ng-book

The Complete Book on Angular 7

FULLSTACK.io

gistia

Nate Murray
Felipe Coury
Ari Lerner
Carlos Taborda

ng-book

The Complete Guide to Angular

Written by Nate Murray, Felipe Coury, Ari Lerner, and Carlos Taborda

Published in San Francisco, California by Fullstack.io.

FULLSTACK.io

We'd like to thank our contributors to this book including:

- *Frode Fikke for technical edits*
- *Nic Raboy, and Burke Holland for their contribution to the NativeScript chapter*
- *Travas Nolte for his work on many chapters*
- *Daniel Rauf for his edits on RxJS, Redux, and Routing*

Contents

Book Revision

Revision 73 - Covers up to Angular 7 (7.2.0, 2019-01-08)

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: us@fullstack.io[1].

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, come join us on Gitter[2]!

Vote for New Content (new!)

We're constantly updating the book, writing new blog posts, and producing new material. You can now cast your vote for new content here[3].

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow @fullstackio[4]

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[5].

[1] mailto:us@fullstack.io?Subject=ng-book%202%20feedback
[2] https://gitter.im/ng-book/ng-book
[3] https://fullstackio.canny.io/ng-book
[4] https://twitter.com/fullstackio
[5] mailto:us@fullstack.io?Subject=ng-book%202%20testimonial

How to Read This Book

This book aims to be the single most useful resource on learning Angular. By the time you're done reading this book, you (and your team) will have everything you need to build reliable, powerful Angular apps.

Angular is a rich and feature-filled framework, but that also means it can be tricky to understand all of its parts. In this book, we'll walk through everything from installing the tools, writing components, using forms, routing between pages, and calling APIs.

But before we dig in, there are a few guidelines I want to give you **in order to get the most out of this book**. Briefly, I want to tell you:

- how to approach **the code examples** and
- **how to get help** if something goes wrong

Running Code Examples

This book comes with a library of runnable code examples. The code is available to download from the same place where you downloaded this book.

We use the program `npm`[6] to run **every example** in this book. This means you can type the following commands to run any example:

```
npm install
npm start
```

> If you're unfamiliar with `npm`, we cover how to get it installed in the Getting Started section in the first chapter.

After running `npm start`, you will see some output on your screen that will tell you what URL to open to view your app.

If you're ever unclear on how to run a particular sample app, check out the `README.md` in that project's directory. Every sample project contains a `README.md` that will give you the instructions you need to run each app.

[6] https://www.npmjs.com/

Angular CLI

With a couple of minor exceptions, every project in this book was built on Angular CLI[7]. Unless specified otherwise, you can use the `ng` commands in each project.

For instance, to run an example you can run `ng serve` (this is, generally, what is run when you type `npm start`). For most projects you can compile them to JavaScript with `ng build` (we'll talk about this more in the first chapter). And you can run end-to-end tests with `ng e2e`, etc.

Without getting too far into the details, Angular CLI is based on Webpack, a tool which helps process and bundle our various TypeScript, JavaScript, CSS, HTML, and image files. **Angular CLI is not a requirement** for using Angular. It's simply a wrapper around Webpack (and some other tooling) that makes it easy to get started.

Code Blocks and Context

Nearly every code block in this book is pulled from a **runnable code example**, which you can find in the sample code. For example, here is a code block pulled from the first chapter:

code/first-app/angular-hello-world/src/app/app.component.ts

```
export class AppComponent {
  title = 'app';
}
```

Notice that the header of this code block states the path to the file which contains this code: `code/first-app/angular-hello-world/src/app/app.component.ts`.

If you ever feel like you're missing the context for a code example, open up the full code file using your favorite text editor. **This book is written with the expectation that you'll also be looking at the example code alongside the manuscript**.

For example, we often need to `import` libraries to get our code to run. In the early chapters of the book we show these `import` statements, because it's not clear where the libraries are coming from otherwise. However, the later chapters of the book are more advanced and they focus on *key concepts* instead of repeating boilerplate code that was covered earlier in the book. **If at any point you're not clear on the context, open up the code example on disk**.

Code Block Numbering

In this book, we sometimes build up a larger example in steps. If you see a file being loaded that has a numeric suffix, that generally means we're building up to something bigger.

[7] https://github.com/angular/angular-cli

For instance, in the Dependency Injection chapter you may see a code block with the filename: `price.service.1.ts`. When you see the `.N.ts` syntax that means we're building up to the ultimate file, which will **not** have a number. So, in this case, the final version would be: `price.service.ts`. We do it this way so that a) we can unit test the intermediate code and b) you can see the whole file in context at a particular stage.

A Word on Versioning

As you may know, the Angular covered in this book is a descendant of an earlier framework called "AngularJS". This can sometimes be confusing, particularly when reading supplementary blogs or documentation.

The official branding guidelines state that "*AngularJS*" is a term reserved for AngularJS 1.x, that is, the early versions of "Angular".

Because the new version of Angular used TypeScript (instead of JavaScript) as the primary language, the 'JS' was dropped, leaving us with just *Angular*. For a long time the only consistent way to distinguish the two was folks referred to the *new* Angular as *Angular 2*.

However, the Angular team in 2017 switched to *semantic versioning* with a new major-release upgrade slated for every 6 months. Instead of calling the next versions *Angular 4*, *Angular 5*, and so on, the number is also dropped and it's just *Angular*.

In this book, when we're referring to *Angular* we'll just say *Angular* or sometimes *Angular X*, just to avoid confusion. When we're talking about "the old-style JavaScript Angular" we'll use the term *AngularJS* or *AngularJS 1.x*.

Getting Help

While we've made every effort to be clear, precise, and accurate you may find that when you're writing your code you run into a problem.

Generally, there are three types of problems:

- A "bug" in the book (e.g. how we describe something is wrong)
- A "bug" in our code
- A "bug" in your code

If you find an inaccuracy in how we describe something, or you feel a concept isn't clear, email us! We want to make sure that the book is both accurate and clear.

Similarly, if you've found a bug in our *code* we definitely want to hear about it.

If you're having trouble getting your own app working (and it isn't *our* example code), this case is a bit harder for us to handle.

Your first line of defense, when getting help with your custom app, should be our unofficial community chat room[8]. We (the authors) are there from time-to-time, but there are hundreds of other readers there who may be able to help you faster than we can.

If you're still stuck, we'd still love to hear from you, and here are some tips for getting a clear, timely response.

Emailing Us

If you're emailing us asking for technical help, here's what we'd like to know:

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.8, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- What have you tried[9] already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

The **absolute best way to get technical support** is to send us a short, self-contained example of the problem.

But in any case email us at us@fullstack.io[10]. We look forward to hearing from you.

Chapter Overview

Before we dive in, I want to give you a feel for the rest of the book and what you can expect inside.

The first few chapters provide the **foundation** you need to get up and running with Angular. You'll create your **first apps**, use **the built-in components**, and start **creating your components**.

Next we'll move into intermediate concepts such as using **forms**, using **APIs**, **routing** to different pages, and using *Dependency Injection* to organize our code.

After that, we'll move into more **advanced concepts**. We spend a good part of the book talking about *data architectures*. Managing state in client/server applications is hard and we dive deep into two popular approaches: using **RxJS Observables** and using **Redux**. In these chapters, we'll show how to build the same app, two different ways, so you can compare and contrast and evaluate what's best for you and your team.

After that, we'll discuss how to write complex, **advanced components** using Angular's most powerful features. Then we talk about how to write **tests** for our app and how we can **upgrade**

[8] https://gitter.im/ng-book/ng-book
[9] http://mattgemmell.com/what-have-you-tried/
[10] mailto:us@fullstack.io

our Angular 1 apps to Angular. Finally, we close with a chapter on writing **native mobile apps** with Angular using **NativeScript**.

By using this book, **you're going to learn how to build real Angular apps** faster than spending hours parsing out-dated blog posts.

So hold on tight - you're about to become an Angular expert, and have a lot of fun along the way. Let's dig in!

- Nate (@eigenjoy[11])

[11]https://twitter.com/eigenjoy

Writing Your First Angular 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[12] or Product Hunt[13].

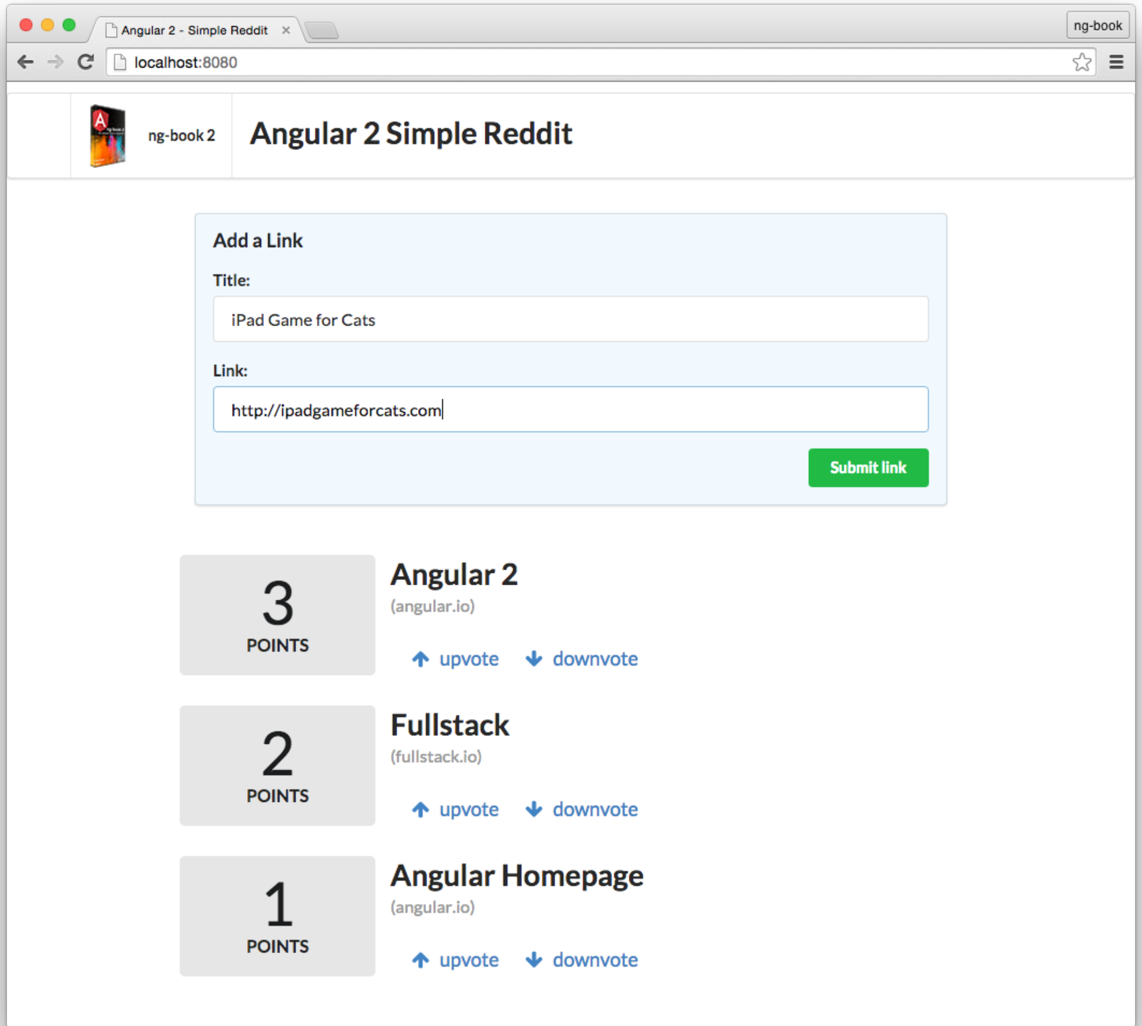
In this simple app we're going to cover most of the essentials of Angular including:

- Building custom components
- Accepting user input from forms
- Rendering lists of objects into views
- Intercepting user clicks and acting on them
- Deploying our app to a server

By the time you're finished with this chapter you'll know how to take an empty folder, build a basic Angular application, and deploy it to production. After working through this chapter you'll have a good grasp on how Angular applications are built and a solid foundation to build your own Angular app.

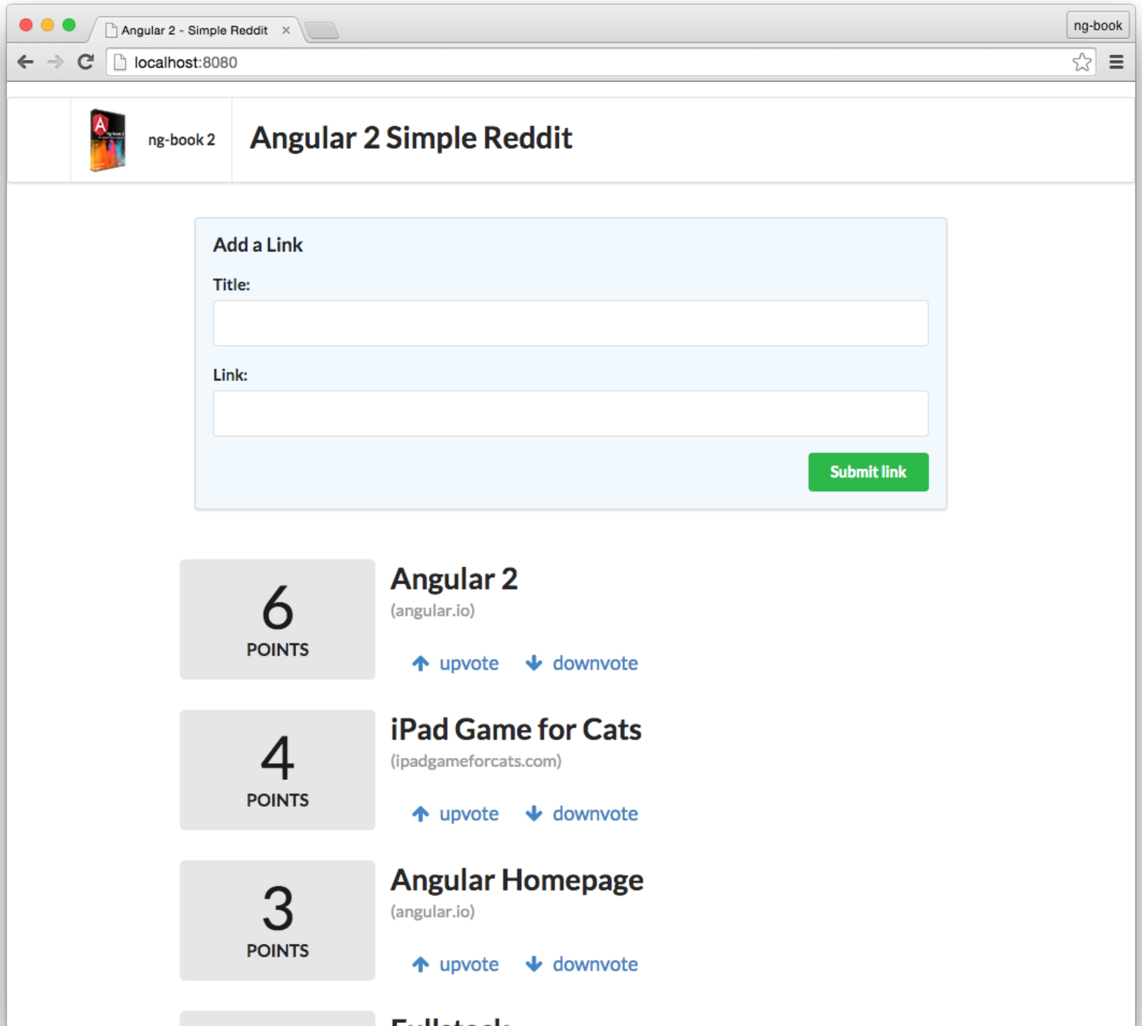
Here's a screenshot of what our app will look like when it's done:

[12] http://reddit.com
[13] http://producthunt.com



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 **we'll go over TypeScript more in depth in the next chapter**.

Don't worry if you're having trouble with some of the new syntax. If you're familiar with ES5 ("normal" JavaScript) / ES6 (ES2015) you should be able to follow along and we'll talk more about TypeScript in a bit.

Getting started

Node.js and `npm`

To get started with Angular, 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[14] for detailed information.

Make sure you install Node `8.9.0` or higher.

> If you're on a Mac, your best bet is to install Node.js directly from the Node.js website instead of through another package manager (like Homebrew). Installing Node.js via Homebrew is known to cause some issues.

The Node Package Manager (`npm` for short) is installed as a part of Node.js. To check if `npm` is available as a part of our development environment, we can open a terminal window and type:

```
$ npm -v
```

If a version number is not printed out and you receive an error, make sure to download a Node.js installer that includes npm.

Your `npm` version should be `5.6.0` or higher.

TypeScript

Once you have Node.js setup, the next step is to install TypeScript. Make sure you install at least version 2.1 or greater. To install it, run the following `npm` command:

```
$ npm install -g typescript
```

> **Do I have to use TypeScript?** No, you don't *have* to use TypeScript to use Angular, but you probably should. Angular does have an ES5 API, but Angular 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 easier. That said, it isn't strictly required.

Browser

We highly recommend using the Google Chrome Web Browser[15] to develop Angular apps. We'll use the Chrome developer toolkit throughout this book. To follow along with our development and debugging we recommend downloading it now.

[14] https://nodejs.org/download/
[15] https://www.google.com/chrome/

Special instruction for Windows users

Throughout this book, we will be using Unix/Mac commands in the terminal. Most of these commands, like `ls` and `cd`, are cross-platform. However, sometimes these commands are Unix/Mac-specific or contain Unix/Mac-specific flags (like `ls -1p`).

As a result, be alert that you may have to occasionally determine the equivalent of a Unix/Mac command for your shell. Fortunately, the amount of work we do in the terminal is minimal and you will not encounter this issue often.

Windows users should be aware that our terminal examples use Unix/Mac commands.

Angular CLI

Angular provides a utility to allow users to create and manage projects from the command line. It automates tasks like creating projects, adding new controllers, etc. It's generally a good idea to use Angular CLI as it will help create and maintain common patterns across our application.

To install Angular CLI, run the following command:

```
$ npm install -g @angular/cli
```

Once it's installed you'll be able to run it from the command line using the `ng` command. When you do, you'll see a lot of output, but if you scroll back, you should be able to see the following:

```
$ ng --version
```

If everything installed correctly, you should see the current version output to your terminal. Congratulations!

If you're running OSX or Linux, you might receive this line in the output:

```
Could not start watchman; falling back to NodeWatcher for file system events.
```

This means that we don't have a tool called **watchman** installed. This tool helps Angular CLI when it needs to monitor files in your filesystem for changes. If you're running OSX, it's recommended to install it using Homebrew with the following command:

```
$ brew install watchman
```

> If you're on OSX and got an error when running brew, it means that you probably don't have Homebrew installed. Please refer to the page http://brew.sh/ to learn how to install it and try again.
>
> If you're on Linux, you may refer to the page https://ember-cli.com/user-guide/#watchman for more information about how to install watchman.
>
> If you're on Windows instead, you don't need to install anything and Angular CLI will use the native Node.js watcher.

If you're curious about all of the things that Angular CLI can do, try out this command:

```
$ ng --help
```

Don't worry about understanding all of the options - we'll be covering the important ones in this chapter.

Now that we have Angular CLI and its dependencies installed, let's use this tool to create our first application.

Example Project

Open up the terminal and run the `ng new` command to create a new project from scratch:

```
$ ng new angular-hello-world
```

Once you run it, you'll see (roughly) following output:

```
CREATE angular-hello-world/README.md (1034 bytes)
CREATE angular-hello-world/angular.json (3504 bytes)
CREATE angular-hello-world/package.json (1323 bytes)
CREATE angular-hello-world/tsconfig.json (384 bytes)
CREATE angular-hello-world/tslint.json (2805 bytes)
CREATE angular-hello-world/.editorconfig (245 bytes)
CREATE angular-hello-world/.gitignore (503 bytes)
CREATE angular-hello-world/src/environments/environment.prod.ts (51 bytes)
CREATE angular-hello-world/src/environments/environment.ts (631 bytes)
CREATE angular-hello-world/src/favicon.ico (5430 bytes)
CREATE angular-hello-world/src/index.html (304 bytes)
CREATE angular-hello-world/src/main.ts (370 bytes)
CREATE angular-hello-world/src/polyfills.ts (3194 bytes)
CREATE angular-hello-world/src/test.ts (642 bytes)
CREATE angular-hello-world/src/assets/.gitkeep (0 bytes)
```

```
CREATE angular-hello-world/src/styles.css (80 bytes)
CREATE angular-hello-world/src/browserslist (375 bytes)
CREATE angular-hello-world/src/karma.conf.js (964 bytes)
CREATE angular-hello-world/src/tsconfig.app.json (194 bytes)
CREATE angular-hello-world/src/tsconfig.spec.json (282 bytes)
CREATE angular-hello-world/src/tslint.json (314 bytes)
CREATE angular-hello-world/src/app/app.module.ts (314 bytes)
CREATE angular-hello-world/src/app/app.component.css (0 bytes)
CREATE angular-hello-world/src/app/app.component.html (1141 bytes)
CREATE angular-hello-world/src/app/app.component.spec.ts (986 bytes)
CREATE angular-hello-world/src/app/app.component.ts (207 bytes)
CREATE angular-hello-world/e2e/protractor.conf.js (752 bytes)
CREATE angular-hello-world/e2e/src/app.e2e-spec.ts (299 bytes)
CREATE angular-hello-world/e2e/src/app.po.ts (208 bytes)
CREATE angular-hello-world/e2e/tsconfig.e2e.json (213 bytes)
...
added 1146 packages in 105.319s
    Successfully initialized git.
```

This will run for a while while it's installing npm dependencies. Once it finishes we'll see a success message.

The exact files that your project generates may vary slightly depending on the version of `@angular/cli` that was installed.

There are a lot of files generated! Don't worry about understanding all of them yet. Throughout the book we'll walk through what each one means and what it's used for.

Let's go inside the `angular-hello-world` directory, which the `ng` command created for us and see what has been created:

```
$ cd angular-hello-world
$ tree -F -L 1
.
|-- README.md                // a useful README
|-- angular.json             // angular-cli configuration file
|-- e2e/                     // end-to-end tests
|-- node_modules/            // installed dependencies
|-- package-lock.json        // npm dependencies lockfile
|-- package.json             // npm configuration
|-- src/                     // our application's code
|-- tsconfig.json            // typescript config
```

```
`-- tslint.json            // linting config

3 directories, 6 files
```

The `tree` command is completely optional. But if you're on OSX it can be installed via `brew install tree`

For now, the folder we're interested in is `src`, where we'll put our custom application code. Let's take a look at what was created there:

```
$ cd src
$ tree -F
.
|-- app/
|   |-- app.component.css
|   |-- app.component.html
|   |-- app.component.spec.ts
|   |-- app.component.ts
|   `-- app.module.ts
|-- assets/
|-- browserslist
|-- environments/
|   |-- environment.prod.ts
|   `-- environment.ts
|-- favicon.ico
|-- index.html
|-- karma.conf.js
|-- main.ts
|-- polyfills.ts
|-- styles.css
|-- test.ts
|-- tsconfig.app.json
|-- tsconfig.spec.json
`-- tslint.json

3 directories, 18 files
```

Using your favorite text editor, let's open `index.html`. You should see this code:

code/first-app/angular-hello-world/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularHelloWorld</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Let's break it down a bit:

code/first-app/angular-hello-world/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularHelloWorld</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
```

If you're familiar with writing HTML files, this first part is straightforward, we're declaring the core structure of the HTML document and a few bits of metadata such as page `charset`, `title` and base `href`.

If we continue to the template body, we see the following:

code/first-app/angular-hello-world/src/index.html

```
<body>
  <app-root></app-root>
</body>
</html>
```

The `app-root` tag is **where our application will be rendered**.

But what *is* the `app-root` tag and where does it come from? `app-root` is a *component* that is defined by our Angular application. In Angular **we can define our own HTML tags** and give them custom functionality. The `app-root` tag will be the "entry point" for our application on the page.

Let's try running this app as-is and then we'll dig in to see how this component is defined.

Writing Application Code

Running the application

Before making any changes, let's load our app from the generated application into the browser. Angular CLI has a built in HTTP server that we can use to run our app.

To use it, head back to the terminal, and change directories into the root of our application.

```
$ cd angular-hello-world
$ ng serve
** NG Live Development Server is running on http://localhost:4200. **
// ...
// a bunch of other messages
// ...
Compiled successfully.
```

Our application is now running on `localhost` port `4200`. Let's open the browser and visit:

http://localhost:4200[16]

[16] http://localhost:4200

Note that if you get the message:

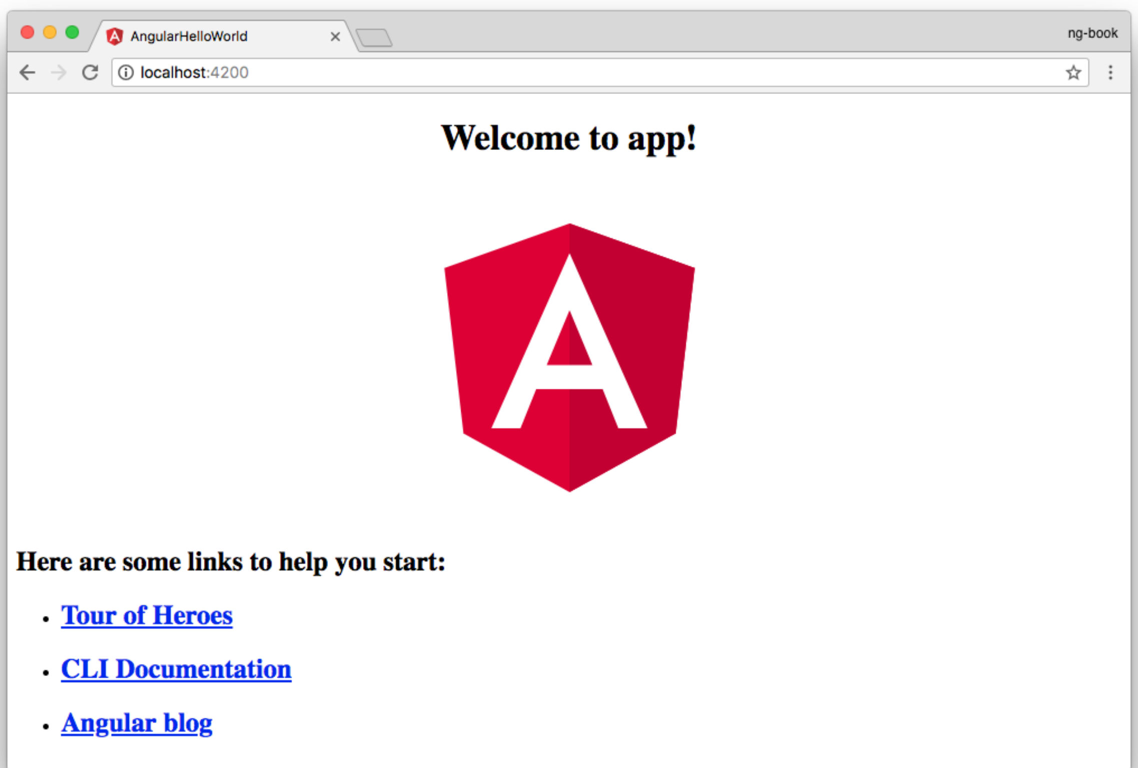
```
Port 4200 is already in use. Use '--port' to specify a different port
```

This means that you already have another service running on port `4200`. If this is the case you can either 1. shut down the other service or 2. use the `--port` flag when running `ng serve` like this:

```
ng serve --port 9001
```

The above command would change the URL you open in your browser to something like: `http://localhost:9001`

Another thing to notice is that, on some machines, the domain `localhost` may not work. You may see a set of numbers such as `127.0.0.1`. When you run `ng serve` it should show you what URL the server is running on, so be sure to read the messages on your machine to find your exact development URL.



Running application

Now that we have the application setup, and we know how to run it, it's time to start writing some code.

Making a Component

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

In our Angular apps, we write HTML markup that becomes our interactive application, but the browser only understands a limited set of markup tags; Built-ins like <select> or <form> or <video> all have functionality **defined by our browser creator**.

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 want to create a <login> tag that shows a login panel?

This is the fundamental idea behind components: we will **teach the browser new tags** that have custom functionality attached to them.

> If you have a background in AngularJS 1.X, you can think of **components as 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 using the app-hello-world tag:

```
<app-hello-world></app-hello-world>
```

To create a new component using Angular CLI, we'll use the **generate** command.

To generate the **hello-world** component, we need to run the following command:

```
$ ng generate component hello-world
CREATE src/app/hello-world/hello-world.component.css (0 bytes)
CREATE src/app/hello-world/hello-world.component.html (30 bytes)
CREATE src/app/hello-world/hello-world.component.spec.ts (657 bytes)
CREATE src/app/hello-world/hello-world.component.ts (288 bytes)
UPDATE src/app/app.module.ts (414 bytes)
```

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

1. A Component decorator
2. A component definition class

Let's look at the component code and then take these one at a time. Open up our first TypeScript file: src/app/hello-world/hello-world.component.ts.

code/first-app/angular-hello-world/src/app/hello-world/hello-world.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

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

> 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 gap, the `ng serve` command live-compiles our `.ts` to a `.js` file automatically.

Importing Dependencies

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

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. In this case, we're telling the compiler that `"@angular/core"` defines and exports two JavaScript/TypeScript objects called `Component` and `OnInit`.

Similarly, we `import OnInit` from the same module. As we'll learn later, `OnInit` helps us to run code when we initialize the component. For now, don't worry about it.

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 by ES6 and TypeScript. We will talk more about it in the next chapter.

The idea with `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.

Component Decorators

After importing our dependencies, we are declaring the component:

code/first-app/angular-hello-world/src/app/hello-world/hello-world.component.ts

```
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
```

If you're new to TypeScript then the syntax of this next statement might seem a little foreign:

```
@Component({
  // ...
})
```

What is going on here? These are called *decorators*.

We can think of decorators as **metadata added to our code**. When we use `@Component` on the `HelloWorld` class, we are "decorating" `HelloWorld` as a `Component`.

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

```
@Component({
  selector: 'app-hello-world'
  // ... more here
})
```

The syntax of Angular's component selectors is similar to CSS selectors (though Angular components have some special syntax for selectors, which we'll cover later on). For now, know that with this `selector` we're **defining a new tag** that we can use in our markup.

The `selector` property here indicates *which DOM element* this component is going to use. In this case, any `<app-hello-world></app-hello-world>` tags that appear within a template will be compiled using the `HelloWorldComponent` class and get any attached functionality.

Adding a template with `templateUrl`

In our component we are specifying a `templateUrl` of `./hello-world.component.html`. This means that we will load our template from the file `hello-world.component.html` in the same directory as our component. Let's take a look at that file:

code/first-app/angular-hello-world/src/app/hello-world/hello-world.component.html

```
<p>
  hello-world works!
</p>
```

Here we're defining a `p` tag with some basic text in the middle. When Angular loads this component it will also read from this file and use it as the template for our component.

Adding a `template`

We can define templates two ways, either by using the `template` key in our `@Component` object or by specifying a `templateUrl`.

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

```
@Component({
  selector: 'app-hello-world',
  template: `
    <p>
      hello-world works inline!
    </p>
  `
})
```

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 you really be putting templates in your 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 switch between a lot of files.

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

The biggest drawback to mixing views and our 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.

Adding CSS Styles with `styleUrls`

Notice the key `styleUrls`:

```
styleUrls: ['./hello-world.component.css']
```

This code says that we want to use the CSS in the file `hello-world.component.css` as the styles for this component. Angular uses a concept called "style-encapsulation" which means that styles specified for a particular component *only apply to that component*. We talk more about this in-depth later on in the book in the Styling section of Advanced Components.

For now, we're not going to use any component-local styles, so you can leave this as-is (or delete the key entirely).

> You may have noticed that this key is different from `template` in that it accepts *an array* as it's argument. This is because we can load multiple stylesheets for a single component.

Loading Our Component

Now that we have our first component code filled out, how do we load it in our page?

If we visit our application again in the browser, we'll see that nothing changed. That's because we only **created** the component, but we're not **using** it yet.

In order to change that, we need to add our component tag to a template that is already being rendered. Open up the file: `first_app/angular-hello-world/src/app/app.component.html`

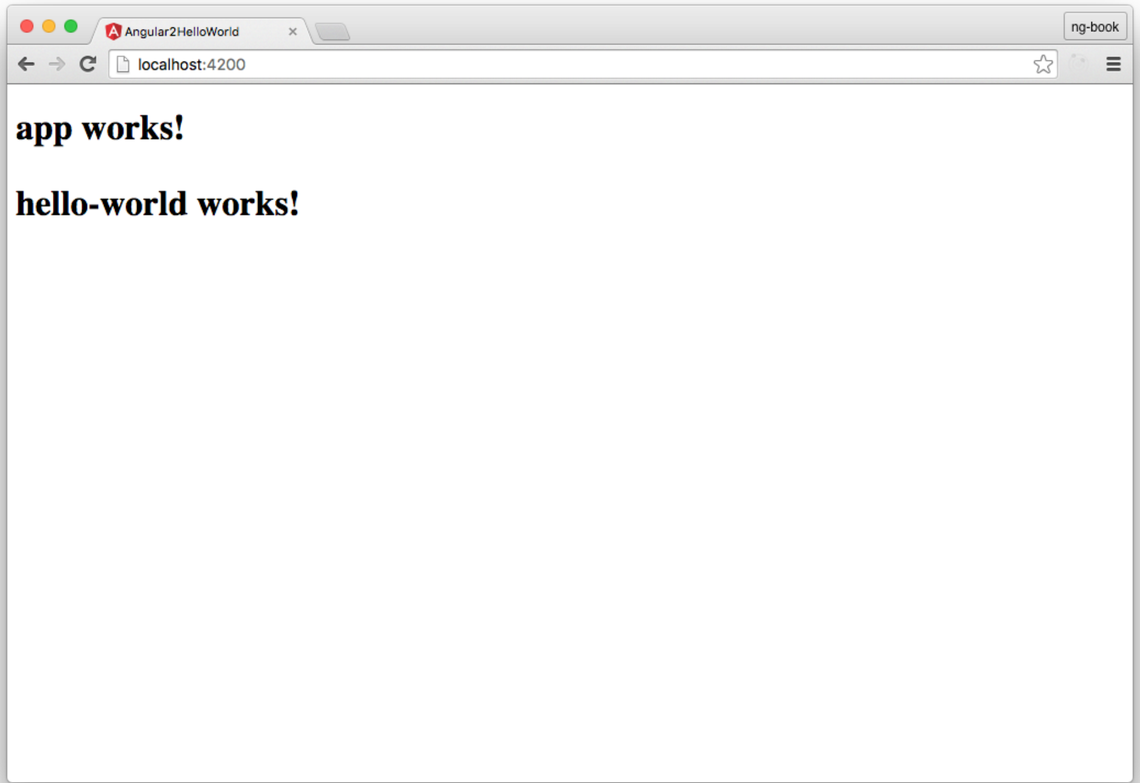
Remember that because we configured our `HelloWorldComponent` with the selector `app-hello-world`, we can use the `<app-hello-world></app-hello-world>` in our template. Let's add the `<app-hello-world>` tag to `app.component.html`

Delete the content in `app.component.html` and replace it with:

code/first-app/angular-hello-world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>
</h1>
```

Now refresh the page and take a look:



Hello world works

It works!

Adding Data to the Component

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

Let's imagine that we have an app which will show **a list of users** and we want to show their names. Before we render the whole list, we first need to render an individual user. So let's create a new component that will show a user's name.

To do this, we will use the `ng generate` command again:

```
ng generate component user-item
```

Remember that in order to see a component we've created, we need to add it to a template.

Let's add our `app-user-item` tag to `app.component.html` so that we can see our changes as we make them. Modify `app.component.html` to look like this:

code/first-app/angular-hello-world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>

  <app-user-item></app-user-item>
</h1>
```

Then refresh the page and confirm that you see the `user-item works!` text on the page.

We want our `UserItemComponent` to show the name of a particular user .

Let's introduce `name` as a new *property* of our component. By having a `name` property, we will be able to reuse this component for different users (but keep the same markup, logic, and styles).

In order to add a name, we'll introduce a property on the `UserItemComponent` class to declare it has a local variable named `name`.

code/first-app/angular-hello-world/src/app/user-item/user-item.component.ts

```
export class UserItemComponent implements OnInit {
  name: string; // <-- added name property

  constructor() {
    this.name = 'Felipe'; // set the name
  }

  ngOnInit() {
  }

}
```

Notice that we've changed two things :

1. `name` Property

On the `UserItemComponent` class we added a *property*. Notice that the syntax is new relative to ES5 JavaScript. When we write `name: string;` it means that we're declaring the `name` property to be of *type* `string`.

Being able to assign a type to a variable is what gives *TypeScript* it's name. By setting the type of this property to `string`, the compiler ensures that `name` variable is a `string` and it will throw an error if we try to assign, say, a `number` to this property.

This syntax is also the way TypeScript defines instance properties. By putting `name: string` in our code like this, we're giving every instance of `UserItemComponent` a property `name`.

2. A Constructor

On the `UserItemComponent` class we defined a *constructor*, i.e. a 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:

code/first-app/angular-hello-world/src/app/user-item/user-item.component.ts

```
  constructor() {
    this.name = 'Felipe'; // set the name
  }
```

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

Rendering The Template

When we have a property on a component, we can show that value in our template by using two curly brackets `{{ }}` to display the value of the variable in our template. For instance:

code/first-app/angular-hello-world/src/app/user-item/user-item.component.html

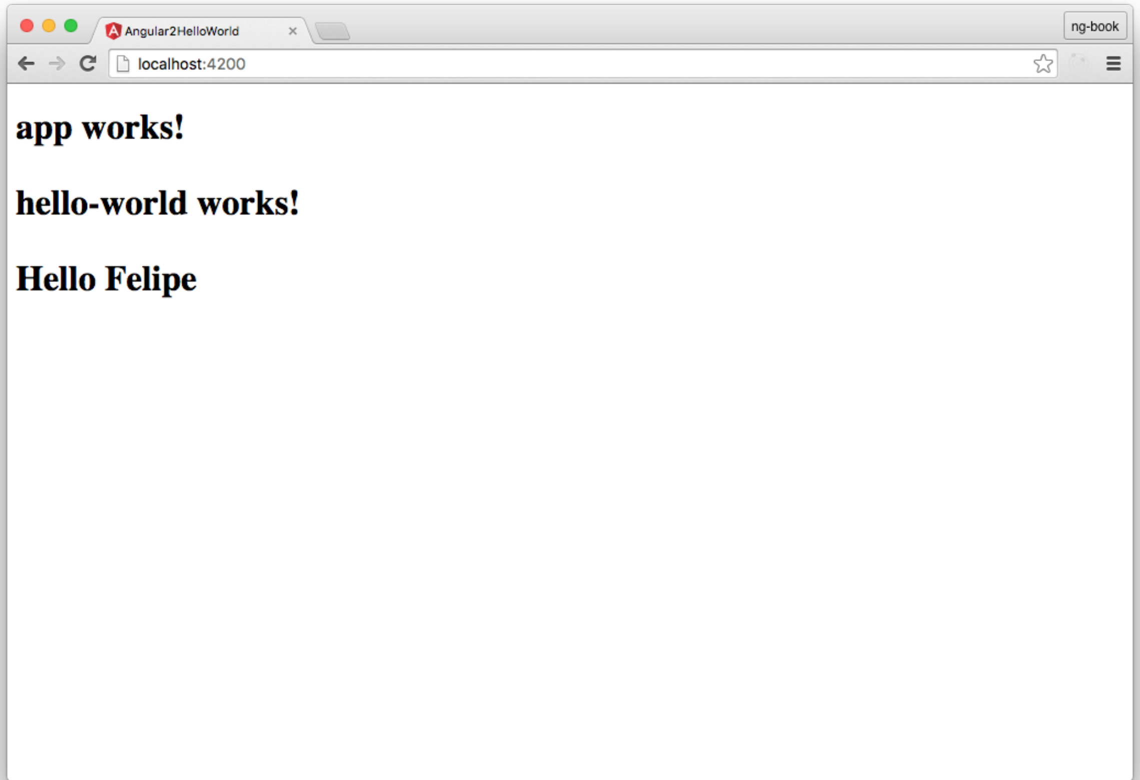
```
<p>
  Hello {{ name }}
</p>
```

On the `template` notice that we added a new syntax: `{{ name }}`. The brackets are called *template tags* (or sometimes *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?

In Angular we can iterate over a list of objects in our template using the syntax `*ngFor`. The idea is that we want to **repeat the same markup for a collection of objects**.

> If you've worked with AngularJS 1.X before, you've probably used the `ng-repeat` directive. `NgFor` works much the same way.

Let's create a new component that will render a *list* of users. We start by generating a new component:

```
ng generate component user-list
```

And let's replace our <app-user-item> tag with <app-user-list> in our app.component.html file:

code/first-app/angular-hello-world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>

  <app-user-list></app-user-list>
</h1>
```

In the same way we added a name property to our UserItemComponent, let's add a names property to this UserListComponent.

However, instead of storing only a single string, let's set the type of this property to *an array of strings*. An array is notated by the [] after the type, and the code looks like this:

code/first-app/angular-hello-world/src/app/user-list/user-list.component.ts

```
export class UserListComponent implements OnInit {
  names: string[];

  constructor() {
    this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
  }

  ngOnInit() {
  }

}
```

The first change to point out is the new string[] property on our UserListComponent 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'].

Now we can update our template to render this list of names. To do this, we will use *ngFor, which will

- iterate over a list of items and
- generate a new tag for each one.

Here's what our new template will look like:

code/first-app/angular-hello-world/src/app/user-list/user-list.component.html

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

We updated the template with one `ul` and one `li` with a new `*ngFor="let name of names"` attribute. The `*` character and `let` syntax 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 `UserListComponent` 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 *local* variable called `name`.

The `NgFor` directive will render one `li` tag for each entry found on the `names` array and 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:

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

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

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

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

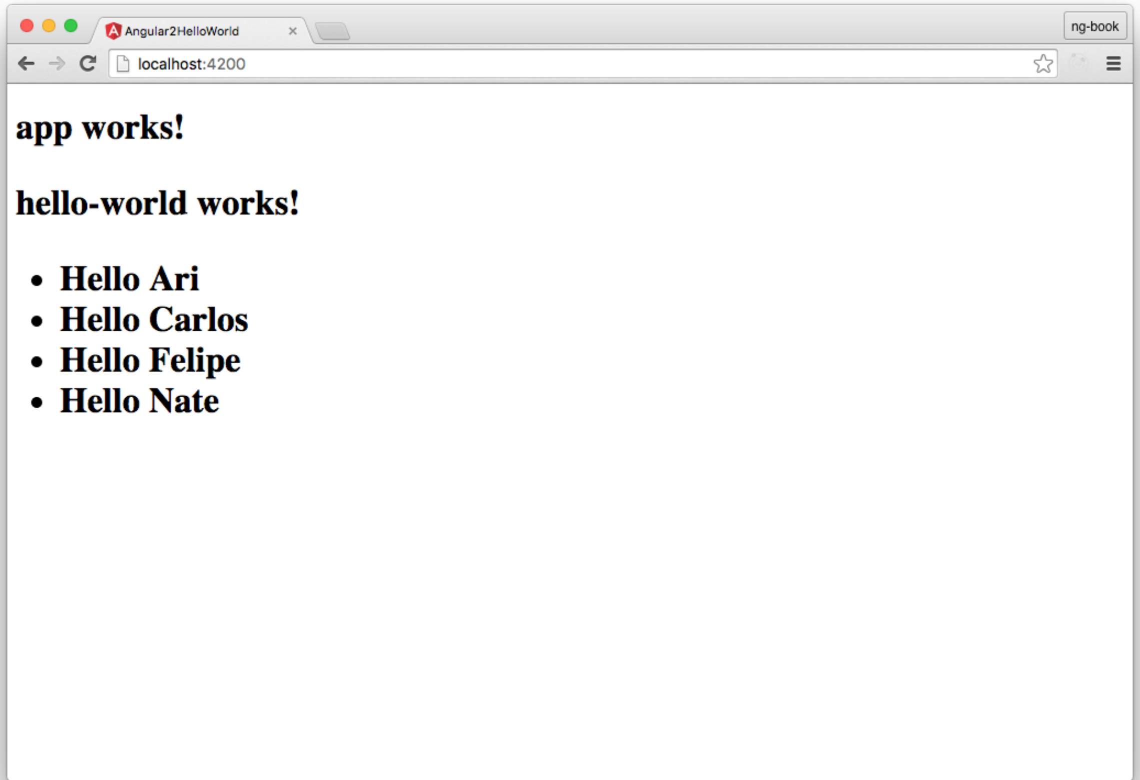
`NgFor` repeats the element that the `ngFor` is called. That is, we put it on the `li` tag and **not** the `ul` tag because we want to repeat the list element (`li`) and not the list itself (`ul`).

Note that the capitalization here isn't a typo: `NgFor` is the capitalization of the *class* that implements the logic and `ngFor` is the "selector" for the attribute we want to use.

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[17].

When we reload the page now, we'll see that we now have one `li` for each string in the array:

[17] https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.ts



Application with Data

Using the User Item Component

Remember that earlier we created a `UserItemComponent`? Instead of rendering each name within the `UserListComponent`, we ought to use `UserItemComponent` as a *child component* - that is, instead of rendering the text `Hello` and the name directly, we should let our `UserItemComponent` specify the template (and functionality) of **each item in the list**.

To do this, we need to do three things:

1. Configure the `UserListComponent` to render to `UserItemComponent` (in the template)
2. Configure the `UserItemComponent` to accept the `name` variable as an *input* and
3. Configure the `UserListComponent` template to **pass the name** to the `UserItemComponent`.

Let's perform these steps one-by-one.

Rendering the `UserItemComponent`

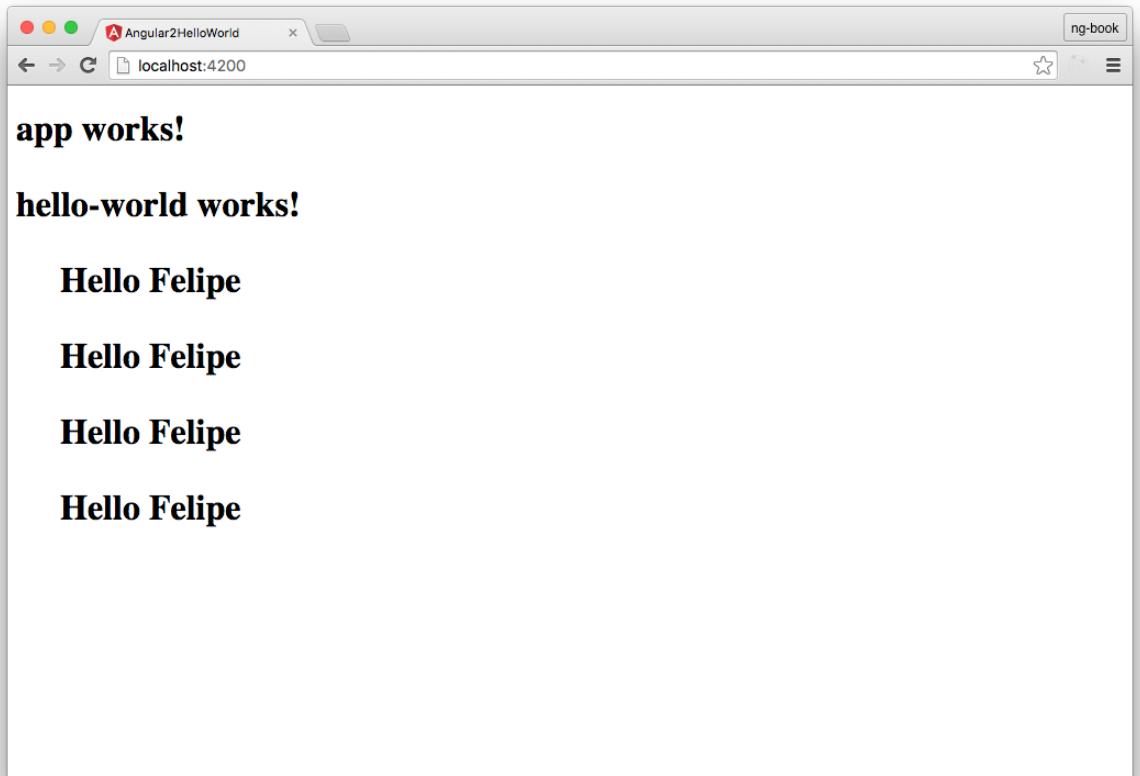
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template:

code/first-app/angular-hello-world/src/app/user-list/user-list.component.html

```
<ul>
  <li *ngFor="let name of names">
    <app-user-item></app-user-item>
  </li>
</ul>
```

Notice that we swapped out the text `Hello` and the `name` for the tag `app-user-item`.

