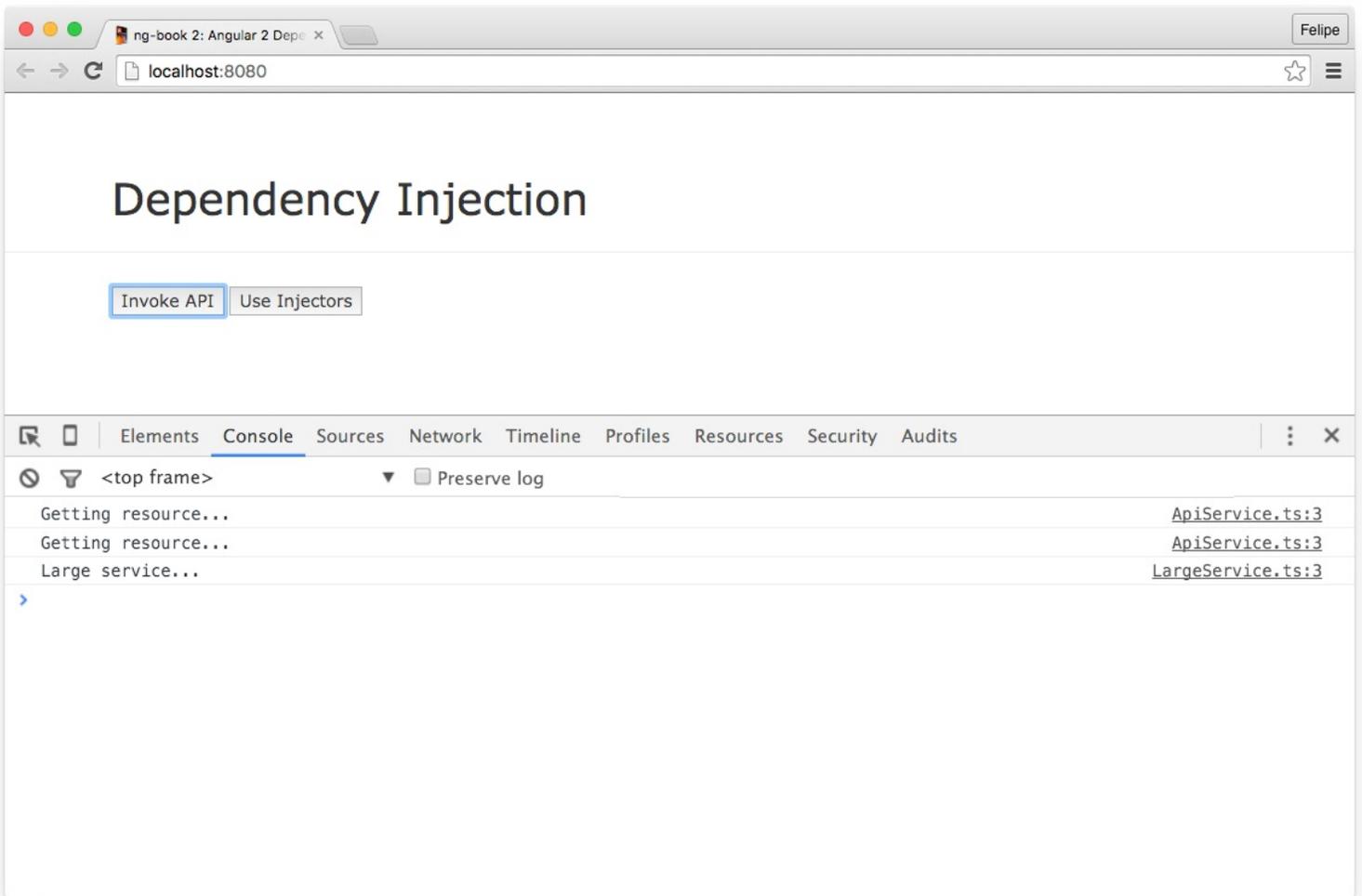
Now if we run the app and click the **Invoke API** button with a small browser window:



Small browser window

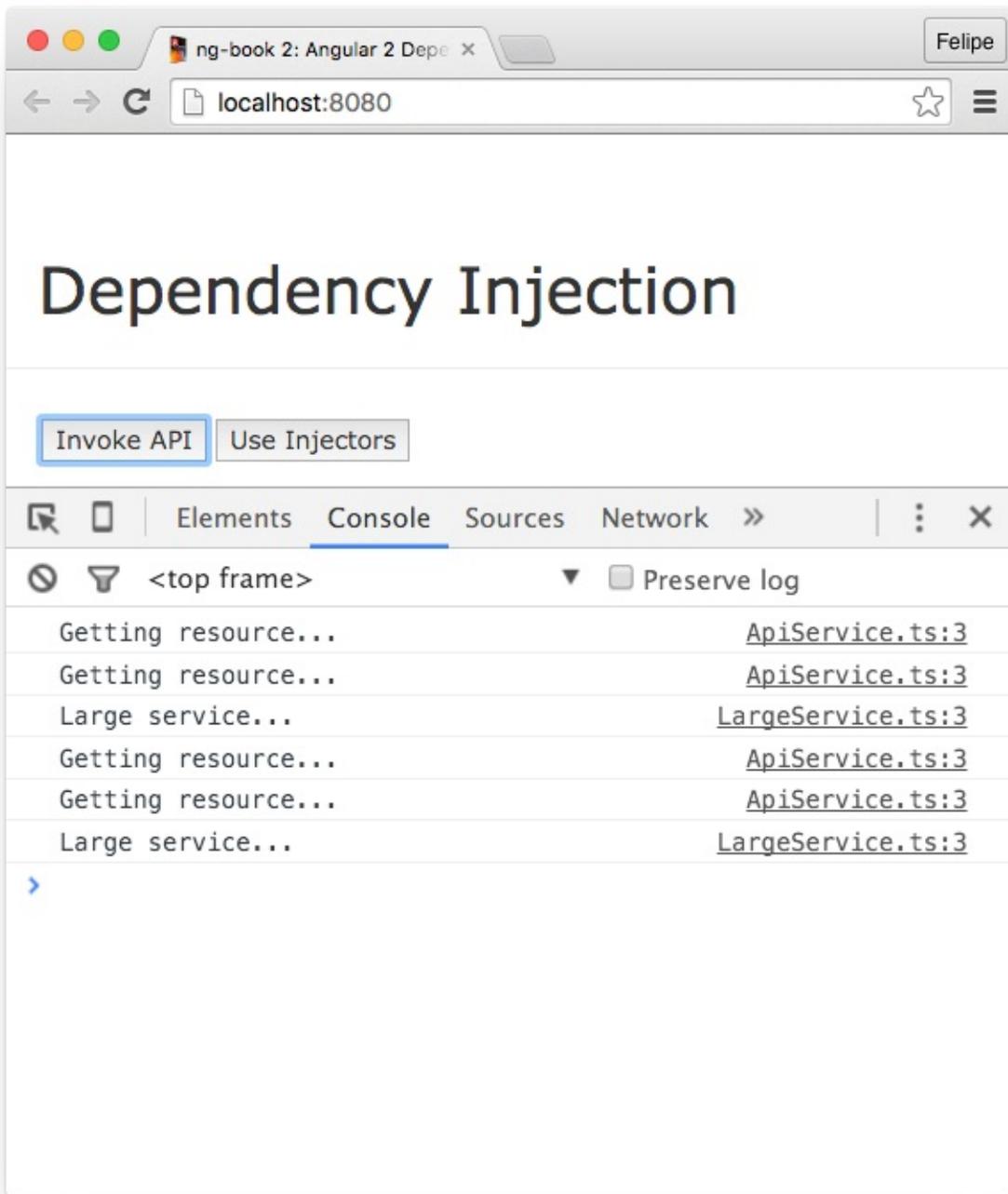
We get one log from the **ApiService**, another one from the aliased service and finally we get a log from the **SmallService**.