If we reload our browser, this is what we will see:



Application with Data

It repeats, but something is wrong here - every name says “Felipe”! We need a way to *pass data into the child component.*

Thankfully, Angular provides a way to do this: the `@Input` decorator.

Accepting Inputs

Remember that in our `UserItemComponent` we had set `this.name = 'Felipe';` in the `constructor` of that component. Now we need to change this component to accept a value for this property.

Here's what we need to change on our `UserItemComponent`:

code/first-app/angular-hello-world/src/app/user-item/user-item.component.ts

```
import {
  Component,
  OnInit,
  Input // <--- added this
} from '@angular/core';

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.css']
})
export class UserItemComponent implements OnInit {
  @Input() name: string; // <-- added Input annotation

  constructor() {
    // removed setting name
  }

  ngOnInit() {}
}
```

Notice that we changed the `name` property to have a *decorator* of `@Input`. We talk a lot more about `Inputs` (and `Outputs`) in the next chapter, but for now, know that this syntax allows us to pass in a value *from the parent template* .

In order to use `Input` we also had to add it to the list of constants in `import`.

Lastly, we don't want to set a default value for `name` so we remove that from the `constructor`.

So now that we have a `name Input`, how do we actually use it?

Passing an Input value

To pass values to a component we use the *bracket* `[]` syntax in our template - let's take a look at our updated template:

code/first-app/angular-hello-world/src/app/user-list/user-list.component.html

```
<ul>
  <li *ngFor="let name of names">
    <app-user-item [name]="name"></app-user-item>
  </li>
</ul>
```

Notice that we've added a new attribute on our `app-user-item` tag: `[name]="name"` . In Angular when we add an attribute in brackets like `[foo]` we're saying we want to pass a value to the *input* named `foo` on that component.

In this case notice that the `name` on the right-hand side comes from the `let name ...` statement in `ngFor`. That is, consider if we had this instead:

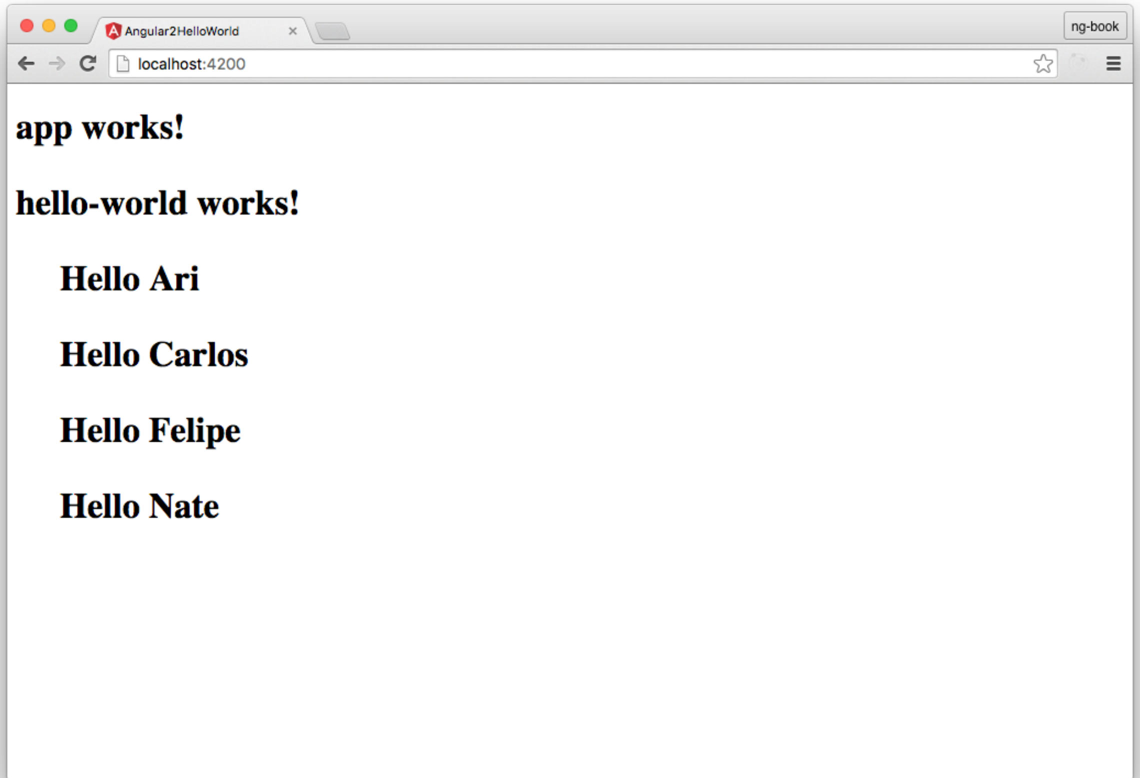
```
<li *ngFor="let individualUserName of names">
  <app-user-item [name]="individualUserName"></app-user-item>
</li>
```

The `[name]` part designates the `Input` on the `UserItemComponent`. Notice that we're *not* passing the literal string `"individualUserName"` instead we're passing the *value* of `individualUserName`, which is, on each pass, the value of an element of `names`.

We talk more about inputs and outputs in detail in the next chapter. For now, know that we're:

1. Iterating over `names`
2. Creating a new `UserItemComponent` for each element in `names` and
3. Passing the value of that name into the `name Input` property on the `UserItemComponent`

Now our list of names works!



Application with Names Working

Congratulations! You've built your first Angular app with components!

Of course, this app is very simple and we'd like to build much more sophisticated applications. Don't worry, in this book we'll show you how to become an expert writing Angular apps. In fact, in this chapter we're going to build a voting-app (think Reddit or Product Hunt). This app will feature user interaction, and even more components!

But before we start building a new app, let's take a closer look at how Angular apps are bootstrapped.

Bootstrapping Crash Course

Every app has a main entry point. This application was built using Angular CLI (which is built on a tool called Webpack). We run this app by calling the command:

```
ng serve
```

`ng` will look at the file `angular.json` to find the entry point to our app. Let's trace how `ng` finds the components we just built.

At a high level, it looks like this:

- `angular.json` specifies a `"main"` file, which in this case is `main.ts`
- `main.ts` is the entry-point for our app and it *bootstraps* our application
- The bootstrap process boots **an Angular module** – we haven't talked about modules yet, but we will in a minute
- We use the `AppModule` to bootstrap the app. `AppModule` is specified in `src/app/app.module.ts`
- `AppModule` specifies which *component* to use as the top-level component. In this case it is `AppComponent`
- `AppComponent` has `<app-user-list>` tags in the template and this renders our list of users.

For now the thing we want to focus on is the Angular module system: `NgModule`.

Angular has a powerful concept of *modules*. When you boot an Angular app, you're not booting a component directly, but instead you create an `NgModule` which points to the component you want to load.

Take a look at this code:

code/first-app/angular-hello-world/src/app/app.module.ts

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

The first thing we see is an `@NgModule` decorator. Like all decorators, this `@NgModule( ... )` code **adds metadata to the class immediately following** (in this case, `AppModule`).

Our `@NgModule` decorator has four keys: `declarations`, `imports`, `providers`, and `bootstrap`.

`declarations`

`declarations` specifies the components that are **defined in this module**. This is an important idea in Angular:

You have to declare components in a `NgModule` before you can use them in your templates.

You can think of an `NgModule` a bit like a "package" and `declarations` states **what components are "owned by" this module**.

You may have noticed that when we used `ng generate`, the tool automatically added our components to this `declarations` list! The idea is that when we generated a new component, the `ng` tool assumed we wanted it to belong to the current `NgModule`.

`imports`

`imports` describes which *dependencies* this module has. We're creating a browser app, so we want to import the `BrowserModule`.

If your module depends on other modules, you list them here.

> **`import` vs. `imports`?**
>
> You might be asking the question, "What's the difference between `import`ing a class at the top of the file and putting a module in `imports`?"
>
> The short answer is that you put something in your `NgModule`'s `imports` if you're going to be using it in your templates or with *dependency injection*. We haven't talked about *dependency injection*, but rest assured, we will.

`providers`

`providers` is used for dependency injection. So to make a service available to be injected throughout our application, we will add it here.

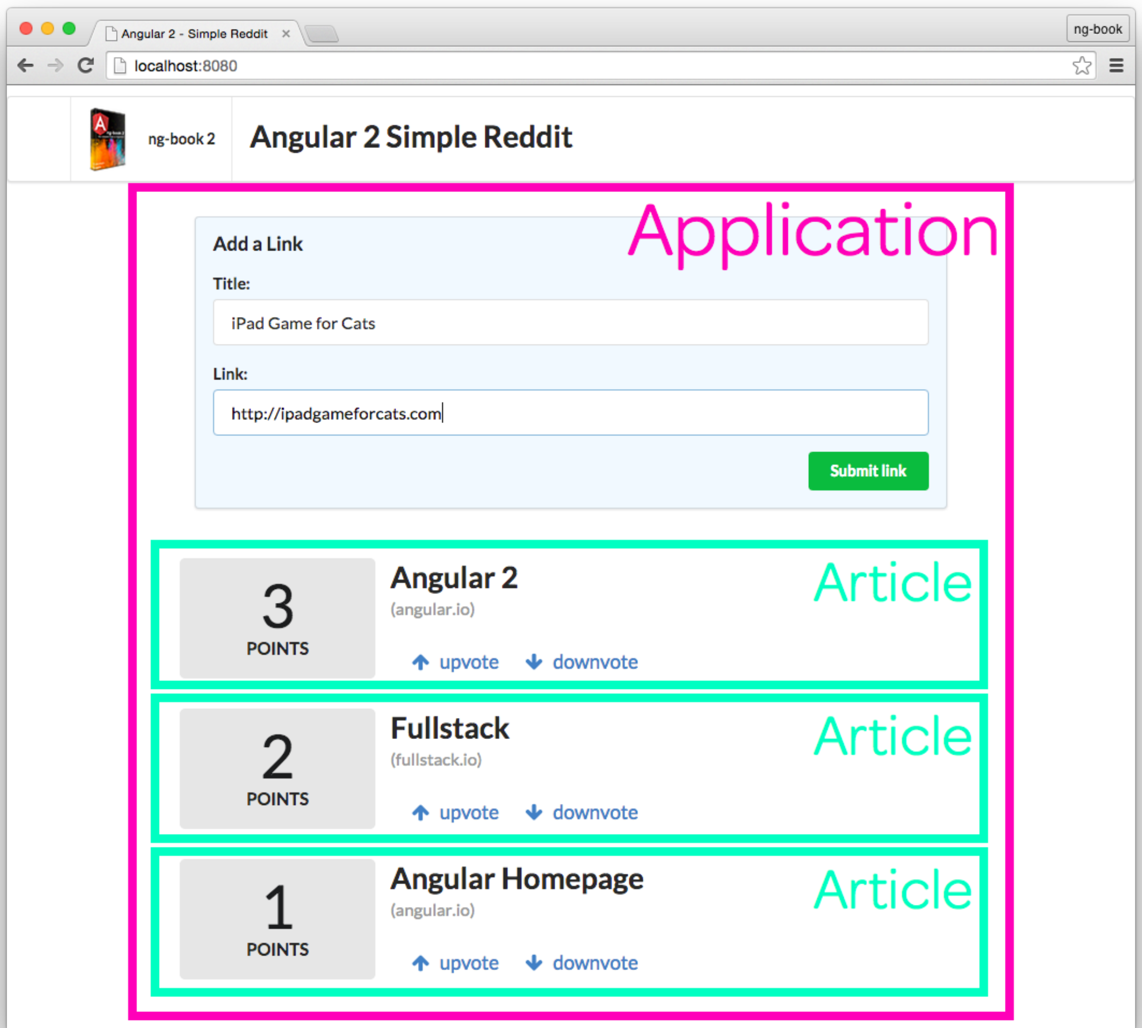
> Learn more about this in the section on `Dependency Injection`.

`bootstrap`

`bootstrap` tells Angular that when this module is used to bootstrap an app, we need to load the `AppComponent` component as the top-level component.

Expanding our Application

Now that we know how to create a basic application, let's build 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 the 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 have only two components.

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

But first thing's first, let's generate a new application by running the same **ng new** command we ran before to create a new application passing it the name of the app we want to create (here, we'll create an application called `angular-reddit`):

```
ng new angular-reddit
```

We've provided a completed version of our `angular-reddit` in the example code download. If you ever need more context, be sure to check it out to see how everything fits together.

Adding CSS

First thing we want to do is add some CSS styling so that our app isn't completely unstyled.

If you're building your app from scratch, you'll want to copy over a few files from our completed example in the `first_app/angular-reddit` folder.

Copy:

- `src/index.html`
- `src/styles.css`
- `src/app/vendor`
- `src/assets/images`

into your application's folder.

For this project we're going to be using Semantic-UI[18] to help with the styling. Semantic-UI is a CSS framework, similar to Zurb Foundation[19] or Twitter Bootstrap[20]. We've included it in the sample code download so all you need to do is copy over the files specified above.

[18] http://semantic-ui.com/
[19] http://foundation.zurb.com
[20] http://getbootstrap.com

The Application Component

Let's now build a new component which will:

1. store our current list of articles
2. contain the form for submitting new articles.

We can find the main application component on the `src/app/app.component.ts` file. Let's open this file. Again, we'll see the same initial contents we saw previously.

code/first-app/angular-reddit/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Notice that the `title` property was automatically generated for us on the `AppComponent`. Remove that line, because we aren't using the component `title`.

Below we're going to be submitting new links that have a 'title', which could be confused with the `AppComponent` title that was auto-generated by Angular CLI. Keep in mind that the form 'title' is a separate form field from the 'title' in the links below.

Let's change the template a bit to include a form for adding links. We'll use a bit of styling from the `semantic-ui` package to make the form look a bit nicer:

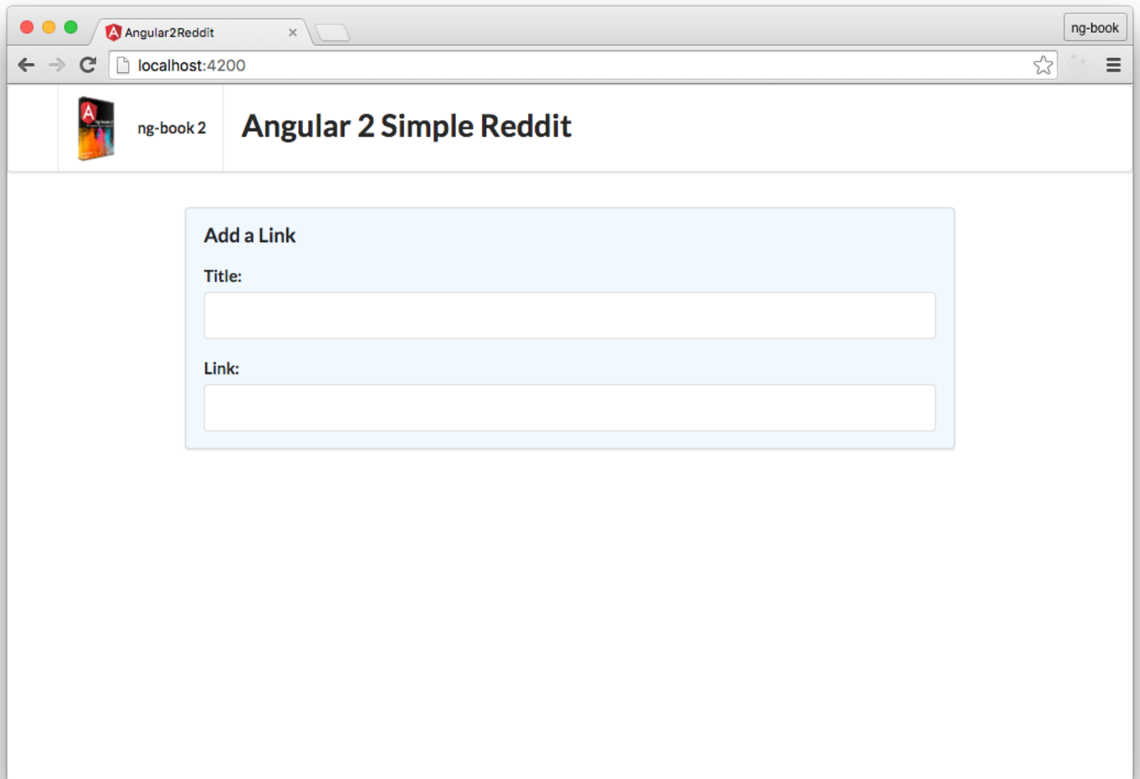
code/first-app/angular-reddit/src/app/app.component.html

```
<form class="ui large form segment">
  <h3 class="ui header">Add a Link</h3>

  <div class="field">
    <label for="title">Title:</label>
    <input name="title" id="title">
  </div>
  <div class="field">
    <label for="link">Link:</label>
```

```
    <input name="link" id="link">
  </div>
</form>
```

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

When we load the browser you should see the rendered form:



Form

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.

When the form is submitted, we'll want to call a function to create and add a link. We can do this by adding an interaction event on the `<button />` element.

We tell Angular we want to respond to an event by surrounding the event name in parentheses `()`. For instance, to add a function call to the `<button />` `onClick` event, we can pass it through like so:

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

Now, when the button is clicked, it will call a function called `addArticle()`, which we need to define on the `AppComponent` class. Let's do that now:

code/first-app/angular-reddit/src/app/app.component.ts

```
export class AppComponent {
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
}
```

With the `addArticle()` function added to the `AppComponent` and the `(click)` event added to the `<button />` element, this function will be called when the button is clicked. Notice that the `addArticle()` function can accept two arguments: the `title` and the `link` arguments. We need to change our template button to pass those into the call to the `addArticle()`.

We do this by populating a *template variable* by adding a special syntax to the `input` elements on our form. Here's what our template will look like:

code/first-app/angular-reddit/src/app/app.component.html

```
<form class="ui large form segment">
  <h3 class="ui header">Add a Link</h3>

  <div class="field">
    <label for="title">Title:</label>
    <input name="title" id="title" #newtitle>  <!-- changed -->
  </div>
  <div class="field">
    <label for="link">Link:</label>
    <input name="link" id="link" #newlink> <!-- changed -->
  </div>

  <!-- added this button -->
  <button (click)="addArticle(newtitle, newlink)"
          class="ui positive right floated button">
    Submit link
  </button>

</form>
```

Notice that in the input tags we used the # (hash) to tell Angular to assign those tags to *a local variable*. By adding the #newtitle and #newlink to the appropriate <input /> elements, we can **pass them as variables** into the addArticle() function on the button!

To recap what we've done, 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:

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

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 AppComponent
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:

```
<button (click)="addArticle(newtitle, newlink)"
        class="ui positive right floated button">
  Submit link
</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 AppComponent` 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.

code/first-app/angular-reddit/src/app/app.component.ts

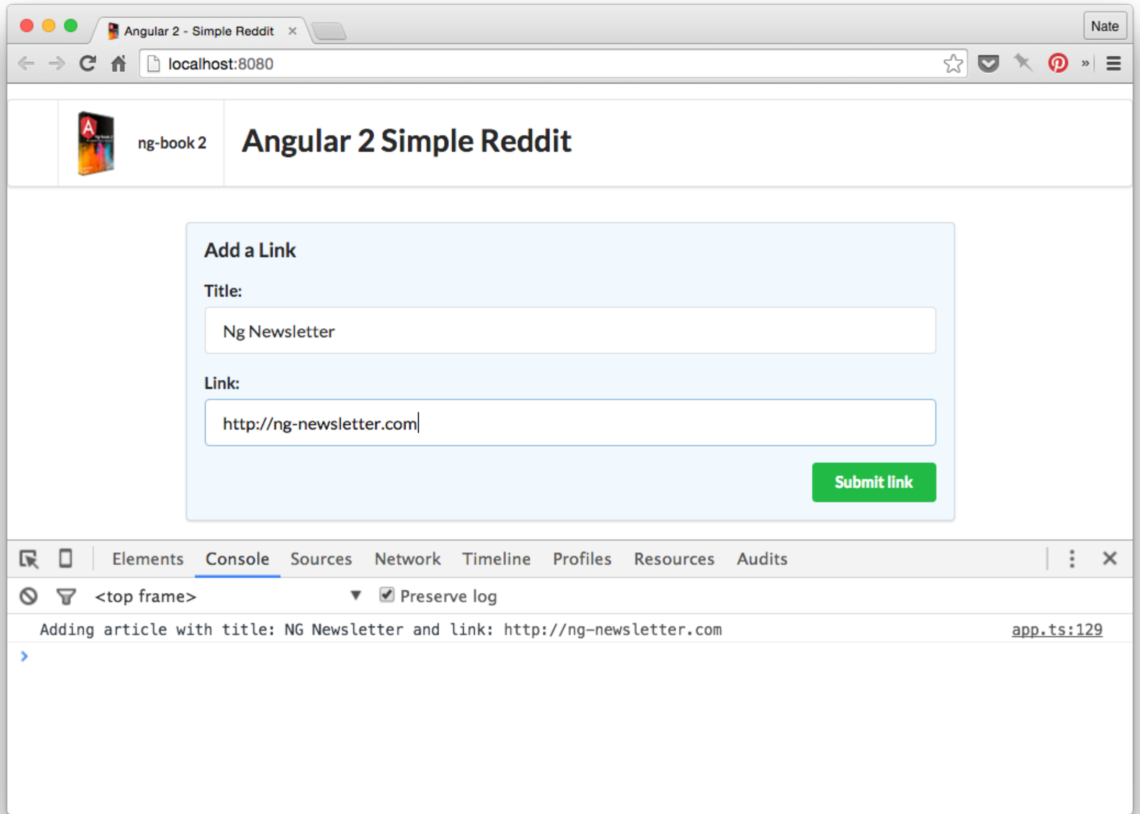
```
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
```

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:

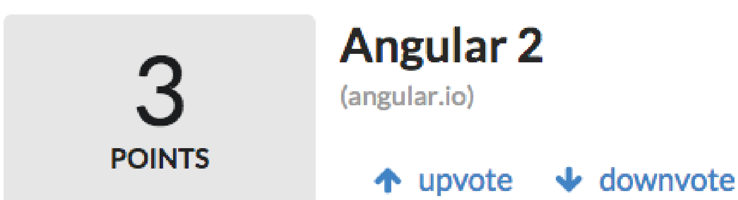


Clicking the Button

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 individual submitted articles.



A reddit-article

For that, let's use the `ng` tool to generate a new component:

```
ng generate component article
```

We have three parts to defining this new component:

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

Let's talk through each part in detail:

Creating the `ArticleComponent` template

We define the template using the file `article.component.html`:

code/first-app/angular-reddit/src/app/article/article.component.html

```
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value">
      {{ votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ link }}">
    {{ title }}
  </a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
          upvote
        </a>
    </li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
      </a>
    </li>
  </ul>
</div>
```

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

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 (remember that these come from SemanticUI's CSS).

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, which we'll define in a minute.

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 `ArticleComponent`

code/first-app/angular-reddit/src/app/article/article.component.ts

```
@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css'],
})
```

First, we define a new Component with `@Component`. The `selector` says that this component is placed on the page by using the tag `<app-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:

```
<app-article>
</app-article>
```

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

Creating the `ArticleComponent` Definition Class

Finally, we create the `ArticleComponent` definition class:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  votes: number;
  title: string;
  link: string;

  constructor() {
    this.title = 'Angular';
    this.link = 'http://angular.io';
    this.votes = 10;
  }

  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }

  ngOnInit() {
  }

}
```

Here we create four properties on `ArticleComponent`:

1. `cssClass` - the CSS `class` we want to apply to the "host" of this component
2. `votes` - a `number` representing the sum of all upvotes, minus the downvotes
3. `title` - a `string` holding the title of the article
4. `link` - a `string` holding the URL of the article

We want each `app-article` to be on its own row. We're using Semantic UI, and Semantic provides a CSS class for rows[21] called `row`.

In Angular, a component *host* is **the element this component is attached to**. We can set properties on the host element by using the `@HostBinding()` decorator. In this case, we're asking Angular to keep the value of the host elements class to be in sync with the property `cssClass`.

> We import `HostBinding` from the package `@angular/core`. For instance we can add `HostBinding` like this:
>
> ```
> import { Component, HostBinding } from '@angular/core';
> ```

By using `@HostBinding()` the **host element** (the `app-article` tag) we want to set the `class` attribute to have "`row`".

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

In the `constructor()` we set some default attributes:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
  constructor() {
    this.title = 'Angular';
    this.link = 'http://angular.io';
    this.votes = 10;
  }
```

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

[21] http://semantic-ui.com/collections/grid.html

code/first-app/angular-reddit/src/app/article/article.component.ts

```
  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }
```

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

Using the `app-article` Component

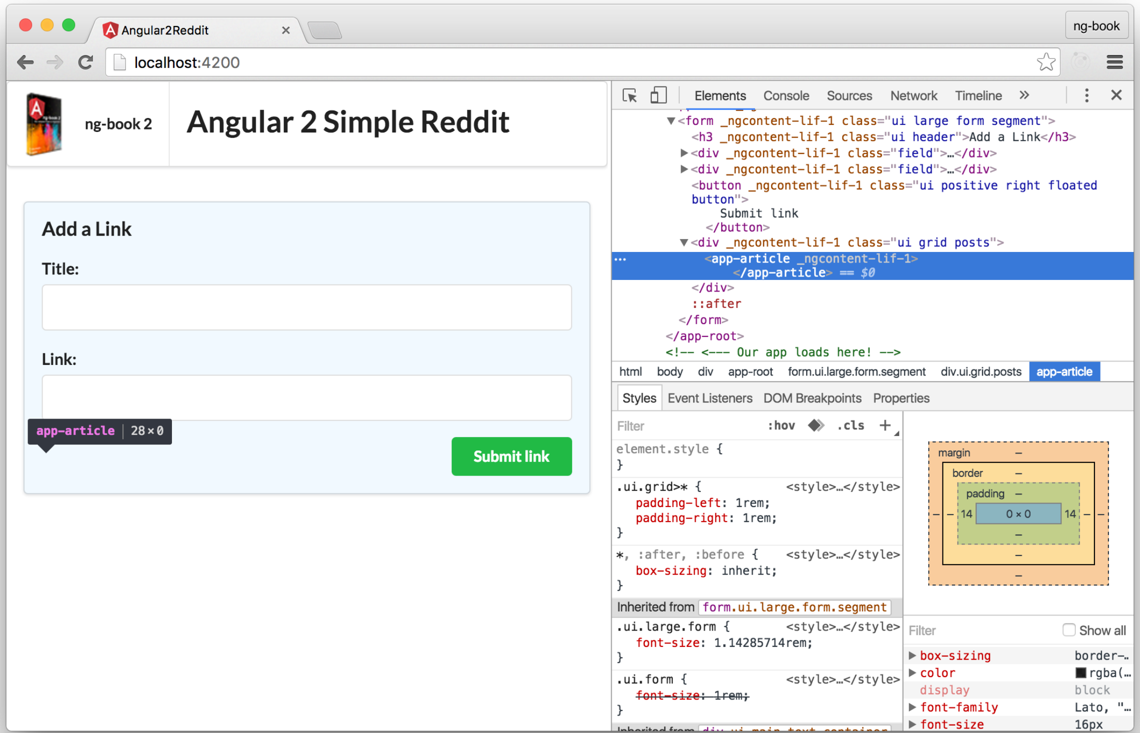
In order to use this component and make the data visible, we have to add a `<app-article></app-article>` tag somewhere in our markup.

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

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

<div class="ui grid posts">
  <app-article>
  </app-article>
</div>
```

If we generated the `ArticleComponent` using Angular CLI (via `ng generate component`), by default it should have "told" Angular about our `app-article` tag (more on that below). However, if we created this component "by hand" and we reload the browser now, we might see that the `<app-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 `app-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 `AppComponent` 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 `app-article` component. This is because in Angular 1, directives match globally. However, in Angular you need to 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 we don't have to share our directive selectors in a global namespace.

In order to tell our `AppComponent` about our new `ArticleComponent` component, we need to **add the `ArticleComponent` to the list of `declarations` in this `NgModule`**.

We add `ArticleComponent` to our `declarations` because `ArticleComponent` is part of this module (`AppModule`). However, if `ArticleComponent` were part of a *different* module, then we might import it with `imports`.

We'll discuss more about `NgModules` later on, but for now, know that when you create a new component, you have to put in a `declarations` in `NgModules`.

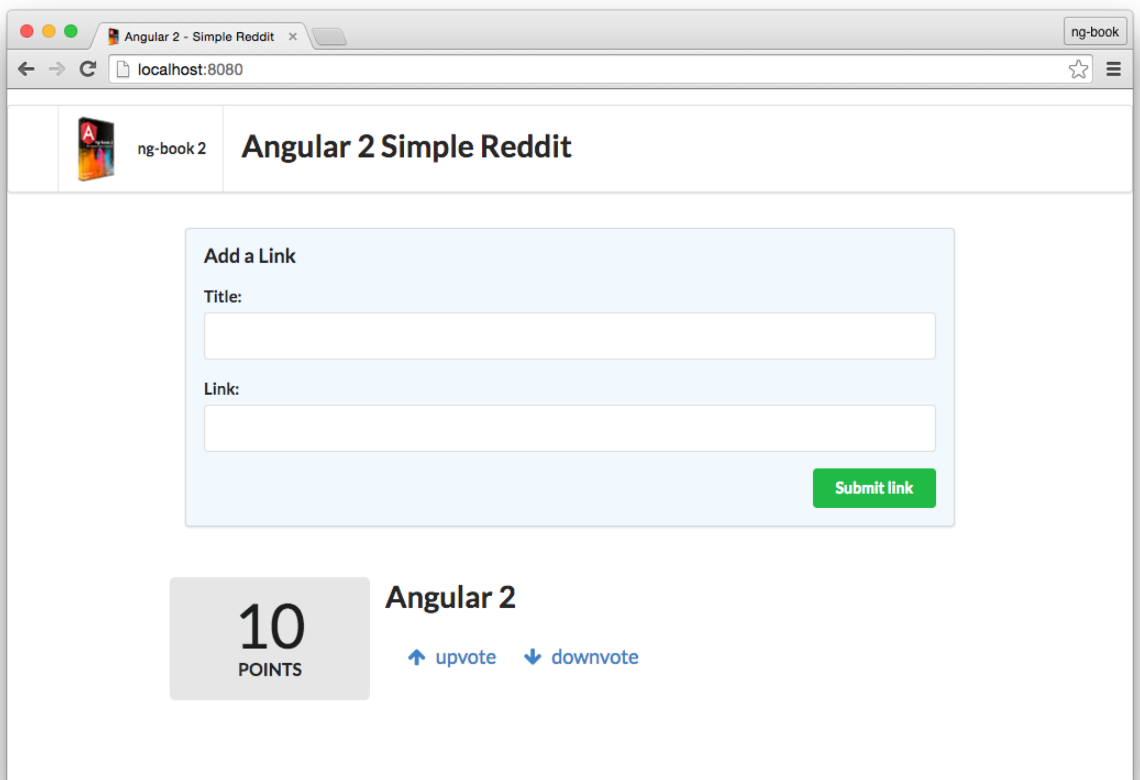
code/first-app/angular-reddit/src/app/app.module.ts

```
import { AppComponent } from './app.component';
import { ArticleComponent } from './article/article.component';

@NgModule({
  declarations: [
    AppComponent,
    ArticleComponent // <-- added this
  ],
```

See here that we are:

1. importing `ArticleComponent` and then
2. Adding `ArticleComponent` to the list of `declarations`

After you've added `ArticleComponent` to `declarations` in the `NgModule`, if we reload the browser we should see the article properly rendered:



Rendered ArticleComponent component

However, clicking on the **vote up** or **vote down** links will **cause the page to reload** instead of updating the article list.

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, which tells the browser to reload.

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. Let's update our code so that each of the functions `voteUp()` and `voteDown()` return a boolean value of `false` (tells the browser *not* to propagate the event upwards):

```
voteDown(): boolean {
  this.votes -= 1;
  return false;
}
// and similarly with `voteUp()`
```

Now when we click the links we'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 `<app-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 we are using from the component code. To do this, let's create a data structure that represents a single article. Let's add a new file `article.model.ts` to define an `Article` class that we can use.

code/first-app/angular-reddit/src/app/article/article.model.ts

```
export class Article {
  title: string;
  link: string;
  votes: number;

  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }
}
```

Here we are creating a new class that represents an `Article`. Note that this is a **plain class and not an Angular 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 (denoted by the `?` at the end of the name) and defaults to zero.

Now let's update 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**.

First let's import the class:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
import { Article } from './article.model';
```

Then let's use it:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular',
      'http://angular.io',
      10);
  }

  voteUp(): boolean {
    this.article.votes += 1;
    return false;
  }

  voteDown(): boolean {
    this.article.votes -= 1;
    return false;
  }

  ngOnInit() {
  }

}
```

Notice what we've changed: instead of storing the `title`, `link`, and `votes` properties directly on the component, 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 }}`, and same with `title` and `link`:

code/first-app/angular-reddit/src/app/article/article.component.html

```
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value">
      {{ article.votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}">
    {{ article.title }}
  </a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
          upvote
        </a>
    </li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
      </a>
    </li>
  </ul>
</div>
```

Reload the browser and everything still works.

This situation is better 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` currently break the Law of Demeter[22] which says that a given object should assume as little as possible about the structure or properties of other objects.

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 (we'll also add a `domain` function, which we'll talk about in a moment):

code/first-app/angular-reddit/src/app/article/article.model.ts

```
export class Article {
  title: string;
  link: string;
  votes: number;

  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }

  voteUp(): void {
    this.votes += 1;
  }

  voteDown(): void {
    this.votes -= 1;
  }

  // domain() is a utility function that extracts
  // the domain from a URL, which we'll explain shortly
  domain(): string {
    try {
      // e.g. http://foo.com/path/to/bar
      const domainAndPath: string = this.link.split('//')[1];
      // e.g. foo.com/path/to/bar
      return domainAndPath.split('/')[0];
    } catch (err) {
      return null;
    }
```

[22] http://en.wikipedia.org/wiki/Law_of_Demeter

```
  }
}
```

We can then change `ArticleComponent` to call these methods:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular',
      'http://angular.io',
      10);
  }

  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }

  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }

  ngOnInit() {
  }

}
```

Why do we have a `voteUp` function in both the model and the component?

The reason we have a `voteUp()` and a `voteDown()` on both classes is because each function does a slightly different thing. The idea is that the `voteUp()` on the `ArticleComponent` relates to the **component view**, whereas the `Article` model `voteUp()` defines what *mutations* happen **in the model**.

That is, it allows the `Article` class to encapsulate what functionality should happen **to a model** when voting happens. In a "real" app, the internals of the `Article` model would probably be more complicated, e.g. make an API request to a webserver, and you wouldn't want to have that sort of model-specific code in your component controller.

Similarly, in the `ArticleComponent` we `return false;` as a way to say "don't propagate the event" - this is a view-specific piece of logic and we shouldn't allow the `Article` model's `voteUp()` function to have to knowledge about that sort of view-specific API. That is, the `Article` model should allow voting apart from the specific view.

After reloading our browser, we'll notice everything works the same way, but we now have clearer, simpler code.

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[23]. The idea is that we want to move most of our logic to our models so that our components do the minimum work possible.

Storing Multiple `Article`s

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

Let's start by changing `AppComponent` to have a collection of articles:

code/first-app/angular-reddit/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { Article } from './article/article.model'; // <-- import this

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  articles: Article[]; // <-- component property

```

[23] http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model

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

Notice that our `AppComponent` has the line:

```
  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 our collection (the `Array`) is typed. That is, the `Array` is a collection that will only hold objects of type `Article`.

In order to have access to the `Article` class, we first have to `import` it, as we do up top.

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

code/first-app/angular-reddit/src/app/app.component.ts

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

Configuring the `ArticleComponent` with `inputs`

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

Here again we use `Inputs`. Previously we had our `ArticleComponent` class defined like this:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular',
      'http://angular.io',
      10);
  }
```

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 `app-article` component by passing an `Article` as a "parameter" to the component like this:

```
<app-article [article]="article1"></app-article>
<app-article [article]="article2"></app-article>
```

Angular allows us to do this by using the `Input` decorator on a property of a `Component`:

```
class ArticleComponent {
  @Input() article: Article;
  // ...
```

Now if we have an `Article` in a variable `myArticle` we could pass it to our `ArticleComponent` in our view. Remember, we can pass a variable in an element by surrounding it in square brackets `[variableName]`, like so:

```
<app-article [article]="myArticle"></app-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 into that input.

Then, and this is important, the `this.article` on the `ArticleComponent` instance will be set to `myArticle`. We can think about the variable `myArticle` as being passed as a *parameter* (i.e. input) to our components.

Here's what our `ArticleComponent` component now looks like using `@Input`:

code/first-app/angular-reddit/src/app/article/article.component.ts

```
import {
  Component,
  OnInit,
  Input,        // <-- added,
  HostBinding
} from '@angular/core';
import { Article } from './article.model'; // <-- added

@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css']
})
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  @Input() article: Article;

  constructor() {
    // article is populated by the Input now,
    // so we don't need anything here
  }

  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }

  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }

  ngOnInit() {
  }

}
```

Don't forget to `import`!

Notice that we `import` the `Input` class from `@angular/core`. We've also imported our `Article` model as we did with the `AppComponent` earlier.

Rendering a List of Articles

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

Let's add this in the `template` of the `AppComponent` `@Component`, just below the closing `<form>` tag:

```
    Submit link
  </button>
</form>

<!-- start adding here -->
<div class="ui grid posts">
  <app-article
    *ngFor="let article of articles"
    [article]="article">
  </app-article>
</div>
<!-- end adding here -->
```

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

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

To specify the `article` input on a component, we are using 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`.

We are 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`:

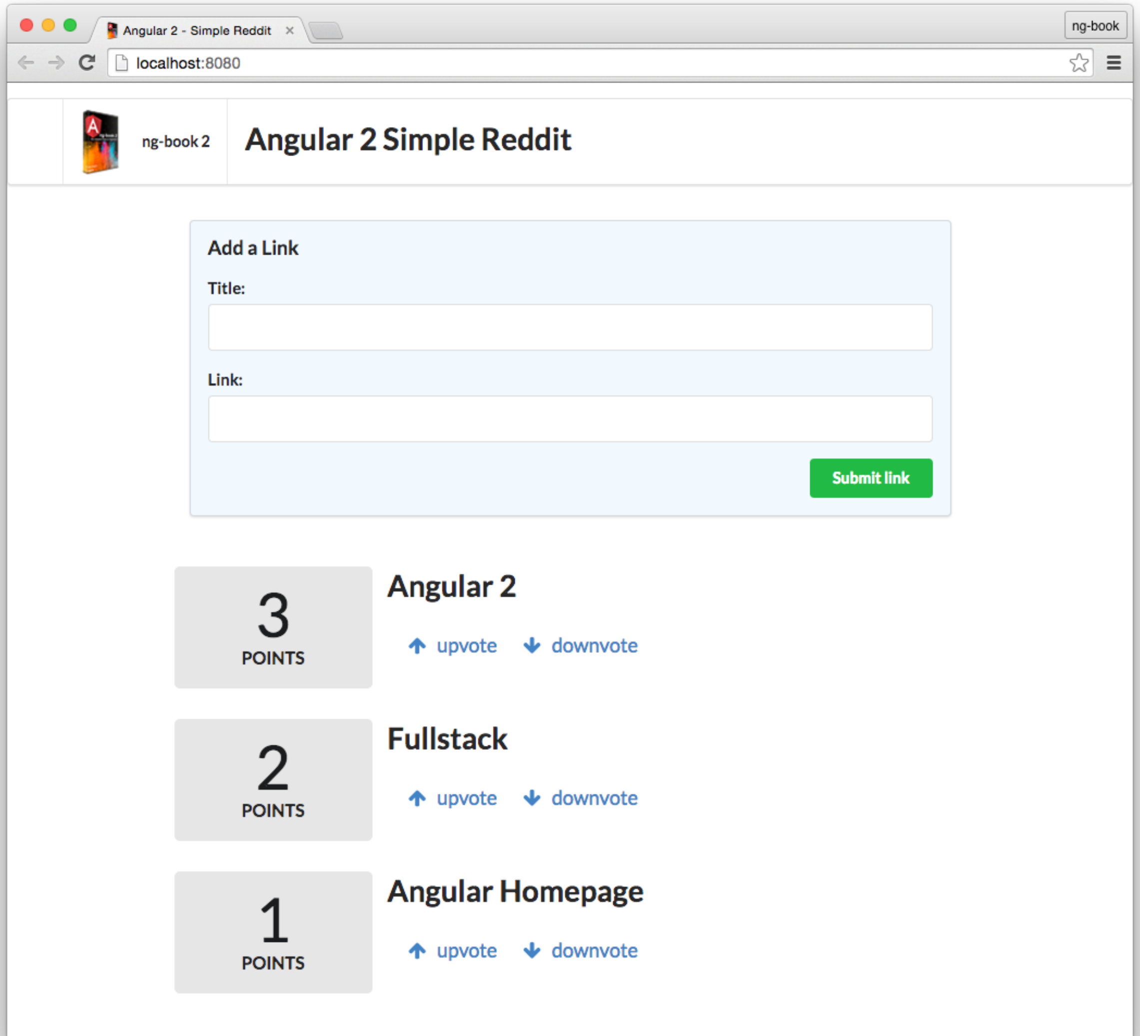
```
<app-article
  *ngFor="let foobar of articles"
  [article]="foobar">
</app-article>
```

So here we have three variables:

1. `articles` which is an `Array` of `Articles`, defined on the `AppComponent`
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 `app-article`

Reloading our browser now, we will see all articles will be rendered:



Multiple articles being rendered

Adding New `Article`s

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/angular-reddit/src/app/app.component.ts

```
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    this.articles.push(new Article(title.value, link.value, 0));
    title.value = '';
    link.value = '';
    return false;
  }
```

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.

After adding a new article in our input fields and clicking the **Submit Link** we 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.

Let's add a `domain` method to the `Article` class:

code/first-app/angular-reddit/src/app/article/article.model.ts

```
  domain(): string {
    try {
      // e.g. http://foo.com/path/to/bar
      const domainAndPath: string = this.link.split('//')[1];
      // e.g. foo.com/path/to/bar
      return domainAndPath.split('/')[0];
    } catch (err) {
      return null;
    }
  }
```

Let's add a call to this function on the `ArticleComponent`'s template:

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

And now when we reload the browser, we will see the domain name of each URL (note: URL must include *http://*).

Re-sorting Based on Score

Clicking and voting on articles, we'll see that something doesn't feel quite right: our articles don't sort based on the score! We definitely want to see the highest-rated items on top and the lower ranking ones sink to the bottom.

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

code/first-app/angular-reddit/src/app/app.component.ts

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

ES6 Arrow Function

The above code snippet uses "arrow" (=>) functions from ES6. You can read more about arrow functions here[24]

`sort()` We're also calling the `sort()` function, which is a built-in which you can read about here[25]

In our `ngFor` we can iterate over `sortedArticles()` (instead of `articles` directly):

```
<div class="ui grid posts">
  <app-article
    *ngFor="let article of sortedArticles()"
    [article]="article">
  </app-article>
</div>
```

Deployment

Now that we have an app that runs, let's get it live on the internet, so that we can share it with our friends!

Deployment and performance in production-ready apps is an intermediate topic that we'll cover in a future chapter. For now, we're going to gloss over the details and just show how easy a basic deployment can be.

Deploying our app is the act of pushing our code to a server, where it can be accessed by others.

Broadly speaking, the idea is that we're going to:

- *compile* all of our TypeScript code into JavaScript (which the browser can read)
- *bundle* all of our JavaScript code files into one or two files
- and then *upload* our JavaScript, HTML, CSS, and images to a server

Ultimately, this Angular app is an HTML file that *loads JavaScript code*. So we need to upload our code to a computer somewhere on the internet.

But first, let's **build** our Angular app.

[24] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

Building Our App for Production

The Angular CLI tool we used to generate this app can be used to build our app for production. In fact, we just type a single command.

In `first_app/angular-reddit`, type the following:

```
ng build --prod
```

This command tells the `ng` tool to **build** our application for a `production` environment.

This command will run for a little while and when it finishes you should have a `dist` folder on your disk.

```
$ ls dist/angular-reddit
136B  assets/
5.3K  favicon.ico
 27K  flags.9c74e172f87984c48ddf.png
306K  icons.2980083682e94d33a66e.svg
119K  icons.706450d7bba6374ca02f.ttf
 55K  icons.97493d3f11c0a3bd5cbd.woff2
 70K  icons.d9ee23d59d0e0e727b51.woff
 59K  icons.f7c2b4b747b1a225eb8d.eot
1.1K  index.html
1.4K  inline.44deb5fed75ee6385e18.bundle.js
 17K  main.c683e6eda100e8873d71.bundle.js
 82K  polyfills.b81504c68200c7bfeb16.bundle.js
503K  styles.7f23e351d688b00e8a5b.bundle.css
440K  vendor.cc4297c08c0803bddc87.bundle.js
```

These files are the full compiled result of your app. Notice that there is a long string of characters in the middle of each file such as:

```
main.c683e6eda100e8873d71.bundle.js
```

Those characters are a hash of the content (and may not match on your computer). If you look at each file, you can see that we have some icons, the `index.html`, a `main.js`, a `polyfills.js`, a `vendor.js`, and some `styles.css`. Now all the need to do is upload these to our server.

Uploading to a Server

There are **lots** of ways to host your HTML and JavaScript. For this demo, we're going to use the easiest way possible: now[26].

[26] https://zeit.co/now

If you don't want to use `now`, you're free to use whatever method you want. For instance, you can host sites on Heroku, AWS S3, upload files to your own server via FTP, etc.

The important thing is that the server exposes all of the files in our `dist` folder onto the internet.

Installing `now`

We can install `now` using `npm`:

```
npm install -g now
```

To deploy a site with now is very easy:

```
cd dist/angular-reddit # change into the dist folder
now
```

The `now` command should ask you a couple of questions (such as your email address) and you'll need to check your email and click the link inside.

After you've confirmed your account (or if you had one already), `now` will **upload your code** and then **give you a URL** to view to see your application.

Visit that URL and view your app. If it works, try sending the URL to a friend!

Congratulations! You've built and deployed your first Angular app!

Full Code Listing

We've been exploring many small pieces of code for this chapter. You can find all of the files and the complete TypeScript code for our app in the example code download included with this book.

Wrapping Up

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

But for now, bask in our success! Much of writing Angular 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

In the future chapters of this book we'll cover everything you need to write sophisticated apps with Angular.

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 community and chat with us on Gitter[27]
- Email us directly at us@fullstack.io[28]

Onward!

[27] https://gitter.im/ng-book/ng-book
[28] mailto:us@fullstack.io

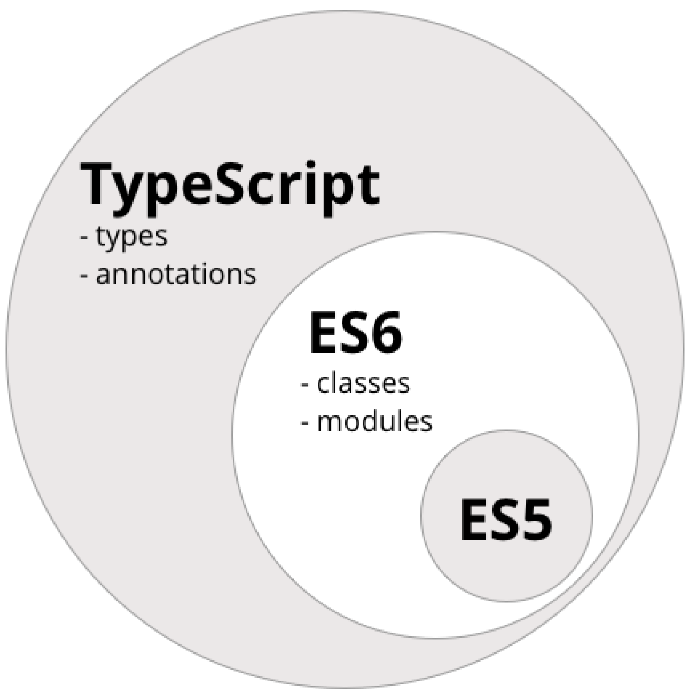
TypeScript

Angular is built in TypeScript

Angular is built in a JavaScript-like language called TypeScript[29].

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.

[29] http://www.typescriptlang.org/

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**[30] by Google and **babel**[31] 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
- decorators
- imports
- language utilities (e.g. destructuring)

Let's deal with these one at a time.

[30] https://github.com/google/traceur-compiler
[31] https://babeljs.io/

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 fullName;`.

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:

```
var fullName: string;
```

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

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

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:

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

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

```
$ tsc compile-error.ts
compile-error.ts(2,12): error TS2322: Type 'number' is not assignable to type 'strin\
g'.
```

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`:

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

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 in this chapter, let's install a nice little utility called **TSUN**[32] (TypeScript Upgraded Node):

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

Now start tsun:

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl

>
```

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 follow along.

[32] https://github.com/HerringtonDarkholme/typescript-repl

Built-in types

String

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

```
var fullName: string = 'Nate Murray';
```

Number

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

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

Boolean

The `boolean` holds either `true` or `false` as the value.

```
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:

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

Or similarly with a `number`:

```
var chickens: Array<number> = [1, 2, 3];
var chickens: 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:

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

The default initial value for an enum is `0`, though you can set the starting enum number like this:

```
enum Role {Employee = 3, Manager, Admin};
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:

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

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

```
enum Role {Employee, Manager, Admin};
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:

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

Void

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

```
function setName(name: string): void {
  this.fullName = name;
}
```

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[33], and you can find a good overview on the Introduction to Object-Oriented JavaScript[34] 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:

```
class Vehicle {
}
```

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

```
class Person {
  first_name: string;
  last_name: string;
  age: number;
}
```

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:

[33] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
[34] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  greet() {
    console.log("Hello", this.first_name);
  }
}
```

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:

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

// instantiate a new Person instance
p = new Person();

// give it a first_name
p.first_name = 'Felipe';

// call the greet method
p.greet();
```

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

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}
```

```
// instantiate a new Person instance
var p: Person = new Person();

// set initial age
p.age = 6;

// how old will he be in 12 years?
p.ageInYears(12);

// -> 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:

```
class Vehicle {
}
var v = new Vehicle();
```

Is the same as:

```
class Vehicle {
  constructor() {
  }
}
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:

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  constructor(first_name: string, last_name: string, age: number) {
    this.first_name = first_name;
    this.last_name = last_name;
    this.age = age;
  }

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}
```

It makes our previous example a little easier to write:

```
var p: Person = new Person('Felipe', 'Coury', 36);
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[35].

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:

```
class Report {
  data: Array<string>;

  constructor(data: Array<string>) {
    this.data = data;
  }

  run() {
    this.data.forEach(function(line) { console.log(line); });
  }
}
```

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:

[35] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

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

Running this should show:

```
First line
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:

```
class TabbedReport extends Report {
  headers: Array<string>;

  constructor(headers: string[], values: string[]) {
    super(values)
    this.headers = headers;
  }

  run() {
    console.log(this.headers);
    super.run();
  }
}
```

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

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

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

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

Parentheses are optional when there's only one parameter. The `=>` syntax can be used both as an expression:

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

Or as a statement:

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

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:

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    var self = this;
    this.guitars.forEach(function(g) {
      // this.name is undefined so we have to use self.name
      console.log(self.name + " plays a " + g);
    });
  }
};
```

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

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    this.guitars.forEach( (g) => {
      console.log(this.name + " plays a " + g);
    });
  }
};
```

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:

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

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:

```
var template = `
<div>
  <h1>Hello</h1>
  <p>This is a great website</p>
</div>
`

// 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
- Decorators
- 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. We're going to take a step back so that we can see how all the pieces fit together.s

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

In the chapters that follow, we won't be taking a deep dive into each concept, but instead we're going to give an overview and explain the foundational ideas.

The first big idea is that an Angular 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 AngularJS 1.x (it turns out, Angular has directives too, but we'll talk more about this distinction later on).

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

Application

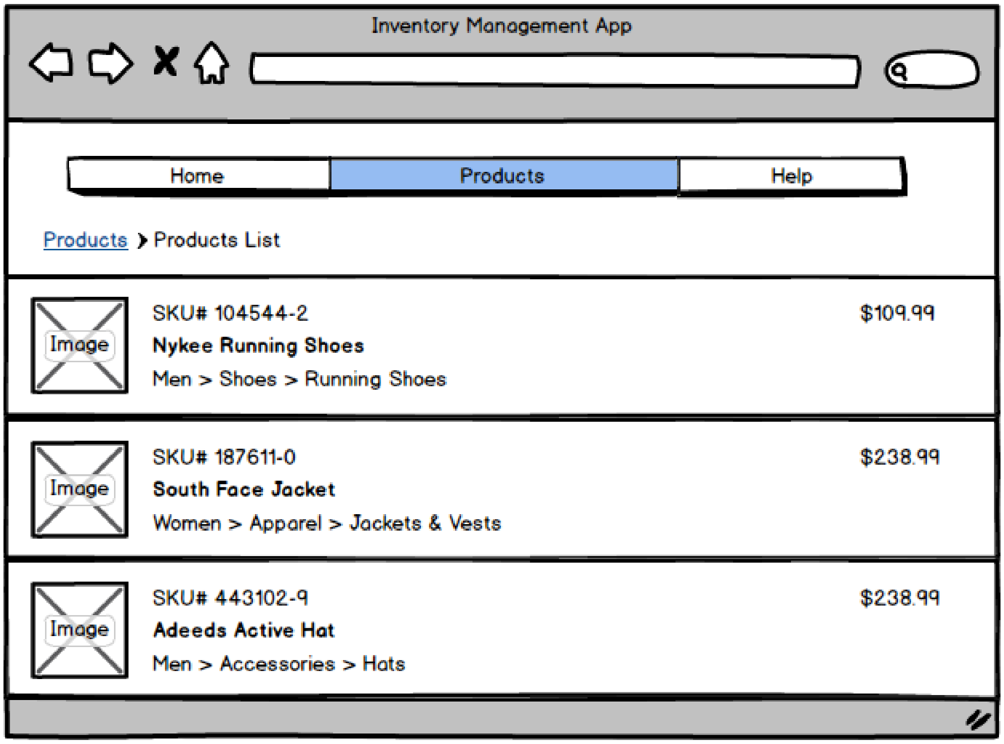
An Angular Application is nothing more than a tree of Components.

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 List 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

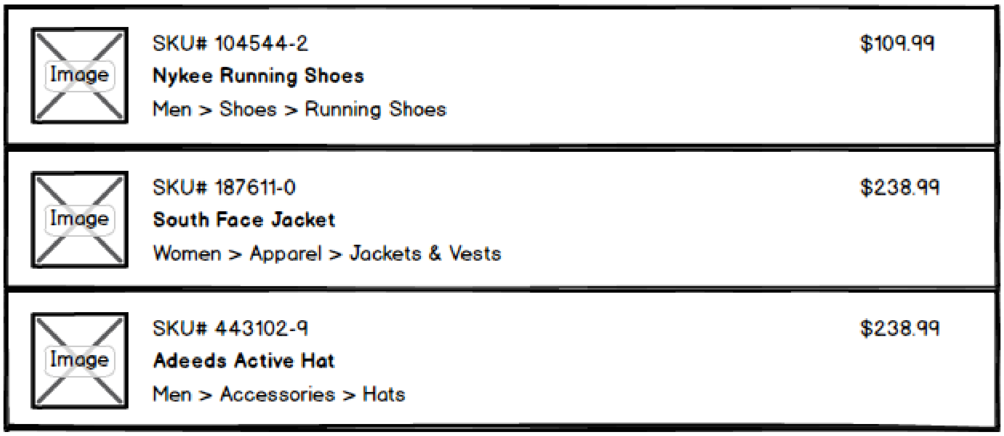
The Breadcrumbs Component

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

Breadcrumbs Component

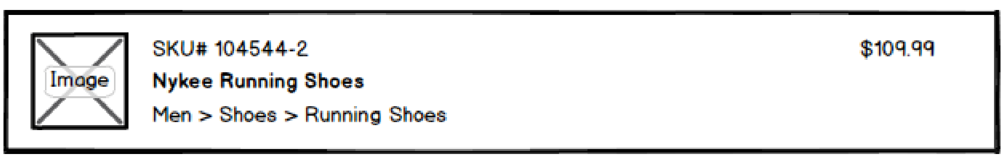
The Product List Component

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



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.

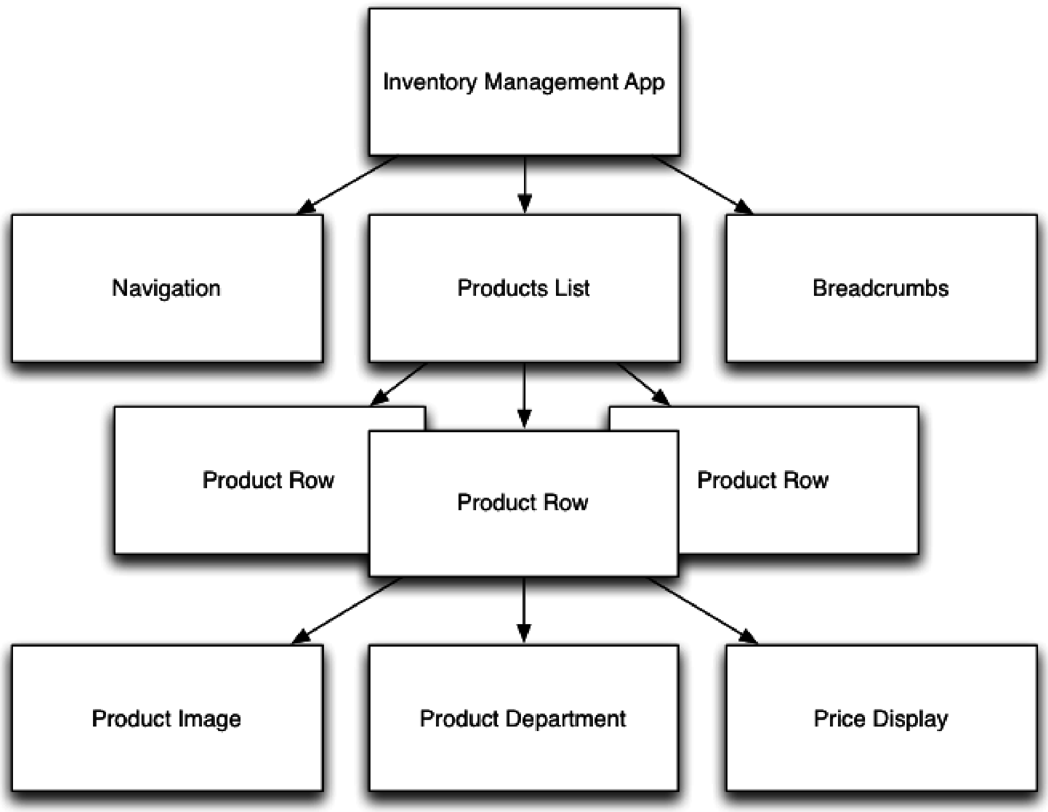


Product Row Component

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.

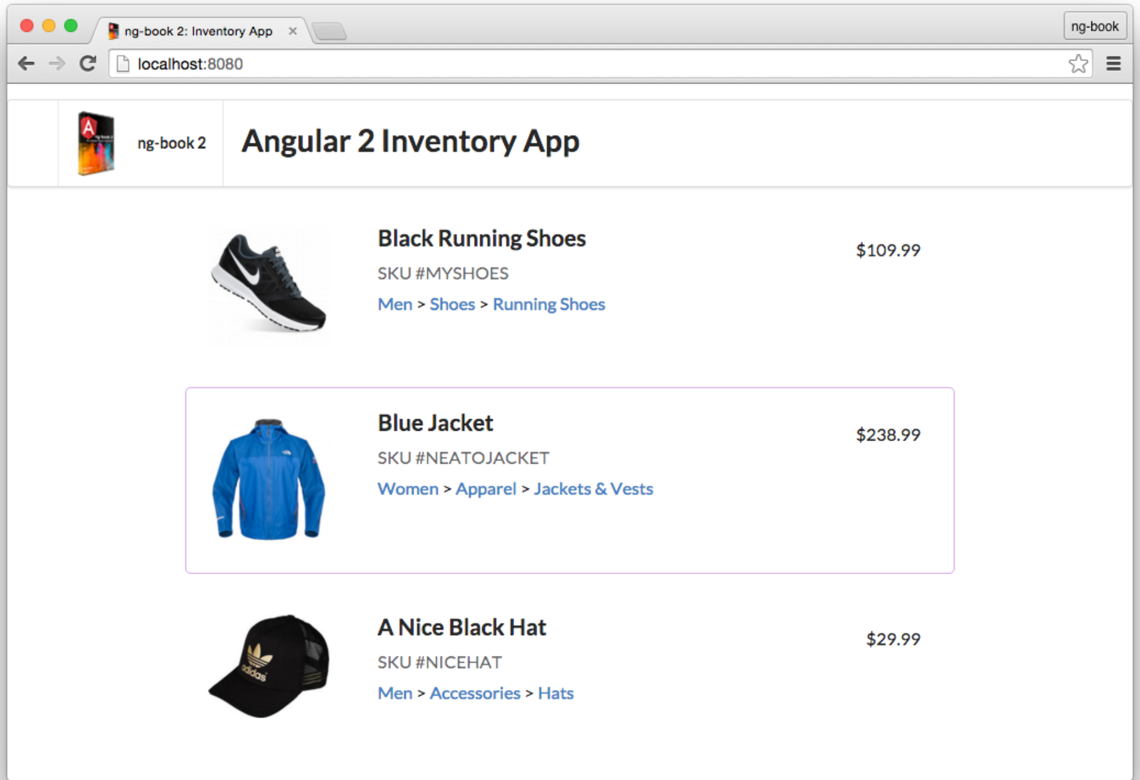
The Products List component has Product Rows, one for each product.

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

How to Use This Chapter

In this chapter we're going to explain the fundamental concepts required when building Angular apps by walking through an app that we've built. We'll explain:

- How to break your app into components
- How to make reusable components using *inputs*
- How to handle user interactions, such as *clicking* on a component

In this chapter, we've used `angular-cli`, just as we did before. This means you can use all of the normal `ng` commands such as:

```
ng serve # runs the app
```

Also, in this chapter, we're not going to give step-by-step instructions on how to create each file in the app. If you'd like to follow along at home, when we introduce a new component you can run:

```
ng generate component NameOfNewComponentHere
```

This will generate the files you need, and you're free to type in your code there or copy and paste code from the book or from our example code.

We've provided the entire, completed application in the code download folder under `how-angular-works/inventory-` If you ever feel lost or need more context, take some time to look at the completed example code.

Let's get started building!

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/src/app/product.model.ts

```
/**
 * Provides a `Product` object
 */
export class Product {
  constructor(
    public sku: string,
    public name: string,
    public imageUrl: string,
    public department: string[],
    public price: number) {
  }
}
```

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[36]

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 applications. The "application" itself is just the top-level Component. Then we break our application into smaller child Components.

> When building new Angular applications, we often follow this process: we mockup the design in wireframes (or on paper) and then we break down the parts into Components.

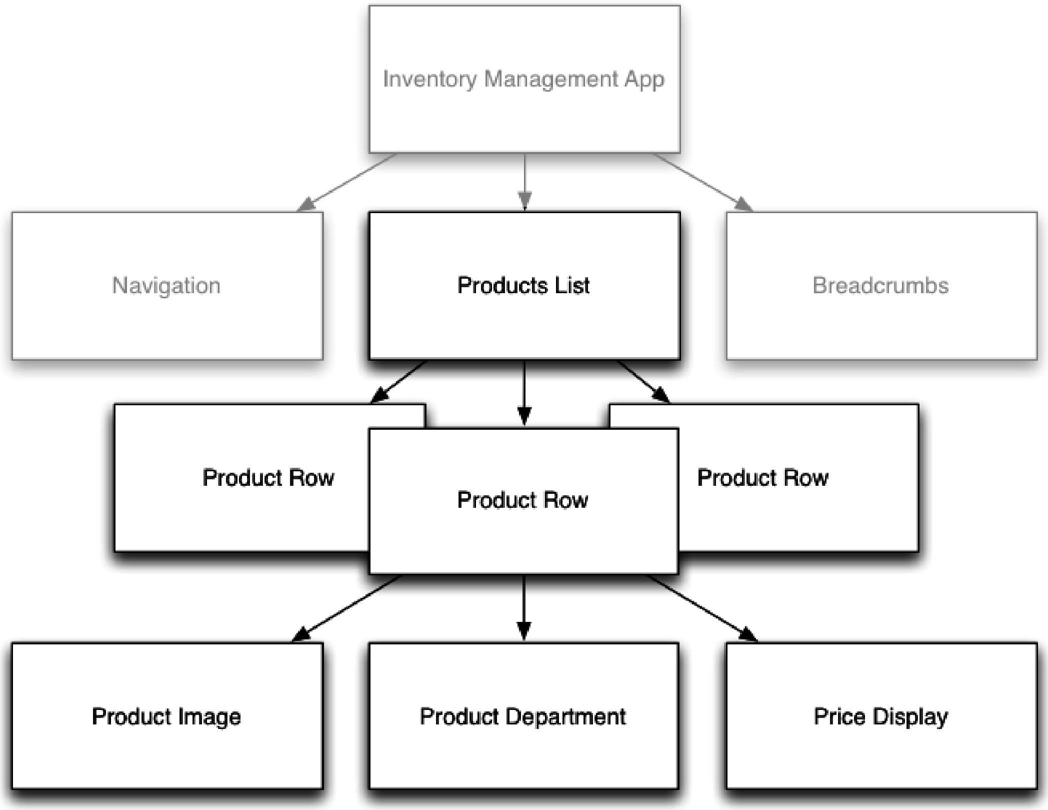
We'll be using Components a lot, so it's worth looking at them more closely.

Each component 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:

[36] https://learnxinyminutes.com/docs/typescript/



Products List Component

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

```
@Component({
  selector: 'inventory-app-root',
  template: `
  <div class="inventory-app">
     (Products will go here soon)
  </div>
  `
})
export class AppComponent {
  // Inventory logic here
}
```

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 (`AppComponent`).

The `@Component` decorator 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 `AppComponent` 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. One of the primary roles of the `@Component` decorator is to configure how the outside world will interact with your component.

> There are lots of options available to configure a component (many of which we cover in the Advanced Components Chapter). In this chapter we're just going to touch on the basics.

Component `selector`

With the `selector` key, you indicate how your component will be recognized when used in a template. 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-root'`, we're saying that in our HTML we want to match the `inventory-app-root` 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:

```
<inventory-app-root></inventory-app-root>
```

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

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 use:

```
@Component({
  selector: 'inventory-app-root',
  template: `
  <div class="inventory-app">
     (Products will go here soon)
  </div>
  `
})
```

For the `template` above, 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.

We can also move our template out to a separate file and use `templateUrl` instead:

```
@Component({
  selector: 'inventory-app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // Inventory logic here
}
```

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:

```
// this is just an example of how to use Product,
// we'll do something similar in our Angular code in a moment

// first, we have to import `Product` so that we can use it
import { Product } from './product.model';

// now we can create a new `Product`
let newProduct = new Product(
      'NICEHAT',                               // sku
      'A Nice Black Hat',                      // name
      '/assets/images/products/black-hat.jpg', // imageUrl
      ['Men', 'Accessories', 'Hats'],          // department
      29.99);                                  // price
```

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 a few arguments to a function. Another option here is to configure the `Product` class to take an Object in the constructor, then we wouldn't have to remember the order of the arguments. That is, `Product` could be changed to do something like this:
>
> ```
> new Product({sku: "MYHAT", name: "A green hat"})
> ```
>
> But for now, this 5 argument constructor is easy to use.

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 example, if we want to access `newProduct` in our view we could write:

```
import { Product } from './product.model';
export class AppComponent {
  product: Product;

  constructor() {
    let newProduct = new Product(
          'NICEHAT',
          'A Nice Black Hat',
          '/resources/images/products/black-hat.jpg',
          ['Men', 'Accessories', 'Hats'],
          29.99);

    this.product = newProduct;
  }
}
```

or more concisely:

```
export class AppComponent {
  product: Product;

  constructor() {
    this.product = new Product(
          'NICEHAT',
          'A Nice Black Hat',
          '/resources/images/products/black-hat.jpg',
          ['Men', 'Accessories', 'Hats'],
          29.99);
  }
}
```

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 `AppComponent`, when we write: `product: Product`, we're specifying that the `AppComponent` 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 to the `AppComponent` instance, we could use that variable in our view template:

```
<div class="inventory-app">
   <h1>{{ product.name }}</h1>
   <span>{{ product.sku }}</span>
</div>
```

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 `AppComponent`.

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

In the code above, we're only able to show a single product in our app, but we want to be able to show a list of products. Let's change our `AppComponent` to store an array of `Products` rather than a single `Product`:

```
export class AppComponent {
  products: Product[];

  constructor() {
    this.products = [];
  }
}
```

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 `AppComponent` holds an array of `Products`. Let's create some `Products` in the constructor:

code/how-angular-works/inventory-app/src/app/app.component.ts

```
export class AppComponent {
  products: Product[];

  constructor() {
    this.products = [
      new Product(
        'MYSHOES',
        'Black Running Shoes',
        '/assets/images/products/black-shoes.jpg',
        ['Men', 'Shoes', 'Running Shoes'],
        109.99),
      new Product(
        'NEATOJACKET',
        'Blue Jacket',
        '/assets/images/products/blue-jacket.jpg',
        ['Women', 'Apparel', 'Jackets & Vests'],
        238.99),
      new Product(
        'NICEHAT',
        'A Nice Black Hat',
        '/assets/images/products/black-hat.jpg',
        ['Men', 'Accessories', 'Hats'],
        29.99)
      ];
  }
```

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

Selecting a `Product`

We (eventually) 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 `AppComponent` 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/src/app/app.component.ts

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

This function accepts a single argument `product` and then it will log out that the product was passed in. We'll use this function in a bit.

Listing products using `<products-list>`

Now that we have our top-level `AppComponent` 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 in our template:

code/how-angular-works/inventory-app/src/app/app.component.html

```
<div class="inventory-app">
  <products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
</div>
```

There is some new syntax here, so let's talk about each part:

Inputs and Outputs

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

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

The `[squareBrackets]` pass inputs and the `(parentheses)` 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:

```
<products-list
  [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 (we'll show how to define that in a moment).

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

That said, we'll define the `products-list` component in a moment, and we'll see exactly how the `productList` input is defined.

`(parens)` handle outputs

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

In our code where we show:

```
<products-list
  ...
  (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 sent to this output

- `$event` is a *special variable* here that represents **the thing emitted on** (i.e. sent to) 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. For now, know that we can pass data to child components through *inputs* (like "arguments" to a function) and we can receive data out of a child component through *outputs* (sort of like "return values" from a function).

Full `AppComponent` Listing

We broke the `AppComponent` up into several chunks above. So that we can see the whole thing together, here's the full code listing of our `AppComponent`:

code/how-angular-works/inventory-app/src/app/app.component.ts

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

import { Product } from './product.model';

/**
 * @InventoryApp: the top-level component for our application
 */
@Component({
  selector: 'inventory-app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  products: Product[];

  constructor() {
    this.products = [
      new Product(
        'MYSHOES',
        'Black Running Shoes',
        '/assets/images/products/black-shoes.jpg',
        ['Men', 'Shoes', 'Running Shoes'],
        109.99),
      new Product(
        'NEATOJACKET',
        'Blue Jacket',
        '/assets/images/products/blue-jacket.jpg',
        ['Women', 'Apparel', 'Jackets & Vests'],
```

```
        238.99),
      new Product(
        'NICEHAT',
        'A Nice Black Hat',
        '/assets/images/products/black-hat.jpg',
        ['Men', 'Accessories', 'Hats'],
        29.99)
      ];
  }

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

and the template:

code/how-angular-works/inventory-app/src/app/app.component.html

```
<div class="inventory-app">
  <products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
</div>
```

The `ProductsListComponent`

Now that we have our top-level application component, let's write the `ProductsListComponent`, 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 `ProductsListComponent` is a great place to do this because it "knows" all of the `Products` at the same time.

Let's write the `ProductsListComponent` in three steps:

- Configuring the `ProductsListComponent` `@Component` options
- Writing the `ProductsListComponent` controller class
- Writing the `ProductsListComponent` view template

Configuring the `ProductsListComponent` `@Component` Options

Let's take a look at the `@Component` configuration for `ProductsListComponent`:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
import {
  Component,
  EventEmitter,
  Input,
  Output
} from '@angular/core';
import { Product } from '../product.model';

/**
 * @ProductsList: A component for rendering all ProductRows and
 * storing the currently selected Product
 */
@Component({
  selector: 'products-list',
  templateUrl: './products-list.component.html'
})
export class ProductsListComponent {
  /**
   * @input productList - the Product[] passed to us
   */
  @Input() productList: Product[];

  /**
   * @output onProductSelected - outputs the current
   *          Product whenever a new Product is selected
   */
  @Output() onProductSelected: EventEmitter<Product>;
```

We start our ProductsListComponent with a familiar option: selector. This selector means we can place our ProductsListComponent with the tag <products-list>. We've also defined two properties productList and onProductSelected. Notice that productList has a @Input() annotation, denoting that it is an *input* and onProductSelected has an @Output() annotation, denoting that it is an *output*.

Component inputs

Inputs **specify the parameters we expect our component to receive**. To designate an input, we use the @Input() decoration on a component class property.

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:

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

@Component({
  selector: 'my-component',
})
class MyComponent {
  @Input() name: string;
  @Input() age: number;
}
```

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

If we need to use two different names for the attribute and the property, we could for example write @Input('firstname') name: String;. But the Angular Style Guide[37] suggests to avoid this.

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. However, these 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:

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

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

```
@Component({
  selector: 'my-component'
})
class MyComponent {
  @Input('shortName') name: string;
  @Input('oldAge') age: number;
}
```

- The **property name** (name, age) represent how that incoming property will be **visible ("bound") in the controller**.
- The **@Input argument** (shortName, oldAge) configures how the property is **visible to the "outside world"**.

[37] https://angular.io/docs/ts/latest/guide/style-guide.html

Passing `products` through via the inputs

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

code/how-angular-works/inventory-app/src/app/app.component.html

```
<div class="inventory-app">
  <products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
</div>
```

Hopefully this syntax makes more sense now: we're passing the value of `this.products` (on the `AppComponent`) in via an input on `ProductsListComponent`.

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:

```
@Component({
  selector: 'counter',
  template: `
    {{ value }}
    <button (click)="increase()">Increase</button>
    <button (click)="decrease()">Decrease</button>
  `
})
class Counter {
  value: number;

  constructor() {
    this.value = 1;
  }

```

```
  increase() {
    this.value = this.value + 1;
    return false;
  }

  decrease() {
    this.value = this.value - 1;
    return false;
  }
}
```

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 the `click` event on this button. There are many other built-in events we can listen to such as: `mousedown`, `mousemove`, `dbl-click`, etc.

In this example, the event is *internal* to the component. That is, calling `increase()` increments `this.value`, but there's no effect that leaves this 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 an object that helps you implement the Observer Pattern[38]. That is, it's an object that will:

1. maintain a list of subscribers and
2. publish events to them.

That's it.

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

```
let ee = new EventEmitter();
ee.subscribe((name: string) => console.log(`Hello ${name}`));
ee.emit("Nate");

// -> "Hello Nate"
```

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

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

```
@Component({
  selector: 'single-component',
  template: `
    <button (click)="liked()">Like it?</button>
  `
})
class SingleComponent {
  @Output() putRingOnIt: EventEmitter<string>;

  constructor() {
    this.putRingOnIt = new EventEmitter();
  }

  liked(): void {
    this.putRingOnIt.emit("oh oh oh");
  }
}
```

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.

[38] https://en.wikipedia.org/wiki/Observer_pattern

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

```
@Component({
  selector: 'club',
  template: `
    <div>
      <single-component
        (putRingOnIt)="ringWasPlaced($event)"
        ></single-component>
    </div>
  `
})
class ClubComponent {
  ringWasPlaced(message: string) {
    console.log(`Put your hands up: ${message}`);
  }
}

// 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 `ProductsListComponent` Controller Class

Back to our store example, our `ProductsListComponent` 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/src/app/products-list/products-list.component.ts

```
export class ProductsListComponent {
  /**
   * @input productList - the Product[] passed to us
   */
  @Input() productList: Product[];

  /**
   * @output onProductSelected - outputs the current
   *          Product whenever a new Product is selected
   */
  @Output() onProductSelected: EventEmitter<Product>;

  /**
   * @property currentProduct - local state containing
   *             the currently selected `Product`
   */
  private currentProduct: Product;

  constructor() {
    this.onProductSelected = new EventEmitter();
  }
```

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 `ProductsListComponent`. You might also hear this being referred to as "local component state". It's only used here within the component.

Writing the `ProductsListComponent` View Template

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

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.html

```
<div class="ui items">
  <product-row
    *ngFor="let myProduct of productList"
    [product]="myProduct"
    (click)='clicked(myProduct)'
    [class.selected]="isSelected(myProduct)">
  </product-row>
</div>
```

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.

> Style-wise, I probably wouldn't call this variable `myProduct` in a real app. Instead, I'd probably call it `product`, or even `p`. But here I want to be explicit about what we're passing, and `myProduct` is slightly clearer because it let's us distinguish the 'local template variable' from the input `product`.

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 `ProductsListComponent` 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 `ProductsListComponent`):

`clicked`

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

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

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/src/app/products-list/products-list.component.ts

```
  isSelected(product: Product): boolean {
    if (!product || !this.currentProduct) {
      return false;
    }
    return product.sku === this.currentProduct.sku;
  }
```

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

The Full `ProductsListComponent` Component

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

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
import {
  Component,
  EventEmitter,
  Input,
  Output
} from '@angular/core';
import { Product } from '../product.model';

/**
 * @ProductsList: A component for rendering all ProductRows and
 * storing the currently selected Product
```

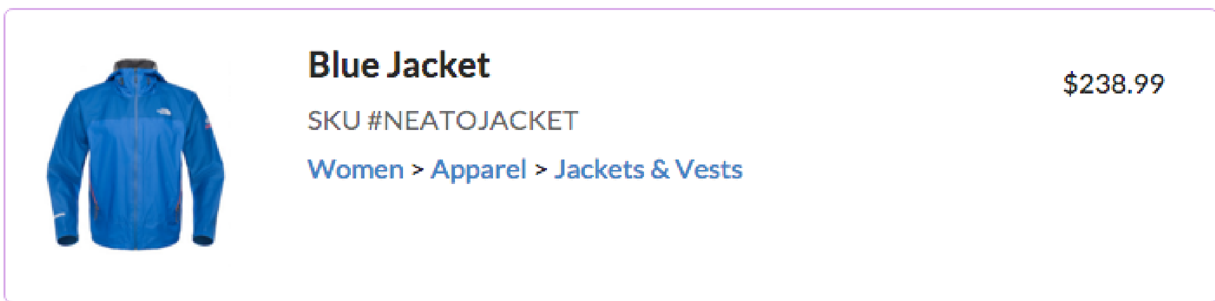
```
 */
@Component({
  selector: 'products-list',
  templateUrl: './products-list.component.html'
})
export class ProductsListComponent {
  /**
   * @input productList - the Product[] passed to us
   */
  @Input() productList: Product[];

  /**
   * @output onProductSelected - outputs the current
   *          Product whenever a new Product is selected
   */
  @Output() onProductSelected: EventEmitter<Product>;

  /**
   * @property currentProduct - local state containing
   *             the currently selected `Product`
   */
  private currentProduct: Product;

  constructor() {
    this.onProductSelected = new EventEmitter();
  }

  clicked(product: Product): void {
    this.currentProduct = product;
    this.onProductSelected.emit(product);
  }

  isSelected(product: Product): boolean {
    if (!product || !this.currentProduct) {
      return false;
    }
    return product.sku === this.currentProduct.sku;
  }

}
```

and the template:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.html

```
<div class="ui items">
  <product-row
    *ngFor="let myProduct of productList"
    [product]="myProduct"
    (click)='clicked(myProduct)'
    [class.selected]="isSelected(myProduct)">
  </product-row>
</div>
```

The `ProductRowComponent` Component

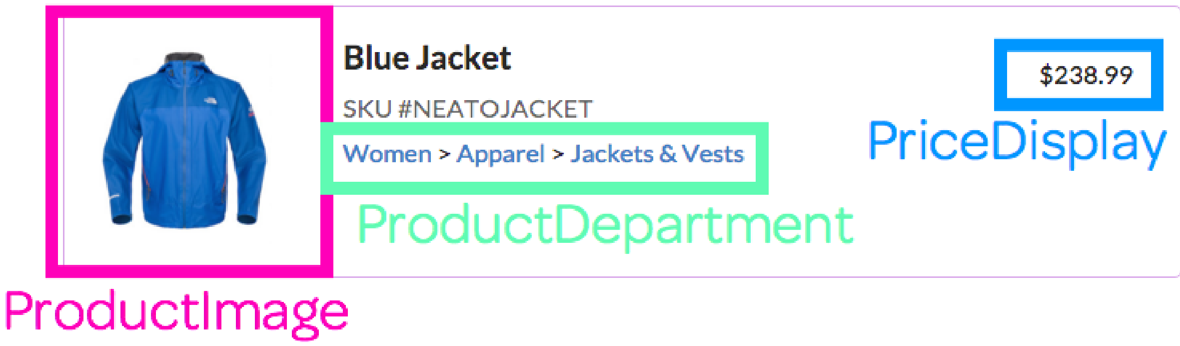


A Selected Product Row Component

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

- `ProductImageComponent` - for the image
- `ProductDepartmentComponent` - for the department "breadcrumbs"
- `PriceDisplayComponent` - for showing the product's price

Here's a visual of the three Components that will be used within the `ProductRowComponent`:



ProductRowComponent's Sub-components

Let's take a look at the `ProductRowComponent`'s Component configuration, definition class, and template:

`ProductRowComponent` Configuration

The `ProductRowComponent` uses a lot of the ideas we've covered so far:

code/how-angular-works/inventory-app/src/app/product-row/product-row.component.ts

```
import {
  Component,
  Input,
  HostBinding
} from '@angular/core';
import { Product } from '../product.model';

/**
 * @ProductRow: A component for the view of single Product
 */
@Component({
  selector: 'product-row',
  templateUrl: './product-row.component.html',
})
export class ProductRowComponent {
  @Input() product: Product;
  @HostBinding('attr.class') cssClass = 'item';
}
```

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 instance variable will be set to the `Product` that was passed in from our parent Component.

The `HostBinding` decoration is new - it lets us **set attributes on the host element**. The *host* is the element this component is attached to.

In this case, we're using the Semantic UI `item` class[39]. Here when we say `@HostBinding('attr.class') cssClass = '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.

Instead of putting a long `template` string in our TypeScript file, instead we're going to move the template to a separate HTML file and use a `templateUrl` to load it. We'll talk about the template in a minute.

`ProductRowComponent template`

Now let's take a look at the template:

code/how-angular-works/inventory-app/src/app/product-row/product-row.component.html

```
<product-image [product]="product"></product-image>
<div class="content">
  <div class="header">{{ product.name }}</div>
  <div class="meta">
    <div class="product-sku">SKU #{{ product.sku }}</div>
  </div>
  <div class="description">
    <product-department [product]="product"></product-department>
  </div>
</div>
<price-display [price]="product.price"></price-display>
```

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 `ProductImageComponent`. 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.

[39] http://semantic-ui.com/views/item.html

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

Now let's talk about the three components we used in this template. They're relatively short.

The `ProductImageComponent` Component

In the `ProductImageComponent` the template is only one line, so we can put it inline:

code/how-angular-works/inventory-app/src/app/product-image/product-image.component.ts

```
/**
 * @ProductImage: A component to show a single Product's image
 */
@Component({
  selector: 'product-image',
  template: `
  <img class="product-image" [src]="product.imageUrl">
  `
})
export class ProductImageComponent {
  @Input() product: Product;
  @HostBinding('attr.class') cssClass = 'ui small image';
}
```

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

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.

We could also have written this tag this way:

```
<img src="{{ product.imageUrl }}">
```

Both styles do essentially the same thing, so feel free to pick the style that works best for your team.

The `PriceDisplayComponent` Component

Next, let's look at `PriceDisplayComponent`:

code/how-angular-works/inventory-app/src/app/price-display/price-display.component.ts

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

/**
 * @PriceDisplay: A component to show the price of a
 * Product
 */
@Component({
  selector: 'price-display',
  template: `
  <div class="price-display">\${{ price }}</div>
  `
})
export class PriceDisplayComponent {
  @Input() price: number;
}
```

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 (in ES6).

The `ProductDepartmentComponent`

Here is our `ProductDepartmentComponent`:

code/how-angular-works/inventory-app/src/app/product-department/product-department.component.ts

```
import {
  Component,
  Input
} from '@angular/core';
import { Product } from '../product.model';

/**
 * @ProductDepartment: A component to show the breadcrumbs to a
 * Product's department
 */
@Component({
  selector: 'product-department',
  templateUrl: './product-department.component.html'
```

```
})
export class ProductDepartmentComponent {
  @Input() product: Product;
}
```

and template:

code/how-angular-works/inventory-app/src/app/product-department/product-department.component.html

```
<div class="product-department">
  <span *ngFor="let name of product.department; let i=index">
    <a href="#">{{ name }}</a>
    <span>{{i < (product.department.length-1) ? '>' : ''}}</span>
  </span>
</div>
```

The thing to note about the `ProductDepartmentComponent` 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: `let 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:

```
Women > Apparel > Jackets & Vests
```

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*.

`NgModule` and Booting the App

The final thing we have to do is ensure we have a `NgModule` for this app and boot it up:

code/how-angular-works/inventory-app/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ProductImageComponent } from './product-image/product-image.component';
import { ProductDepartmentComponent } from './product-department/product-department.\
component';
import { PriceDisplayComponent } from './price-display/price-display.component';
import { ProductRowComponent } from './product-row/product-row.component';
import { ProductsListComponent } from './products-list/products-list.component';

@NgModule({
  declarations: [
    AppComponent,
    ProductImageComponent,
    ProductDepartmentComponent,
    PriceDisplayComponent,
    ProductRowComponent,
    ProductsListComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular provides a *module* system that helps organize our code. Unlike AngularJS 1.x, where all directives are essentially globals, in Angular you must specify exactly *which components* you're going to be using in your app.

While it is a bit more configuration to do it this way, it's a lifesaver for larger apps.

When you create new components in Angular, in order to use them they must be *accessible* from the current module. That is, if we want to use the `ProductsListComponent` component with the `products-list` selector in the `AppComponent` `template`, then we need to make sure that the `AppComponent`'s module either:

1. is in the same module as the `ProductsListComponent` component or
2. The `AppComponent`'s module `imports` the module that contains `ProductsListComponent`

Remember **every** component you write must be declared in one `NgModule` before it can be used in a template.

In this case, we're putting `AppComponent`, `ProductsListComponent`, and **all** the other components for this app in one module. This is easy and it means they can all "see" each other.

Notice that we tell `NgModule` that we want to `bootstrap` with `AppComponent`. This says that `AppComponent` will be the top-level component.

Because we are writing a browser app, we also put `BrowserModule` in the `imports` of the `NgModule`.

Booting the app

To bootstrap this app we write this in our `main.ts`:

code/how-angular-works/inventory-app/src/main.ts

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

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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

The last line in this file is what boots our `AppModule` and subsequently boots our Angular app.

Because this app was written with `angular-cli`, we can use the `ng` tool to run the app by running `ng serve`.

That said, it can be tricky to understand what's going on there. When we run our app with `ng serve` this is what happens:

- `ng serve` looks at `.angular-cli.json` which specifies `main.ts` as our entry point (and `index.html` as our index file)
- `main.ts` bootstraps `AppModule`
- `AppModule` specifies that `AppComponent` is the top level component
- … and then `AppComponent` renders the rest of our app!

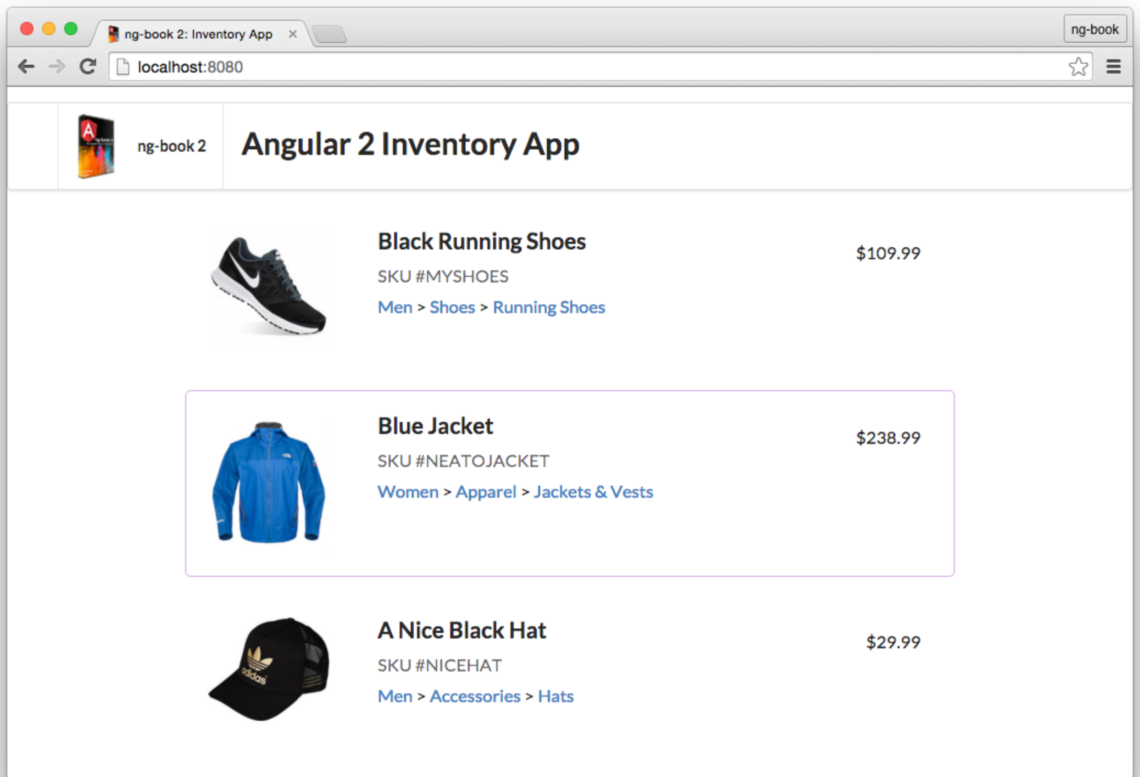
The Completed Project

To try it out, change into the project directory and type:

```
npm install
ng serve
```

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

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.

Deploying the App

We can deploy this app in the same way we deployed the app in the first chapter:

```
ng build --target=production --base-href /
```

And then push the files in `dist` to our server!

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 an item in the cart changes.

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

The recommended way in Angular, 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 Directives

Introduction

Angular provides a number of built-in *directives*, which are attributes we add to our HTML elements that give us dynamic behavior. In this chapter, we're going to cover each built-in directive and show you examples of how to use them.

By the end of this chapter you'll be able to use the basic built-in directives that Angular offers.

How To Use This Chapter

Instead of building an app step-by-step, this chapter is a tour of the built-in directives in Angular. Since we're early in the book, we won't explain every detail, but we will provide plenty of example code.

Remember: at any time you can reference the sample code for this chapter to get the complete context.

If you'd like to run the examples in this chapter then see the folder `code/built-in-directives` and run:

```
npm install
npm start
```

And then open http://localhost:4200[40] in your browser.

`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 into the directive.

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

Some examples are:

[40] http://localhost:4200

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

Note for AngularJS 1.x Users

If you've used AngularJS 1.x, you may have used the `ngIf` directive before. You can think of the Angular version as a direct substitute.

On the other hand, Angular 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:

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

But as you can see, the scenario where `myVar` is neither `A` nor `B` is verbose when all we're trying to express is an `else`.

To illustrate this growth in complexity, say we wanted to handle a new value `C`.