If we make our browser window larger, reload the page and click the button again:



Large browser window

We get the **LargeService** log instead. However, if we try to make the browser window smaller and click the button without reloading the page, we still get the **LargeService** log:



Small browser window - resized

That's because the factory was executed once: during the application bootstrap.

To overcome that, we can create our own injectors and get the instance of the proper service by doing the following:

code/dependency_injection/complex/app/ts/app.ts

```
42 useInjectors(): void {
43   let injector: any = ReflectiveInjector.resolveAndCreate([
44     ViewPortService,
45     provide('OtherSizeService', {useFactory: (viewport: any) => {
46       return viewport.determineService();
47     }, deps: [ViewPortService]})
48   ]);
49   let sizeService: any = injector.get('OtherSizeService');
50   sizeService.run();
```

Here we are creating an injector that knows the **ViewPortService** and another injectable with the string `OtherSizeService` as its token. This injectable uses the same factory as the `SizeService` we used before.

Finally, it uses the injector we created to get an instance of the `OtherSizeService`.line

Now if we run the app with a large browser window and click the **Use Injector** button, we get the large service log. However, if we resize it to a small width, even without reloading we now get the proper log. That's because the factory is being executed every time we click the button, since the injector is being created on demand. Neat!

Substituting values

Another reason to use DI is to change the hard value of the injection at runtime. That could happen if we have an API service that performs the HTTP requests to our application's API. On the context of our unit or integration tests, we don't want our code to hit the production database. In this case, we could write a Mock API service that seamlessly *replaces* our concrete implementation. Let's take a look at that now.

For instance, if we are running the app in development we might hit a different API server than if we were running the app in production.

This is even more true if we were publishing an open-source or reusable service. In that case we may want the allow the client application define or override an API URL.

Let's write a simple example of an application that injects different values as the API URL depending on whether it's running on production or dev mode. We start with the `ApiService` class:

code/dependency_injection/value/app/ts/services/ApiService.ts

```
1 import { Inject } from '@angular/core';
2
3 export const API_URL: string = 'API_URL';
4
5 export class ApiService {
6   constructor(@Inject(API_URL) private apiUrl: string) {
7   }
8
9   get(): void {
10    console.log(`Calling ${this.apiUrl}/endpoint...`);
11  }
12 }
```

We are declaring a constant that will be used as the *token* for our API URL dependency. In other words, Angular will use the string `'API_URL'` to store the information about which URL to call. This way when we `@Inject(API_URL)` the proper value will be injected into the variable.

Notice we are exporting the `API_URL` constant so that client applications can use `API_URL` to inject the correct value from outside the service.

Now that we have the service, let's write the application component that will use the service and provide different values for the URL, depending on the environment the app will be running on.

code/dependency_injection/value/app/ts/app.ts

```
20 @Component({
21   selector: 'di-value-app',
22   template: `
23     <button (click)="invokeApi()">Invoke API</button>
24   `
25 })
26 class DiValueApp {
27   constructor(private apiService: ApiService) {
28   }
29
30   invokeApi(): void {
31     this.apiService.get();
32   }
33 }
```

This is the component code. On the constructor we can see that we are declaring a `ApiService` variable called `apiService`. Here Angular will infer that we need to get the `ApiService` dependency and inject it at runtime. If we wanted to be explicit about it we could have used:

```
1 constructor(@Inject(ApiService) private apiService: ApiService) {
2 }
```

The idea behind this component is to have an **Invoke API** button. When we click this button, we'll call the `get()` method of the `ApiService`. This method will then log to the console the `API_URL` we're using.