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

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar == 'C'">Var is C</div>
  <div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something else</d\
iv>
</div>
```

For cases like this, Angular 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 `ngSwitchCase` directive
- Handle all the other unknown cases with `ngSwitchDefault`

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

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

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

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchCase="'C'">Var is C</div>
  <div *ngSwitchDefault>Var is something else</div>
</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 `*ngSwitchCase` value for different elements, so you're not limited to matching only a single time. Here's an example:

code/built-in-directives/src/app/ng-switch-example/ng-switch-example.component.html

```html
<h4 class="ui horizontal divider header">
  Current choice is {{ choice }}
</h4>

<div class="ui raised segment">
  <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">First choice</li>
    <li *ngSwitchCase="2">Second choice</li>
    <li *ngSwitchCase="3">Third choice</li>
    <li *ngSwitchCase="4">Fourth choice</li>
    <li *ngSwitchCase="2">Second choice, again</li>
    <li *ngSwitchDefault>Default choice</li>
  </ul>
</div>

<div style="margin-top: 20px;">
  <button class="ui primary button" (click)="nextChoice()">
    Next choice
  </button>
</div>
```

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-directives/src/app/ng-style-example/ng-style-example.component.html

```html
<div [style.background-color]="'yellow'">
  Uses fixed yellow background
</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-directives/src/app/ng-style-example/ng-style-example.component.html

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

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>

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

<button class="ui primary button" (click)="apply(colorinput.value, fontinput.value)">
  Apply settings
</button>
```

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-directives/src/app/ng-style-example/ng-style-example.component.html

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

The `.px` suffix indicates that we're setting the `font-size` property value in pixels. You could easily replace that by `[style.font-size.em]` to express the font size in ems or even in percentage using `[style.font-size.%]`.

The other two elements use the `#colorinput` to set the text and background colors:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.html

```
<h4 class="ui horizontal divider header">
  ngStyle with object property from variable
</h4>

<div>
  <span [ngStyle]="{color: color}">
    {{ color }} text
  </span>
</div>

<h4 class="ui horizontal divider header">
  style from variable
</h4>

<div [style.background-color]="color"
     style="color: white;">
  {{ color }} background
</div>
```

This way, when we click the **Apply settings** button, we call a method that sets the new values:

code/built-in-directives/src/app/ng-style-example/ng-style-example.component.ts

```
  apply(color: string, fontSize: number): void {
    this.color = color;
    this.fontSize = fontSize;
  }
```

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.

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-directives/src/styles.css

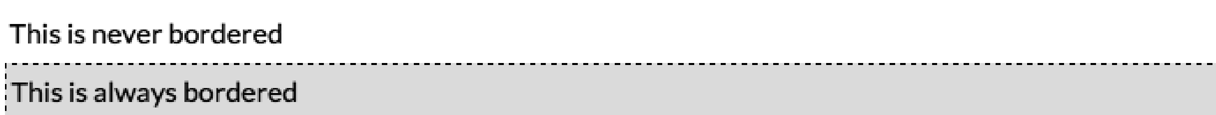
```
.bordered {
  border: 1px dashed black;
  background-color: #eee; }
```

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-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div [ngClass]="{bordered: false}">This is never bordered</div>
  <div [ngClass]="{bordered: true}">This is always bordered</div>
```

As expected, this is how those two `divs` would be rendered:



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-directives/src/app/ng-class-example/ng-class-example.component.html

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

Alternatively, we can define a `classesObj` object in our component:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.ts

```
@Component({
  selector: 'app-ng-class-example',
  templateUrl: './ng-class-example.component.html'
})
export class NgClassExampleComponent implements OnInit {
  isBordered: boolean;
  classesObj: Object;
  classList: string[];

  constructor() {
  }

  ngOnInit() {
    this.isBordered = true;
    this.classList = ['blue', 'round'];
    this.toggleBorder();
  }

  toggleBorder(): void {
    this.isBordered = !this.isBordered;
    this.classesObj = {
      bordered: this.isBordered
    };
  }
```

And use the object directly:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

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

> Again, be careful when you have class names that contains dashes, like `bordered-box`. JavaScript requires that object-literal keys with dashes be quoted like a string, as in:
>
> ```
> <div [ngClass]="{'bordered-box': false}">...
> ```

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-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div class="base" [ngClass]="['blue', 'round']">
    This will always have a blue background and
    round corners
  </div>
```

Or assign an array of values to a property in our component:

```
this.classList = ['blue', 'round'];
```

And pass it in:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
  <div class="base" [ngClass]="classList">
    This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
    and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
  </div>
```

In this last example, the `[ngClass]` assignment works alongside existing values assigned by the HTML `class` attribute.

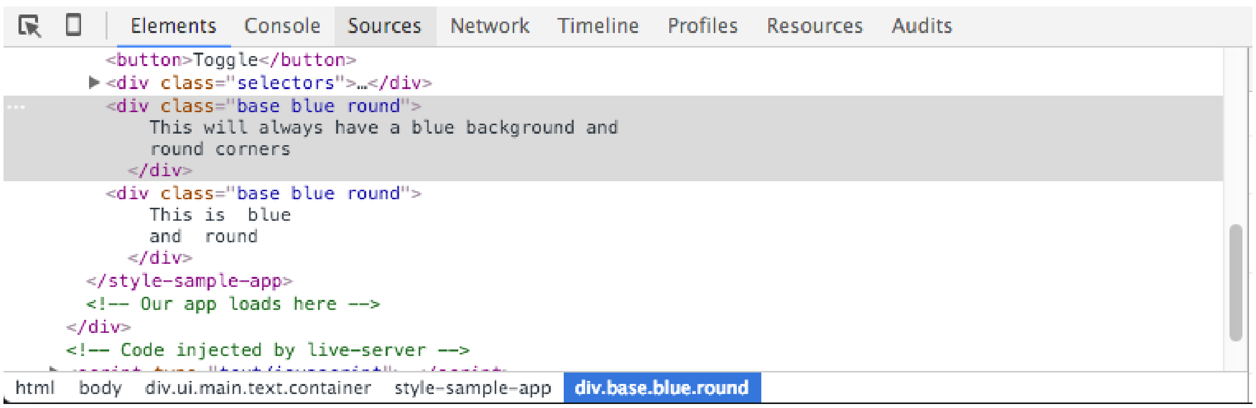
The resulting classes added to the element will always be the set of the classes provided by usual `class` HTML attribute and the result of the evaluation of the `[class]` directive.

In this example:

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
<div class="base" [ngClass]="['blue', 'round']">
  This will always have a blue background and
  round corners
</div>
```

The element will have all three classes: `base` from the `class` HTML attribute and also `blue` and `round` from the `[class]` assignment:



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) and pass an element of the array on each iteration.

The syntax is `*ngFor="let item of items"`.

- The `let item` syntax specifies a (template) variable that's receiving each element of the `items` array;
- The `items` is the collection of items from your controller.

To illustrate, we can take a look at the code example. We declare an array of cities on our component controller:

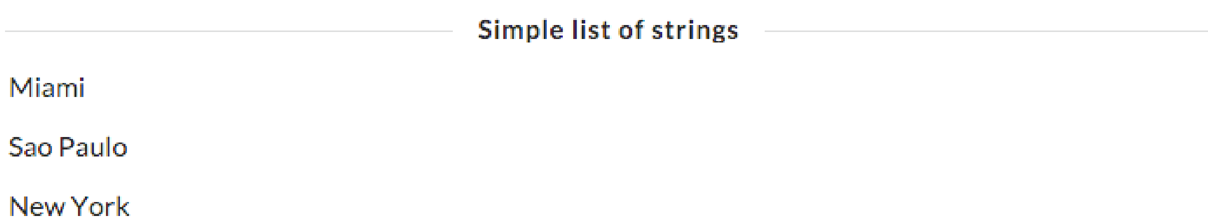
```
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

And then, in our template we can have the following HTML snippet:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<h4 class="ui horizontal divider header">
  Simple list of strings
</h4>

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

And it will render each city inside the `div` as you would expect:



Result of the ngFor directive usage

We can also iterate through an array of objects like these:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.ts

```
    this.people = [
      { name: 'Anderson', age: 35, city: 'Sao Paulo' },
      { name: 'John', age: 12, city: 'Miami' },
      { name: 'Peter', age: 22, city: 'New York' }
    ];
```

And then render a table based on each row of data:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<h4 class="ui horizontal divider header">
  List of objects
</h4>

<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
```

```
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</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-directives/src/app/ng-for-example/ng-for-example.component.ts

```
    this.peopleByCity = [
      {
        city: 'Miami',
        people: [
          { name: 'John', age: 12 },
          { name: 'Angel', age: 22 }
        ]
      },
      {
        city: 'Sao Paulo',
        people: [
          { name: 'Anderson', age: 35 },
          { name: 'Felipe', age: 36 }
        ]
      }
```

```
  ];
}
```

And then we could use `NgFor` to render one `h2` for each city:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>
```

And use a nested directive to iterate through the people for a given city:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tr *ngFor="let p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</table>
```

Resulting in the following template code:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<h4 class="ui horizontal divider header">
  Nested data
</h4>

<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>

  <table class="ui celled table">
    <thead>
      <tr>
        <th>Name</th>
```

```
        <th>Age</th>
      </tr>
    </thead>
    <tr *ngFor="let p of item.people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
    </tr>
  </table>
</div>
```

And it would render one table for each city:

Nested data

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`).

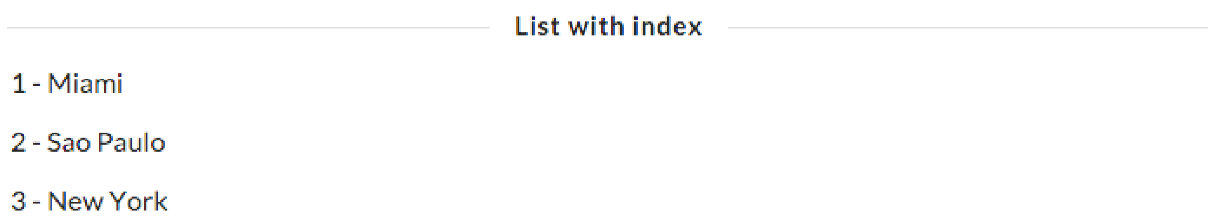
Note that, like JavaScript, the index is always zero based. So the index for first element is `0`, `1` for the second and so on…

Making some changes to our first example, adding the `let num = index` snippet like below:

code/built-in-directives/src/app/ng-for-example/ng-for-example.component.html

```
<div class="ui list" *ngFor="let c of cities; let num = index">
  <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

It will add the position of the city before the name, like this:



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.

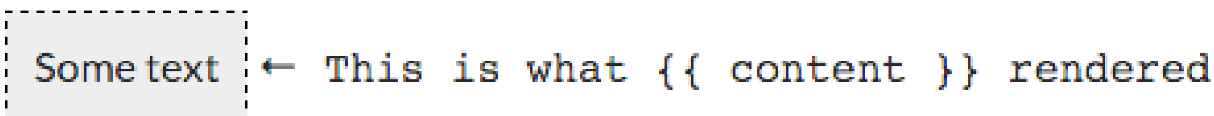
Let's say we want to have a `div` that renders the contents of that `content` variable and right after we want to point that out by outputting *<- this is what {{ content }} rendered* next to the actual value of the variable.

To do that, here's the template we'd have to use:

code/built-in-directives/src/app/ng-non-bindable-example/ng-non-bindable-example.component.html

```
<div class='ngNonBindableDemo'>
  <span class="bordered">{{ content }}</span>
  <span class="pre" ngNonBindable>
    &larr; This is what {{ content }} rendered
  </span>
</div>
```

And with that `ngNonBindable` attribute, ng2 will not compile within that second `span`'s context, leaving it intact:



Result of using ngNonBindable

Conclusion

Angular has only a few core directives, but we can combine these simple pieces to create dynamic, powerful apps. However, all of these directives help us **output** dynamic data, they don't let us **accept user interaction**.

In the next chapter we'll learn how to let our **user input data using forms**.

Forms in Angular

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 be very 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 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.

`FormControl`s and `FormGroup`s

The two fundamental objects in Angular 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 being valid, dirty (changed), or has errors.

For instance, here's how we might use a `FormControl` in TypeScript:

```
// create a new FormControl with the value "Nate"
let nameControl = new FormControl("Nate");

let name = nameControl.value; // -> Nate

// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
// etc.
```

To build up forms we create `FormControl`s (and groups of `FormControl`s) 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:

```
<!-- part of some bigger form -->
<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 `FormControl`s. If we wanted to check the validity of our form, it's cumbersome to iterate over an array of `FormControl`s and check each `FormControl` for validity. `FormGroup`s solve this issue by providing a wrapper interface around a collection of `FormControl`s.

Here's how you create a `FormGroup`:

```
let personInfo = new FormGroup({
    firstName: new FormControl("Nate"),
    lastName: new FormControl("Murray"),
    zip: new FormControl("90210")
})
```

`FormGroup` and `FormControl` have a common ancestor (`AbstractControl`[41]). That means we can check the `status` or `value` of `personInfo` just as easily as a single `FormControl`:

[41]https://angular.io/docs/ts/latest/api/forms/index/AbstractControl-class.html

```
personInfo.value; // -> {
//   firstName: "Nate",
//   lastName: "Murray",
//   zip: "90210"
// }

// now we can query this control group for certain values, which have sensible
// values depending on the children FormControl's values:
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty  // -> false
personInfo.valid  // -> true
// etc.
```

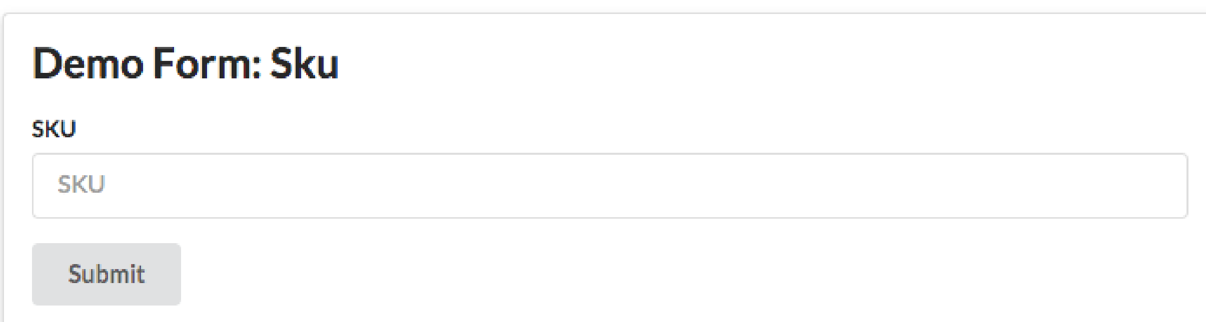
Notice that when we tried to get the `value` from the `FormGroup` we received an **object** with key-value pairs. This is a really handy way to get the full set of values from our form without having to iterate over each `FormControl` individually.

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:



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()` decorator
- Create the template
- Implement custom functionality in the component definition class

Let’s take these in turn:

Loading the `FormsModule`

In order to use the new forms library we need to first make sure we import the forms library in our `NgModule`.

There are two ways of using forms in Angular and we’ll talk about them both in this chapter: using `FormsModule` or using `ReactiveFormsModule`. Since we’ll use both, we’ll import them both into our module. To do this, we do the following in our `app.ts` where we bootstrap the app:

```
// app.module.ts
import {
  FormsModule,
  ReactiveFormsModule
} from '@angular/forms';

// farther down...

@NgModule({
  declarations: [
    AppComponent,
    DemoFormSkuComponent,
    // ... our declarations here
  ],
  imports: [
    BrowserModule,
    FormsModule,          // <-- add this
    ReactiveFormsModule  // <-- and this
  ],
  bootstrap: [ AppComponent ]
})
class AppModule {}
```

This ensures that we're able to use the form directives in our views. At the risk of jumping ahead, the `FormsModule` gives us *template driven* directives such as:

- `ngModel` and
- `NgForm`

Whereas `ReactiveFormsModule` gives us *reactive driven* directives like

- `formControl` and
- `ngFormGroup`

… 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 importing `FormsModule` and `ReactiveFormsModule` into our `NgModule` means we can *use any of the directives in that list* in our view template or *inject any of their respective providers* into our components.

Reactive- vs. template-driven Forms

Angular allows you to define forms in two different ways: "reactive" or "template" driven. You can see a comparison of two ways here[42]. Rather than describe how they're different, we're going to show you examples of different ways you can build forms - then you can decide which is right for your application.

Simple SKU Form: @Component Decorator

First, let's start by creating what's called a "template driven" form. Starting with our component:

code/forms/src/app/demo-form-sku/demo-form-sku.component.ts

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

@Component({
  selector: 'app-demo-form-sku',
  templateUrl: './demo-form-sku.component.html',
```

Here we define a `selector` of `app-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 `app-demo-form-sku` tag like so:

[42] https://angular.io/guide/forms-overview

```
<app-demo-form-sku></app-demo-form-sku>
```

Simple SKU Form: `template`

Let's look at our template:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku</h2>
  <form #f="ngForm"
        (ngSubmit)="onSubmit(f.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             name="sku" ngModel>
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

`form` & `NgForm`

Now things get interesting: because we imported `FormsModule`, 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 import `FormsModule`, `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/src/app/demo-form-sku/demo-form-sku.component.html

```
  <form #f="ngForm"
        (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 and pass the value of the form as the argument".

`input` & `NgModel`

Our `input` tag has a few things we should touch on before we talk about `NgModel`:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
      class="ui form">

  <div class="field">
    <label for="skuInput">SKU</label>
    <input type="text"
           id="skuInput"
           placeholder="SKU"
           name="sku" ngModel>
  </div>
```

- `class="ui form"` and `class="field"` - these classes are totally optional. They come from the CSS framework Semantic UI[43]. 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[44]
- 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"`.

[43] http://semantic-ui.com/
[44] http://www.w3.org/TR/WCAG20-TECHS/H44.html

`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/src/app/demo-form-sku/demo-form-sku.component.ts

```
export class DemoFormSkuComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }

}
```

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/src/app/demo-form-sku/demo-form-sku.component.ts

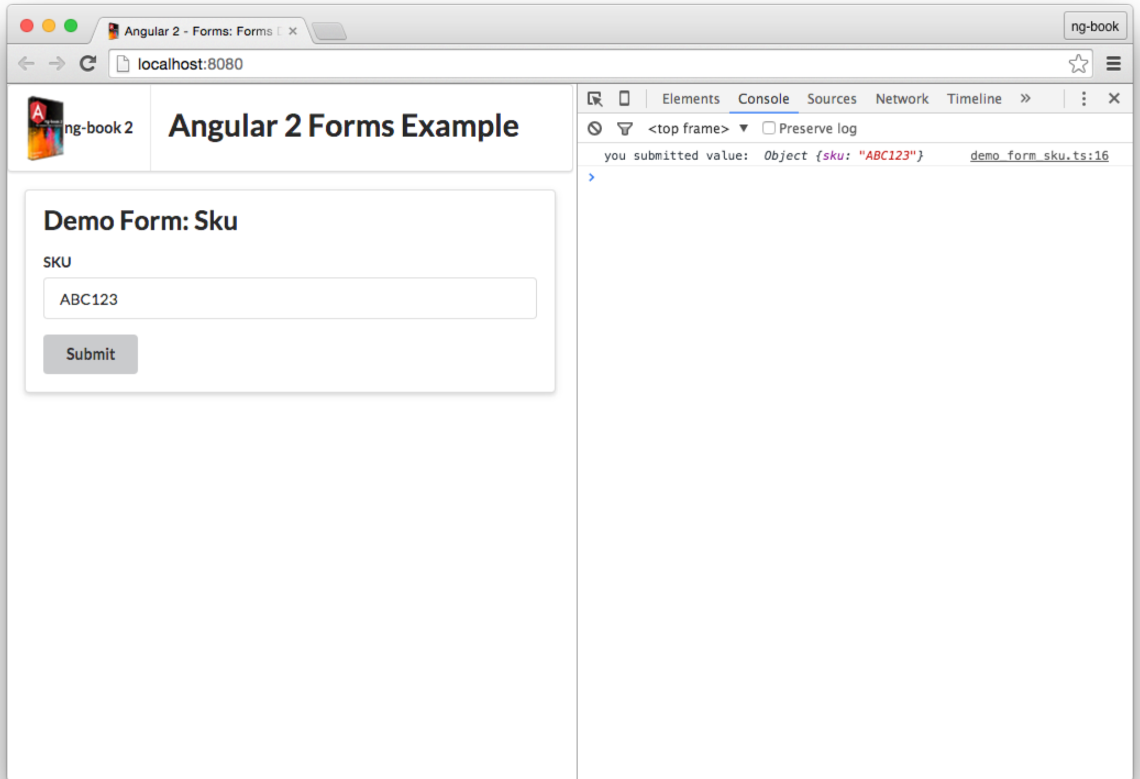
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-demo-form-sku',
  templateUrl: './demo-form-sku.component.html',
  styles: []
})
export class DemoFormSkuComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }

}
```

and the template:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku</h2>
  <form #f="ngForm"
        (ngSubmit)="onSubmit(f.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             name="sku" ngModel>
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

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` 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

Reactive Forms with `FormBuilder`

For this component we're going to be using the `formGroup` and `formControl` directives which means we need to import the appropriate classes. We start by importing them like so:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

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/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
```

```
      'sku': ['ABC123']
    });
  }

  ngOnInit() {
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }

}
```

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 `FormControl`s 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 use `FormsModule`. 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/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
  <h2 class="ui header">Demo Form: Sku with Builder</h2>
  <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 FormsModule 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:
>
> ```
> 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.

When we want to bind an **existing FormControl** to an input we use formControl:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             [formControl]="myForm.controls['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/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  ngOnInit() {
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }

}
```

and the template:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku with Builder</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
```

```
            id="skuInput"
            placeholder="SKU"
            [formControl]="myForm.controls['sku']">
    </div>

  <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

Remember:

To create a new `FormGroup` and `FormControls` implicitly use:

- `ngForm` and
- `ngModel`

But to bind to an existing `FormGroup` and `FormControls` 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:

```
let control = new FormControl('sku', Validators.required);
```

Or in our case, because we're using `FormBuilder` we will use the following syntax:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }
```

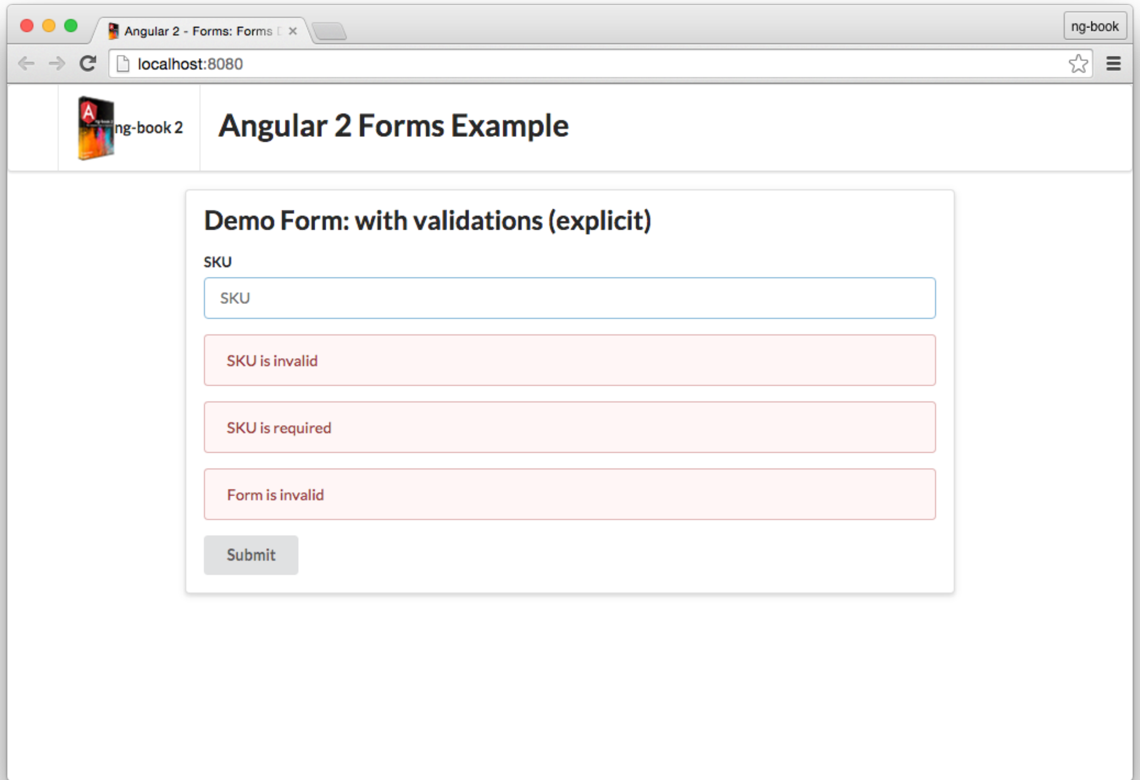
Now we need to use our validation in the view. There are two ways we can access the validation value in the view:

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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

```
export class DemoFormWithValidationsExplicitComponent {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
```

```
    console.log('you submitted value: ', value);
  }

}
```

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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
    <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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
              [formControl]="sku">
        <div *ngIf="!sku.valid"
          class="ui error message">SKU is invalid</div>
        <div *ngIf="sku.hasError('required')"
```

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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
    <div class="field"
        [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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
        <div *ngIf="sku.hasError('required')"
          class="ui error message">SKU is required</div>
```

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:

```
    <div *ngIf="myForm.hasError('required', 'sku')"
      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/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

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

@Component({
  selector: 'app-demo-form-with-validations-explicit',
  templateUrl: './demo-form-with-validations-explicit.component.html',
  styles: []
})
export class DemoFormWithValidationsExplicitComponent {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }

}
```

And the template:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with validations (explicit)</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form"
        [class.error]="!myForm.valid && myForm.touched">

    <div class="field"
        [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             [formControl]="sku">
       <div *ngIf="!sku.valid"
         class="ui error message">SKU is invalid</div>
       <div *ngIf="sku.hasError('required')"
         class="ui error message">SKU is required</div>
    </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

Removing the `sku` instance variable

In the example above we set `sku: AbstractControl` as an instance variable. We often won't want to create an instance variable for each `AbstractControl`, 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/src/app/demo-form-with-validations-shorthand/demo-form-with-validations-shorthand.component.html

```
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             [formControl]="myForm.controls['sku']">
       <div *ngIf="!myForm.controls['sku'].valid"
         class="ui error message">SKU is invalid</div>
       <div *ngIf="myForm.controls['sku'].hasError('required')"
```

In this way we can access the `sku` control without being forced to explicitly add it as an instance variable on the component class.

> We used bracket-notation, e.g. `myForm.controls['sku']`. We could also use the dot-notation, e.g `myForm.controls.sku`. In general, be aware that TypeScript may give a warning if you use the dot-notation and the object is not properly typed (but that is not a problem here).

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:

```
export class Validators {
  static required(c: FormControl): StringMap<string, boolean> {
    return isBlank(c.value) || c.value == "" ? {"required": true} : null;
  }
```

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/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.ts

```
function skuValidator(control: FormControl): { [s: string]: boolean } {
  if (!control.value.match(/^123/)) {
    return {invalidSku: true};
  }
}
```

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/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.ts

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.compose([
        Validators.required, skuValidator])]
    });
```

`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/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.html

```
       <div *ngIf="sku.hasError('invalidSku')"
         class="ui error message">SKU must begin with <span>123</span></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[45]

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 `.subscribe` method

Here's an example:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.ts

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];

    this.sku.valueChanges.subscribe(
      (value: string) => {
        console.log('sku changed to:', value);
      }
    );

    this.myForm.valueChanges.subscribe(
      (form: any) => {
        console.log('form changed to:', form);
      }
    );

  }
```

[45] https://github.com/jhusain/observable-spec

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:

```
sku changed to:  k
form changed to:  Object {sku: "k"}
sku changed to:  kj
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 **mimics two-way data binding**.

Two-way data binding is almost always more complicated and difficult to reason about vs. one-way data binding. Angular 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[46]. Of course, `ngModel` can be really handy, but know that we don't necessarily rely on two-way data binding 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:

[46] https://www.quora.com/Why-is-the-two-way-data-binding-being-dropped-in-Angular-2

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.ts

```
export class DemoFormNgModelComponent {
  productName: string;

  constructor() {
    this.productName = "ng-book: The Complete Guide to Angular"
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

Notice that we're simply storing `productName: string` as an instance variable.

Next, let's use `ngModel` on our `input` tag:

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.html

```
      <label for="productNameInput">Product Name</label>
      <input type="text"
             id="productNameInput"
             placeholder="Product Name"
             name="productName"
             [(ngModel)]="productName">
```

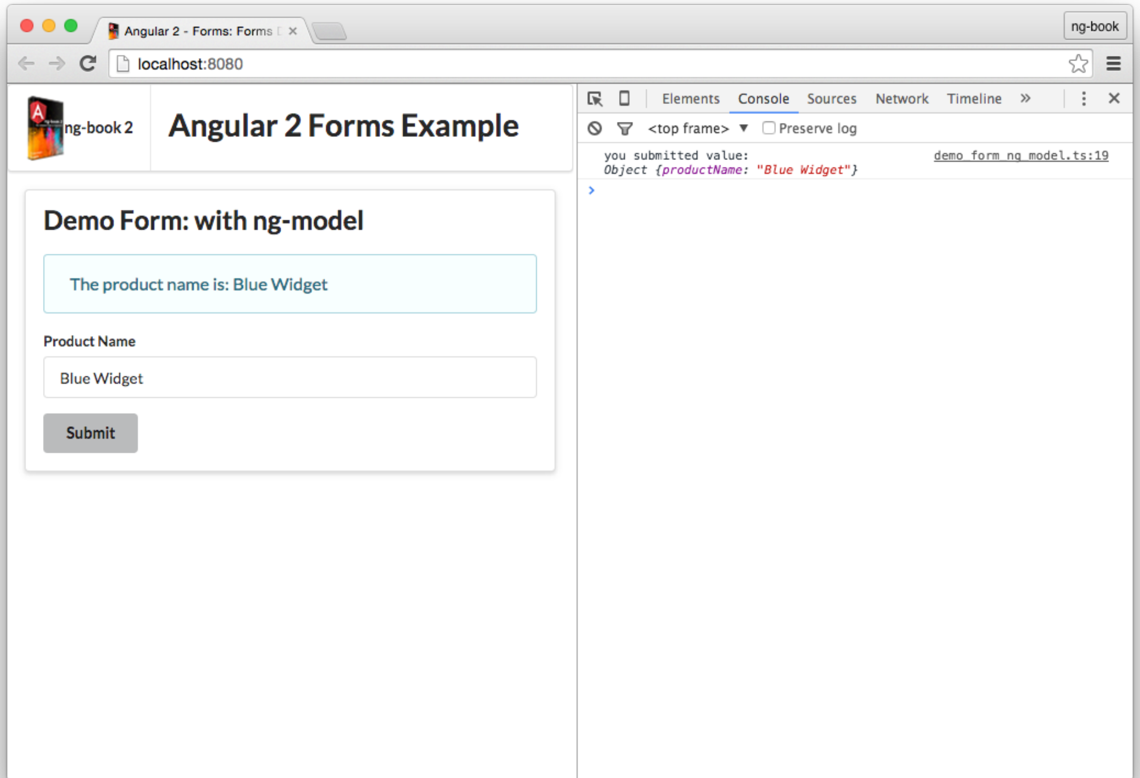
Now notice something - the syntax for `ngModel` is funny: we are using both brackets and parentheses around the `ngModel` attribute! The idea this is intended to invoke is that we're using both the *input* `[]` brackets and the *output* `()` parentheses. It's an indication of the two-way bind.

Last, let's display our `productName` value in the view:

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.html

```
  <div class="ui info message">
    The product name is: {{productName}}
  </div>
```

Here's what it looks like:



Demo Form with ngModel

Easy!

Wrapping Up

Forms have a lot of moving pieces, but Angular makes it fairly straightforward. Once you get a handle on how to use `FormGroups`, `FormControls`, and `Validations`, it's pretty easy going from there!

Dependency Injection

As our programs grow in size, 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:

```
// in A.ts
import {B} from 'B'; // a dependency!

B.foo(); // using B
```

In many cases, simply `import`ing code is sufficient, but other times we need to provide dependencies in a more sophisticated way. For instance, we may want to:

- substitute out the implementation of `B` for `MockB` during testing
- share a *single instance* of the `B` class across our whole app (e.g. the *Singleton* pattern)
- create a *new instance* of the `B` class every time it is 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 "the injector" is as a replacement for the `new` operator. That is, instead of using the language-provided `new` operator, Dependency Injection let's us configure *how* objects are created.

The term Dependency Injection is used to describe both a design pattern (used in many different frameworks) and also the *specific implementation* of DI that is built-in to Angular.

The major benefit of using Dependency Injection is that the client component needn't be aware of ***how* to create the dependencies**. All the client component needs to know is how to *interact* with those dependencies. This is all very abstract, so let's dive in to some code.

How to use this chapter

This chapter is a tour of Angular DI system and concepts. You can find the code for this chapter in `code/dependency-injection`.

While reading this chapter, run the demo project by changing into the project directory and running:

```
npm install
npm start
```

As a preview, to get Dependency Injection to work involves configuration in your `NgModules`. It can feel a bit confusing at first to figure out "where" things are coming from.

The example code has full, runnable examples with all of the context. So if you feel lost, we'd encourage you to checkout the sample code alongside reading this chapter.

Injections Example: `PriceService`

Let's imagine we're building a store that has `Products` and we need to calculate the final price of that product after sales tax. In order to calculate the full price for this product, we use a `PriceService` that takes as input:

- the **base price** of the `Product` and
- the **state** we're selling it to.

and then returns the final price of the `Product`, plus tax:

code/dependency-injection/src/app/price-service-demo/price.service.1.ts

```
export class PriceService {
  constructor() { }

  calculateTotalPrice(basePrice: number, state: string) {
    // e.g. Imgine that in our "real" application we're
    // accessing a real database of state sales tax amounts
    const tax = Math.random();

    return basePrice + tax;
  }

}
```

In this service, the `calculateTotalPrice` function will take the `basePrice` of a product and the `state` and return the total price of product.

Say we want to use this service on our `Product` model. Here's how it could look without dependency injection:

code/dependency-injection/src/app/price-service-demo/product.model.1.ts

```
import { PriceService } from './price.service';

export class Product {
  service: PriceService;
  basePrice: number;

  constructor(basePrice: number) {
    this.service = new PriceService(); // <-- create directly ("hardcoded")
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Now imagine we need to write a test for this `Product` class. We could write a test like this:

```
import { Product } from './product';

describe('Product', () => {

  let product;

  beforeEach(() => {
    product = new Product(11);
  });

  describe('price', () => {
    it('is calculated based on the basePrice and the state', () => {
      expect(product.totalPrice('FL')).toBe(11.66); // <-- hmmm
    });
  })

});
```

The problem with this test is that we don't actually know what the exact value for tax in Florida (`'FL'`) is going to be. Even if we implemented the `PriceService` the 'real' way by calling an API or calling a database, we have the problem that:

- The API needs to be available (or the database needs to be running) and
- We need to know the exact Florida tax at the time we write the test.

What should we do if we want to test the `price` method of the `Product` *without* relying on this external resource? In this case we often *mock* the `PriceService`. For example, if we know the *interface* of a `PriceService`, we could write a `MockPriceService` which will always give us a predictable calculation (and not be reliant on a database or API).

Here's the interface for `IPriceService`:

code/dependency-injection/src/app/price-service-demo/price-service.interface.ts

```
export interface IPriceService {
  calculateTotalPrice(basePrice: number, state: string): number;
}
```

This interface defines just one function: `calculateTotalPrice`. Now we can write a `MockPriceService` that conforms to this interface, which we will use only for our tests:

code/dependency-injection/src/app/price-service-demo/price.service.mock.ts

```
import { IPriceService } from './price-service.interface';

export class MockPriceService implements IPriceService {
  calculateTotalPrice(basePrice: number, state: string) {
    if (state === 'FL') {
      return basePrice + 0.66; // it's always 66 cents!
    }

    return basePrice;
  }
}
```

Now, just because we've written a `MockPriceService` doesn't mean our `Product` will use it. In order to use this service, we need to modify our `Product` class:

code/dependency-injection/src/app/price-service-demo/product.model.ts

```
import { IPriceService } from './price-service.interface';

export class Product {
  service: IPriceService;
  basePrice: number;

  constructor(service: IPriceService, basePrice: number) {
    this.service = service; // <-- passed in as an argument!
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Now, when creating a `Product` **the client using the `Product` class becomes responsible for deciding which concrete implementation of the `PriceService` is going to be given to the new instance**.

And with this change, we can tweak our test slightly and get rid of the dependency on the unpredictable `PriceService`:

code/dependency-injection/src/app/price-service-demo/product.spec.ts

```
import { Product } from './product.model';
import { MockPriceService } from './price.service.mock';

describe('Product', () => {
  let product;

  beforeEach(() => {
    const service = new MockPriceService();
    product = new Product(service, 11.00);
  });

  describe('price', () => {
    it('is calculated based on the basePrice and the state', () => {
      expect(product.totalPrice('FL')).toBe(11.66);
    });
  });
});
```

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.

While the predictability is nice, it's a bit laborious to pass a concrete implementation of a service every time we want a new `Product`. Thankfully, Angular's DI library helps us deal with that problem, too. More on that below.

Within Angular's DI system, instead of directly `import`ing and creating a `new` instance of a class, instead we will:

- Register the "dependency" with Angular
- Describe *how* the dependency will be *injected*
- Inject the dependency

One benefit of this model is that the dependency *implementation* can be swapped at run-time (as in our mocking example above). But another significant benefit is that we can configure **how the dependency is created**.

That is, often in the case of program-wide services, we may want to have **only one instance** - that is, a Singleton. With DI we're able to configure Singletons easily.

A third use-case for DI is for configuration or environment-specific variables. For instance, we might define a "constant" `API_URL`, but then inject a different value in production vs. development.

Let's learn how to create our own services and the different ways of injecting them.

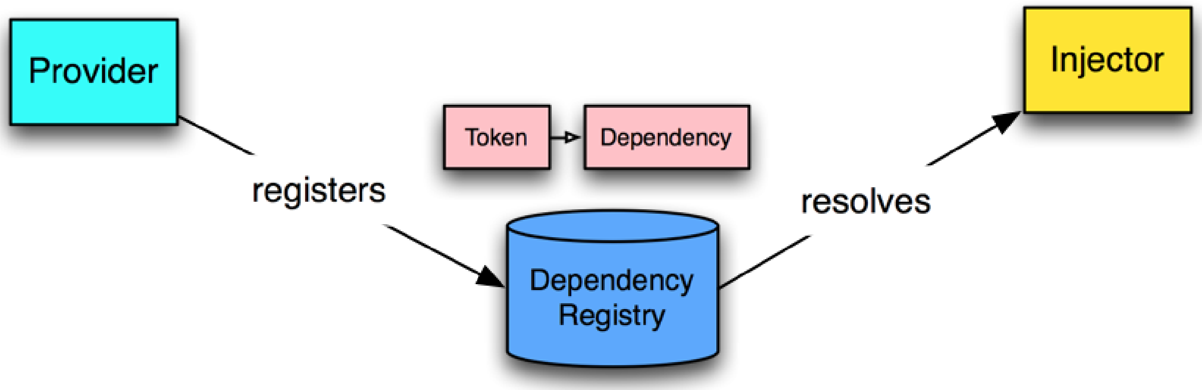
Dependency Injection Parts

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:



Dependency Injection

A way of thinking about this is that when we configure DI we specify **what** is being injected and **how** it will be resolved.

Playing with an Injector

Above with our `Product` and `PriceService` we **manually** created the `PriceService` using the `new` operator. This mimics what Angular itself does.

Angular uses an *injector* to **resolve** a dependency and **create the instance**. This is done for us behind the scenes, but as an exercise, it's useful to explore what's happening. It can be enlightening to use the injector manually, because we can see what Angular does for us behind the scenes.

Let's **manually use the injector** in our component to resolve and create a service. (After we've resolved a dependency manually, we'll show the typical, easy way of injecting dependencies.)

One of the common use-cases for services is to have a 'global' Singleton object. For instance, we might have a `UserService` which contains the information for the currently logged in user. Many different components will want to have logic based on the current user, so this is a good case for a service.

Here's a basic `UserService` that stores the user object as a property:

code/dependency-injection/src/app/services/user.service.ts

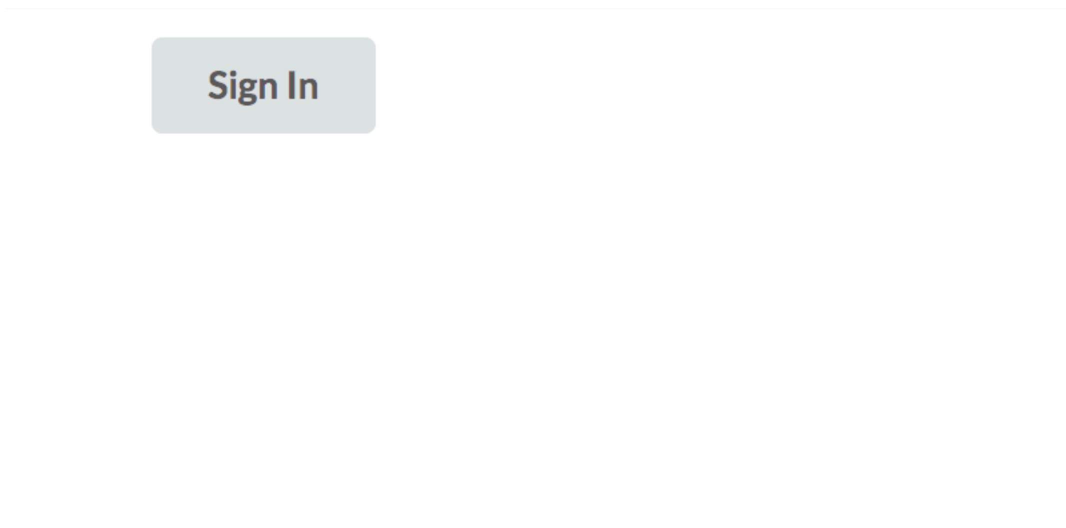
```
import { Injectable } from '@angular/core';

@Injectable()
export class UserService {
  user: any;

  setUser(newUser) {
    this.user = newUser;
  }

  getUser(): any {
    return this.user;
  }
}
```

Say we want to create a toy sign-in form:

code/dependency-injection/src/app/user-demo/user-demo.component.html

```
<div>
  <p
    *ngIf="userName"
    class="welcome">
    Welcome: {{ userName }}!
  </p>
  <button
    (click)="signIn()"
    class="ui button"
    >Sign In
  </button>
</div>
```

Above, we click the "Sign In" button to call the `signIn()` function (which we'll define in a moment). If we have a `userName`, we'll display a greeting.



Simple Sign In Button

Now let's implement this functionality in our component by **using the injector directly**.

code/dependency-injection/src/app/user-demo/user-demo.injector.component.ts

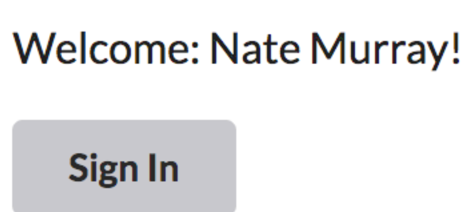
```
import {
  Component,
  ReflectiveInjector
 } from '@angular/core';

 import { UserService } from '../services/user.service';

@Component({
  selector: 'app-injector-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoInjectorComponent {
  userName: string;
  userService: UserService;

  constructor() {
    // Create an _injector_ and ask for it to resolve and create a UserService
    const injector: any = ReflectiveInjector.resolveAndCreate([UserService]);

    // use the injector to **get the instance** of the UserService
    this.userService = injector.get(UserService);
  }

```

```
  signIn(): void {
    // when we sign in, set the user
    // this mimics filling out a login form
    this.userService.setUser({
      name: 'Nate Murray'
    });

    // now **read** the user name from the service
    this.userName = this.userService.getUser().name;
    console.log('User name is: ', this.userName);
  }
}
```

This starts as a basic component: we have a selector, template, and CSS. Note that we have two properties: `userName`, which holds the currently logged-in user's name and `userService`, which holds a reference to the `UserService`.

In our component's `constructor` 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 `UserService` 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.



Signed In

Providing Dependencies with `NgModule`

While it's interesting to see how an `injector` is created directly, that isn't the typical way we'd use injections. Instead, what we'd normally do is

- use `NgModule` to *register* what we'll inject – these are called *providers* and
- use decorators (generally on a `constructor`) to specify **what we're injecting**

By doing these two steps **Angular** will manage creating the injector and resolving the dependencies.

Let's convert our `UserService` to be *injectable* as a singleton across our app. First, we're going to add it to the `providers` key of our `NgModule`:

code/dependency-injection/src/app/user-demo/user-demo.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

// imported here
import { UserService } from '../services/user.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [
    UserService // <-- added right here
  ],
  declarations: []
})
export class UserDemoModule { }
```

Now we can inject `UserService` into our component like this:

code/dependency-injection/src/app/user-demo/user-demo.component.ts

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

import { UserService } from '../services/user.service';

@Component({
  selector: 'app-user-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoComponent {
  userName: string;
  // removed `userService` because of constructor shorthand below

  // Angular will inject the singleton instance of `UserService` here.
  // We set it as a property with `private`.
  constructor(private userService: UserService) {
    // empty because we don't have to do anything else!
  }

  // below is the same...
  signIn(): void {
    // when we sign in, set the user
    // this mimics filling out a login form
    this.userService.setUser({
      name: 'Nate Murray'
    });

    // now **read** the user name from the service
    this.userName = this.userService.getUser().name;
    console.log('User name is: ', this.userName);
  }
}
```

Notice in the `constructor` above that we have made `userService: UserService` an argument to the `UserDemoComponent`. When this component is created on our page **Angular will resolve and inject the `UserService` singleton**. What's great about this is that because Angular is managing the instance, we don't have to worry about doing it ourselves. Every class that injects the `UserService` will receive the same singleton.

Providers are the Key

It's important to know that when we put the `UserService` on the constructor of the `UserDemoComponent`, Angular knows what to inject (and how) **because we listed `UserService` in the `providers` key of our `NgModule`.

It **does not** inject arbitrary classes. You **must** configure an `NgModule` for DI to work.

We've been talking a lot about Singleton services, but we can inject things in lots of other ways. Let's take a look.

Providers

There are several ways we can configure resolving injected dependencies in Angular. For instance we can:

- Inject a (singleton) instance of a class (as we've seen)
- Inject a **value**
- **Call any function** and inject the return value of that function

Let's look into detail at how we create each one:

Using a Class

As we've discussed, injecting a singleton instance of a class is probably the most common type of injection.

When we put the class itself into the list of `providers` like this:

```
providers: [ UserService ]
```

This tells Angular that we want to provide a singleton *instance* of `UserService` whenever `UserService` is injected. Because this pattern is so common, the class by itself is actually shorthand notation for the following, equivalent configuration:

```
providers: [
  { provide: UserService, useClass: UserService }
]
```

What's interesting to note is that the object configuration with `provide` takes **two** keys. `provide` is the *token* that we use **to identify the injection** and the second `useClass` is **how and what** to inject.

Here we're mapping the `UserService` *class* to the `UserService` *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 *aren't required 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, the singleton hasn't been instantiated yet, so when creating the `UserService` instance for the first time, the DI system will trigger the class `constructor` method.

Using a Value

Another way we can use DI is to provide a value, much like we might use a global constant. For instance, we might configure an API Endpoint URL depending on the environment.

To do this, in our `NgModule` `providers`, we use the key `useValue`:

```
providers: [
  { provide: 'API_URL', useValue: 'http://my.api.com/v1' }
]
```

Above, for the `provide` token we're using a *string* of `API_URL`. If we use a string for the `provide` value, Angular can't infer which dependency we're resolving by the type. For instance we **can't** write:

```
// doesn't work - anti-example
export class AnalyticsDemoComponent {
  constructor(apiUrl: 'API_URL') { // <--- this isn't a type, just a string
                                   // if we put `string` that is ambiguous
  }
}
```

So what can we do? In this case, we'll use the `@Inject()` decorator like this:

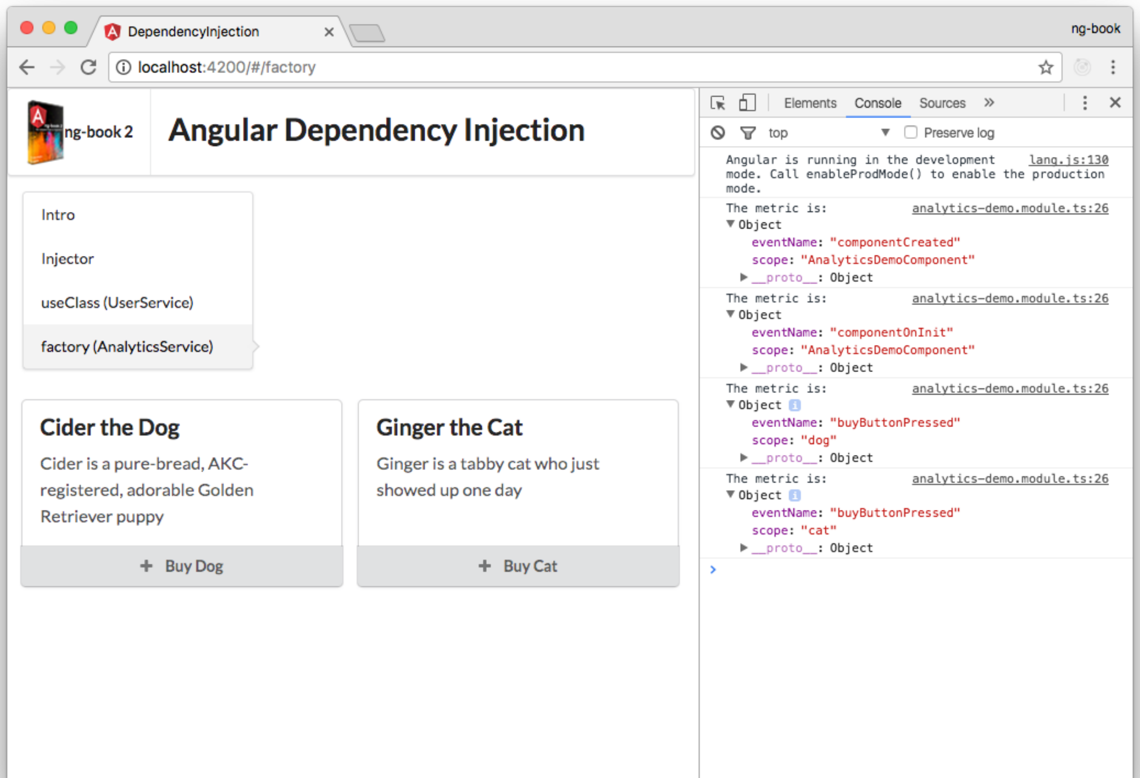
```
import { Inject } from '@angular/core';

export class AnalyticsDemoComponent {
  constructor(@Inject('API_URL') apiUrl: string) {
   // works! do something w/ apiUrl
  }
}
```

Now that we know how to do simple values with `useValue` and Singleton classes with `useClass`, we're ready to talk about the more advanced case: writing configurable services using factories.

Configurable Services

In the case of the `UserService`, no arguments are required for the `constructor`. But what happens if a service's `constructor` requires arguments? We can implement this by using a *factory* which is a function that can return **any object** when injected.

For instance, let's say we're writing a library for recording user analytics (that is, keeping a record of events of actions a user took on the page). In this scenario, we want to have an `AnalyticsService` with a catch: the `AnalyticsService` should define the interface for *recording* events, **but not the implementation for handling the event**.



Tracking Analytics on the events

Our user may, for instance, want to record these metrics with Google Analytics or they may want to use Optimizely, or some other in-house solution. Let's write an injectable `AnalyticsService` which can take an implementation configuration.

First, a couple of definitions. Let's define a `Metric`:

code/dependency-injection/src/app/analytics-demo/analytics-demo.interface.ts

```
export interface Metric {
  eventName: string;
  scope: string;
}
```

A `Metric` will store an `eventName` and a `scope`. We could use this for say, when a the user `nate` logs-in the `eventName` could be `loggedIn` and the `scope` would be `nate`.

```
// just an example
let metric: Metric = {
  eventName: 'loggedIn',
  scope: 'nate'
}
```

This way we could, in theory, count the number of user logins by counting the events with `eventName` `loggedIn` and count the number of times the specific user `nate` logged in by counting the `loggedIn` events with user `nate`.

We also need to define what an analytics implementation would look like:

code/dependency-injection/src/app/analytics-demo/analytics-demo.interface.ts

```
export interface AnalyticsImplementation {
  recordEvent(metric: Metric): void;
}
```

Here we define an `AnalyticsImplementation` interface to have one function: `recordEvent` which takes a `Metric` as an argument.

Now let's define the `AnalyticsService`:

code/dependency-injection/src/app/services/analytics.service.ts

```
import { Injectable } from '@angular/core';
import {
  Metric,
  AnalyticsImplementation
} from '../analytics-demo/analytics-demo.interface';

@Injectable()
export class AnalyticsService {
  constructor(private implementation: AnalyticsImplementation) {
  }
```

```

  record(metric: Metric): void {
    this.implementation.recordEvent(metric);
  }
}
```

Above our `AnalyticsService` defines one method: `record` which accepts a `Metric` and then passes it on to the `implementation`.

> Of course, this `AnalyticsService` is a bit trivial and in this case, we probably wouldn't need the indirection. But this same pattern could be used in the case where you had a more advanced `AnalyticsService`. For instance, we could add middleware or broadcast to several implementations.

Notice how its constructor method takes a phrase as a parameter? If we try to use the "regular" `useClass` injection mechanism we would see an error on the browser like:

```
Cannot resolve all parameters for AnalyticsService.
```

This happens because we didn't provide the injector with the implementation necessary for the `constructor`. In order to resolve this problem, we need to configure the provider to **use a factory**.

Using a Factory

So to use our `AnalyticsService`, we need to:

- create an implementation that conforms to `AnalyticsImplementation` and
- add it to `providers` with `useFactory`

Here's how:

code/dependency-injection/src/app/analytics-demo/analytics-demo.module.1.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

@NgModule({
```

```
  imports: [
    CommonModule
  ],
  providers: [
    {
      // `AnalyticsService` is the _token_ we use to inject
      // note, the token is the class, but it's just used as an identifier!
      provide: AnalyticsService,

      // useFactory is a function - whatever is returned from this function
      // will be injected
      useFactory() {

        // create an implementation that will log the event
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('The metric is:', metric);
          }
        };

        // create our new `AnalyticsService` with the implementation
        return new AnalyticsService(loggingImplementation);
      }
    }
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

Here in `providers` we're using the syntax:

```
providers: [
  { provide: AnalyticsService, useFactory: () => ... }
]
```

`useFactory` takes a function and **whatever this function returns** will be injected.

Also note that we `provide AnalyticsService`. Again, when we use `provide` this way, we're using the class `AnalyticsService` as the *identifying token* of what we're going to inject. (If you wanted to be confusing, you could use a completely separate class, or less-confusingly a string.)

In `useFactory` we're creating an `AnalyticsImplementation` object that has one function: `recordEvent`. `recordEvent` is where we could, in theory, configure **what** happens when an event is recorded.

Again, in a real app this would probably send an event to Google Analytics or a custom event logging software.

Lastly, we instantiate our `AnalyticsService` and return it.

Factory Dependencies

Using a factory is the most powerful way to create injectables, because we can do whatever we want within the factory function. Sometimes our factory function will have dependencies of it's own.

Say that we wanted to configure our `AnalyticsImplementation` to make an HTTP request to a particular URL. In order to do this we'd need:

- The Angular `Http` client and
- Our `API_URL` value

Here's how we could set that up:

code/dependency-injection/src/app/analytics-demo/analytics-demo.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

// added this ->
import {
  HttpModule,
  Http
} from '@angular/http';

@NgModule({
  imports: [
    CommonModule,
    HttpModule, // <-- added
  ],
  providers: [
    // add our API_URL provider
    { provide: 'API_URL', useValue: 'http://devserver.com' },
    {
      provide: AnalyticsService,

```

```
      // add our `deps` to specify the factory depencies
      deps: [ Http, 'API_URL' ],

      // notice we've added arguments here
      // the order matches the deps order
      useFactory(http: Http, apiUrl: string) {

        // create an implementation that will log the event
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('The metric is:', metric);
            console.log('Sending to: ', apiUrl);
            // ... You'd send the metric using http here ...
          }
        };

        // create our new `AnalyticsService` with the implementation
        return new AnalyticsService(loggingImplementation);
      }
    },
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

Here we're importing the `HttpModule`, both in the ES6 `import` (which makes the class *constants* available) and in our `NgModule` `imports` (which makes it available for dependency injection).

We've added an `API_URL` provider, as we did above. And then in our `AnalyticsService` provider, we've added a new key: `deps`. `deps` is an array of injection tokens and these tokens will be resolved and **passed as arguments to the factory function**.

Dependency Injection in Apps

To review, when writing our apps there are three steps we need to take in order to perform an injection:

1. Create the dependency (e.g. the service class)
2. Configure the injection (i.e. register the injection with Angular in our `NgModule`)
3. Declare the dependencies on the **receiving component**

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*.

Reminder on terminology: a *provider* provides (creates, instantiates, etc.) the *injectable* (the thing you want). In Angular when you want to access an *injectable* you *inject* a dependency into a function (often a `constructor`) and Angular's dependency injection framework will locate it and provide it to you.

As we can see, Dependency Injection provides a powerful way to manage dependencies within our app.

More Resources

- Official Angular DI Docs[47]
- Victor Savkin Compare DI in Angular 1 vs. Angular 2[48]

[47] https://angular.io/docs/ts/latest/guide/dependency-injection.html
[48] http://victorsavkin.com/post/126514197956/dependency-injection-in-angular-1-and-angular-2

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, 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 `HttpClient`
2. create a YouTube search-as-you-type component
3. discuss API details about the `HttpClient` 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/common/http`

HTTP has been split into a separate module in Angular. This means that to use it you need to `import` constants from `@angular/common/http`. For instance, we might import constants from `@angular/common/http` like this:

```
import {
  // The NgModule for using @angular/common/http
  HttpClientModule,

  // the class constants
  HttpClient
} from '@angular/common/http';
```

`import` from `@angular/common/http`

In our `app.module.ts` we're going to import `HttpClientModule` which is a convenience collection of modules.

code/http/src/app/app.module.ts

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

In our `NgModule` we will add `HttpClientModule` to the list of `imports`. The effect is that we will be able to inject `Http` (and a few other modules) into our components.

code/http/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    SimpleHttpComponent,
    MoreHttpRequestsComponent,
    YouTubeSearchComponent,
    SearchResultComponent,
    SearchBoxComponent
  ],
  imports: [
    BrowserModule,
```

```
    FormsModule,
    HttpClientModule // <-- right here
  ],
  providers: [youTubeSearchInjectables],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Notice that we have custom components in `declarations` as well as a custom `provider`. We'll talk about these later in the chapter.

Now we can inject the `HttpClient` service into our components (or anywhere we use dependency injection).

```
class MyFooComponent {
  constructor(public http: HttpClient) {
  }

  makeRequest(): void {
    // do something with this.http ...
  }
}
```

A Basic Request

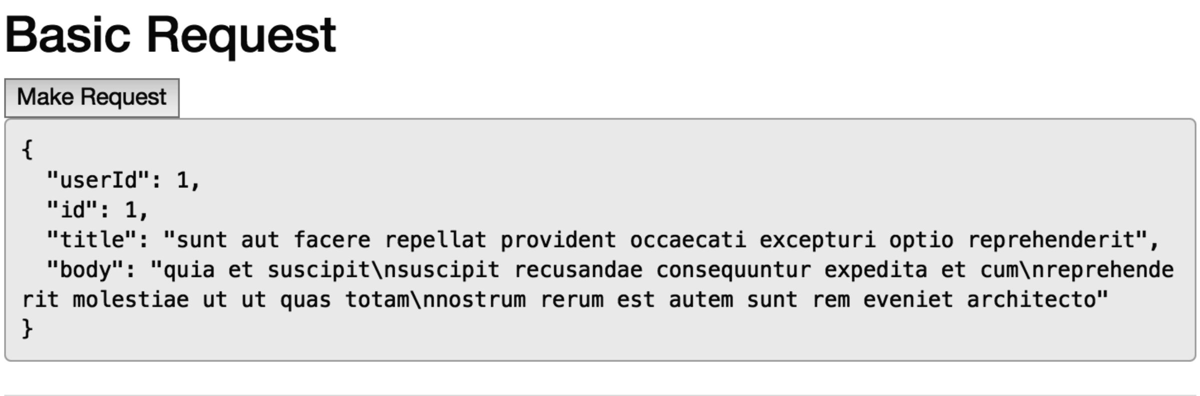
The first thing we're going to do is make a simple `GET` request to the `jsonplaceholder` API[49].

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:

[49] https://jsonplaceholder.typicode.com



Basic Request

Building the `SimpleHttpComponent` Component Definition

The first thing we're going to do is import a few modules and then specify a `selector` for our `@Component`:

code/http/src/app/simple-http/simple-http.component.ts

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

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: HttpClient) {}
```

Building the `SimpleHttpComponent template`

Next we build our view:

code/http/src/app/simple-http/simple-http.component.html

```html
<h2>Basic Request</h2>
<button type="button" (click)="makeRequest()">Make Request</button>
<div *ngIf="loading">loading...</div>
<pre>{{data | json}}</pre>
```

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/src/app/simple-http/simple-http.component.ts

```typescript
export class SimpleHttpComponent implements OnInit {
  data: Object;
  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/src/app/simple-http/simple-http.component.ts

```typescript
  constructor(private http: HttpClient) {}
```

The `constructor` body is empty, but we inject one key module: `HttpClient`.

Remember that when we use the `public` keyword in `public http: HttpClient` TypeScript will assign `http` to `this.http`. It's a shorthand for:

```
// other instance variables here
http: HttpClient;

constructor(http: HttpClient) {
  this.http = http;
}
```

Now let's make our first HTTP request by implementing the `makeRequest` function:

code/http/src/app/simple-http/simple-http.component.ts

```
  makeRequest(): void {
    this.loading = true;
    this.http
      .get('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
```

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.get` and pass the URL to which we want to make a `GET` request.

`http.get` returns an `Observable`. We can subscribe to changes (akin to using `then` from a Promise) using `subscribe`.

code/http/src/app/simple-http/simple-http.component.ts

```
    this.http
      .get('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
```

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/src/app/simple-http/simple-http.component.ts

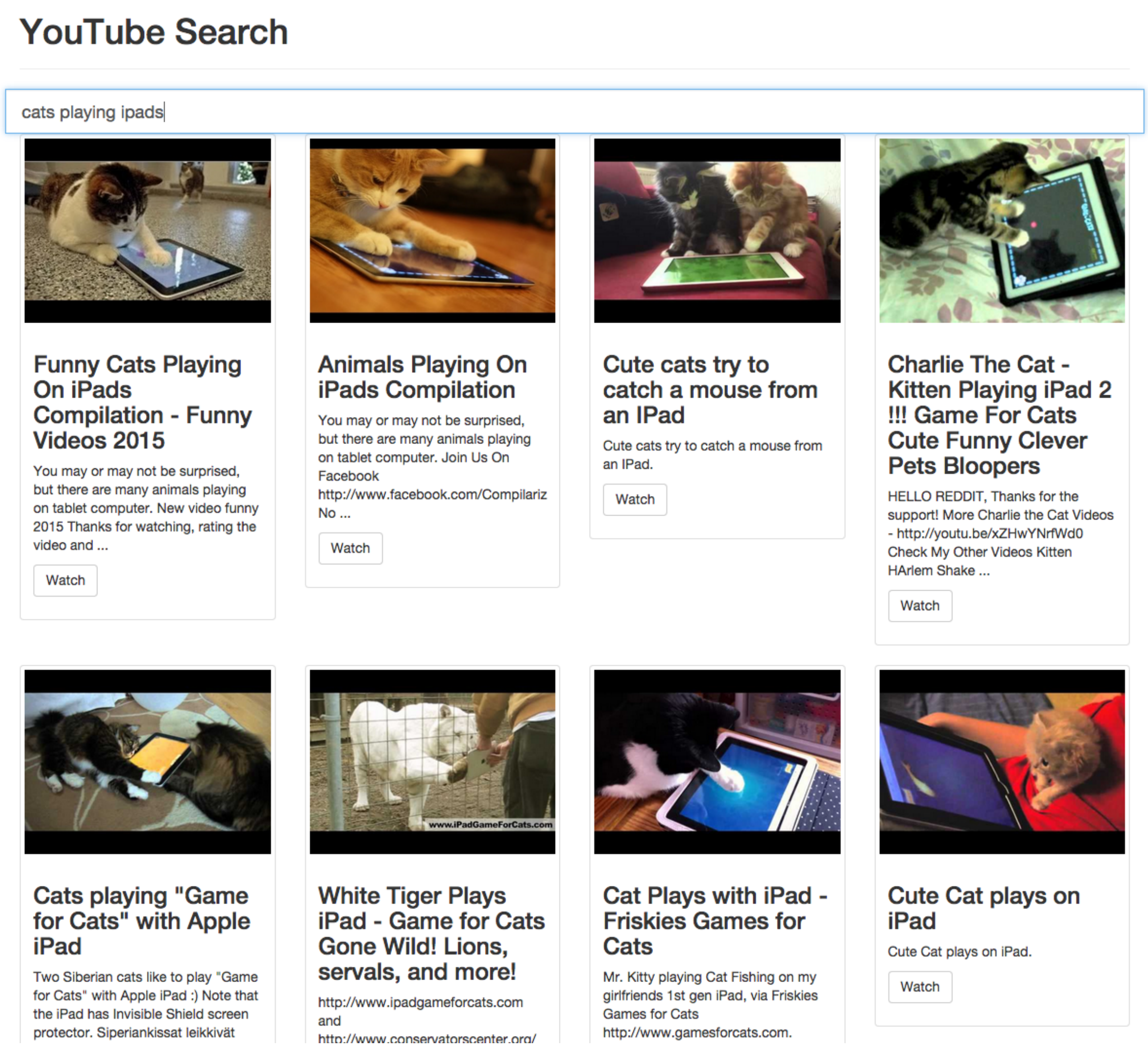
```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: HttpClient) {}

  ngOnInit() {}

  makeRequest(): void {
    this.loading = true;
    this.http
      .get('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
}
```

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":



Can I get my cat to write Angular?

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 `YouTubeSearchService` which will manage the API request to YouTube and convert the results to a stream of `SearchResult[]`
3. A `SearchBoxComponent` 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[50]. 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/src/app/you-tube-search/search-result.model.ts

```
/**
 * SearchResult is a data-structure that holds an individual
 * record from a YouTube video search
 */
export class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id            || null;
    this.title           = obj && obj.title         || null;
    this.description     = obj && obj.description   || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl  || null;
    this.videoUrl        = obj && obj.videoUrl      ||
                              `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

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.

[50] https://github.com/angular-class/angular2-webpack-starter

Writing the `YouTubeSearchService`

The API

For this example we're going to be using the YouTube v3 search API[51].

> 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[52]. For the sake of simplicity, 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 `YouTubeSearchService` mapping to our API key and the API URL:

```
let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
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 `{ provide: ... , useValue: ... }` syntax like this:

[51] https://developers.google.com/youtube/v3/docs/search/list
[52] https://developers.google.com/youtube/registering_an_application#Create_API_Keys

code/http/src/app/you-tube-search/you-tube-search.injectables.ts

```
import {
  YouTubeSearchService,
  YOUTUBE_API_KEY,
  YOUTUBE_API_URL
} from './you-tube-search.service';

export const youTubeSearchInjectables: Array<any> = [
  {provide: YouTubeSearchService, useClass: YouTubeSearchService},
  {provide: YOUTUBE_API_KEY, useValue: YOUTUBE_API_KEY},
  {provide: YOUTUBE_API_URL, useValue: YOUTUBE_API_URL}
];
```

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 `YouTubeSearchService` in a minute.)

> To get a refresher on the different ways to create 'injectables', checkout the chapter on dependency injection

If you recall, to make something available to be injected throughout our application, we need to put it in `providers` for our `NgModule`. Since we're exporting `youTubeServiceInjectables` here we can use it in our `app.module.ts`

```
// http/app.ts
import { HttpClientModule } from '@angular/common/http';
import { youTubeServiceInjectables } from "components/YouTubeSearchComponent";

// ...
// further down
// ...

@NgModule({
  declarations: [
    HttpApp,
    // others ....
  ],
  imports: [ BrowserModule, HttpClientModule ],
  bootstrap: [ HttpApp ],
  providers: [
    youTubeServiceInjectables // <--- right here
  ]
```

```
})
class HttpAppModule {}
```

Now we can inject `YOUTUBE_API_KEY` (from the `youTubeServiceInjectables`) instead of using the variable directly.

`YouTubeSearchService` constructor

We create our `YouTubeSearchService` by making a service `class`:

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(
    private http: HttpClient,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string
  ) {}
```

The `@Injectable` annotation allows us to inject things into this classes `constructor`.

In the `constructor` we inject three things:

1. `HttpClient`
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.

`YouTubeSearchService` 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/src/app/you-tube-search/you-tube-search.service.ts

```
  search(query: string): Observable<SearchResult[]> {
    const params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${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[53].)

Then we build the `queryUrl` by concatenating the `apiUrl` and the `params`.

Now that we have a `queryUrl` we can make our request. In this case we are going to use `http.get`, although `HttpClient` can make any kind of request (POST, DELETE, GET, etc.):

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
  search(query: string): Observable<SearchResult[]> {
    const params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl).map(response => {
      return <any>response['items'].map(item => {
        // console.log("raw item", item); // uncomment if you want to debug
        return new SearchResult({
          id: item.id.videoId,
          title: item.snippet.title,
          description: item.snippet.description,
          thumbnailUrl: item.snippet.thumbnails.high.url
        });
      });
    });
  }
```

[53] https://developers.google.com/youtube/v3/docs/search/list

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 browsers 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

`YouTubeSearchService` Full Listing

Here's the full listing of our `YouTubeSearchService`.

In this chapter we are adding some style using the CSS framework Bootstrap[54]

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(
    private http: HttpClient,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string
  ) {}

  search(query: string): Observable<SearchResult[]> {
    const params: string = [
```

[54] http://getbootstrap.com

```
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl).map(response => {
      return <any>response['items'].map(item => {
        // console.log("raw item", item); // uncomment if you want to debug
        return new SearchResult({
          id: item.id.videoId,
          title: item.snippet.title,
          description: item.snippet.description,
          thumbnailUrl: item.snippet.thumbnails.high.url
        });
      });
    });
  }
}
```

Writing the `SearchBoxComponent`

The `SearchBoxComponent` plays a key role in our app: it is the mediator between our UI and the `YouTubeSearchService`.

The `SearchBoxComponent` will :

1. Watch for `keyup` on an `input` and submit a search to the `YouTubeSearchService`
2. Emit a `loading` event when we're loading (or not)
3. Emit a `results` event when we have new results

`SearchBoxComponent @Component` Definition

Let's define our `SearchBoxComponent @Component`:

code/http/src/app/you-tube-search/search-box.component.ts

```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>\
();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }
```

The `selector` we've seen many times before: this allows us to create a `<app-search-box>` tag.

The two `@Outputs` specify that events 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 `app-search-box` tag in our view later on:

```
<app-search-box
 (loading)="loading = $event"
 (results)="updateResults($event)"
  ></app-search-box>
```

In this example, when the `SearchBoxComponent` emits a `loading` event, we will set the variable `loading` in the parent context. Likewise, when the `SearchBoxComponent` emits a `results` event, we will call the `updateResults()` function, with the value, in the parent's context.

In the `@Component` class we're specifying the properties of the events with the names `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.

`SearchBoxComponent` `template` Definition

Our template is straightforward. We have one `input` tag:

code/http/src/app/you-tube-search/search-box.component.ts

```
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
```

`SearchBoxComponent` Controller Definition

Our `SearchBoxComponent` controller is a new class:

code/http/src/app/you-tube-search/search-box.component.ts

```
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>\
();
```

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`.

Here we create the `EventEmitters` for both `loading` and the `results`. `loading` will emit a `boolean` when this search is loading and `results` will emit an array of `SearchResults` when the search is finished.

`SearchBoxComponent` Controller Definition `constructor`

Let's talk about the `SearchBoxComponent constructor`:

code/http/src/app/you-tube-search/search-box.component.ts

```
  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }
```

In our `constructor` we inject :

1. Our `YouTubeSearchService` 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.

`SearchBoxComponent` 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/src/app/you-tube-search/search-box.component.ts

```
  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    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:

```
.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:

```
.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.

```
.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.

```
.do(() => this.loading.emit(true))         // enable loading
```

Using `do` on a stream is a 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.emit(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).

```
.map((query: string) => this.youtube.search(query))
.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". That is, if a new search comes in, we want to use the most recent and discard the rest.

> Reactive experts will note that I'm handwaving here. `switch` has a more specific technical definition which you can read about in the RxJS docs here[55].

For each `query` that comes in, we're going to perform a `search` on our `YouTubeSearchService`.

Putting the chain together we have this:

[55] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/switch.md

code/http/src/app/you-tube-search/search-box.component.ts

```
  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.emit(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
```

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 `YouTubeSearchService` 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/src/app/you-tube-search/search-box.component.ts

```
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.emit(false);
          this.results.emit(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.emit(false);
        },
        () => { // on completion
          this.loading.emit(false);
        }
      );
  }
```

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.emit(false)`, indicating we've stopped loading

2. We call `this.results.emit(results)`, which will emit an event containing the list of results

The second argument specifies what should happen when the stream has an error event. Here we set `this.loading.emit(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.

`SearchBoxComponent`: Full Listing

All together, here's the full listing of our `SearchBoxComponent` Component:

code/http/src/app/you-tube-search/search-box.component.ts

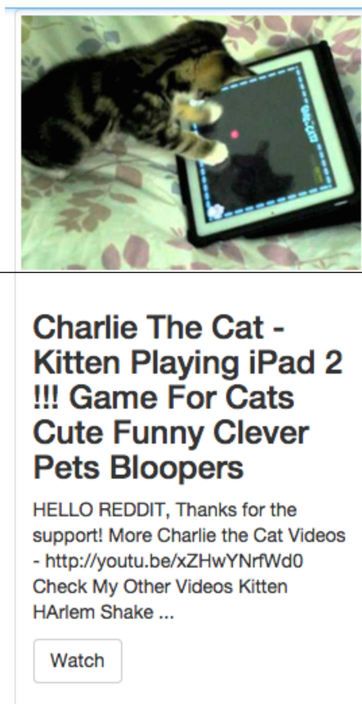
```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>\
();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.emit(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.emit(false);
          this.results.emit(results);
```

```
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.emit(false);
        },
        () => { // on completion
          this.loading.emit(false);
        }
      );
  }
}
```

Writing `SearchResultComponent`

The `SearchBoxComponent` was fairly complicated . Let's handle a **much** easier component now: the `SearchResultComponent`. The `SearchResultComponent`'s job is to render a single `SearchResult`.

Given what we've already covered there aren't any new ideas here, so let's take it all at once:



Single Search Result Component

code/http/src/app/you-tube-search/search-result.component.ts

```
import {
  Component,
  OnInit,
  Input
} from '@angular/core';
import { SearchResult } from './search-result.model';


@Component({
  selector: 'app-search-result',
  templateUrl: './search-result.component.html'
})
export class SearchResultComponent implements OnInit {
  @Input() result: SearchResult;

  constructor() { }

  ngOnInit() {
  }
```

```
}
```

A few things:

The `@Component` takes a single input `result`, on which we will put the `SearchResult` assigned to this component.

The template shows the title, description, and thumbnail of the video and then links to the video via a button.

code/http/src/app/you-tube-search/search-result.component.html

```html
<div class="col-sm-6 col-md-3">
   <div class="thumbnail">
     <img src="{{result.thumbnailUrl}}">
     <div class="caption">
       <h3>{{result.title}}</h3>
       <p>{{result.description}}</p>
       <p><a href="{{result.videoUrl}}"
             class="btn btn-default" role="button">
             Watch</a></p>
     </div>
   </div>
 </div>
```

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/src/app/you-tube-search/you-tube-search.component.ts

```
@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;
```

Our `@Component` decorator is straightforward: use the `selector` `app-you-tube-search`.

`YouTubeSearchComponent` Controller

Before we look at the template, let's take a look at the `YouTubeSearchComponent` controller:

code/http/src/app/you-tube-search/you-tube-search.component.ts

```
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;

  constructor() { }
  ngOnInit() { }

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

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 lets look at our template. Let's build some basic structure and show the loading gif next to the header:

code/http/src/app/you-tube-search/you-tube-search.component.html

```
<div class='container'>
    <div class="page-header">
      <h1>YouTube Search
        <img
          style="float: right;"
          *ngIf="loading"
          src='assets/images/loading.gif' />
      </h1>
    </div>
```

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/src/app/you-tube-search/you-tube-search.component.html

```
    <div class="row">
      <div class="input-group input-group-lg col-md-12">
        <app-search-box
           (loading)="loading = $event"
           (results)="updateResults($event)"
            ></app-search-box>
      </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 `SearchBoxComponent`, when we call `this.loading.emit(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 components `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/src/app/you-tube-search/you-tube-search.component.html

```
    <div class="row">
      <app-search-result
        *ngFor="let result of results"
        [result]="result">
      </app-search-result>
    </div>
</div>
```

`YouTubeSearchComponent` Full Listing

Here's the full listing for the `YouTubeSearchComponent`:

code/http/src/app/you-tube-search/you-tube-search.component.ts

```
@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;

  constructor() { }
  ngOnInit() { }

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

and the template:

code/http/src/app/you-tube-search/you-tube-search.component.html

```
<div class='container'>
    <div class="page-header">
      <h1>YouTube Search
        <img
          style="float: right;"
          *ngIf="loading"
          src='assets/images/loading.gif' />
      </h1>
    </div>

    <div class="row">
      <div class="input-group input-group-lg col-md-12">
        <app-search-box
           (loading)="loading = $event"
           (results)="updateResults($event)"
            ></app-search-box>
      </div>
    </div>

    <div class="row">
      <app-search-result
        *ngFor="let result of results"
        [result]="result">
      </app-search-result>
    </div>
</div>
```

There we have it! A functional search-as-you-type implemented for YouTube video search! Try running it from the code examples if you haven't already.

`@angular/common/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/common/http` is very much like making a `GET` request except that we have one additional parameter: a body.

jsonplaceholder API[56] also provides a convent URL for testing our POST requests, so let's use it for a `POST`:

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makePost(): void {
    this.loading = true;
    this.http
      .post(
        'https://jsonplaceholder.typicode.com/posts',
        JSON.stringify({
          body: 'bar',
          title: 'foo',
          userId: 1
        })
      )
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
```

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:

[56] http://jsonplaceholder.typicode.com

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makeDelete(): void {
    this.loading = true;
    this.http
      .delete('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
```

Custom HTTP Headers

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/src/app/more-http-requests/more-http-requests.component.ts

```
  makeHeaders(): void {
    const headers: HttpHeaders = new HttpHeaders({
      'X-API-TOKEN': 'ng-book'
    });

    const req = new HttpRequest(
      'GET',
      'https://jsonplaceholder.typicode.com/posts/1',
      {
        headers: headers
      }
    );

    this.http.request(req).subscribe(data => {
      this.data = data['body'];
    });
  }
```

Summary

`@angular/common/http` is flexible and suitable for a wide variety of APIs.

One of the great things about `@angular/common/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[57] you might write something like this:

```
var express = require('express');
var router = express.Router();

// define the about route
router.get('/about', function(req, res) {
  res.send('About us');
});
```

Or with Ruby on Rails[58] you might have:

```
# routes.rb
get '/about', to: 'pages#about'

# PagesController.rb
class PagesController < ActionController::Base
  def about
    render
  end
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?

[57] http://expressjs.com/guide/routing.html
[58] http://rubyonrails.org/

The beginning: using anchor tags

Client-side routing started out with a clever hack: Instead of using a normal server-side URL for a page in our SPA, we 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:

```
<!-- ... lots of page content here ... -->
<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, 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

The Angular docs recommends using HTML5 mode routing[59]. But due to the challenges mentioned in the previous section we will for simplicity be using hash based routing in our examples.

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 routing

There are three main components that we use to configure routing in Angular:

- `Routes` describes the routes our application supports
- `RouterOutlet` is a "placeholder" component that shows Angular where to put the content of each route
- `RouterLink` directive is used to link to routes

Let's look at each one more closely.

Imports

In order to use the router in Angular, we import constants from the `@angular/router` package:

[59] https://angular.io/docs/ts/latest/guide/router.html#!#browser-url-styles

code/routes/routing/src/app/app.module.ts

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

Now we can define our router configuration.

Routes

To define routes for our application, create a `Routes` configuration and then use `RouterModule.forRoot(routes)` to provide our application with the dependencies necessary to use the router. First, let's look at the routes definitions:

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

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

We'll dive into the details of each route in this chapter, but at a high-level, 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:4200/#/[60], we'll be redirected to the route `home`.

Another example is the `contactus` route:

code/routes/routing/src/app/app.module.ts

```
  { path: 'contactus', redirectTo: 'contact' },
```

In this case, if we visit the URL http://localhost:4200/#/contactus[61], 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/routing` 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.

Installing our Routes

Now that we have our `Routes routes`, we need to install it. To use the routes in our app we do two things to our `NgModule`:

1. Import the `RouterModule`
2. Install the routes using `RouterModule.forRoot(routes)` in the `imports` of our `NgModule`

Here's our routes configured into our `NgModule` for this app:

[60] http://localhost:4200/#/
[61] http://localhost:4200/#/contactus

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
```

code/routes/routing/src/app/app.module.ts

```
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(routes), // <-- routes

    // added this for our child module
    ProductsModule
  ],
```

RouterOutlet using <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.

Our component `@Component` has a `template` which specifies some `div` structure, a section for `Navigation`, and a directive called `router-outlet`.

The `router-outlet` element indicates where the contents of each route component will be rendered.

> We are are able to use the `router-outlet` directive in our template because we imported the `RouterModule` in our `NgModule`.

Here's the component and template for the navigation wrapper of our app:

code/routes/routing/src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private router: Router) {
  };
}
```

and the template:

code/routes/routing/src/app/app.component.html

```
<div class="page-header">
  <div class="container">
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="['/home']">Home</a>
      <a [routerLink]="['/about']">About Us</a>
      <a [routerLink]="['/contact']">Contact Us</a>
      |
      <a [routerLink]="['/products']">Products</a>
      <a [routerLink]="['/login']">Login</a>
      <a [routerLink]="['/protected']">Protected</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>
```

If we look at the template 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 to navigate to a given route?

We might try linking to the routes directly using pure HTML:

```
<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 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/routing/src/app/app.component.html

```
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="['/home']">Home</a>
      <a [routerLink]="['/about']">About Us</a>
      <a [routerLink]="['/contact']">Contact Us</a>
      |
```

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/routing/src/index.html

```html
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Routing</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

The code should be familiar by now, with the exception of this line:

```html
<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 case. You can declare the application base path programmatically, when configuring our `NgModule` by using the `APP_BASE_HREF` provider:

```
@NgModule({
  declarations: [ RoutesDemoApp ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy },
    { provide: APP_BASE_HREF, useValue: '/' } // <--- this right here
  ]
})
```

Putting `{ provide: APP_BASE_HREF, useValue: '/' }` in the `providers` is the equivalent of using <base href="/"> on our application HTML header.

> When deploying to production we can also set the value of the base-href by using the `--base-href` command-line option

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/routing/src/app/home/home.component.ts

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

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

  constructor() { }

```

```
  ngOnInit() {
  }

}
```

And template:

code/routes/routing/src/app/home/home.component.html

```
<h1>Welcome Home!</h1>
```

`AboutComponent`

Similarly, the `AboutComponent` will just have a basic `h1`:

code/routes/routing/src/app/about/about.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

And template:

code/routes/routing/src/app/about/about.component.html

```
<h1>About Us</h1>
```

`ContactComponent`

And, likewise with `AboutComponent`:

code/routes/routing/src/app/contact/contact.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

And template:

code/routes/routing/src/app/contact/contact.component.html

```
<h1>Contact Us</h1>
```

Nothing really very interesting about those components, so let's move on to the main `app.module.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/routing/src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import {
  RouterModule,
  Routes
```

Next step is to import the three components we created above:

code/routes/routing/src/app/app.module.ts

```
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ContactComponent } from './contact/contact.component';
import { AboutComponent } from './about/about.component';
```

For our root component, we're going to use two router directives: `RouterOutlet` and the `RouterLink`. Those directives, along with all other common router directives are imported when we put `RouterModule` in the `imports` section of our `NgModule`.

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 `AppComponent` template.

The `RouterLink` directive is used to create navigation links to our routes:

code/routes/routing/src/app/app.component.html

```
<div class="page-header">
  <div class="container">
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="['/home']">Home</a>
      <a [routerLink]="['/about']">About Us</a>
      <a [routerLink]="['/contact']">Contact Us</a>
      |
      <a [routerLink]="['/products']">Products</a>
      <a [routerLink]="['/login']">Login</a>
      <a [routerLink]="['/protected']">Protected</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>
```

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 `Routes` type:

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
```

code/routes/routing/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactComponent,
    AboutComponent,
    LoginComponent,
    ProtectedComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(routes), // <-- routes

    // added this for our child module
    ProductsModule
  ],
  providers: [
    // uncomment this for "hash-bang" routing
    // { provide: LocationStrategy, useClass: HashLocationStrategy }
    AUTH_PROVIDERS,
    LoggedInGuard
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Notice that we put all necessary components in our `declarations`. If we're going to route to a component, then it needs to be declared in *some* `NgModule` (either this module or imported).

In our `imports` we have `RouterModule.forRoot(routes)`. `RouterModule.forRoot(routes)` 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.

In our `providers` we have this:

```
{ 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 called *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, how do you set this up?
>
> In order to use HTML5 mode routing, you have to configure your server to redirect every "missing" route to the root URL.
>
> Angular CLI supports this natively, but know that it doesn't necessarily work by default on your server. In the `routes/routing` project you can use HTML5 routes by simply doing `ng serve`

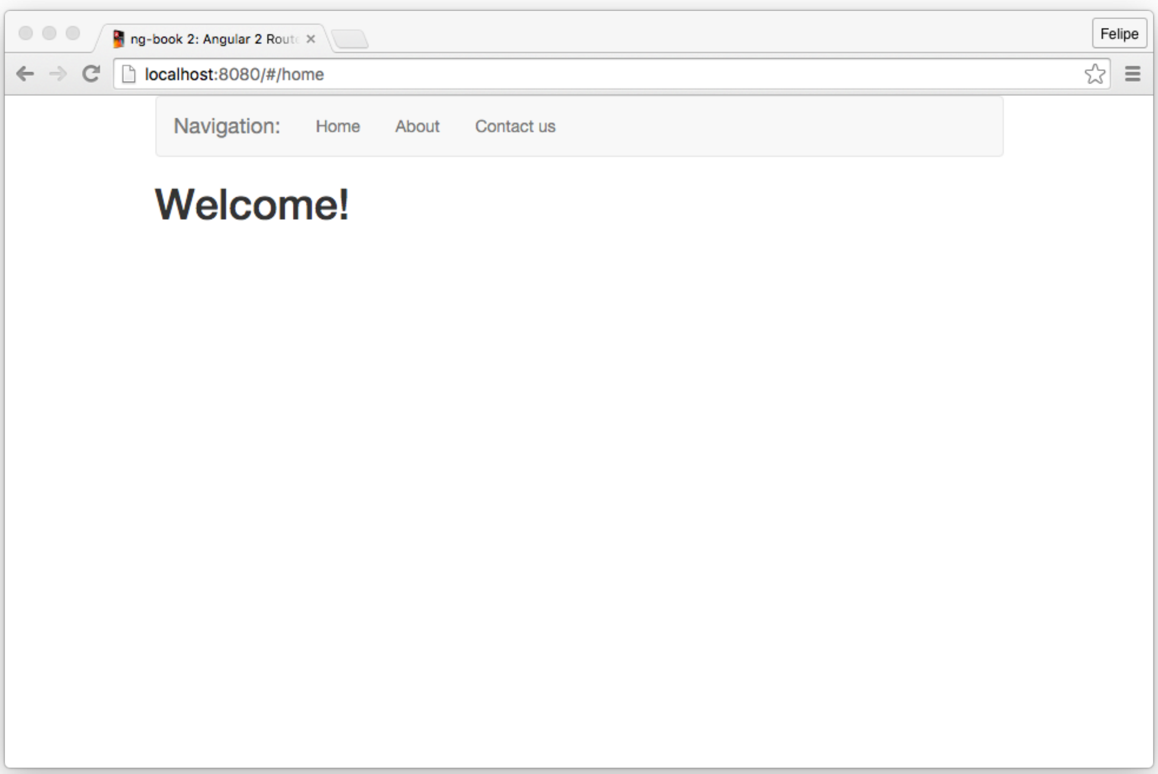
If we wanted to make our example application work with this new strategy, first we have to import `LocationStrategy` and `HashLocationStrategy` and then add that location strategy to the providers of our `NgModule`.

> **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 source for the `HashLocationStrategy` or `PathLocationStrategy` classes.

Running the application

You can now go into the application root folder (`code/routes/routing`) and run `npm start` to boot the application.

When you type http://localhost:4200/[62] into your browser you should see the home route rendered:

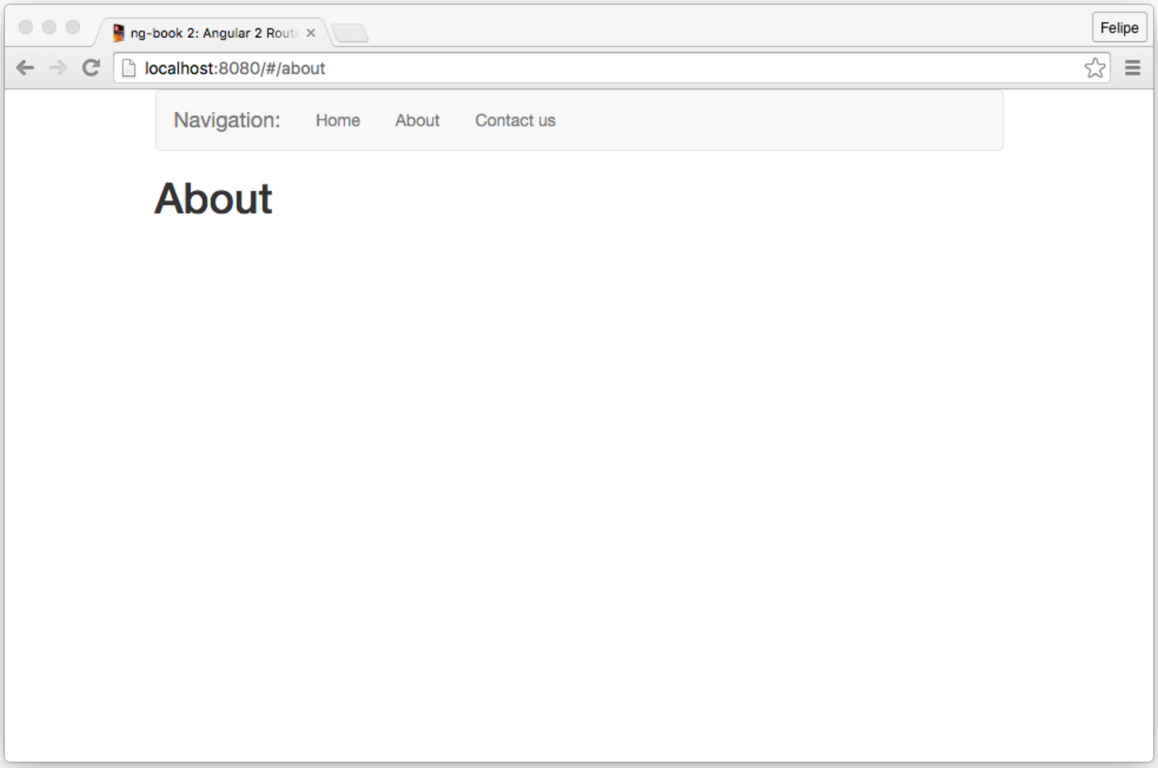


Home Route

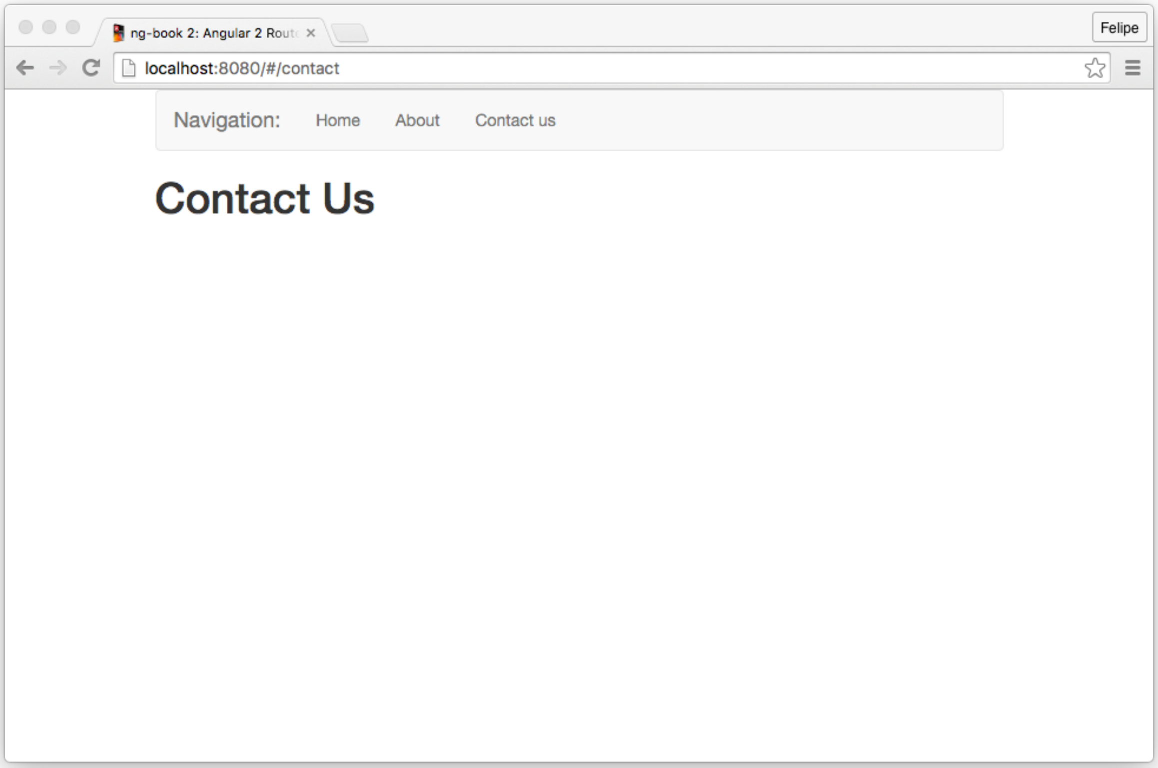
Notice that the URL in the browser was redirected to http://localhost:4200/home[63].