The next step is to configure the application with the providers:

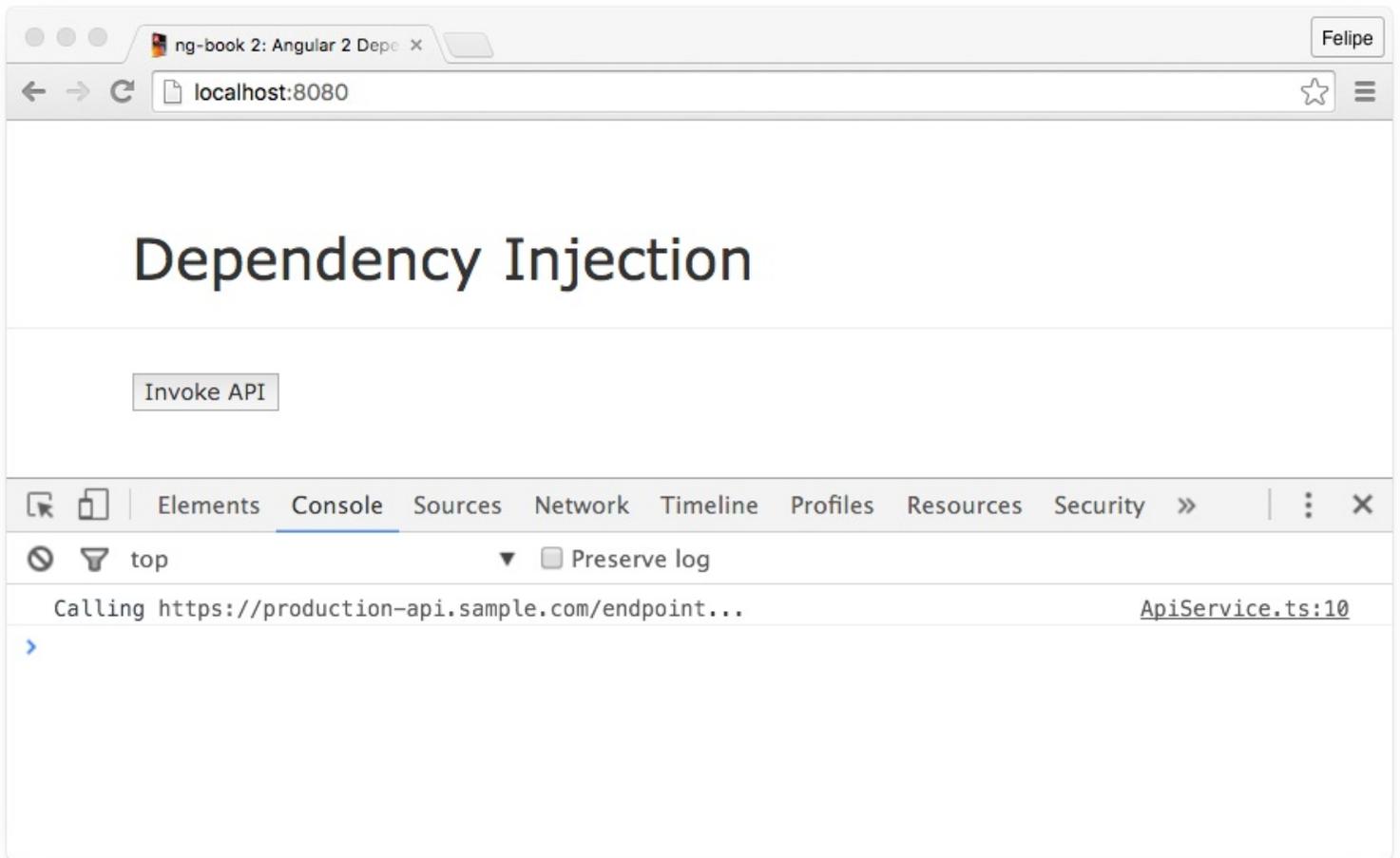
code/dependency_injection/value/app/ts/app.ts

```
35 const isProduction: boolean = false;
36
37 bootstrap(DiValueApp, [
38   provide(ApiService, { useClass: ApiService }),
39   provide(API_URL, {
40     useValue: isProduction ?
41       'https://production-api.sample.com' :
42       'http://dev-api.sample.com'
43   })
44 ]).catch((err: any) => console.error(err));
```

First we declare a constant called `isProduction` and set it to `false`. We can pretend that we're doing something here to determine whether or not we are in production mode. This setting could be hardcoded like we're doing, or it could be set using some technique like using WebPack and an `.env` file, for instance.