[62] http://localhost:4200/
[63] http://localhost:4200/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:

`/product/:id`

To add a parameter to our router configuration, we specify the route path like this:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent },
];
```

When we visit the route `/product/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`:

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

Next, we inject the `ActivatedRoute` into the constructor of our component. For example, let's say we have a `Routes` that specifies the following:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent }
];
```

Then when we write the `ProductComponent`, we add the `ActivatedRoute` as one of the constructor arguments:

```
export class ProductComponent {
  id: string;

  constructor(private route: ActivatedRoute) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
}
```

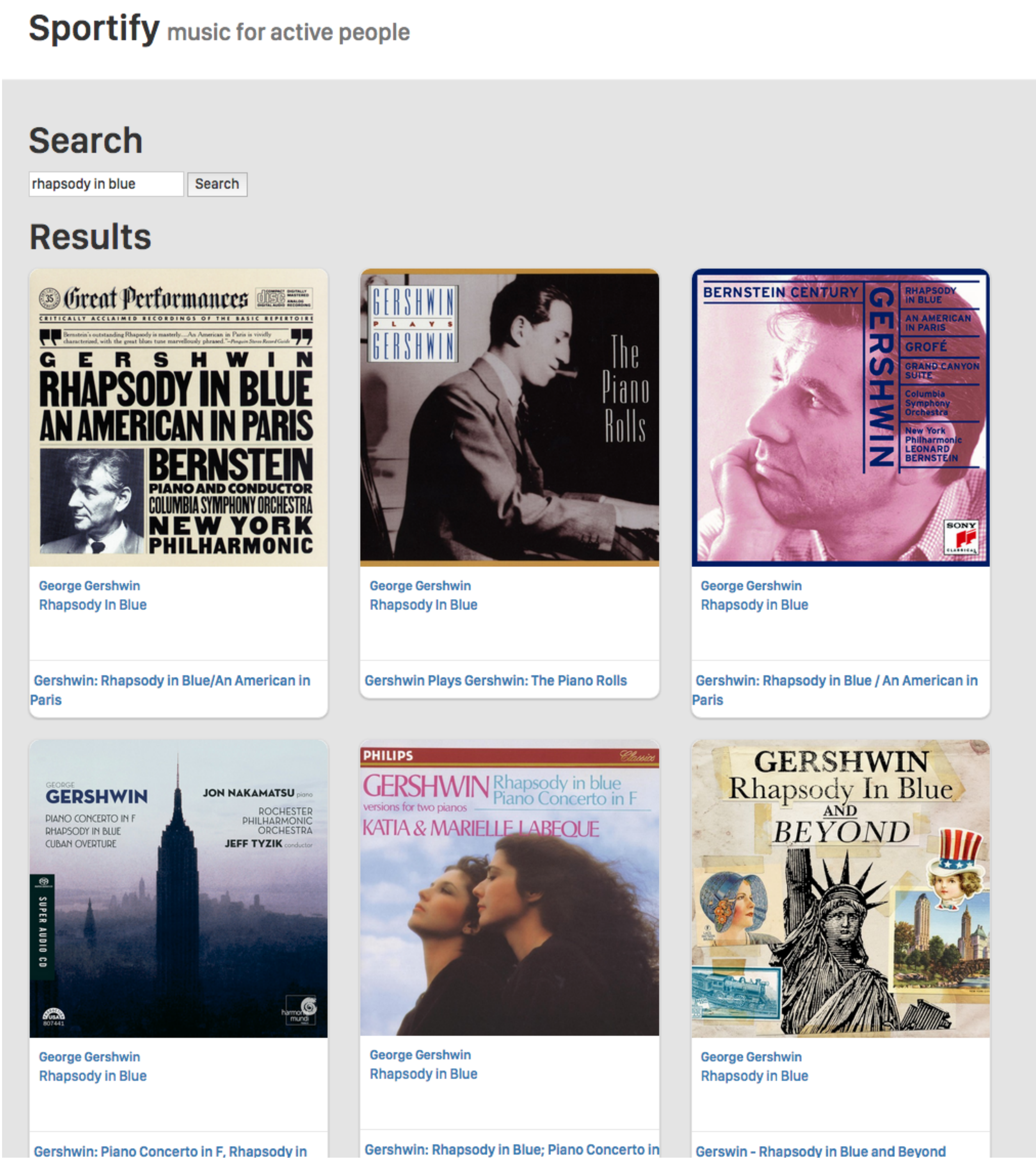
Notice that `route.params` is an *observable*. We can extract the value of the param into a hard value by using `.subscribe`. In this case, we assign the value of `params['id']` to the `id` instance variable on the component.

Now when we visit `/product/230`, our component's `id` attribute will 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 **artist details** when the artist 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



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[64] to get information about tracks, artists and albums.

First Steps

The first file we need work on is `app.module.ts`. Let's start by importing classes we'll use from Angular:

code/routes/music/src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import {
  RouterModule,
  Routes
} from '@angular/router';
import {
  LocationStrategy,
  HashLocationStrategy,
  APP_BASE_HREF
} from '@angular/common';

import { AppComponent } from './app.component';
import { AlbumComponent } from './album/album.component';
import { ArtistComponent } from './artist/artist.component';
```

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

[64] https://developer.spotify.com/web-api

- 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 final version of the SpotifyService in the routes/music/src/app folder of the sample code.

The first method we'll implement is searchTrack which will search for a track, given a search term.

One of the endpoints documented on Spotify API docs is the Search endpoint[65].

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:

```
class SpotifyService {
  constructor(public http: Http) {
  }

  searchTrack(query: string) {
    let params: string = [
      `q=${query}`,
      `type=track`
    ].join("&");
    let queryURL: string = `https://api.spotify.com/v1/search?${params}`;
    return this.http.request(queryURL).map(res => res.json());
  }
}
```

This code performs an HTTP GET request to the URL https://api.spotify.com/v1/search[66], passing our query as the search term and type hardcoded to track.

[65] https://developer.spotify.com/web-api/search-item/
[66] https://api.spotify.com/v1/search

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 `searchTrack` will then have to use the `Observable` API to subscribe to the response like this:

```
service
  .searchTrack('query')
  .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`.

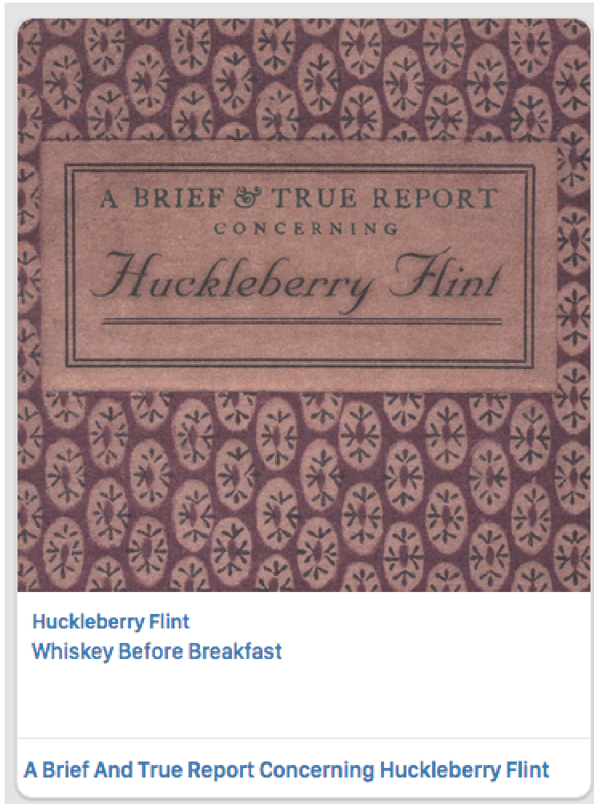
Again, we start with an import section:

code/routes/music/src/app/search/search.component.ts

```
/*
 * Angular
 */

import {Component, OnInit} from '@angular/core';
import {
  Router,
  ActivatedRoute,
} from '@angular/router';

/*
 * Services
 */
import {SpotifyService} from '../spotify.service';
```

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, making a few imports and using the following template. The template is a bit long because we're putting some reasonable styles on it using the CSS framework Bootstrap[67], but it isn't particularly complicated, relative to what we've done so far:

code/routes/music/src/app/search/search.component.html

```
<h1>Search</h1>

<p>
  <input type="text" #newquery
    [value]="query"
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</p>

<div *ngIf="results">
  <div *ngIf="!results.length">
    No tracks were found with the term '{{ query }}'
  </div>
```

[67]http://getbootstrap.com

```

  <div *ngIf="results.length">
    <h1>Results</h1>

    <div class="row">
      <div class="col-sm-6 col-md-4" *ngFor="let t of results">
        <div class="thumbnail">
          <div class="content">
            <img src="{{ t.album.images[0].url }}" class="img-responsive">
            <div class="caption">
              <h3>
                <a [routerLink]="['/artists', t.artists[0].id]">
                  {{ t.artists[0].name }}
                </a>
              </h3>
              <br>
              <p>
                <a [routerLink]="['/tracks', t.id]">
                  {{ t.name }}
                </a>
              </p>
            </div>
            <div class="attribution">
              <h4>
                <a [routerLink]="['/albums', t.album.id]">
                  {{ t.album.name }}
                </a>
              </h4>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

The Search Field

Let’s break down the HTML template a bit.

This first section will have the search field:

code/routes/music/src/app/search/search.component.html

```
<p>
  <input type="text" #newquery
    [value]="query"
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</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/src/app/search/search.component.html

```
    <div class="row">
      <div class="col-sm-6 col-md-4" *ngFor="let t of results">
        <div class="thumbnail">
```

For each track, we display the artist name:

code/routes/music/src/app/search/search.component.html

```
              <h3>
                <a [routerLink]="['/artists', t.artists[0].id]">
                  {{ t.artists[0].name }}
                </a>
              </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/src/app/search/search.component.html

```
              <p>
                <a [routerLink]="['/tracks', t.id]">
                  {{ t.name }}
                </a>
              </p>
```

And the album:

code/routes/music/src/app/search/search.component.html

```
                <a [routerLink]="['/albums', t.album.id]">
                  {{ t.album.name }}
                </a>
              </h4>
```

`SearchComponent` Class

Let's take a look at the constructor first:

code/routes/music/src/app/search/search.component.ts

```
export class SearchComponent implements OnInit {
  query: string;
  results: Object;

  constructor(private spotify: SpotifyService,
              private router: Router,
              private route: ActivatedRoute) {
    this.route
      .queryParams
      .subscribe(params => { this.query = params['query'] || ''; });
  }
```

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/src/app/search/search.component.ts

```
  search(): void {
    console.log('this.query', this.query);
    if (!this.query) {
      return;
    }

    this.spotify
      .searchTrack(this.query)
      .subscribe((res: any) => this.renderResults(res));
  }
```

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/src/app/search/search.component.ts

```
renderResults(res: any): void {
  this.results = null;
  if (res && res.tracks && res.tracks.items) {
    this.results = res.tracks.items;
  }
}
```

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/src/app/search/search.component.ts

```
ngOnInit(): void {
  this.search();
}
```

> 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/src/app/search/search.component.ts

```
  submit(query: string): void {
    this.router.navigate(['search'], { queryParams: { query: query } })
      .then(_ => this.search() );
  }
```

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:

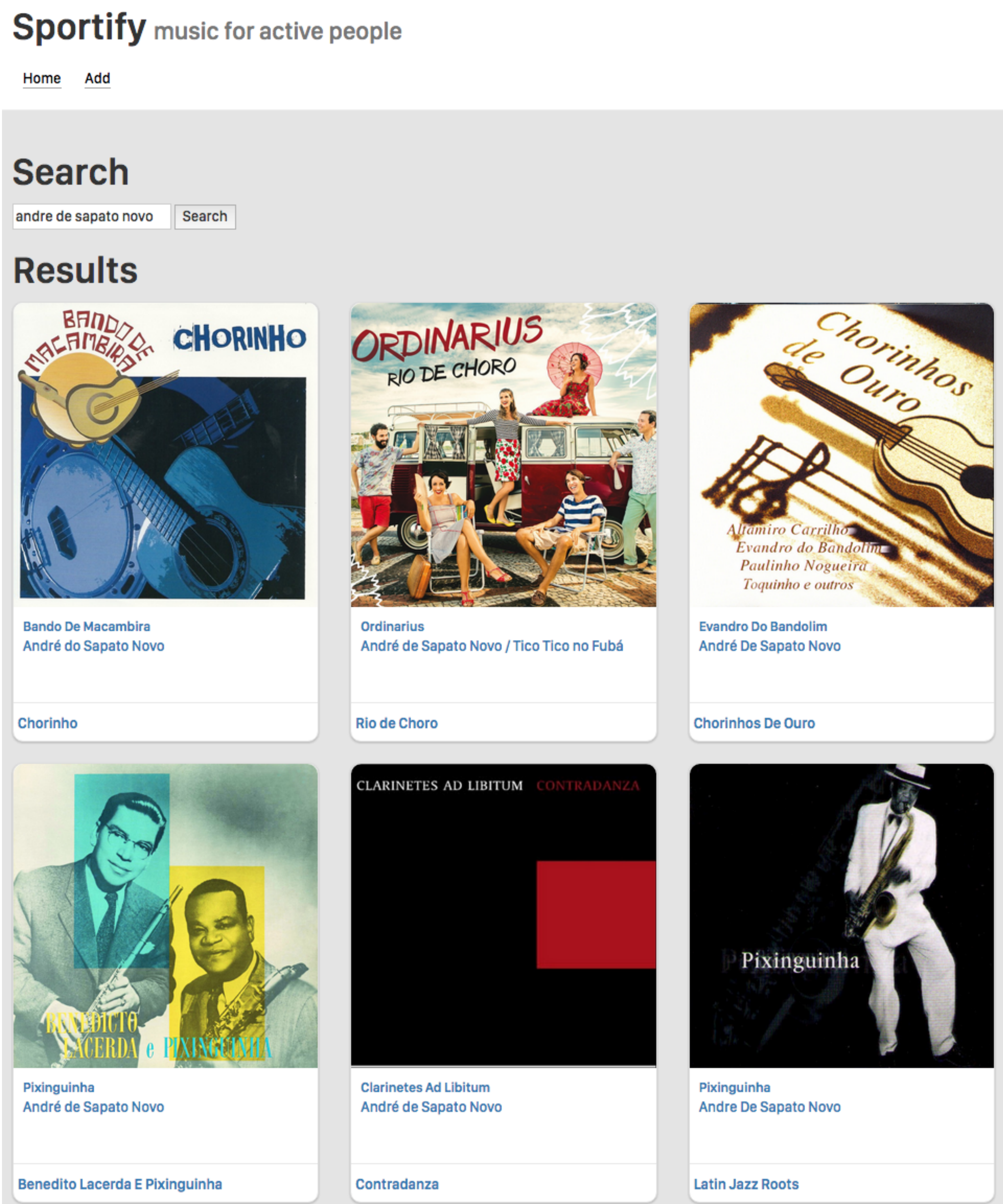
code/routes/music/src/app/search/search.component.ts

```
/*
 * Angular
 */

import {Component, OnInit} from '@angular/core';
import {
  Router,
  ActivatedRoute,
} from '@angular/router';

/*
 * Services
 */
import {SpotifyService} from '../spotify.service';
;

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent implements OnInit {
  query: string;
  results: Object;

  constructor(private spotify: SpotifyService,
```

```
              private router: Router,
              private route: ActivatedRoute) {
    this.route
      .queryParams
      .subscribe(params => { this.query = params['query'] || ''; });
  }

  ngOnInit(): void {
    this.search();
  }

  submit(query: string): void {
    this.router.navigate(['search'], { queryParams: { query: query } })
      .then(_ => this.search() );
  }

  search(): void {
    console.log('this.query', this.query);
    if (!this.query) {
      return;
    }

    this.spotify
      .searchTrack(this.query)
      .subscribe((res: any) => this.renderResults(res));
  }

  renderResults(res: any): void {
    this.results = null;
    if (res && res.tracks && res.tracks.items) {
      this.results = res.tracks.items;
    }
  }
}
```

Trying the search

Now that we have completed the code for the search, let's try it out:



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/src/app/track/track.component.html

```
<div *ngIf="track">
  <h1>{{ track.name }}</h1>

  <p>
    <img src="{{ track.album.images[1].url }}">
  </p>

  <p>
    <audio controls src="{{ track.preview_url }}"></audio>
  </p>

  <p><a href (click)="back()">Back</a></p>
</div>
```

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/src/app/spotify.service.ts

```
export class SpotifyService {
  static BASE_URL = 'https://api.spotify.com/v1';

  constructor(private http: Http) {}

  query(
    URL: string,
    params?: Array<string>
  ): Observable<any[]> {
    let queryURL = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join('&')}`;
    }
    const apiKey = environment.spotifyApiKey;
    const headers = new Headers({
      Authorization: `Bearer ${apiKey}`
    });
    const options = new RequestOptions({
```

```
      headers: headers
    });

    return this.http
      .request(queryURL, options)
      .map((res: any) => res.json());
  }

  search(query: string, type: string): Observable<any[]> {
    return this.query(`/search`, [
      `q=${query}`,
      `type=${type}`
    ]);
  }
```

Now that we've extracted those methods into the `SpotifyService`, notice how much simpler `searchTrack` becomes:

code/routes/music/src/app/spotify.service.ts

```
  searchTrack(query: string): Observable<any[]> {
    return this.search(query, 'track');
  }
```

Now let's create a method to allow the component we're building retrieve track information, based in the track ID:

code/routes/music/src/app/spotify.service.ts

```
  getTrack(id: string): Observable<any[]> {
    return this.query(`/tracks/${id}`);
  }
```

And now we can use `getTrack` from a new `ngOnInit` method on the `TrackComponent`:

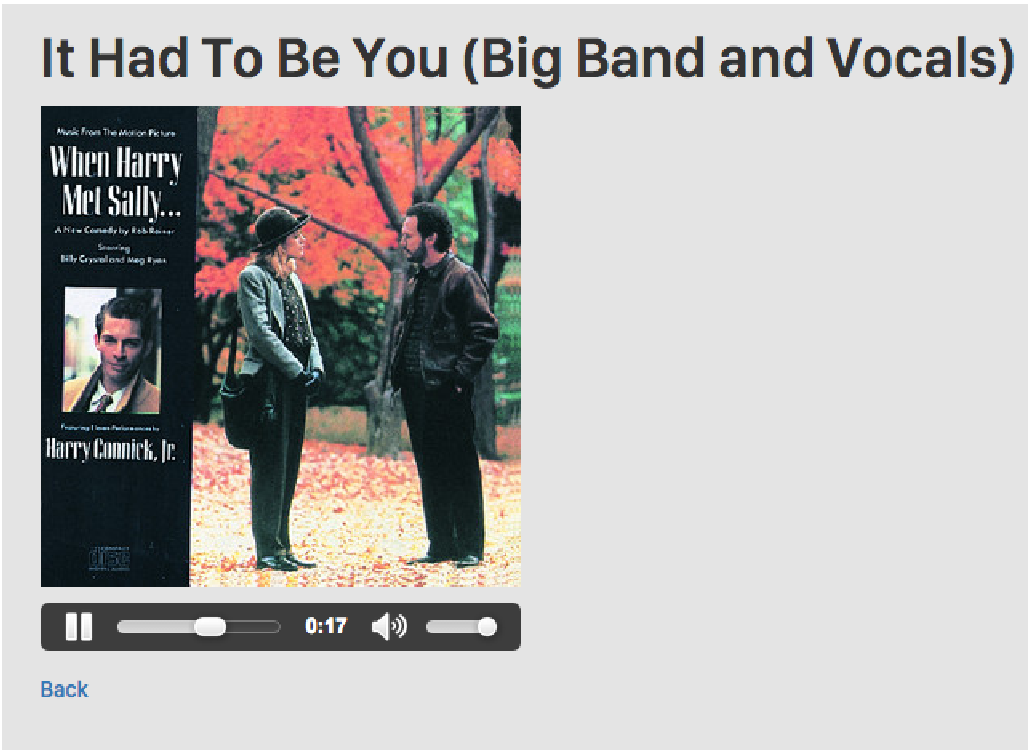
code/routes/music/src/app/track/track.component.ts

```
  ngOnInit(): void {
    this.spotify
      .getTrack(this.id)
      .subscribe((res: any) => this.renderTrack(res));
  }
```

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 classic 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 build on the first section and can be found in the `routes/routing` 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/routing/src/app/auth.service.ts

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

@Injectable()
export class AuthService {
  login(user: string, password: string): boolean {
    if (user === 'user' && password === 'password') {
      localStorage.setItem('username', user);
      return true;
    }

    return false;
  }
```

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[68]

The `logout` method just clears the `username` value:

[68] https://developer.mozilla.org/en-US/docs/Web/API/Storage

code/routes/routing/src/app/auth.service.ts

```
  logout(): any {
    localStorage.removeItem('username');
  }
```

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/routing/src/app/auth.service.ts

```
  getUser(): any {
    return localStorage.getItem('username');
  }

  isLoggedIn(): boolean {
    return this.getUser() !== null;
  }
```

The last thing we do is export an `AUTH_PROVIDERS`, so it can be injected into our app:

code/routes/routing/src/app/auth.service.ts

```
export const AUTH_PROVIDERS: Array<any> = [
  { provide: AuthService, useClass: AuthService }
];
```

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/routing/src/app/login/login.component.ts

```
export class LoginComponent {
  message: string;

  constructor(public authService: AuthService) {
    this.message = '';
  }

  login(username: string, password: string): boolean {
    this.message = '';
    if (!this.authService.login(username, password)) {
      this.message = 'Incorrect credentials.';
      setTimeout(function() {
        this.message = '';
      }.bind(this), 2500);
    }
    return false;
  }

  logout(): boolean {
    this.authService.logout();
    return false;
  }
```

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/routing/src/app/login/login.component.html

```
</div>

<form class="form-inline" *ngIf="!authService.getUser()">
  <div class="form-group">
    <label for="username">User: (type <em>user</em>)</label>
    <input class="form-control" name="username" #username>
  </div>

  <div class="form-group">
    <label for="password">Password: (type <em>password</em>)</label>
    <input class="form-control" type="password" name="password" #password>
```

```
  </div>

  <a class="btn btn-default" (click)="login(username.value, password.value)">
    Submit
```

And the information banner, containing the logout link, protected by the inverse - `*ngIf="authService.getUser()"`:

code/routes/routing/src/app/login/login.component.html

```
<div class="well" *ngIf="authService.getUser()">
  Logged in as <b>{{ authService.getUser() }}</b>
  <a href (click)="logout()">Log out</a>
</div>
```

There's another snippet of code that is displayed when we have an authentication error:

code/routes/routing/src/app/login/login.component.html

```
<div class="alert alert-danger" role="alert" *ngIf="message">
  {{ message }}
</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/routing/src/app/protected/protected.component.ts

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

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

  constructor() { }
```

```

  ngOnInit() {
  }

}
```

And the template will show some protected content:

code/routes/routing/src/app/protected/protected.component.html

```html
<h1>Protected</h1>
<p>
  Protected content
</p>
```

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 file `logged-in.guard.ts`:

code/routes/routing/src/app/logged-in.guard.ts

```typescript
/* tslint:disble max-line-length */
import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
} from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { AuthService } from './auth.service';

@Injectable()
export class LoggedInGuard implements CanActivate {
  constructor(private authService: AuthService) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
      const isLoggedIn = this.authService.isLoggedIn();
      console.log('canActivate', isLoggedIn);
```

```
      return isLoggedIn;
  }
}
```

Angular CLI contains a generator for creating guards. So this file could be created with the command: `ng generate guard logged-in`

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.module.ts`.

We import the `LoggedInGuard`:

code/routes/routing/src/app/app.module.ts

```
import { AUTH_PROVIDERS } from './auth.service';
import { LoggedInGuard } from './logged-in.guard';
```

We add `canActivate` with our guard to the `protected` route:

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

We add `LoggedInGuard` to our list of providers:

code/routes/routing/src/app/app.module.ts

```
  providers: [
    // uncomment this for "hash-bang" routing
    // { provide: LocationStrategy, useClass: HashLocationStrategy }
    AUTH_PROVIDERS,
    LoggedInGuard
  ],
```

Logging in

We import the `LoginComponent`:

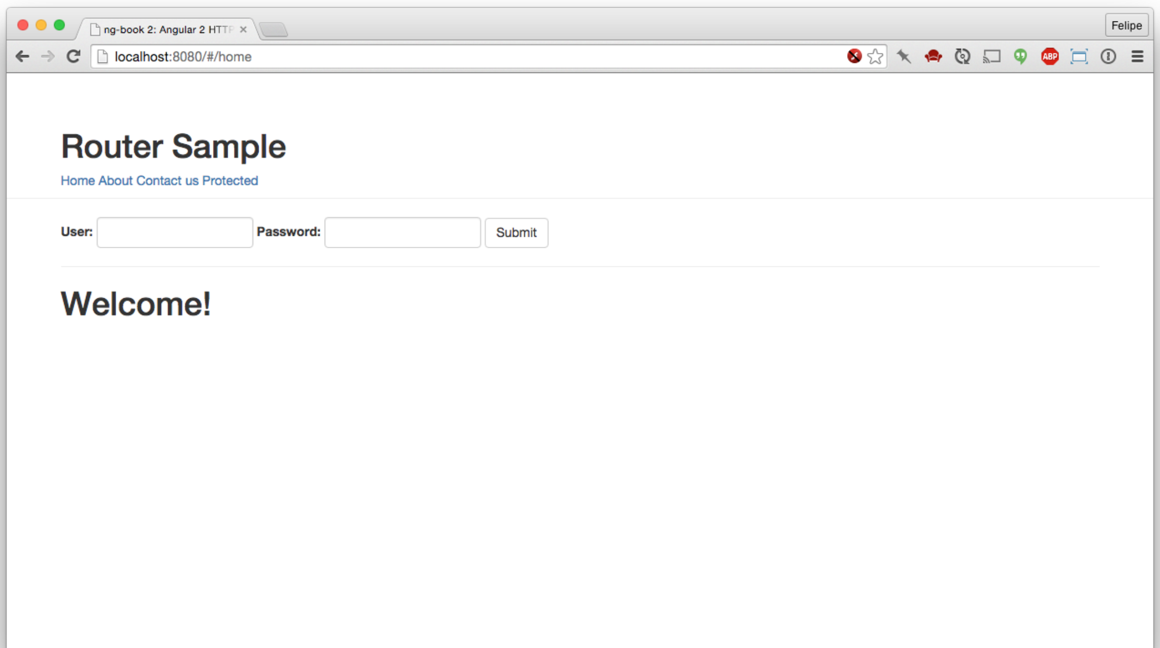
code/routes/routing/src/app/app.module.ts

```
import { LoginComponent } from './login/login.component';
```

And then to access it we have:

1. a route that links to the `LoginComponent`
2. a new link to the protected route

Now when we open the application on the browser, we can see the new login form and the new protected link:

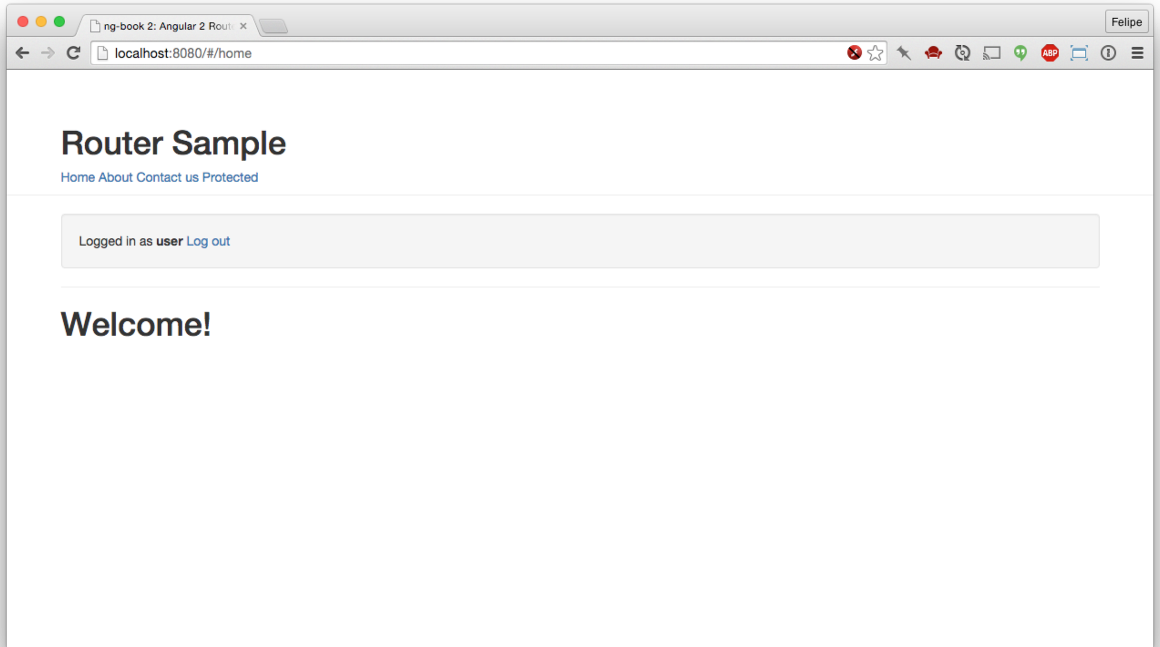


Auth App - Initial Page

If you click the Protected link, you'll see nothing happens. The same happens if you try to manually visit http://localhost:4200/protected[69].

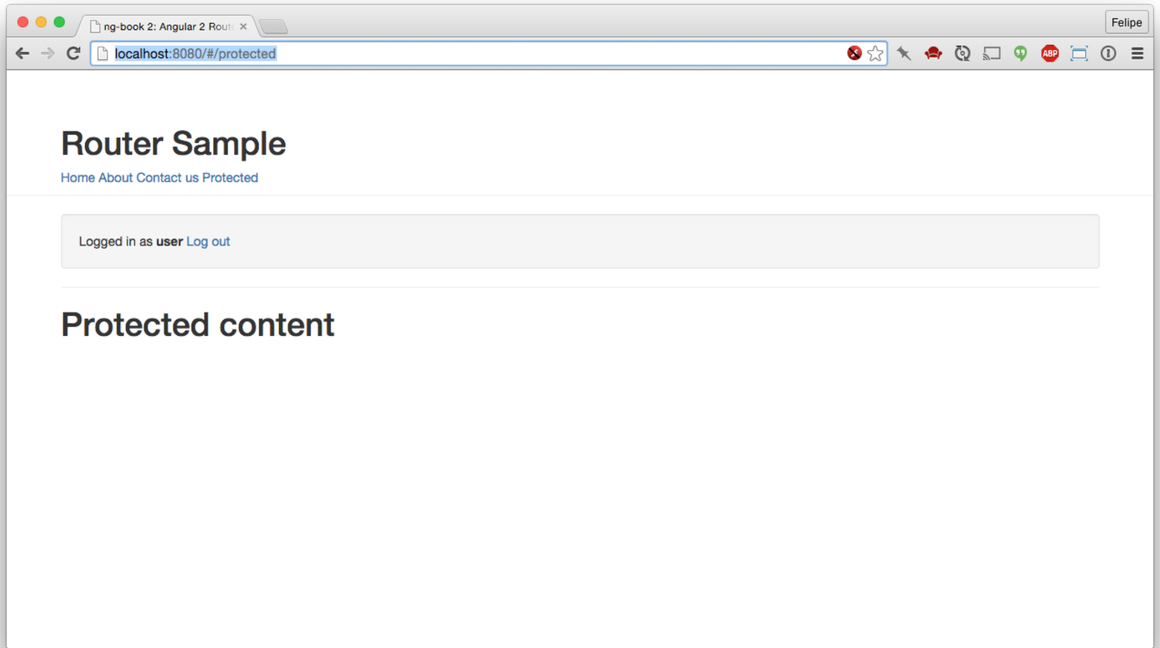
Now enter the string `user` for the user and `password` for the password on the form and click **Submit**. You'll see that we now get the current user displayed on a banner:

[69] http://localhost:4200/protected



Auth App - Logged In

And, sure enough, if we click the Protected link, it gets redirected and the component is rendered:



Auth App - Protected Area

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 doesn'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 these 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.

Configuring Routes

We will start by describing the `products` route on the `app.module.ts` file:

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

Notice that `products` has a `children` parameter. Where does this come from? We've defined the `childRoutes` **in a new module**: the `ProductsModule`. Let's take a look:

`ProductsModule`

The `ProductsModule` will have its own route configuration:

code/routes/routing/src/app/products/products.module.ts

```
export const routes: Routes = [
  { path: '', redirectTo: 'main', pathMatch: 'full' },
  { path: 'main', component: MainComponent },
  { path: 'more-info', component: MoreInfoComponent },
  { path: ':id', component: ProductComponent },
];
```

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/routing/src/app/products/products.component.html

```
<div class="navLinks">
  <a [routerLink]="['./main']">Main</a> |
  <a [routerLink]="['./more-info']">More Info</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/routing/src/app/products/products.component.html

```
<h2>Products</h2>

<div class="navLinks">
  <a [routerLink]="['./main']">Main</a> |
  <a [routerLink]="['./more-info']">More Info</a> |
  Enter id: <input #id size="6">
  <button (click)="goToProduct(id.value)">Go</button>
</div>

<div class="products-area">
  <router-outlet></router-outlet>
</div>
```

Let's look at the `ProductsComponent` definition:

code/routes/routing/src/app/products/products.component.ts

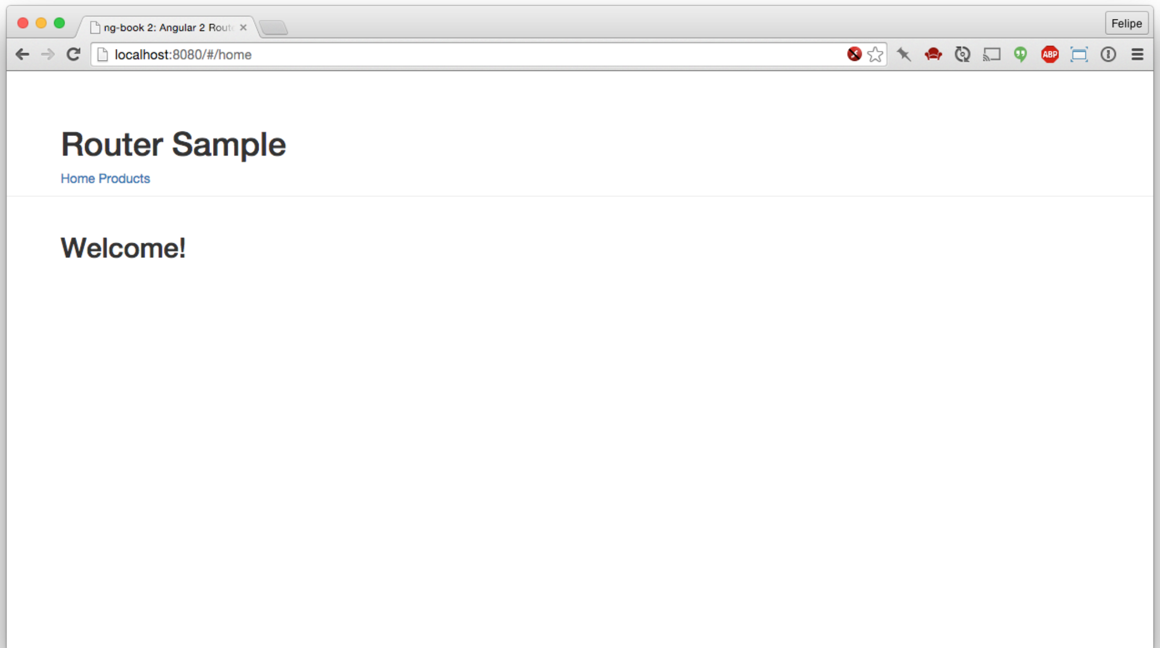
```
import { Component } from '@angular/core';
import {
  ActivatedRoute,
  Router
} from '@angular/router';

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent {
  constructor(private router: Router, private route: ActivatedRoute) {
  }

  goToProduct(id: string): void {
    this.router.navigate(['./', id], {relativeTo: this.route});
  }
}
```

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`.

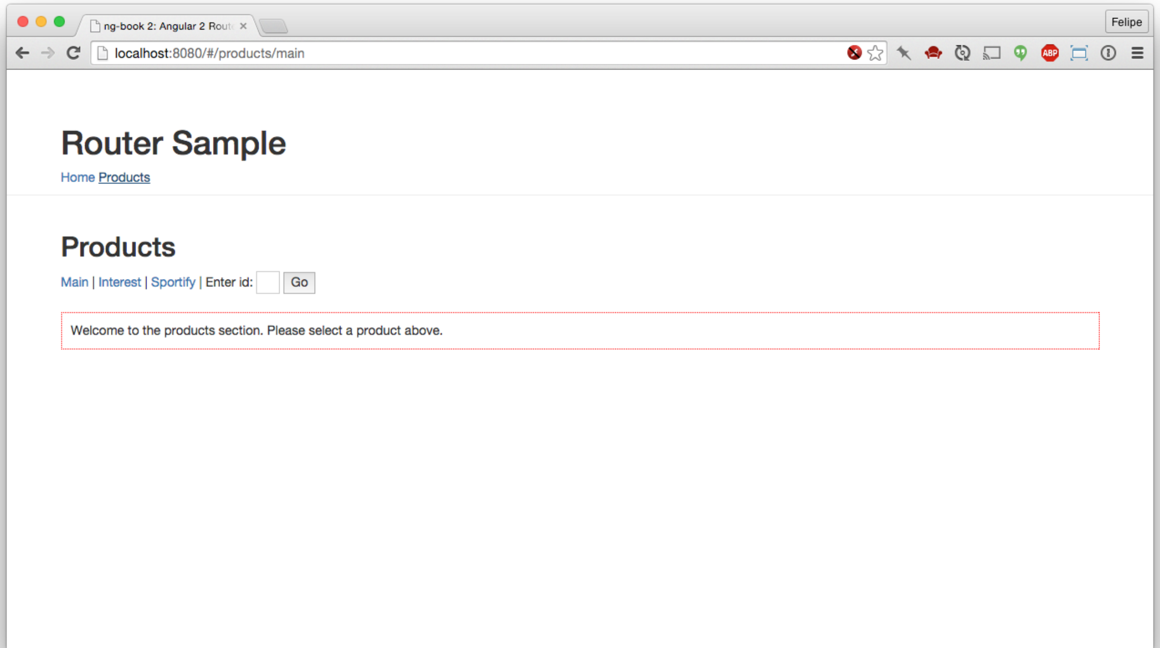
Notice that we use the relative `./` path in the `navigate` function. In order to use this we also pass the `relativeTo` object to the options, which tells the router what that route is relative to.

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:

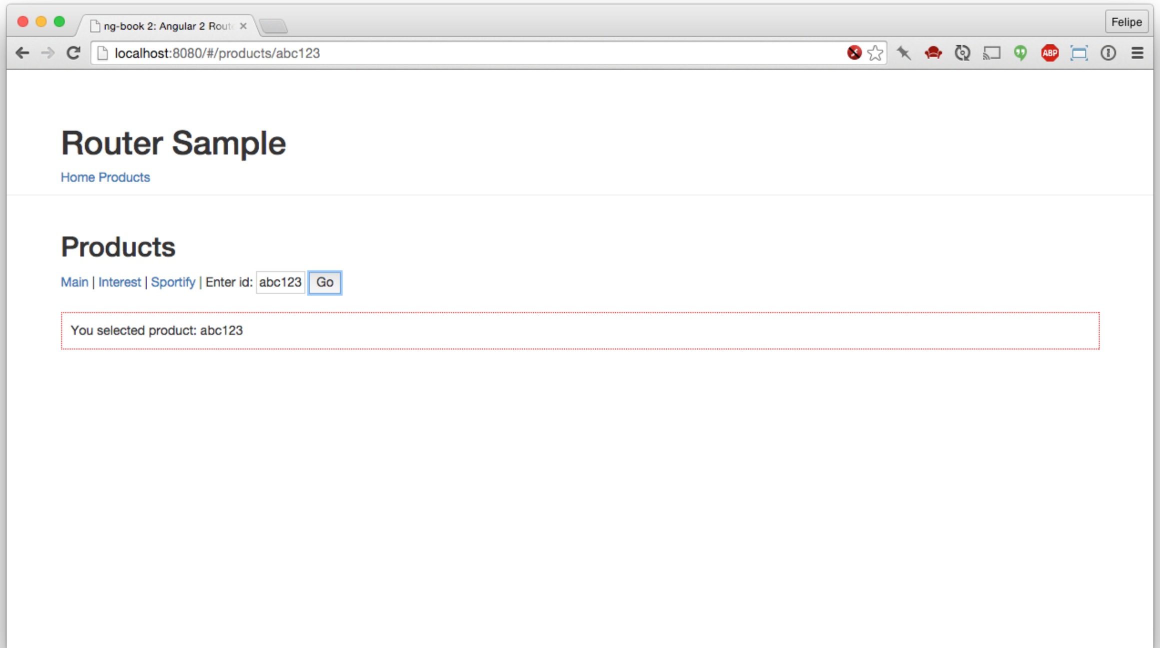


Nested Routes App - Products Section

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 enter 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/spotify`) over the parameterized ones (like `/products/123`). This way `/products/spotify` 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!

Data Architecture in Angular

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
- Indexdb
- 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 used[70] 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**[71]: 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.

[70]See: Model View Whatever
[71]https://facebook.github.io/flux/

- **Observables**: Observables give us streams of data. We subscribe to the streams and then perform operations to react to changes. RxJs[72] 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[73] is a powerful framework that helps bind your client-side models to the server-side data. Falcor is often used with an Observables-type data architecture.

Data Architecture in Angular

Angular 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.

[72] https://github.com/Reactive-Extensions/RxJS
[73] http://netflix.github.io/falcor/

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[74] 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[75].

Learning Reactive Programming and RxJS

If you're just learning RxJS I recommend that you read this article first:

[74] http://underscorejs.org/
[75] http://underscorejs.org/

- The introduction to Reactive Programming you've been missing[76] 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?[77]
- Which instance operators to use on streams?[78]
- RxMarbles[79] - 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?** - No, you definitely don't. Observables are just one pattern out of many that you can use with Angular. We talk more about other data patterns you can use here.

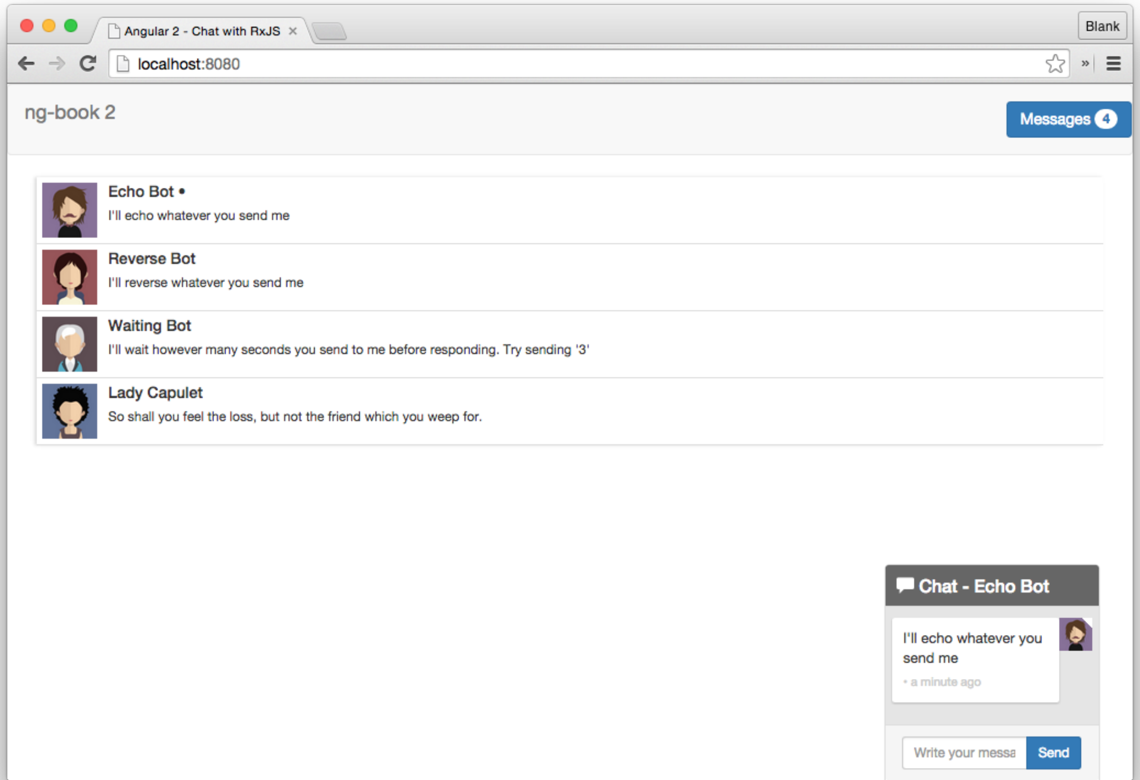
I want to give you fair warning: learning RxJS can be a bit mind-bending at first. But trust me, you'll get the hang of it and it's worth it. Here's a few big ideas about streams that you might find helpful:

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:

[76] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
[77] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md
[78] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md
[79] http://rxmarbles.com



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/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:

```
cd code/rxjs/rxjs-chat
npm install
npm start
```

Now open your browser to `http://localhost:4200`.

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

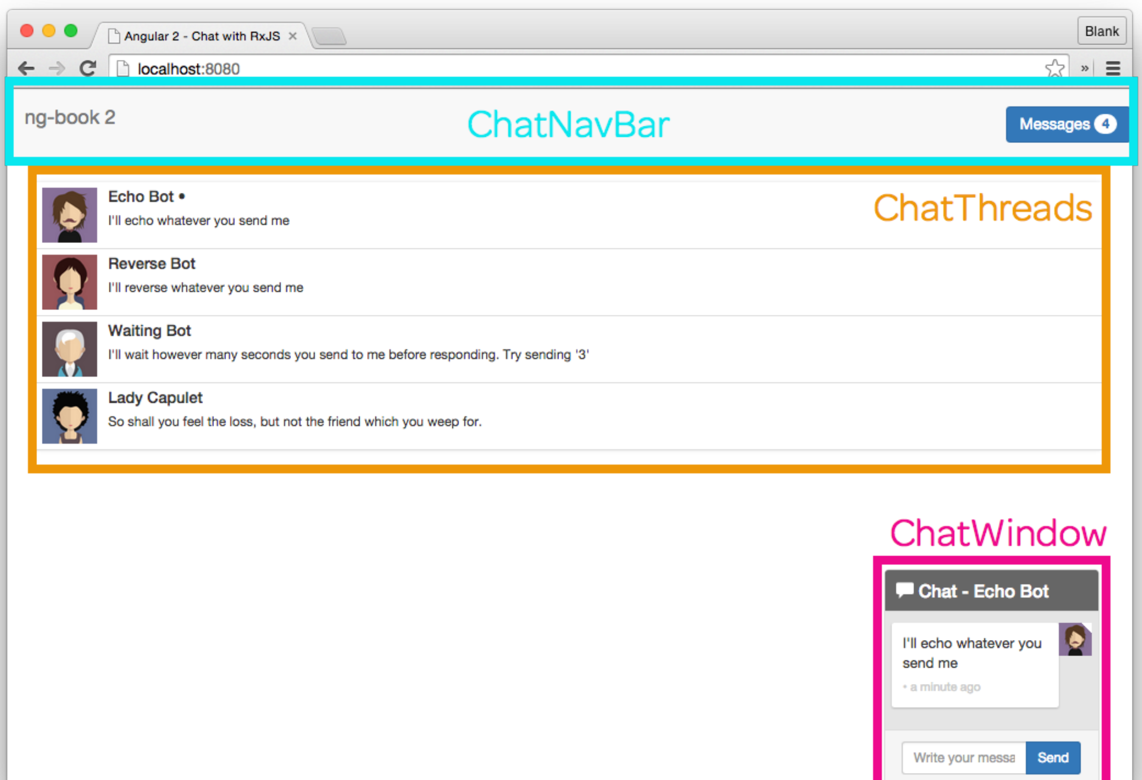
Let's look at an overview of how this app is constructed. We have

- 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:



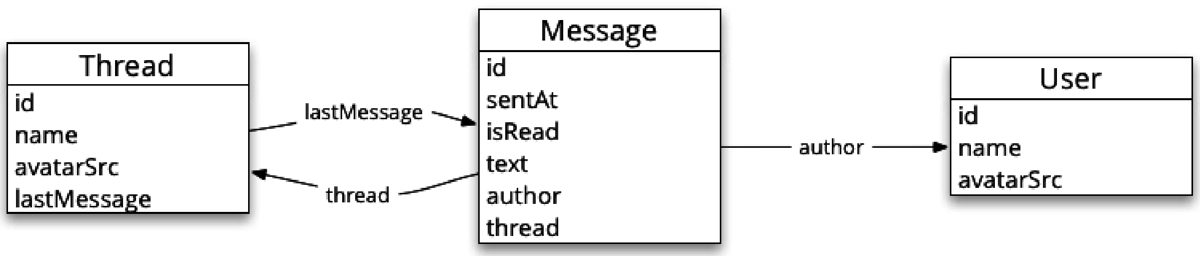
Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count

- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - 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 `UsersService`:

- 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 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/rxjs-chat/src/app/user/user.model.ts

```
import { uuid } from '../util/uuid';

/**
 * A User represents an agent that sends messages
 */
export class User {
  id: string;

  constructor(public name: string,
              public avatarSrc: string) {
    this.id = uuid();
  }
}
```

> 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/rxjs-chat/src/app/thread/thread.model.ts

```
import { Message } from '../message/message.model';
import { uuid } from '../util/uuid';

/**
 * Thread represents a group of Users exchanging Messages
 */
 export class Thread {
   id: string;
   lastMessage: Message;
   name: string;
   avatarSrc: string;

   constructor(id?: string,
               name?: string,
               avatarSrc?: string) {
     this.id = id || uuid();
     this.name = name;
     this.avatarSrc = avatarSrc;
   }
 }
```

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/rxjs-chat/src/app/message/message.model.ts

```
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';
import { uuid } from './../util/uuid';

/**
 * Message represents one message being sent in a Thread
 */
export class Message {
  id: string;
  sentAt: Date;
  isRead: boolean;
  author: User;
  text: string;
  thread: Thread;

  constructor(obj?: any) {
    this.id              = obj && obj.id              || uuid();
    this.isRead          = obj && obj.isRead          || false;
    this.sentAt          = obj && obj.sentAt          || new Date();
    this.author          = obj && obj.author          || null;
    this.text            = obj && obj.text            || null;
    this.thread          = obj && obj.thread          || null;
  }
}
```

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:

```
let msg1 = new Message();

# or this

let msg2 = new Message({
  text: "Hello Nate Murray!"
})
```

Now that we've looked at our models, let's take a look at our first service: the `UsersService`.

Implementing `UsersService`

The point of the `UsersService` 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 add the `@Injectable` decorator.

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
export class UsersService {
  // `currentUser` contains the current user
  currentUser: Subject<User> = new BehaviorSubject<User>(null);

  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
}
```

We make a class that we will be able to use 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, but the result is that we can now inject other dependencies into our constructor like so:

```
class UsersService {
  constructor(public someOtherService: SomeOtherService) {
    // do something with `someOtherService` here
  }
}
```

`currentUser` stream

Next we setup a stream which we will use to manage our current user:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
  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`[80] inherits from both `Observable`[81] and `Observer`[82]

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`[83] 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 `UsersService.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 `UsersService` simply publish a new `User` directly to the stream like this:

[80] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/subject.md
[81] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md
[82] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md
[83] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md

```
UsersService.currentUser.subscribe((newUser) => {
  console.log('New User is: ', newUser.name);
})

// => New User is: originalUserName

let u = new User('Nate', 'anImgSrc');
UsersService.currentUser.next(u);

// => 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 `UsersService` like this:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
```

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 `UsersService` 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.

`UsersService.ts`

Putting it together, our `UsersService` looks like this:

code/rxjs/rxjs-chat/src/app/user/users.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject } from 'rxjs';
import { User } from './user.model';


/**
 * UserService manages our current user
 */
@Injectable()
export class UsersService {
  // `currentUser` contains the current user
  currentUser: Subject<User> = new BehaviorSubject<User>(null);

  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
}

export const userServiceInjectables: Array<any> = [
  UsersService
];
```

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 `UsersService`. 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/rxjs-chat/src/app/message/messages.service.ts

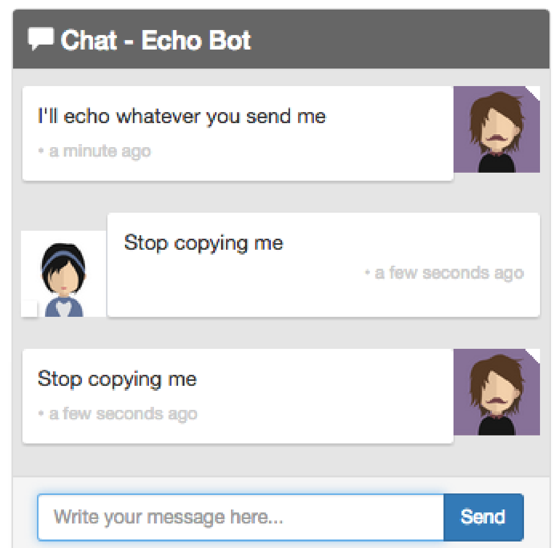
```
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();
```

If we want, we can define a helper method to add `Messages` to this stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
  addMessage(message: Message): void {
    this.newMessages.next(message);
  }
```

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/rxjs-chat/src/app/message/messages.service.ts

```
  messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
    return this.newMessages
      .filter((message: Message) => {
               // belongs to this thread
        return (message.thread.id === thread.id) &&
               // and isn't authored by this user
               (message.author.id !== user.id);
      });
  }
```

`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/rxjs-chat/src/app/message/messages.service.ts

```
  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/rxjs-chat/src/app/message/messages.service.ts

```
interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}
```

Let's define our `updates` stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  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 `MessagesService`) like this:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
```

This code introduces a new stream function: `scan`[84]. 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[]`.

[84] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md

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`[85] and `replay`[86])
- `refCount`[87] - 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`[88]
- Introduction to Rx: Hot and Cold observables[89]
- RefCount Marble Diagram[90]

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .publishReplay(1)
      .refCount();
```

[85]https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/publish.md
[86]https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/replay.md
[87]https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md
[88]https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md
[89]http://www.introtorx.com/Content/v1.0.10621.0/14_HotAndColdObservables.html#RefCount
[90]http://reactivex.io/documentation/operators/refcount.html

Adding `Message`s to the `messages` Stream

Now we could add a `Message` to the `messages` stream like so:

```
var myMessage = new Message(/* params here... */);

updates.next( (messages: Message[]): Message[] => {
  return messages.concat(myMessage);
})
```

Above, we're adding an operation to the `updates` stream. The effect is that `messages` is "subscribed" 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:

```
addMessage(newMessage: Message) {
  updates.next( (messages: Message[]): Message[] => {
    return messages.concat(newMessage);
  })
}

// somewhere else

var myMessage = new Message(/* params here... */);
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/rxjs-chat/src/app/message/messages.service.ts

```
  // action streams
  create: Subject<Message> = new Subject<Message>();
```

Next, in our `constructor` we configure the `create` stream:

code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
```

The `map`[91] 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`[92].

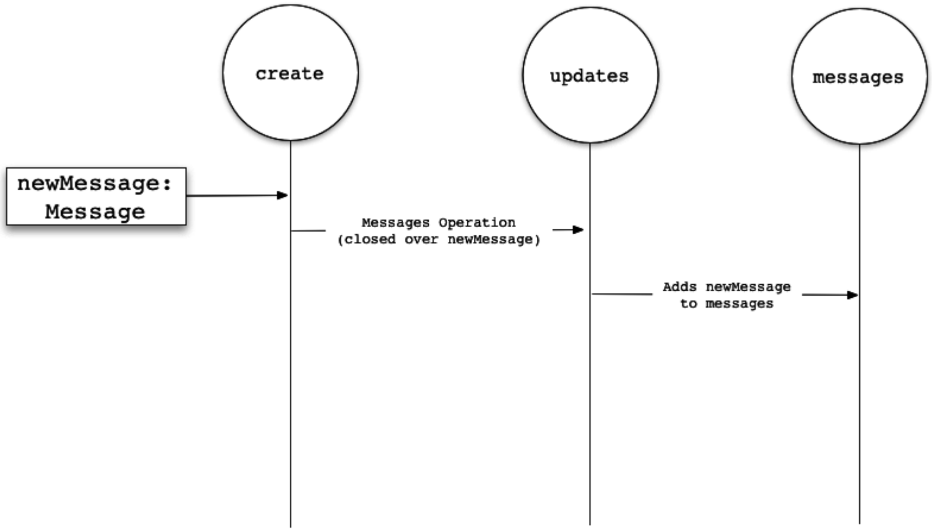
code/rxjs/rxjs-chat/src/app/message/messages.service.ts

```
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
      .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:

[91] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/select.md
[92] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/subscribe.md



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`)

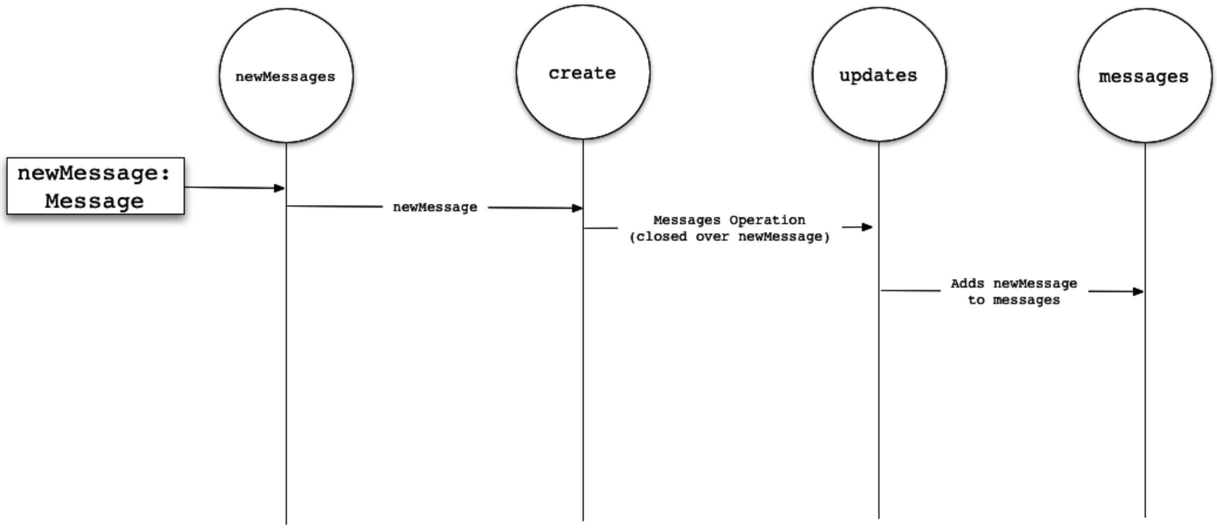
Anywhere in our code, if we want to get the most current list of messages, we just have to go to the `messages` stream. But we have a problem, **we still haven't connected this flow to the `newMessages` stream.**

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/rxjs-chat/src/app/message/messages.service.ts

```
    this.newMessages
      .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/rxjs-chat/src/app/message/messages.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, Observable } from 'rxjs';
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';
import { Message } from '../message/message.model';

const initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();

  // `messages` is a stream that emits an array of the most up to date messages
  messages: Observable<Message[]>;

  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  updates: Subject<any> = new Subject<any>();

  // action streams
  create: Subject<Message> = new Subject<Message>();
  markThreadAsRead: Subject<any> = new Subject<any>();

  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .publishReplay(1)
      .refCount();
```

```

    // `create` takes a Message and then puts an operation (the inner function)
    // on the `updates` stream to add the Message to the list of messages.
    //
    // That is, for each item that gets added to `create` (by using `next`)
    // this stream emits a concat operation function.
    //
    // Next we subscribe `this.updates` to listen to this stream, which means
    // that it will receive each operation that is created
    //
    // Note that it would be perfectly acceptable to simply modify the
    // "addMessage" function below to simply add the inner operation function to
    // the update stream directly and get rid of this extra action stream
    // entirely. The pros are that it is potentially clearer. The cons are that
    // the stream is no longer composable.
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
      .subscribe(this.updates);

    this.newMessages
      .subscribe(this.create);

    // similarly, `markThreadAsRead` takes a Thread and then puts an operation
    // on the `updates` stream to mark the Messages as read
    this.markThreadAsRead
      .map( (thread: Thread) => {
        return (messages: Message[]) => {
          return messages.map( (message: Message) => {
            // note that we're manipulating `message` directly here. Mutability
            // can be confusing and there are lots of reasons why you might want
            // to, say, copy the Message object or some other 'immutable' here
            if (message.thread.id === thread.id) {
              message.isRead = true;
            }
            return message;
          });
        };
      })
      .subscribe(this.updates);
```

```

  }

  // an imperative function call to this action stream
  addMessage(message: Message): void {
    this.newMessages.next(message);
  }

  messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
    return this.newMessages
      .filter((message: Message) => {
               // belongs to this thread
        return (message.thread.id === thread.id) &&
               // and isn't authored by this user
               (message.author.id !== user.id);
      });
  }
}

export const messagesServiceInjectables: Array<any> = [
  MessagesService
];
```

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 `code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts`.

> To run the tests in this project, open up your terminal then:
>
> ```
> cd /path/to/code/rxjs/rxjs-chat // <-- your path will vary
> npm install
> npm run test
> ```

Let's start by creating a few instances of our models to use:

code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts

```
import { MessagesService } from './messages.service';

import { Message } from './message.model';
import { Thread } from './../thread/thread.model';
import { User } from './../user/user.model';

describe('MessagesService', () => {
  it('should test', () => {

    const user: User = new User('Nate', '');
    const thread: Thread = new Thread('t1', 'Nate', '');
    const m1: Message = new Message({
      author: user,
      text: 'Hi!',
      thread: thread
    });

    const m2: Message = new Message({
      author: user,
      text: 'Bye!',
      thread: thread
    });
```

Next let's subscribe to a couple of our streams:

code/rxjs/rxjs-chat/src/app/message/messages.service.spec.ts

```
    const messagesService: MessagesService = new MessagesService();

    // listen to each message indivdually as it comes in
    messagesService.newMessages
      .subscribe( (message: Message) => {
        console.log('=> newMessages: ' + message.text);
      });

    // listen to the stream of most current messages
    messagesService.messages
      .subscribe( (messages: Message[]) => {
        console.log('=> messages: ' + messages.length);
      });

    messagesService.addMessage(m1);
```

```
    messagesService.addMessage(m2);

    // => messages: 1
    // => newMessages: Hi!
    // => messages: 2
    // => newMessages: Bye!
  });


});
```

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 `MessagesService` 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 `MessagesService` 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` were 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/rxjs-chat/src/app/thread/threads.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject, Observable } from 'rxjs/Rx';
import { Thread } from './thread.model';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';
import * as _ from 'lodash';

@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list of threads
  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/rxjs-chat/src/app/thread/threads.service.ts

```
  threads: Observable<{ [key: string]: Thread }>;
```

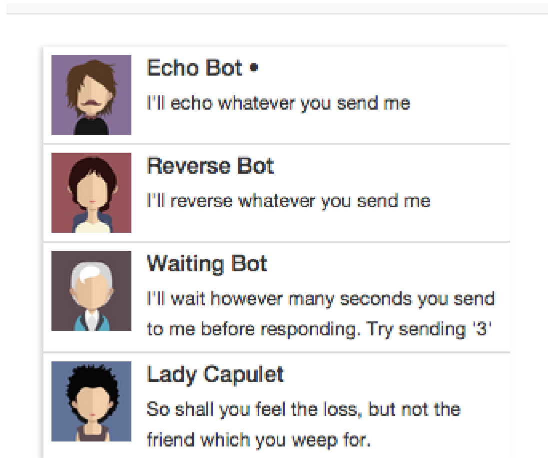
Recall that each time a new `Message` is added to the steam, `messages` will emit an array of the current `Messages`. We're going to look at each `Message` and we want to return a unique list of the `Threads`.

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        const threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            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/rxjs-chat/src/app/thread/threads.service.ts

```
   // Cache the most recent message for each thread
   const messagesThread: Thread = threads[message.thread.id];
   if (!messagesThread.lastMessage ||
     messagesThread.lastMessage.sentAt < message.sentAt) {
    messagesThread.lastMessage = message;
   }
  });
  return threads;
 });
```

Putting it all together, `threads` looks like this:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
this.threads = messagesService.messages
 .map( (messages: Message[]) => {
  const threads: {[key: string]: Thread} = {};
  // Store the message's thread in our accumulator `threads`
  messages.map((message: Message) => {
   threads[message.thread.id] = threads[message.thread.id] ||
    message.thread;

   // Cache the most recent message for each thread
   const messagesThread: Thread = threads[message.thread.id];
```

```
        if (!messagesThread.lastMessage ||
            messagesThread.lastMessage.sentAt < message.sentAt) {
          messagesThread.lastMessage = message;
        }
      });
      return threads;
    });
```

Trying out the `ThreadsService`

Let's try out our `ThreadsService`. First we'll create a few models to work with:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
import { Message } from './../message/message.model';
import { Thread } from './thread.model';
import { User } from './../user/user.model';

import { ThreadsService } from './threads.service';
import { MessagesService } from './../message/messages.service';
import * as _ from 'lodash';

describe('ThreadsService', () => {
  it('should collect the Threads from Messages', () => {

    const nate: User = new User('Nate Murray', '');
    const felipe: User = new User('Felipe Coury', '');

    const t1: Thread = new Thread('t1', 'Thread 1', '');
    const t2: Thread = new Thread('t2', 'Thread 2', '');

    const m1: Message = new Message({
      author: nate,
      text: 'Hi!',
      thread: t1
    });

    const m2: Message = new Message({
      author: felipe,
      text: 'Where did you get that hat?',
      thread: t1
    });

```

```
    const m3: Message = new Message({
      author: nate,
      text: 'Did you bring the briefcase?',
      thread: t2
    });
```

Now let's create an instance of our services:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
    const messagesService: MessagesService = new MessagesService();
    const threadsService: ThreadsService = new ThreadsService(messagesService);
```

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.

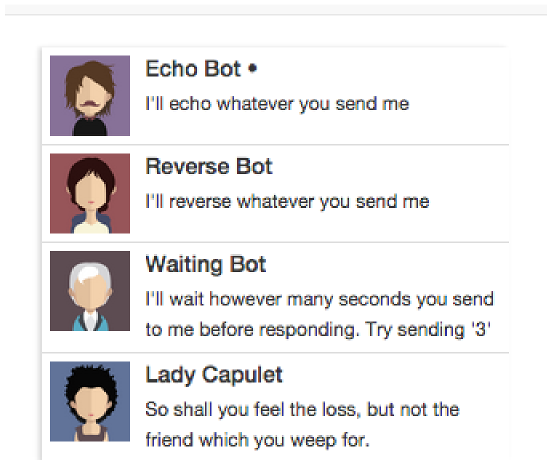
Let's subscribe to `threads` and log out what comes through:

code/rxjs/rxjs-chat/src/app/thread/threads.service.spec.ts

```
    const threadsService: ThreadsService = new ThreadsService(messagesService);

    threadsService.threads
      .subscribe( (threadIdx: { [key: string]: Thread }) => {
        const threads: Thread[] = _.values(threadIdx);
        const threadNames: string = _.map(threads, (t: Thread) => t.name)
                                .join(', ');
        console.log(`=> threads (${threads.length}): ${threadNames} `);
      });

    messagesService.addMessage(m1);
    messagesService.addMessage(m2);
    messagesService.addMessage(m3);

    // => threads (1): Thread 1
    // => threads (1): Thread 1
    // => threads (2): Thread 1, Thread 2

  });
});
```

A chronological list of `Thread`s, 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 to 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/rxjs-chat/src/app/thread/threads.service.ts

```
  // `orderedThreads` contains a newest-first chronological list of threads
  orderedThreads: Observable<Thread[]>;
```

Next, in the `constructor` we’ll define `orderedThreads` by subscribing to `threads` and ordered by the most recent message:

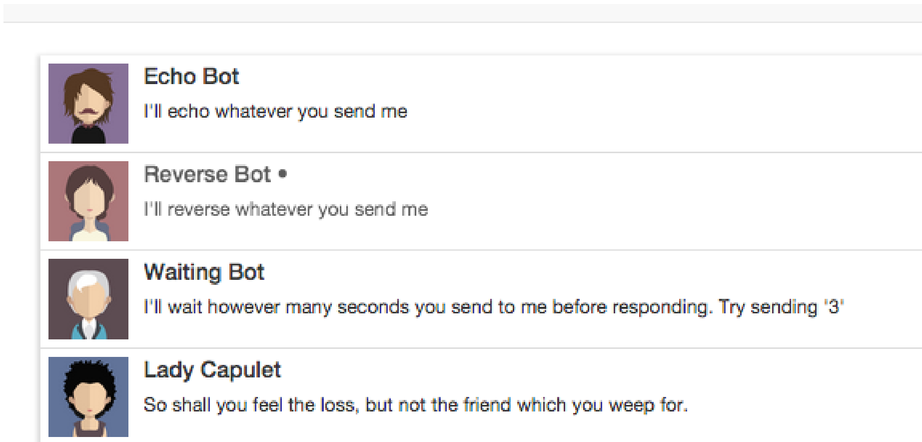
code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    this.orderedThreads = this.threads
      .map((threadGroups: { [key: string]: Thread }) => {
        const threads: Thread[] = _.values(threadGroups);
        return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
      });
```

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 dot symbol

Let's create a `BehaviorSubject` that will store the `currentThread`:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
  // `currentThread` contains the currently selected thread
  currentThread: Subject<Thread> =
    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/rxjs-chat/src/app/thread/threads.service.ts

```
  setCurrentThread(newThread: Thread): void {
    this.currentThread.next(newThread);
  }
```

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.markThreadAsRead` 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`

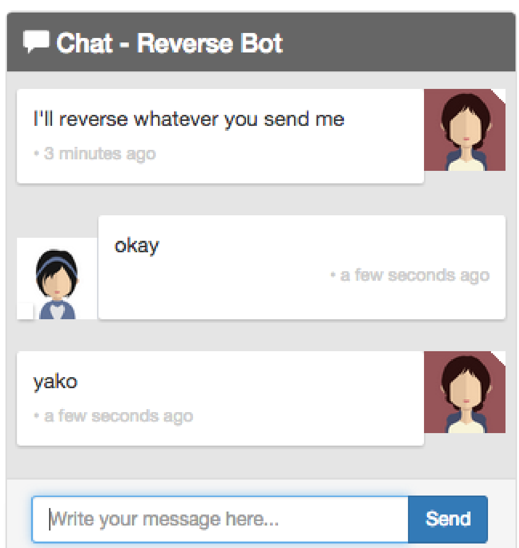
So all we need to do is hook them together:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    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:

```
var theCurrentThread: Thread;

this.currentThread.subscribe((thread: Thread) => {
  theCurrentThread = thread;
})

this.currentThreadMessages.map(
  (messages: Message[]) => {
    return _.filter(messages,
      (message: Message) => {
        return message.thread.id == theCurrentThread.id;
      })
  })
```

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`[93] operator.

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (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:

[93]https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
        if (currentThread && messages.length > 0) {
          return _.chain(messages)
            .filter((message: Message) =>
                    (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/rxjs-chat/src/app/thread/threads.service.ts

```
          return _.chain(messages)
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
            .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/rxjs-chat/src/app/thread/threads.service.ts

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
        if (currentThread && messages.length > 0) {
          return _.chain(messages)
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
```

```
            .value();
        } else {
          return [];
        }
      });
```

Our Completed `ThreadsService`

Here's what our `ThreadsService` looks like:

code/rxjs/rxjs-chat/src/app/thread/threads.service.ts

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject, Observable } from 'rxjs/Rx';
import { Thread } from './thread.model';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';
import * as _ from 'lodash';

@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list of threads
  threads: Observable<{ [key: string]: Thread }>;

  // `orderedThreads` contains a newest-first chronological list of threads
  orderedThreads: Observable<Thread[]>;

  // `currentThread` contains the currently selected thread
  currentThread: Subject<Thread> =
    new BehaviorSubject<Thread>(new Thread());

  // `currentThreadMessages` contains the set of messages for the currently
  // selected thread
  currentThreadMessages: Observable<Message[]>;

  constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        const threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
```

```
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;

          // Cache the most recent message for each thread
          const messagesThread: Thread = threads[message.thread.id];
          if (!messagesThread.lastMessage ||
              messagesThread.lastMessage.sentAt < message.sentAt) {
            messagesThread.lastMessage = message;
          }
        });
        return threads;
      });

    this.orderedThreads = this.threads
      .map((threadGroups: { [key: string]: Thread }) => {
        const threads: Thread[] = _.values(threadGroups);
        return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
      });

    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
        if (currentThread && messages.length > 0) {
          return _.chain(messages)
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
            .value();
        } else {
          return [];
        }
      });

    this.currentThread.subscribe(this.messagesService.markThreadAsRead);
  }

  setCurrentThread(newThread: Thread): void {
    this.currentThread.next(newThread);
  }

```

```
}

export const threadsServiceInjectables: Array<any> = [
  ThreadsService
];
```

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

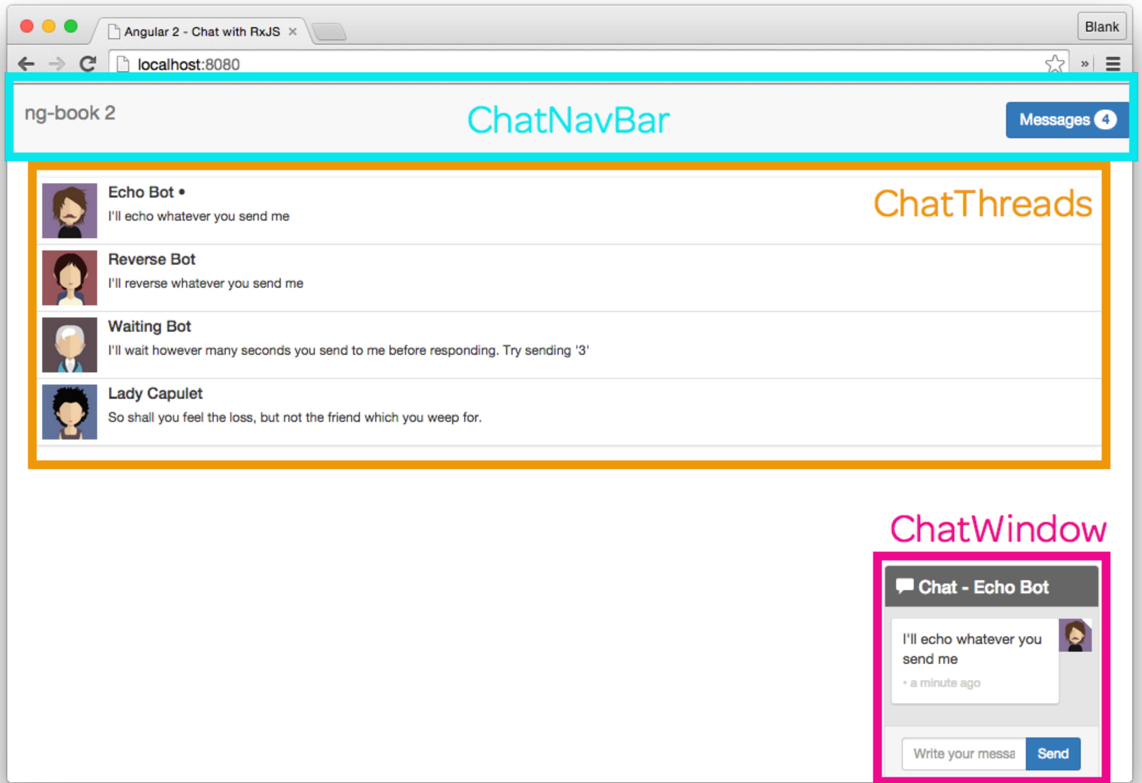
Building Our Views: The `AppComponent` Top-Level Component

Let's turn our attention to our app and implement our view components.

> 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 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:



Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count
- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - shows the messages in the current thread with an input box to send new messages

Here's what our top-level component looks like in code:

code/rxjs/rxjs-chat/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import { ChatExampleData } from './data/chat-example-data';

import { UsersService } from './user/users.service';
import { ThreadsService } from './thread/threads.service';
import { MessagesService } from './message/messages.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public usersService: UsersService) {
    ChatExampleData.init(messagesService, threadsService, usersService);
  }
}
```

and the template:

code/rxjs/rxjs-chat/src/app/app.component.html

```
<div>
  <chat-page></chat-page>
</div>
```

In this chapter we are adding some style using the CSS framework Bootstrap[94]

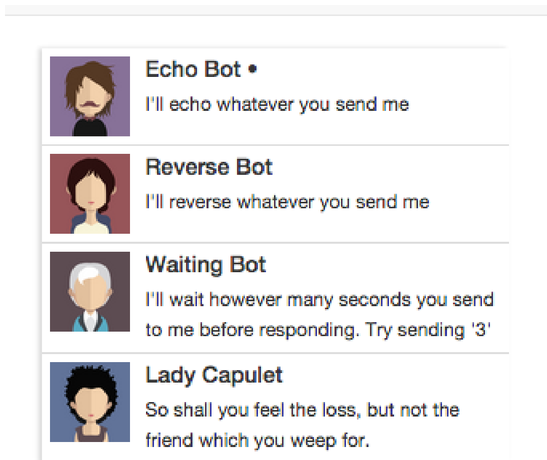
Take a look at the `constructor`. Here we're injecting our three services: the `MessagesService`, `ThreadsService`, and `UsersService`. We're using those services to initialize our example data.

If you're interested in the example data you can find it in `code/rxjs/rxjs-chat/src/app/data/chat-example-data.ts`.

We'll build our `chat-page` in a moment, but first let's build our thread list in the `ChatThreadsComponent`.

[94] http://getbootstrap.com

The `ChatThreadsComponent`



Time Ordered List of Threads

code/rxjs/rxjs-chat/src/app/chat-threads/chat-threads.component.ts

```
import {
  Component,
  OnInit,
  Inject
} from '@angular/core';
import { Observable } from 'rxjs';
import { Thread } from '../thread/thread.model';
import { ThreadsService } from './../thread/threads.service';

@Component({
  selector: 'chat-threads',
  templateUrl: './chat-threads.component.html',
  styleUrls: ['./chat-threads.component.css']
})
export class ChatThreadsComponent {
  threads: Observable<any>;

  constructor(public threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

Here we're injecting `ThreadsService` and then we're keeping a reference to the `orderedThreads` .

`ChatThreadsComponent template`

Lastly, let's look at the `template` and its configuration:

code/rxjs/rxjs-chat/src/app/chat-threads/chat-threads.component.html

```
<!-- conversations -->
<div class="row">
  <div class="conversation-wrap">

    <chat-thread
         *ngFor="let thread of threads | async"
         [thread]="thread">
    </chat-thread>

  </div>
</div>
```

There's three things to look at here: `NgFor` with the `async` pipe, the `ChangeDetectionStrategy` and `ChatThreadComponent`.

The `ChatThreadComponent` 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 `ChatThreadComponent` 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 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.

> We talk more about various change-detection strategies in the Advanced Components Chapter

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.

The Single `ChatThreadComponent`

Let's look at our `ChatThreadComponent`. This is the component that will be used to display a **single thread**. Starting with the `@Component`:

code/rxjs/rxjs-chat/src/app/chat-thread/chat-thread.component.ts

```
import {
  Component,
  OnInit,
  Input,
  Output,
  EventEmitter
} from '@angular/core';
import { Observable } from 'rxjs';
import { ThreadsService } from './../thread/threads.service';
import { Thread } from '../thread/thread.model';

@Component({
  selector: 'chat-thread',
  templateUrl: './chat-thread.component.html',
  styleUrls: ['./chat-thread.component.css']
})
export class ChatThreadComponent implements OnInit {
  @Input() thread: Thread;
  selected = false;

  constructor(public threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```

We'll come back and look at the `template` in a minute, but first let's look at the component definition controller.

`ChatThreadComponent` Controller and `ngOnInit`

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 Advanced Components chapter.

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.

`ChatThreadComponent` template

Here's the code for our template:

code/rxjs/rxjs-chat/src/app/chat-thread/chat-thread.component.html

```
<div class="media conversation">
  <div class="pull-left">
    <img class="media-object avatar"
         src="{{thread.avatarSrc}}">
  </div>
  <div class="media-body">
    <h5 class="media-heading contact-name">{{thread.name}}
      <span *ngIf="selected">&bull;</span>
    </h5>
    <small class="message-preview">{{thread.lastMessage.text}}</small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
```

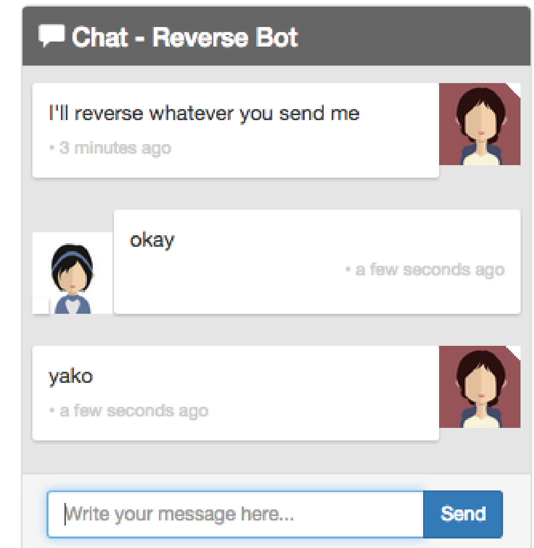
Notice we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`, and `{{thread.lastMessage.text}}`.

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.

The `ChatWindowComponent`

The `ChatWindowComponent` 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/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
@Component({
  selector: 'chat-window',
  templateUrl: './chat-window.component.html',
  styleUrls: ['./chat-window.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
```

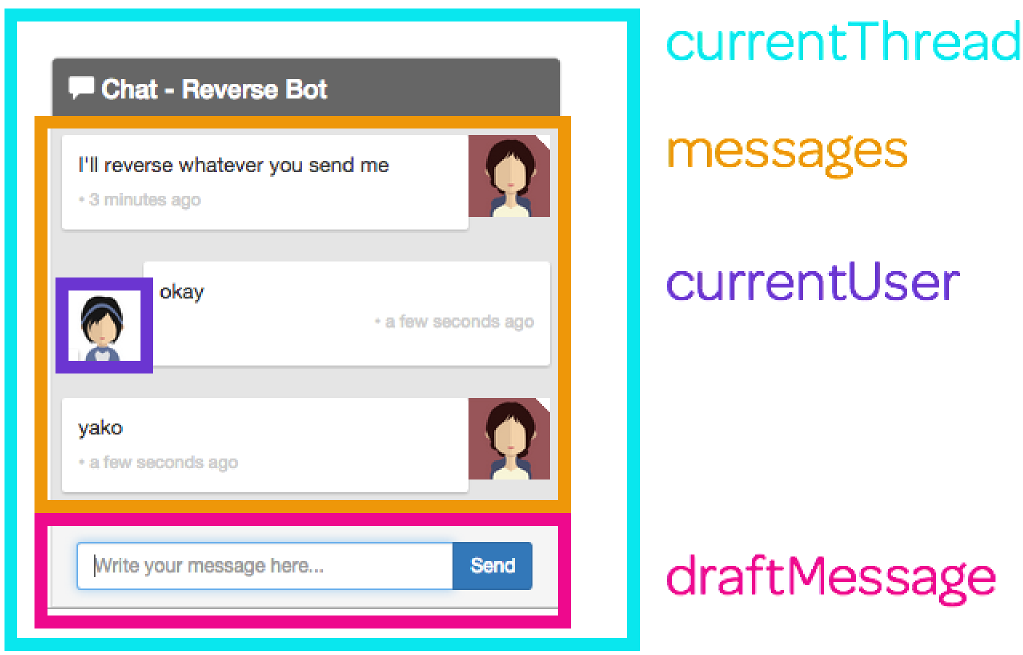
`ChatWindowComponent` Class Properties

Our `ChatWindowComponent` class has four properties :

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
export class ChatWindowComponent implements OnInit {
  messages: Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;
```

Here's a diagram of where each one is used:



Chat Window Properties

In our `constructor` we're going to inject four things:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public UsersService: UsersService,
              public el: ElementRef) {
  }
```

The first three are our services. The fourth, `el` 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`

`ChatWindowComponent 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/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
  ngOnInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    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/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
    this.threadsService.currentThread.subscribe(
      (thread: Thread) => {
        this.currentThread = thread;
      });
```

We also want new messages to be sent from the current user, so let's do the same with `currentUser`:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
    this.UsersService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
        });
```

`ChatWindowComponent sendMessage`

Since we're talking about it, let's implement a `sendMessage` function that will send a new message:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
  sendMessage(): void {
    const m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
  }
```

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.

`ChatWindowComponent` `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/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }
```

`ChatWindowComponent` `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/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
  scrollToBottom(): void {
    const scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }
```

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 anytime we get a new message:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
    this.messages
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
  }
```

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.

`ChatWindowComponent` template

The opening of our `template` should look familiar, we start by defining some markup and the panel header:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.html

```
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">

        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment"></span>
              Chat - {{currentThread.name}}
            </h3>
          </div>
          <div class="panel-buttons-container">
            <!-- you could put minimize or close buttons here -->
          </div>
        </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/rxjs-chat/src/app/chat-window/chat-window.component.html

```
        <div class="panel-body msg-container-base">
          <chat-message
          *ngFor="let message of messages | async"
          [message]="message">
        </chat-message>
      </div>
```

Lastly we have the message input box and closing tags :

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.html

```
      <div class="panel-footer">
        <div class="input-group">
          <input type="text"
          class="chat-input"
          placeholder="Write your message here..."
          (keydown.enter)="onEnter($event)"
          [(ngModel)]="draftMessage.text" />
          <span class="input-group-btn">
            <button class="btn-chat"
```

```
            (click)="onEnter($event)"
            >Send</button>
          </span>
        </div>
      </div>

    </div>
  </div>
</div>
```

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, on the `input` tag above, 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)`.

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/rxjs-chat/src/app/chat-window/chat-window.component.html

```
          <input type="text"
          class="chat-input"
          placeholder="Write your message here..."
          (keydown.enter)="onEnter($event)"
          [(ngModel)]="draftMessage.text" />
```

Clicking "Send"

On our "Send" button we bind the `(click)` property to the `onEnter` function of our component:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.html

```
          <span class="input-group-btn">
            <button class="btn-chat"
            (click)="onEnter($event)"
            >Send</button>
          </span>
```

The Entire `ChatWindowComponent`

We broke that up into a lot of tiny pieces. So that we can get a view of the whole thing, here's the code listing for the entire `ChatWindowComponent`:

code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.ts

```
import {
  Component,
  Inject,
  ElementRef,
  OnInit,
  ChangeDetectionStrategy
} from '@angular/core';
import { Observable } from 'rxjs';

import { User } from '../user/user.model';
import { UsersService } from '../user/users.service';
import { Thread } from '../thread/thread.model';
import { ThreadsService } from '../thread/threads.service';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';

@Component({
  selector: 'chat-window',
  templateUrl: './chat-window.component.html',
  styleUrls: ['./chat-window.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChatWindowComponent implements OnInit {
  messages: Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;

```

```
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public UsersService: UsersService,
              public el: ElementRef) {
  }

  ngOnInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    this.draftMessage = new Message();

    this.threadsService.currentThread.subscribe(
      (thread: Thread) => {
        this.currentThread = thread;
      });

    this.UsersService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
        });

    this.messages
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
  }

  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }

  sendMessage(): void {
    const m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
```

```
  }

  scrollToBottom(): void {
    const scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }
}
```

and template:

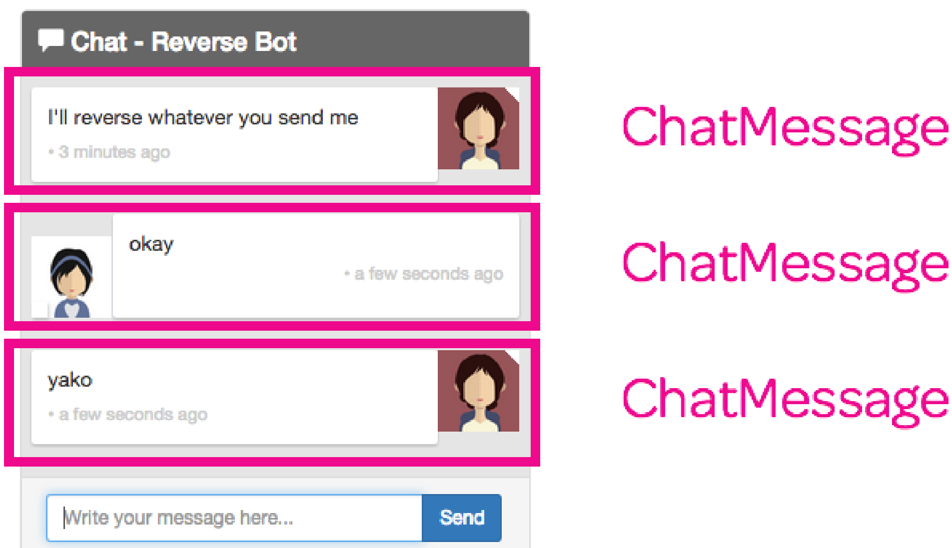
code/rxjs/rxjs-chat/src/app/chat-window/chat-window.component.html

```html
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">

        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment"></span>
              Chat - {{currentThread.name}}
            </h3>
          </div>
          <div class="panel-buttons-container">
            <!-- you could put minimize or close buttons here -->
          </div>
        </div>

        <div class="panel-body msg-container-base">
          <chat-message
          *ngFor="let message of messages | async"
          [message]="message">
        </chat-message>
      </div>

      <div class="panel-footer">
        <div class="input-group">
          <input type="text"
          class="chat-input"
          placeholder="Write your message here..."
          (keydown.enter)="onEnter($event)"
          [(ngModel)]="draftMessage.text" />
```

```
          <span class="input-group-btn">
            <button class="btn-chat"
            (click)="onEnter($event)"
            >Send</button>
          </span>
        </div>
      </div>

    </div>
  </div>
</div>
```

The `ChatMessageComponent`

Each `Message` is rendered by the `ChatMessageComponent`.



The `ChatMessageComponent`

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`.

Remember that each `ChatMessageComponent` 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:

We start by defining the `@Component`

code/rxjs/rxjs-chat/src/app/chat-message/chat-message.component.ts

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

import { UsersService } from './../user/users.service';
import { ThreadsService } from './../thread/threads.service';
import { MessagesService } from './../message/messages.service';

import { Message } from './../message/message.model';
import { Thread } from './../thread/thread.model';
import { User } from './../user/user.model';

@Component({
  selector: 'chat-message',
  templateUrl: './chat-message.component.html',
  styleUrls: ['./chat-message.component.css']
})
export class ChatMessageComponent implements OnInit {
  @Input() message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(public UsersService: UsersService) {
  }

  ngOnInit(): void {
    this.UsersService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
        });
  }
}
```

The `ChatMessageComponent` `template`

In our `template` we have two interesting ideas:

1. the `FromNowPipe`
2. `[ngClass]`

First, here's the code:

code/rxjs/rxjs-chat/src/app/chat-message/chat-message.component.html

```
<div class="msg-container"
     [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

  <div class="avatar"
       *ngIf="!incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>

  <div class="messages"
    [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
    <p>{{message.text}}</p>
    <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
  </div>

  <div class="avatar"
       *ngIf="incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>
</div>
```

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`[95] library. If you'd like to learn about creating your own custom pipes read the source of the `FromNowPipe` in `code/rxjs/rxjs-chat/src/app/pipes/from-now.pipe.ts`

We also make extensive use of `ngClass` in this view. The idea is, when we say:

[95] http://momentjs.com/

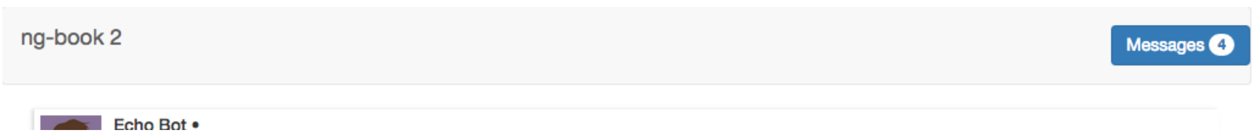
```
[ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

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 `ChatNavBarComponent`

The last component we have to talk about is the `ChatNavBarComponent`. In the nav-bar we'll show an unread messages count to the user.



The Unread Count in the `ChatNavBarComponent`

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 `ChatNavBarComponent` `@Component`

The only thing the `ChatNavBarComponent` 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/rxjs-chat/src/app/chat-nav-bar/chat-nav-bar.component.ts

```
import {
  Component,
  Inject,
  OnInit
} from '@angular/core';
import * as _ from 'lodash';

import { ThreadsService } from './../thread/threads.service';
import { MessagesService } from './../message/messages.service';

import { Thread } from './../thread/thread.model';
import { Message } from './../message/message.model';

@Component({
  selector: 'chat-nav-bar',
  templateUrl: './chat-nav-bar.component.html',
  styleUrls: ['./chat-nav-bar.component.css']
})
export class ChatNavBarComponent implements OnInit {
  unreadMessagesCount: number;

  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _.reduce(
            messages,
            (sum: number, m: Message) => {
              const messageIsInCurrentThread: boolean = m.thread &&
                currentThread &&
                (currentThread.id === m.thread.id);
              // note: in a "real" app you should also exclude
              // messages that were authored by the current user b/c they've
```

```
            // already been "read"
            if (m && !m.isRead && !messageIsInCurrentThread) {
              sum = sum + 1;
            }
            return sum;
          },
          0);
      });
  }
}
```

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 `ChatNavBarComponent` template

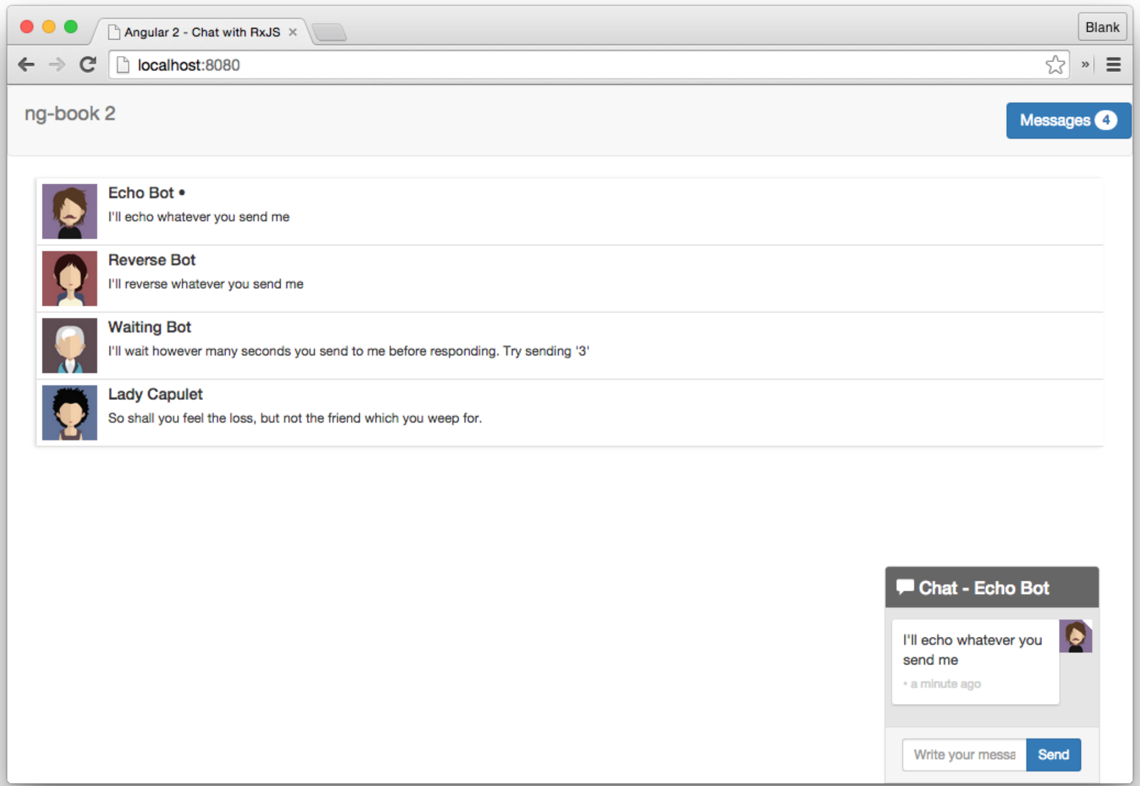
In our view, the only thing we have left to do is display our `unreadMessagesCount`:

code/rxjs/rxjs-chat/src/app/chat-nav-bar/chat-nav-bar.component.html

```html
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="https://ng-book.com/2">
        <img src="assets/images/logos/ng-book-2-minibook.png"/>
         ng-book 2
      </a>
    </div>
    <p class="navbar-text navbar-right">
      <button class="btn btn-primary" type="button">
        Messages <span class="badge">{{ unreadMessagesCount }}</span>
      </button>
    </p>
  </div>
</nav>
```

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/rxjs-chat/src/app/data/chat-example-data.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:

```
let rev: User = new User("Reverse Bot", require("images/avatars/female-avatar-4.png"\
));
let tRev: Thread = new Thread("tRev", rev.name, rev.avatarSrc);
```

code/rxjs/rxjs-chat/src/app/data/chat-example-data.ts

```
    messagesService.messagesForThreadUser(tRev, rev)
      .forEach( (message: Message): void => {
        messagesService.addMessage(
          new Message({
            author: rev,
            text: message.text.split('').reverse().join(''),
            thread: tRev
          })
        );
      },
```

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.