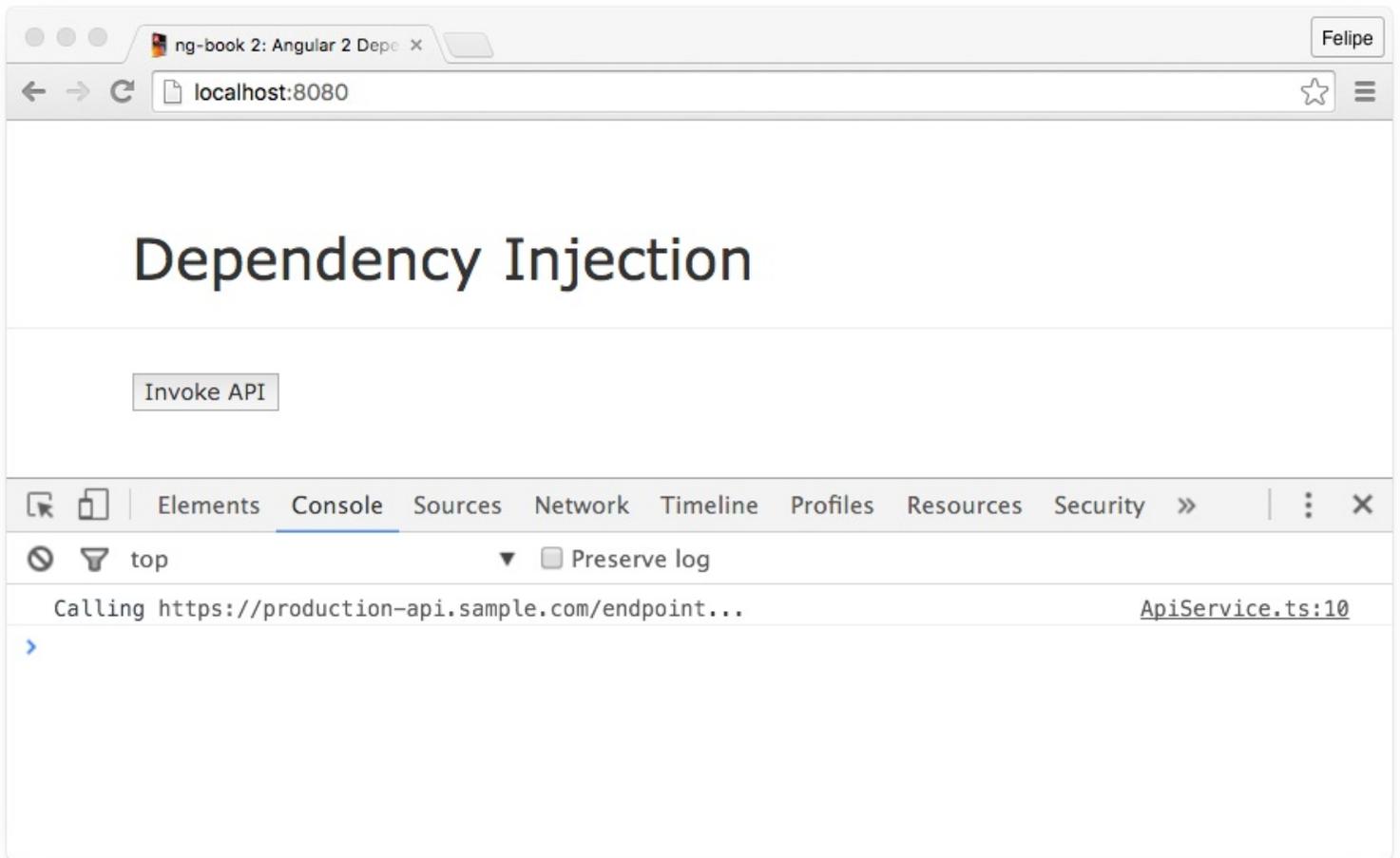
Finally we bootstrap the application and setup 2 providers: one for `ApiService` using the real class implementation and the other for `API_URL`. If we are in production we're using one value and if not, we're using another value.

To test this we can run the application with `isProduction = true` and when we click the button we'll see the production URL being logged:



Production environment

And if we change it to `isProduction = false`, we see the dev URL instead:



Dev environment

Conclusion

As we can see, Dependency Injection is a powerful way to manage dependencies within our app. Here are a few more resources where you can learn more about it:

- [Official Angular DI Docs](#)
- [Victor Savkin Compare DI in Angular 1 vs. Angular 2](#)

Changelog

Revision 35 - 2016-06-30

Book and code up to date with angular-2.0.0-rc.4

- Routing upgraded to new router
- Forms upgraded to new forms library
- Testing chapter updated to match new routing and forms

Revision 34 - 2016-06-15

Book and code up to date with angular-2.0.0-rc.2

Note: still using router-deprecated at this time.

Revision 33 - 2016-05-11

New chapter: Dependency Injection!

Revision 32 - 2016-05-06

Entire book up to date with angular-2.0.0-rc.1!

- Entire book changes:
 - Renamed all imports to match the new packages (see below)
 - Upgrade to typings (removes all tsd references)
 - Directive local variables now use `let` instead of `#`. E.g. `*ngFor="#item in items"` becomes `*ngFor="let item in items"`
 - In projects that use System.js, create an external file for configuration (instead of writing it in the `index.html` `<script>` tags)
- “Testing” Chapter:
 - `injectAsync` has been removed. Instead you use `async` and `inject` together, both come from `@angular/core/testing`
- “Advanced Components” Chapter:
 - In `ngBookRepeat`, when creating a child view manually with `createEmbeddedView`, the context is passed as the second argument (instead of calling `setLocal`).

Details:

Renamed libraries:

- `angular2/core` -> `@angular/core`
- `angular2/compiler` -> `@angular/compiler`
- `angular2/common` -> `@angular/common`

- angular2/platform/common -> @angular/common
- angular2/common_dom -> @angular/common
- angular2/platform/browser -> @angular/platform-browser-dynamic
- angular2/platform/server -> @angular/platform-server
- angular2/testing -> @angular/core/testing
- angular2/upgrade -> @angular/upgrade
- angular2/http -> @angular/http
- angular2/router -> @angular/router
- angular2/platform/testing/browser -> @angular/platform-browser-dynamic/testing

Revision 31 - 2016-04-28

All chapters up to date with angular-2.0.0-beta.16

Revision 30 - 2016-04-20

All chapters up to date with angular-2.0.0-beta.15

Revision 29 - 2016-04-08

All chapters up to date with angular-2.0.0-beta.14

Revision 28 - 2016-04-01

All chapters up to date with angular-2.0.0-beta.13 - (no joke!)

Revision 27 - 2016-03-25

All chapters up to date with angular-2.0.0-beta.12

Revision 26 - 2016-03-24

Advanced Components chapter added!

Revision 25 - 2016-03-21

All chapters up to date with angular-2.0.0-beta.11

Note: angular-2.0.0-beta.10 skipped because the release had a couple of bugs.

Revision 24 - 2016-03-10

All chapters up to date with angular-2.0.0-beta.9

Revision 23 - 2016-03-04

All chapters up to date with angular-2.0.0-beta.8

- “Routing” Chapter
 - Fixed a few typos - Németh T.
 - Fixed path to nested routes description - Dante D.

- “First App” Chapter
 - Fixed typos - Luca F.
 - Removed unnecessary import of NgFor - Neufeld M.
- “Forms” Chapter
 - Typos - Miha Z., Németh T.
- “How Angular Works” Chapter
 - Typos - Koen R., Jeremy T., Németh T.
- “Typescript” Chapter
 - Typos - Németh T.
- “Data Architecture with RxJS” Chapter
 - Typos - Németh T.
- “HTTP” Chapter
 - Typos - Németh T.
- “Testing” Chapter
 - Typos - Németh T.

Revision 22 - 2016-02-24

- r20 & beta.6 introduced some bugs regarding the typescript compiler and new typing files that were required to be included. This revision fixes those bugs
- Added a note about how to deal with the error: `error TS2307: Cannot find module 'angular2/platform/browser'`
- “First App” Chapter - added a tiny note about the typings references
- Updated all non-webpack examples to have a `clean npm` command as well as change the `tsconfig.json` to include the `app.ts` when appropriate

Revision 21 - 2016-02-20

All chapters up to date with `angular-2.0.0-beta.7`

Revision 20 - 2016-02-11

All chapters up to date with `angular-2.0.0-beta.6` (see note below)

- “How Angular Works” Chapter
 - Fixed Typo. Thanks [@AndreaMiotto](#)
 - Added missing brackets in attributes on `MyComponent` - Thanks Németh T.
- “Forms” Chapter
 - Grammar fix - Németh T.
 - Added missing line of code in “Field coloring” - Németh T.
- “RxJs” Chapters
 - Grammar fix - Németh T.
- Note: `beta.4` and `beta.5` were replaced with `beta.6`. [See the angular 2 CHANGELOG](#)

Revision 19 - 2016-02-04

All chapters up to date with `angular-2.0.0-beta.3`

Revision 18 - 2016-01-29

All chapters up to date with angular-2.0.0-beta.2

Revision 17 - 2016-01-28

- Added Testing Chapter

Revision 16 - 2016-01-14

- Added “How to Convert ng1 App to ng2” Chapter
- All chapters now up to date with angular-2.0.0-beta.1
- All package.json files pinned to specific versions
- “HTTP” Chapter
 - Fixed typo - Thanks Ole S!
- “Built-in-Components” Chapter
 - Fixed ngIf typo

Revision 15 - 2016-01-07

All chapters now up to date with angular-2.0.0-beta.0!

- “RxJS” Chapters
 - Updated to angular-2.0.0-beta.0
- “HTTP” Chapter
 - Updated to angular-2.0.0-beta.0
- Fixed line numbers for code that loads from files to match the line numbers on file
- “How Angular Works” Chapter - Fixed swapped LHS / RHS language. - Thanks, Miroslav J.

Revision 14 - 2015-12-23

- “First App” Chapter
 - Fixed typo on hello-world @Component - Thanks Matt D.
 - Fixed typescript dependency in hello_world package.json
- “Forms Chapter”
 - Updated to angular-2.0.0-beta.0
- “How Angular Works Chapter”
 - Significant rewrite to make it clearer
 - Updated to angular-2.0.0-beta.0
- “Routing Chapter”
 - Significant rewrite to make it clearer
 - Updated to angular-2.0.0-beta.0

Revision 13 - 2015-12-17

Angular 2 beta.0 is out!

- “First App” Chapter

- Updated reddit app to angular-2.0.0-beta.0
- Updated hello_world app to angular-2.0.0-beta.0
- Added [Semantic UI](#) styles
- “Built-in Components” Chapter
 - Updated built-in components sample apps to angular-2.0.0-beta.0
 - Added Semantic UI

Revision 12 - 2015-11-16

- “Routing” Chapter
 - Fixed ROUTER_DIRECTIVES typo - Wayne R.
- “First App” Chapter
 - Updated example to angular-2.0.0-alpha-46
 - Fixed some bolding around NgFor to clarify the code example - Henrique M.
 - Fixed Duplicate identifier 'Promise'. errors due to a bad tsconfig.json in angular2-reddit-base/ - Todd F.
 - Fixed language typos caught by Steffen G.
 - “Forms” Chapter
 - Updated example to angular-2.0.0-alpha-46
 - Fixes the method of subscribing to Observables in the “Form with Events” section
 - Fixed a few typos and language issues - Christopher C., Travis P.
 - “TypeScript” Chapter
 - Fixed some unclear language about enum - Frede H.
 - “Built-in Components” Chapter
 - Fixed a typo where [class] needed to be [ng-class] - Neal B.
 - “How Angular Works” Chapter
 - Fixed language typos - Henrique M.

Revision 11 - 2015-11-09

- Fixed explanation of TypeScript benefits - Thanks Don H!
- Fixed tons of typos found by Wayne R - Thanks Wayne!
- “How Angular Works” Chapter
 - Fixed typos - Jegor U.
 - Converted a component to use inputs/outputs - Jegor U.
 - Fixed number to myNumber typo - Wayne R.
- “Built-in Components” Chapter
 - Fixed language typos - Wayne R., Jek C., Jegor U.
 - Added a tip-box explaining object keys with dashes - Wayne R.
 - Use controller view value for ng-style color instead of the form field value - Wayne R.
- “Forms” Chapter
 - Fixed language typos - Wayne R., Jegor U.
- “Data Architecture in Angular 2”
 - Was accidentally part of “Forms” and is now promoted to an introductory mini-chapter - Wayne R.
- “RxJS Pt 1.” Chapter

- Fixed language typos - Wayne R.
- “RxJS Pt 2.” Chapter
 - Fixed Unicode problem - Birk S.
 - Clarified language around combineLatest return value - Birk S.
- “Typescript” Chapter
 - Fixed language typo - Travis P., Don H.
- “Routing” Chapter
 - Fixed language typos - Jegor U., Birk S.
- “First App” Chapter
 - Fixed link to ng_for - Mickey V.
- “HTTP” Chapter
 - Fixed language typos - Birk S.
 - Clarified ElementRef role in YouTubeSearchComponent
 - Fixed link to RequestOptions - Birk S.

Revision 10 - 2015-10-30

- Upgraded Writing your First Angular2 Web Application chapter to angular-2.0.0-alpha.44
- Upgraded Routing chapter to angular-2.0.0-alpha.44
- Fixed ‘pages#about’ on the rails route example. - Thanks Rob Y!

Revision 9 - 2015-10-15

- Added Routing Chapter

Revision 8 - 2015-10-08

- Upgraded chapters 1-5 to angular-2.0.0-alpha.39
- properties and events renamed to inputs and outputs
- Fixed an issue in the First App chapter that said #newtitle bound to the value of the input (it’s really binding to the Control object) - Danny L
- CSSClass renamed to NgClass
- ng-non-bindable is now built-in so you don’t need to inject it as a directive
- Updated the forms chapter as there were several changes to the forms API
- Fixed NgFor source url in First App chapter - Frede H.

Revision 7 - 2015-09-23

- Added HTTP Chapter
- Fixed For -> NgFor typo - Sanjay S.

Revision 6 - 2015-08-28

- Added RxJS Chapter Data Architecture with Observables - Part 1 : Services
- Added RxJS Chapter Data Architecture with Observables - Part 2 : View Components

Revision 5

- Finished built-in components chapter

Revision 4

- Added built-in components chapter draft
- Added a warning about linewrapping of long URLs - Thanks Kevin B!
- Explained how annotations are bound to components on the First App chapter - thanks Richard M. and others
- Copy typo fixes - thanks Richard M.!
- Fixed TypeScript using `integer` instead of `number` - Richard M. and Roel V.
- Fixed “`var nate =`” listings require a comma to be a valid JS object - thanks Roel V.
- Renamed a few “For” directive mentions to “NgFor” - thanks Richard M.
- Fixed type on “RedditArticle” - thanks Richard M.
- Explained how annotations are bound to components on the First App chapter (thanks Richard M. and others)
- Typos and grammar improvements on First App chapter (thanks Kevin B)
- Typos and code improvements on How Angular Works (thanks Roel V.)

Revision 3

- Added forms chapter

Revision 2

- Updated For directive to NgFor accross all chapters and examples (templates changed from `*for=` to `*ng-for=` as well)
- Changed the suggested static web server from `http-server` to `live-server` so the execution command is valid both in OSX/Linux and Windows
- Changed the `@Component`'s `properties` property to match the latest AngularJS 2 format
- Updated `angular2.dev.js` bundle to latest version for all examples
- Updated `typings` folder with latest version for all examples

Revision 1

Initial version of the book

NOTES

1 See: [Model View Whatever↵](#)