Introduction to Redux with TypeScript

> In this chapter and the next we'll be looking at a data-architecture pattern called Redux. In this chapter we're going to discuss the ideas behind Redux, build our own mini version, and then hook it up to Angular. In the next chapter we'll use Redux to build a bigger application.

In most of our projects so far, we've managed state in a fairly direct way: We tend to grab data from services and render them in components, passing values down the component tree along the way.

Managing our apps in this way works fine for smaller apps, but as our apps grow, having multiple components manage different parts of the state becomes cumbersome. For instance, passing all of our values down our component tree suffers from the following downsides:

Intermediate property passing - In order to get state to any component we have to pass the values down through `inputs`. This means we have many intermediate components passing state that it isn't directly using or concerned about

Inflexible refactoring - Because we're passing `inputs` down through the component tree, we're introducing a coupling between parent and child components that often isn't necessary. This makes it more difficult to put a child component somewhere else in the hierarchy because we have to change all of the new parents to pass the state

State tree and DOM tree don't match - The "shape" of our state often doesn't match the "shape" of our view/component hierarchy. By passing all data through the component tree via `props` we run into difficulties when we need to reference data in a far branch of the tree

State throughout our app - If we manage state via components, it's difficult to get a snapshot of the total state of our app. This can make it hard to know which component "owns" a particular bit of data, and which components are concerned about changes

Pulling data out of our components and into services helps a lot. At least if services are the "owners" of our data, we have a better idea of where to put things. But this opens a new question: what are the best practices for "service-owned" data? Are there any patterns we can follow? In fact, there are.

In this chapter, we're going to discuss a data-architecture pattern called *Redux* which was designed to help with these issues. We'll implement our own version of Redux which will store **all of our state in a single place**. This idea of holding **all** of our application's state in one place might sound a little crazy, but the results are surprisingly delightful.

Redux

If you haven't heard of Redux yet you can read a bit about it on the official website[96]. Web application data architecture is evolving and the traditional ways of structuring data aren't quite adequate for large web apps. Redux has been extremely popular because it's both powerful and easy to understand.

Data architecture can be a complex topic and so Redux's best feature is probably its simplicity. If you strip Redux down to the essential core, Redux is fewer than 100 lines of code.

We can build rich, easy to understand, web apps by using Redux as the backbone of our application. But first, let's walk through how to write a minimal Redux and later we'll work out patterns that emerge as we work out these ideas in a larger app.

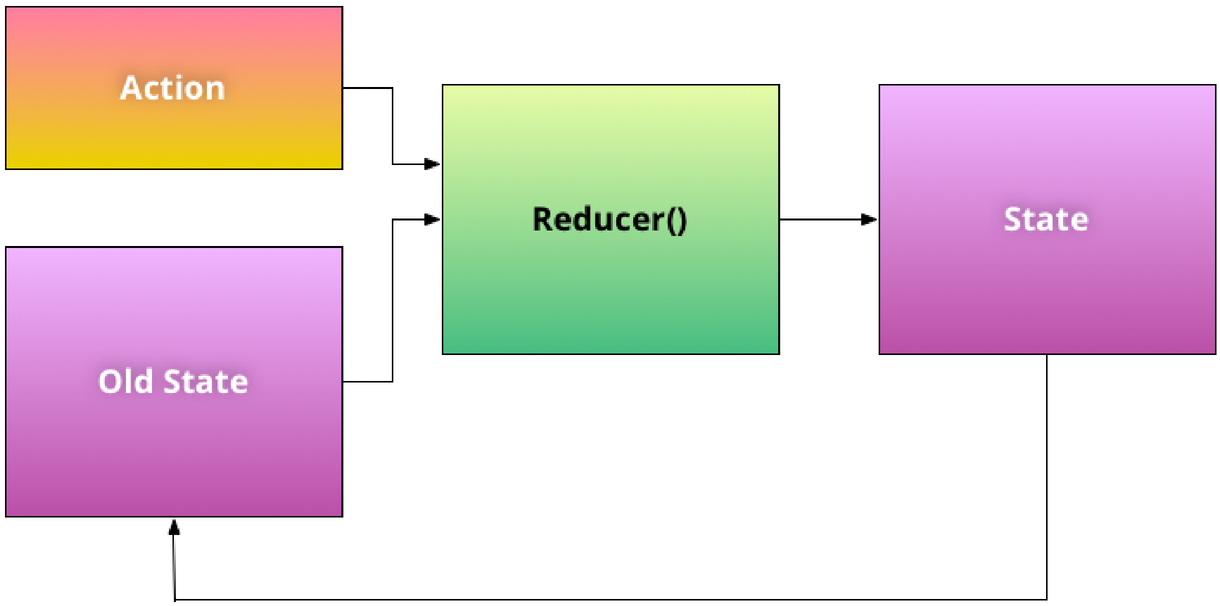
> There are several attempts to use Redux or create a Redux-inspired system that works with Angular. Two notable examples are:
>
> - ngrx/store[97] and
> - angular2-redux[98]
>
> `ngrx` is a Redux-inspired architecture that is heavily observables-based. `angular2-redux` uses Redux itself as a dependency, and adds some Angular helpers (dependency-injection, observable wrappers).
>
> Here we're not going to use either. Instead, we're going to use Redux directly in order to show the concepts without introducing a new dependency. That said, both of these libraries may be helpful to you when writing your apps.

Redux: Key Ideas

The key ideas of Redux are this:

- All of your application's data is in a single data structure called the *state* which is held in the *store*
- Your app reads the **state** from this **store**
- This **store** is never mutated directly
- User interaction (and other code) fires *actions* which describe what happened
- A *new state* is created by combining the **old state** and the **action** by a function called the *reducer*.

[96] http://redux.js.org/
[97] https://github.com/ngrx/store
[98] https://github.com/InfomediaLtd/angular2-redux



Redux Core

If the above bullet list isn't clear yet, don't worry about it - putting these ideas into practice is the goal of the rest of this chapter.

Core Redux Ideas

What's a *reducer*?

Let's talk about the *reducer* first. Here's the idea of a *reducer*: it takes the *old state* and an *action* and returns a *new state*.

A reducer must be a **pure function**[99]. That is:

1. It must not mutate the current state directly
2. It must not use any data outside of its arguments

Put another way, a pure function will always **return the same value, given the same set of arguments**. And a pure function won't call any functions which have an effect on the outside world, e.g. no database calls, no HTTP calls, and no mutating outside data structures.

Reducers should always treat the current state as **read-only**. A reducer **does not change the state** instead, it **returns a new state**. (Often this new state will start with a copy of old state, but let's not get ahead of ourselves.)

Let's define our very first reducer. Remember, there are three things involved:

[99] https://en.wikipedia.org/wiki/Pure_function

1. An `Action`, which defines what to do (with optional arguments)
2. The `state`, which stores *all* of the data in our application
3. The `Reducer` which takes the `state` and the `Action` and returns a new state.

Defining `Action` and `Reducer` Interfaces

Since we're using TypeScript we want to make sure this whole process is typed, so let's set up an interface for our `Action` and our `Reducer`:

The `Action` Interface

Our `Action` interface looks like this:

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}
```

Notice that our `Action` has two fields:

1. `type` and
2. `payload`

The `type` will be an identifying string that describes the action like `INCREMENT` or `ADD_USER`. The `payload` can be an object of any kind. The `?` on `payload?` means that this field is optional.

The `Reducer` Interface

Our `Reducer` interface looks like this:

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Reducer<T> {
  (state: T, action: Action): T;
}
```

Our `Reducer` is using a feature of TypeScript called *generics*. In this case type `T` is the type of the `state`. Notice that we're saying that a valid `Reducer` has a function which takes a `state` (of type `T`) and an `action` and returns a new `state` (also of type `T`).

Creating Our First `Reducer`

The simplest possible reducer returns the state itself. (You might call this the *identity* reducer because it applies the identity function[100] on the state. This is the default case for all reducers, as we will soon see).

[100] https://en.wikipedia.org/wiki/Identity_function

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};
```

Notice that this `Reducer` makes the generic type concrete to `number` by the syntax `Reducer<number>`. We'll define more sophisticated states beyond a single number soon.

We're not using the `Action` yet, but let's try this `Reducer` just the same.

Running the examples in this section

You can find the code for this chapter in the folder `code/redux`. If the example is runnable you will see the filename the code is from above each code box.

In this first section, these examples are run **outside of the browser and run by node.js**. Because we're using TypeScript in these examples, you should run them using the commandline tool `ts-node`, (instead of `node` directly).

You can install `ts-node` by running:

```
npm install -g ts-node
```

Or by doing an `npm install` in the `code/redux/redux-chat` directory and then calling `./node_modules/.bin/ts-node [filename]`

For instance, to run the example above you might type (not including the `$`):

```
$ cd code/redux/redux-chat/tutorial
$ npm install
$ ./node_modules/.bin/ts-node 01-identity-reducer.ts
```

Use this same procedure for the rest of the code in this chapter until we instruct you to switch to your browser.

Running Our First `Reducer`

Let's put it all together and run this reducer:

code/redux/redux-chat/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}

interface Reducer<T> {
  (state: T, action: Action): T;
}

let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};

console.log( reducer(0, null) ); // -> 0
```

And run it:

```
$ cd code/redux/redux-chat/tutorial
$ ./node_modules/.bin/ts-node 01-identity-reducer.ts
0
```

It seems almost silly to have that as a code example, but it teaches us our first principle of reducers:

By default, reducers return the original state.

In this case, we passed a state of the number `0` and a `null` action. The result from this reducer is the state `0`.

But let's do something more interesting and make our state change.

Adjusting the Counter With *actions*

Eventually our state is going to be **much more** sophisticated than a single number. We're going to be holding **all** of the data for our app in the `state`, so we'll need a better data structure for the state eventually.

That said, using a single number for the state lets us focus on other issues for now. So let's continue with the idea that our `state` is simply a single number that is storing a counter.

Let's say we want to be able to change the `state` number. Remember that in Redux we do not modify the state. Instead, we create *actions* which instruct the *reducer* on how to generate a *new state*.

Let's create an `Action` to change our counter. Remember that the only required property is a `type`. We might define our first action like this:

```
let incrementAction: Action = { type: 'INCREMENT' }
```

We should also create a second action that instructs our reducer to make the counter smaller with:

```
let decrementAction: Action = { type: 'DECREMENT' }
```

Now that we have these actions, let's try using them in our reducer:

code/redux/redux-chat/tutorial/02-adjusting-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  if (action.type === 'INCREMENT') {
    return state + 1;
  }
  if (action.type === 'DECREMENT') {
    return state - 1;
  }
  return state;
};
```

And now we can try out the whole reducer:

code/redux/redux-chat/tutorial/02-adjusting-reducer.ts

```
let incrementAction: Action = { type: 'INCREMENT' };

console.log( reducer(0, incrementAction )); // -> 1
console.log( reducer(1, incrementAction )); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };

console.log( reducer(100, decrementAction )); // -> 99
```

Neat! Now the new value of the state is returned according to which action we pass into the reducer.

Reducer `switch`

Instead of having so many `if` statements, the common practice is to convert the reducer body to a `switch` statement:

code/redux/redux-chat/tutorial/03-adjusting-reducer-switch.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  default:
    return state; // <-- dont forget!
  }
};

let incrementAction: Action = { type: 'INCREMENT' };
console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };
console.log(reducer(100, decrementAction)); // -> 99

// any other action just returns the input state
let unknownAction: Action = { type: 'UNKNOWN' };
console.log(reducer(100, unknownAction)); // -> 100
```

Notice that the `default` case of the `switch` returns the original `state`. This ensures that if an unknown action is passed in, there's no error and we get the original `state` unchanged.

Q: **Wait, all of my application state is in one giant `switch` statement?**

A: Yes and no.

If this is your first exposure to Redux reducers it might feel a little weird to have all of your application state changes be the result of a giant `switch`. There are two things you should know:

1. Having your state changes centralized in one place can help a **ton** in maintaining your program, particularly because it's easy to track down where the changes are happening when they're all together. (Furthermore, you can easily locate what state changes as the result of any action because you can search your code for the token specified for that action's `type`)
2. You can (and often do) break your reducers down into several sub-reducers which each manage a different branch of the state tree. We'll talk about this later.

Action "Arguments"

In the last example our actions contained only a `type` which told our reducer either to increment or decrement the state.

But often changes in our app can't be described by a single value - instead we need parameters to describe the change. This is why we have the `payload` field in our `Action`.

In this counter example, say we wanted to add `9` to the counter. One way to do this would be to send 9 `INCREMENT` actions, but that wouldn't be very efficient, especially if we wanted to add, say, 9000.

Instead, let's add a `PLUS` action that will use the `payload` parameter to send a number which specifies how much we want to add to the counter. Defining this action is easy enough:

```
let plusSevenAction = { type: 'PLUS', payload: 7 };
```

Next, to support this action, we add a new `case` to our reducer that will handle a `'PLUS'` action:

code/redux/redux-chat/tutorial/04-plus-action.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  case 'PLUS':
    return state + action.payload;
  default:
    return state;
  }
};
```

`PLUS` will add whatever number is in the `action.payload` to the `state`. We can try it out:

code/redux/redux-chat/tutorial/04-plus-action.ts

```
console.log( reducer(3, { type: 'PLUS', payload: 7}) );    // -> 10
console.log( reducer(3, { type: 'PLUS', payload: 9000}) ); // -> 9003
console.log( reducer(3, { type: 'PLUS', payload: -2}) );   // -> 1
```

In the first line we take the state `3` and `PLUS` a payload of `7`, which results in `10`. Neat! However, notice that while we're passing in a `state`, it doesn't really ever *change*. That is, we're not storing the result of our reducer's changes and reusing it for future actions.

Storing Our State

Our reducers are pure functions, and do not change the world around them. The problem is, in our app, things *do* change. Specifically, our state changes and we need to keep the new state somewhere.

In Redux, we keep our state in the *store*. The store has the responsibility of **running the reducer and then keeping the new state**. Let's take a look at a minimal store:

code/redux/redux-chat/tutorial/05-minimal-store.ts

```
class Store<T> {
  private _state: T;

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
  }
}
```

Notice that our `Store` is generically typed - we specify the type of the *state* with generic type `T`. We store the state in the private variable `_state`.

We also give our `Store` a `Reducer`, which is also typed to operate on `T`, the state type this is because **each store is tied to a specific reducer**. We store the `Reducer` in the private variable `reducer`.

> In Redux, we generally have 1 store and 1 top-level reducer per application.

Let's take a closer look at each method of our `State`:

- In our `constructor` we set the `_state` to the initial state.
- `getState()` simply returns the current `_state`

- `dispatch` takes an action, sends it to the reducer and then **updates the value of `_state`** with the return value

Notice that **`dispatch` doesn't return anything**. It's only *updating* the store's state (once the result returns). This is an important principle of Redux: dispatching actions is a "fire-and-forget" maneuver. **Dispatching actions is not a direct manipulation of the state, and it doesn't return the new state.**

When we dispatch actions, we're sending off a notification of what happened. If we want to know what the current state of the system is, we have to check the state of the store.

Using the Store

Let's try using our store:

code/redux/redux-chat/tutorial/05-minimal-store.ts

```
// create a new store
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 1

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 2

store.dispatch({ type: 'DECREMENT' });
console.log(store.getState()); // -> 1
```

We start by creating a new `Store` and we save this in `store`, which we can use to get the current state and dispatch actions.

The state is set to `0` initially, and then we `INCREMENT` twice and `DECREMENT` once and our final state is `1`.

Being Notified with `subscribe`

It's great that our `Store` keeps track of what changed, but in the above example we have to *ask* for the state changes with `store.getState()`. It would be nice for us to know immediately when a new action was dispatched so that we could respond. To do this we can implement the Observer pattern - that is, we'll register a callback function that will *subscribe* to all changes.

Here's how we want it to work:

1. We will register a *listener* function using `subscribe`
2. When `dispatch` is called, we will iterate over all listeners and call them, which is the notification that the state has changed.

Registering Listeners

Our listener callbacks are going to be a function that takes *no arguments*. Let's define an interface that makes it easy to describe this:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
interface ListenerCallback {
  (): void;
}
```

After we subscribe a listener, we might want to unsubscribe as well, so lets define the interface for an *unsubscribe* function as well:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
interface UnsubscribeCallback {
  (): void;
}
```

Not much going on here - it's another function that takes no arguments and has no return value. But by defining these types it makes our code clearer to read.

Our store is going to keep a list of `ListenerCallbacks` let's add that to our `Store`:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];
```

Now we want to be able to add to that list of `_listeners` with a `subscribe` function:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
```

`subscribe` accepts a `ListenerCallback` (i.e. a function with no arguments and no return value) and returns an `UnsubscribeCallback` (the same signature). Adding the new listener is easy: we `push` it on to the `_listeners` array.

The return value is a function which will update the list of `_listeners` to be the list of `_listeners` without the `listener` we just added. That is, it returns the `UnsubscribeCallback` that we can use to remove this listener from the list.

Notifying Our Listeners

Whenever our state changes, we want to call these listener functions. What this means is, whenever we `dispatch` a new action, whenever the state changes, we want to call all of the listeners:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }
```

The Complete Store

We'll try this out below, but before we do that, here's the complete code listing for our new `Store`:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
```

```
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }

  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
}
```

Trying Out `subscribe`

Now that we can `subscribe` to changes in our store, let's try it out:

code/redux/redux-chat/tutorial/06-store-w-subscribe.ts

```
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

// subscribe
let unsubscribe = store.subscribe(() => {
  console.log('subscribed: ', store.getState());
});

store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 1
store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 2

unsubscribe();
store.dispatch({ type: 'DECREMENT' }); // (nothing logged)

// decrement happened, even though we weren't listening for it
console.log(store.getState()); // -> 1
```

Above we subscribe to our store and in the callback function we'll log `subscribed:` and then the current store state.

> Notice that the listener function is **not** given the current state as an argument. This might seem like an odd choice, but because there are some nuances to deal with, it's easier to think of *the notification of state changed* as separate from *the current state*. Without digging too much into the weeds, you can read more about this choice here[101], here[102], and here[103].

We store the `unsubscribe` callback and then notice that after we call `unsubscribe()` our log message isn't called. We can still dispatch actions, we just won't see the results until we ask the store for them.

> If you're the type of person who likes RxJS and Observables, you might notice that implementing our own subscription listeners could also be implemented using RxJS. You could rewrite our `Store` to use Observables instead of our own subscriptions.
>
> In fact, we've already done this for you and you can find the sample code in the file `code/redux/redux-chat/tutorial/06b-rx-store.ts`.
>
> Using RxJS for the `Store` is an interesting and powerful pattern if you're willing to use RxJS for the backbone of our application data.
>
> Here we're not going to use Observables very heavily, particularly because we want to discuss Redux itself and how to think about data architecture with a single state tree. Redux itself is powerful enough to use in our applications without Observables.
>
> Once you get the concepts of using "straight" Redux, adding in Observables isn't difficult (if you already understand RxJS, that is). For now, we're going to use "straight" Redux and we'll give you some guidance on some Observable-based Redux-wrappers at the end.

The Core of Redux

The above store is the essential core of Redux. Our reducer takes the current state and action and returns a new state, which is held by the store.

There are obviously many more things that we need to add to build a large, production web app. However, all of the new ideas that we'll cover are patterns that flow from building on this simple idea of an immutable, central store of state. If you understand the ideas presented above, you would be likely to invent many of the patterns (and libraries) you find in more advanced Redux apps.

There's still a lot for us to cover about day-to-day use of Redux though. For instance, we need to know:

- How to carefully handle more complex data structures in our state

[101]https://github.com/reactjs/redux/issues/1707
[102]https://github.com/reactjs/redux/issues/1513
[103]https://github.com/reactjs/redux/issues/303

- How to be notified when our state changes without having to poll the state (with subscriptions)
- How to intercept our dispatch for debugging (a.k.a. middleware)
- How to compute derived values (with *selectors*)
- How to split up large reducers into more manageable, smaller ones (and recombine them)
- How to deal with asynchronous data

We'll explain each of these issues and describe common patterns over the rest of this chapter and the next.

Let's first deal with handling more complex data structures in our state. To do that, we're going to need an example that's more interesting than a counter. Let's start building a chat app where users can send each other messages.

A Messaging App

In our messaging app, as in all Redux apps, there are three main parts to the data model:

1. The state
2. The actions
3. The reducer

Messaging App `state`

The `state` in our counter app was a single number. However in our messaging app, the `state` is going to be **an object**.

This state object will have a single property, `messages`. `messages` will be an array of strings, with each string representing an individual message in the application. For example:

```
// an example `state` value
{
  messages: [
    'here is message one',
    'here is message two'
  ]
}
```

We can define the type for the app's state like this:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
interface AppState {
  messages: string[];
}
```

Messaging App `actions`

Our app will process two actions: `ADD_MESSAGE` and `DELETE_MESSAGE`.

The `ADD_MESSAGE` action object will always have the property `message`, the message to be added to the state. The `ADD_MESSAGE` action object has this shape:

```
{
  type: 'ADD_MESSAGE',
  message: 'Whatever message we want here'
}
```

The `DELETE_MESSAGE` action object will delete a specified message from the state. A challenge here is that we have to be able to specify *which message* we want to delete.

If our messages were objects, we could assign each message an `id` property when it is created. However, to simplify this example, our messages are just simple strings, so we'll have to get a handle to the message another way. The easiest way for now is to just use the index of the message in the array (as a proxy for the ID).

With that in mind, the `DELETE_MESSAGE` action object has this shape:

```
{
  type: 'DELETE_MESSAGE',
  index: 2                // <- or whatever index is appropriate
}
```

We can define the types for these actions by using the `interface ... extends` syntax in TypeScript:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
interface AddMessageAction extends Action {
  message: string;
}

interface DeleteMessageAction extends Action {
  index: number;
}
```

In this way our `AddMessageAction` is able to specify a `message` and the `DeleteMessageAction` will specify an `index`.

Messaging App `reducer`

Remember that our reducer needs to handle two actions: `ADD_MESSAGE` and `DELETE_MESSAGE`. Let's talk about these individually.

Reducing `ADD_MESSAGE`

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
let reducer: Reducer<AppState> =
  (state: AppState, action: Action): AppState => {
  switch (action.type) {
  case 'ADD_MESSAGE':
    return {
      messages: state.messages.concat(
        (<AddMessageAction>action).message
      ),
    };
```

We start by switching on the `action.type` and handling the `ADD_MESSAGE` case.

TypeScript objects already have a type, so why are we adding a `type` field?

There are many different ways we might choose to handle this sort of "polymorphic dispatch". Keeping a string in a `type` field (where `type` means "action-type") is a straightforward, portable way we can use to distinguish different types of actions and handle them in one reducer. In part, it means that you don't *have* to create a new `interface` for every action.

That said, it would be more satisfying to be able to use reflection to switch on the concrete type. While this might become possible with more advanced type guards[104], this isn't currently possible in today's TypeScript.

Broadly speaking, types are a compile-time construct and this code is compiled down to JavaScript and we can lose some of the typing metadata.

That said, if switching on a `type` field bothers you and you'd like to use language features directly, you could use the decoration reflection metadata[105]. For now, a simple `type` field will suffice.

Adding an Item Without Mutation

When we handle an `ADD_MESSAGE` action, we need to add the given message to the state. As will all reducer handlers, we need to **return a new state**. Remember that our reducers must be *pure* and not mutate the old state.

What would be the problem with the following code?

```
case 'ADD_MESSAGE':
  state.messages.push( action.message );
  return { messages: messages };
  // ...
```

The problem is that this code **mutates** the `state.messages` array, which changes our old state! Instead what we want to do is create a *copy* of the `state.messages` array and add our new message to the copy.

[104]https://basarat.gitbooks.io/typescript/content/docs/types/typeGuard.html
[105]http://blog.wolksoftware.com/decorators-metadata-reflection-in-typescript-from-novice-to-expert-part-4

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
  case 'ADD_MESSAGE':
    return {
      messages: state.messages.concat(
        (<AddMessageAction>action).message
      ),
    };
```

The syntax `<AddMessageAction>action` will cast our `action` to the more specific type. That is, notice that our reducer takes the more general type `Action`, which does not have the `message` field. If we leave off the cast, then the compiler will complain that `Action` does not have a field `message`.

Instead, we know that we have an `ADD_MESSAGE` action so we cast it to an `AddMessageAction`. We use parentheses to make sure the compiler knows that we want to cast `action` and not `action.message`.

Remember that the reducer **must return a new `AppState`**. When we return an object from our reducer it must match the format of the `AppState` that was input. In this case we only have to keep the key `messages`, but in more complicated states we have more fields to worry about.

Deleting an Item Without Mutation

Remember that when we handle the `DELETE_MESSAGE` action we are passing the index of the item in the array as the faux ID. (Another common way of handling the same idea would be to pass a real item ID.) Again, because we do not want to mutate the old `messages` array, we need to handle this case with care:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
  case 'DELETE_MESSAGE':
    let idx = (<DeleteMessageAction>action).index;
    return {
      messages: [
        ...state.messages.slice(0, idx),
        ...state.messages.slice(idx + 1, state.messages.length)
      ]
```

Here we use the `slice` operator twice. First we take all of the items up until the item we are removing. And we concatenate the items that come after.

There are four common non-mutating operations:

- Adding an item to an array
- Removing an item from an array
- Adding / changing a key in an object
- Removing a key from an object

The first two (array) operations we just covered. We'll talk more about the object operations further down, but for now know that a common way to do this is to use `Object.assign`. As in:

```
Object.assign({}, oldObject, newObject)
             // <-------<-------------
```

You can think of `Object.assign` as merging objects in from the right into the object on the left. `newObject` is merged into `oldObject` which is merged into `{}`. This way all of the fields in `oldObject` will be kept, except for where the field exists in `newObject`. Neither `oldObject` nor `newObject` will be mutated.

Of course, handling all of this on your own takes great care and it is easy to make a mistake. This is one of the reasons many people use Immutable.js[106], which is a set of data structures that help enforce immutability.

Trying Out Our Actions

Now let's try running our actions:

code/redux/redux-chat/tutorial/07-messages-reducer.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Would you say the fringe was made of silk?'
} as AddMessageAction);

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Wouldnt have no other kind but silk'
} as AddMessageAction);

store.dispatch({
```

[106] https://facebook.github.io/immutable-js/

```
  type: 'ADD_MESSAGE',
  message: 'Has it really got a team of snow white horses?'
} as AddMessageAction);

console.log(store.getState());
// ->
// { messages:
//    [ 'Would you say the fringe was made of silk?',
//      'Wouldnt have no other kind but silk',
//      'Has it really got a team of snow white horses?' ] }
```

Here we start with a new store and we call `store.getState()` and see that we have an empty `messages` array.

Next we add three messages[107] to our store. For each message we specify the `type` as `ADD_MESSAGE` and we cast each object to an `AddMessageAction`.

Finally we log the new state and we can see that `messages` contains all three messages.

Our three `dispatch` statements are a bit ugly for two reasons:

1. we manually have to specify the `type` string each time. We could use a constant, but it would be nice if we didn't have to do this and
2. we're manually casting to an `AddMessageAction`

Instead of creating these objects as an object directly we should create a *function* that will create these objects. This idea of writing a function to create actions is so common in Redux that the pattern has a name: *Action Creators*.

Action Creators

Instead of creating the `ADD_MESSAGE` actions directly as objects, let's create a function to do this for us:

[107] https://en.wikipedia.org/wiki/The_Surrey_with_the_Fringe_on_Top

code/redux/redux-chat/tutorial/08-action-creators.ts

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message
    };
  }
  static deleteMessage(index: number): DeleteMessageAction {
    return {
      type: 'DELETE_MESSAGE',
      index: index
    };
  }
}
```

Here we've created a class with two static methods `addMessage` and `deleteMessage`. They return an `AddMessageAction` and a `DeleteMessageAction` respectively.

> You definitely don't *have* to use static methods for your action creators. You could use plain functions, functions in a namespace, even instance methods on an object, etc. The key idea is to keep them organized in a way that makes them easy to use.

Now let's use our new action creators:

code/redux/redux-chat/tutorial/08-action-creators.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

console.log(store.getState());
// ->
// { messages:
```

```
//     [ 'Would you say the fringe was made of silk?',
//       'Wouldnt have no other kind but silk',
//       'Has it really got a team of snow white horses?' ] }
```

This feels much nicer!

An added benefit is that if we eventually decided to change the format of our messages, we could do it without having to update all of our `dispatch` statements. For instance, say we wanted to add the time each message was created. We could add a `created_at` field to `addMessage` and now all `AddMessageActions` will be given a `created_at` field:

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message,
      // something like this
      created_at: new Date()
    };
  }
  // ....
```

Using Real Redux

Now that we've built our own mini-redux you might be asking, "What do I need to do to use the *real* Redux?" Thankfully, not very much. Let's update our code to use the real Redux now!

> If you haven't already, you'll want to run `npm install` in the `code/redux/redux-chat/tutorial` directory.

The first thing we need to do is import `Action`, `Reducer`, and `Store` from the `redux` package. We're also going to import a helper method `createStore` while we're at it:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
import {
  Action,
  Reducer,
  Store,
  createStore
} from 'redux';
```

Next, instead of specifying our initial state when we create the *store* instead we're going to let the *reducer* create the initial state. Here we'll do this as the default argument to the reducer. This way if there is no state passed in (e.g. the first time it is called at initialization) we will use the initial state:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let initialState: AppState = { messages: [] };

let reducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action) => {
```

What's neat about this is that the rest of our reducer stays the same!

The last thing we need to do is create the store using the `createStore` helper method from Redux:

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
```

After that, everything else just works!

code/redux/redux-chat/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

console.log(store.getState());
```

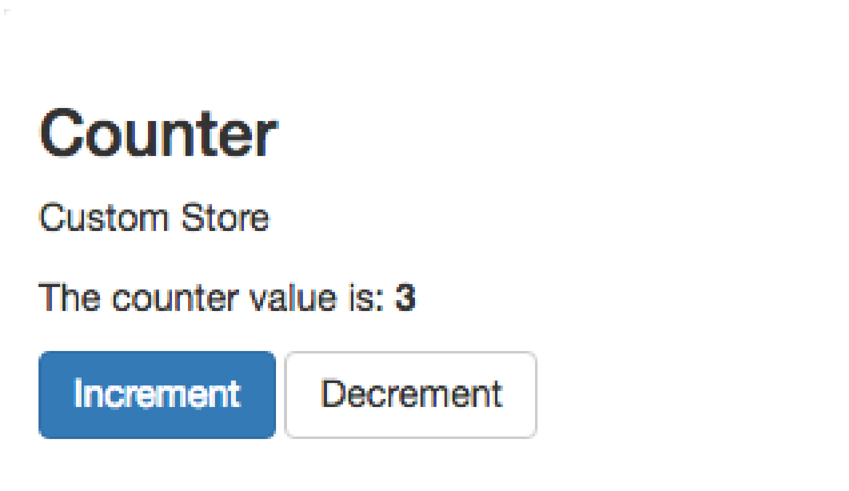
```
// ->
// { messages:
//    [ 'Would you say the fringe was made of silk?',
//      'Wouldnt have no other kind but silk',
//      'Has it really got a team of snow white horses?' ] }
```

Now that we have a handle on using Redux in isolation, the next step is to hook it up to our web app. Let's do that now.

Using Redux in Angular

In the last section we walked through the core of Redux and showed how to create reducers and use stores to manage our data in isolation. Now it's time to level-up and integrate Redux with our Angular components.

In this section we're going to create a minimal Angular app that contains just a counter which we can increment and decrement with a button.



Counter App

By using such a small app we can focus on the integration points between Redux and Angular and then we can move on to a larger app in the next section. But first, let's see how to build this counter app!

> Here we are going to be integrating Redux directly with Angular without any helper libraries in-between. There are several open-source libraries with the goal of making this process easier, and you can find them in the references section below.
>
> That said, it can be much easier to use those libraries once you understand what is going on underneath the hood, which is what we work through here.

Planning Our App

If you recall, the three steps to planning our Redux apps are to:

1. Define the structure of our central app state
2. Define actions that will change that state and
3. Define a reducer that takes the old state and an action and returns a new state.

For this app, we're just going to increment and decrement a counter. We did this in the last section, and so our actions, store, and reducer will all be very familiar.

The other thing we need to do when writing Angular apps is decide where we will create components. In this app, we'll have a top-level `AppComponent` which contains the view we see in the screenshot.

At a high level we're going to do the following:

1. Create our `Store` and make it accessible to our whole app via dependency injection
2. Subscribe to changes to the `Store` and display them in our components
3. When something changes (a button is pressed) we will dispatch an action to the `Store`.

Enough planning, let's look at how this works in practice!

Setting Up Redux

Defining the Application State

Let's take a look at our `AppState`:

code/redux/redux-chat/redux-counter/src/app/app.state.ts

```
export interface AppState {
  counter: number;
};
```

Here we are defining our core state structure as `AppState` - it is an object with one key, `counter` which is a `number`. In the next example (the chat app) we'll talk about how to have more sophisticated states, but for now this will be fine.

Defining the Reducers

Next lets define the reducer which will handle incrementing and decrementing the counter in the application state:

code/redux/redux-chat/redux-counter/src/app/counter.reducer.ts

```
import {
  INCREMENT,
  DECREMENT
} from './counter.actions';

const initialState: AppState = { counter: 0 };

// Create our reducer that will handle changes to the state
export const counterReducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action): AppState => {
    switch (action.type) {
    case INCREMENT:
      return Object.assign({}, state, { counter: state.counter + 1 });
    case DECREMENT:
      return Object.assign({}, state, { counter: state.counter - 1 });
    default:
      return state;
    }
  };
```

We start by importing the constants `INCREMENT` and `DECREMENT`, which are exported by our action creators. They're just defined as the strings `'INCREMENT'` and `'DECREMENT'`, but it's nice to get the extra help from the compiler in case we make a typo. We'll look at those action creators in a minute.

The `initialState` is an `AppState` which sets the counter to `0`.

The `counterReducer` handles two actions: `INCREMENT`, which adds `1` to the current counter and `DECREMENT`, which subtracts `1`. Both actions use `Object.assign` to ensure that we don't *mutate* the old state, but instead create a new object that gets returned as the new state.

Since we're here, let's look at the action creators

Defining Action Creators

Our action creators are functions which return objects that define the action to be taken. `increment` and `decrement` below return an object that defines the appropriate `type`.

code/redux/redux-chat/redux-counter/src/app/counter.actions.ts

```
import {
  Action,
  ActionCreator
} from 'redux';

export const INCREMENT: string = 'INCREMENT';
export const increment: ActionCreator<Action> = () => ({
  type: INCREMENT
});

export const DECREMENT: string = 'DECREMENT';
export const decrement: ActionCreator<Action> = () => ({
  type: DECREMENT
});
```

Notice that our action creator functions return the type `ActionCreator<Action>`. `ActionCreator` is a generic class defined by Redux that we use to define functions that create actions. In this case we're using the concrete class `Action`, but we could use a more specific `Action` class, such as `AddMessageAction` that we defined in the last section.

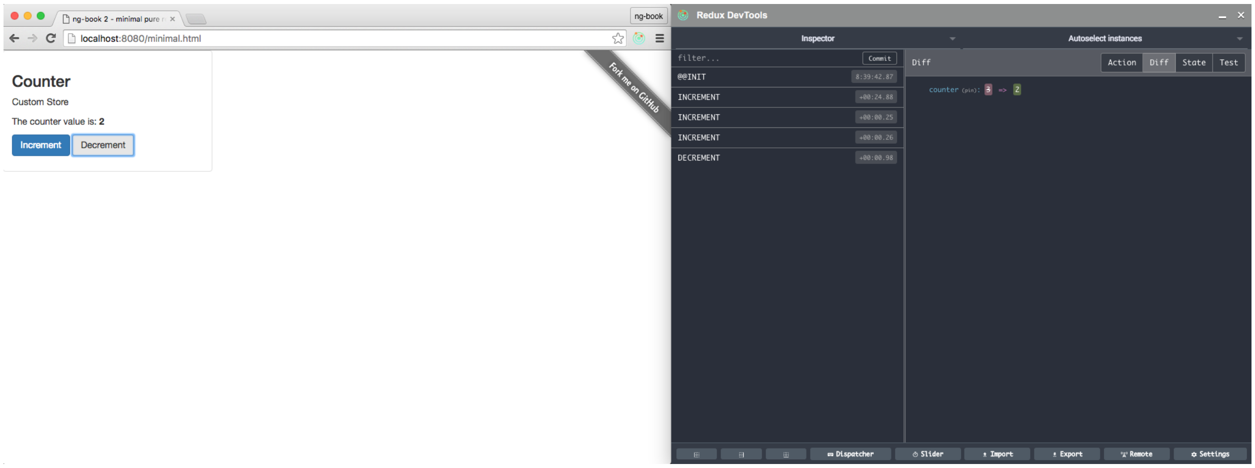
Creating the Store

Now that we have our reducer and state, we could create our store like so:

```
let store: Store<AppState> = createStore<AppState>(counterReducer);
```

However, one of the awesome things about Redux is that it has a robust set of developer tools. Specifically, there is a Chrome extension[108] that will let us monitor the state of our application and dispatch actions.

[108] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en



Counter App With Redux Devtools

What's really neat about the Redux Devtools is that it gives us clear insight to every action that flows through the system and it's affect on the state.

> Go ahead and install the Redux Devtools Chrome extension[109] now!

In order to use the Devtools we have to do one thing: add it to our store.

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
const devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;
```

Not everyone who uses our app will necessarily have the Redux Devtools installed. The code above will check for `window.devToolsExtension`, which is defined by Redux Devtools, and if it exists, we will use it. If it doesn't exist, we're just returning an *identity function* (`f => f`) that will return whatever is passed to it.

> *Middleware* is a term for a function that enhances the functionality of another library. The Redux Devtools is one of many possible middleware libraries for Redux. Redux supports lots of interesting middleware and it's easy to write our own.
>
> You can read more about Redux middleware here[110]

In order to use this `devtools` we pass it as *middleware* to our Redux store:

[109] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en
[110] http://redux.js.org/docs/advanced/Middleware.html

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}
```

Now whenever we dispatch an action and change our state, we can inspect it in our browser!

Providing the Store

Now that we have the Redux core setup, let's turn our attention to our Angular components. Let's create our top-level app component, `AppComponent`. This will be the component we use to `bootstrap` Angular:

We're going to use the `AppComponent` as the root component. Remember that since this is a Redux app, we need to make our store instance accessible everywhere in our app. How should we do this? We'll use dependency injection (DI).

If you recall from the dependency injection chapter, when we want to make something available via DI, then we use the `providers` configuration to add it to the list of `providers` in our `NgModule`.

When we provide something to the DI system, we specify two things:

1. the *token* to use to refer this injectable dependency
2. the *way* to inject the dependency

Oftentimes if we want to provide a singleton service we might use the `useClass` option as in:

```
{ provide: SpotifyService, useClass: SpotifyService }
```

In the case above, we're using the class `SpotifyService` as the *token* in the DI system. The `useClass` option tells Angular to *create an instance* of `SpotifyService` and reuse that instance whenever the `SpotifyService` injection is requested (e.g. maintain a Singleton).

One problem with us using this method is that we don't want Angular to create our store - we did it ourselves above with `createStore`. We just want to use the `store` we've already created.

To do this we'll use the `useValue` option of `provide`. We've done this before with configurable values like `API_URL`:

```
{ provide: API_URL, useValue: 'http://localhost/api' }
```

The one thing we have left to figure out is what token we want to use to inject. Our `store` is of type `Store<AppState>`:

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}

export const appStoreProviders = [
   { provide: AppStore, useFactory: createAppStore }
];
```

`Store` is an *interface*, not a class and, unfortunately, we can't use interfaces as a dependency injection key.

> If you're interested in *why* we can't use an interface as a DI key, it's because TypeScript interfaces are removed after compilation and not available at runtime.
>
> If you'd like to read more, see here[111], here[112], and here[113].

This means we need to create our own token that we'll use for injecting the store. Thankfully, Angular makes this easy to do. Let's create this token in it's own file so that way we can `import` it from anywhere in our application;

code/redux/redux-chat/redux-counter/src/app/app.store.ts

```
export const AppStore = new InjectionToken('App.store');
```

Here we have created a `const AppStore` which uses the `InjectionToken` class from Angular. `InjectionToken` is a better choice than injecting a string directly because it helps us avoid collisions.

Now we can use this token `AppStore` with `provide`. Let's do that now.

Bootstrapping the App

Back in `app.module.ts`, let's create the `NgModule` we'll use to bootstrap our app:

[111] http://stackoverflow.com/questions/32254952/binding-a-class-to-an-interface
[112] https://github.com/angular/angular/issues/135
[113] http://victorsavkin.com/post/126514197956/dependency-injection-in-angular-1-and-angular-2

code/redux/redux-chat/redux-counter/src/app/app.module.ts

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

import { appStoreProviders } from './app.store';

import { AppComponent } from './app.component';

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

Now we are able to get a reference to our Redux store anywhere in our app by injecting `AppStore`. The place we need it most now is our `AppComponent`.

> Notice that we exported the function `appStoreProviders` from `app.store.ts` and then used that function in `providers`. Why not use the `{ provide: ..., useFactory: ... }` syntax directly? The answer is related to AOT - if we want to ahead-of-time compile a provider that uses a function, we must first export is as a function from another module.

The `AppComponent`

With our setup out of the way, we can start creating our component that actually displays the counter to the user and provides buttons for the user to change the state.

`imports`

Let's start by looking at the imports:

code/redux/redux-chat/redux-counter/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';
```

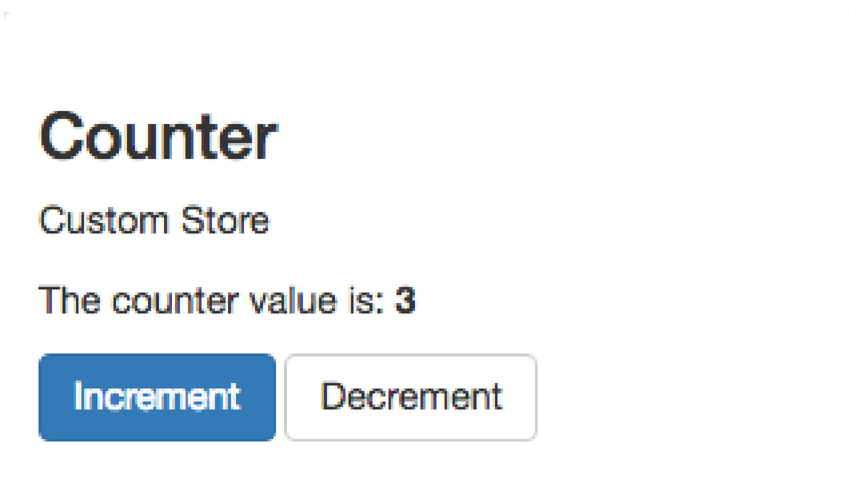
We import `Store` from Redux as well as our injector token `AppStore`, which will get us a reference to the singleton *instance* of our store. We also import the `AppState` type, which helps us know the structure of the central state.

Lastly, we import our action creators with `* as CounterActions`. This syntax will let us call `CounterActions.increment()` to create an `INCREMENT` action.

The template

Let's look at the template of our `AppComponent`.

> In this chapter we are adding some style using the CSS framework Bootstrap[114]



Counter App Template

[114] http://getbootstrap.com

code/redux/redux-chat/redux-counter/src/app/app.component.html

```
<div class="row">
  <div class="col-sm-6 col-md-4">
    <div class="thumbnail">
      <div class="caption">
        <h3>Counter</h3>
        <p>Custom Store</p>

        <p>
          The counter value is:
          <b>{{ counter }}</b>
        </p>

        <p>
          <button (click)="increment()"
                  class="btn btn-primary">
            Increment
          </button>
          <button (click)="decrement()"
                  class="btn btn-default">
             Decrement
          </button>
        </p>
      </div>
    </div>
  </div>
</div>
```

The three things to note here are that we're:

1. displaying the value of the counter in `{{ counter }}`
2. calling the `increment()` function in a button and
3. calling the `decrement()` function in a button.

The `constructor`

Remember that this component depends on the `Store`, so we need to inject it in the constructor. This is how we use our custom `AppStore` token to inject a dependency:

code/redux/redux-chat/redux-counter/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  counter: number;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.readState());
    this.readState();
  }

  readState() {
    const state: AppState = this.store.getState() as AppState;
    this.counter = state.counter;
  }

  increment() {
    this.store.dispatch(CounterActions.increment());
  }

  decrement() {
    this.store.dispatch(CounterActions.decrement());
  }
}
```

We use the `@Inject` decorator to inject `AppStore` - notice that we define the type of the variable `store` to `Store<AppState>`. Having a different injection token than the type of the dependency injected is a little different than when we use the class as the injection token (and Angular infers what to inject).

We set the `store` to an instance variable (with `private store`). Now that we have the store we can listen for changes. Here we call `store.subscribe` and call `this.readState()`, which we define below.

The store will call `subscribe` only when a new action is dispatched, so in this case we need to make sure we manually call `readState` at least once to ensure that our component gets the initial data.

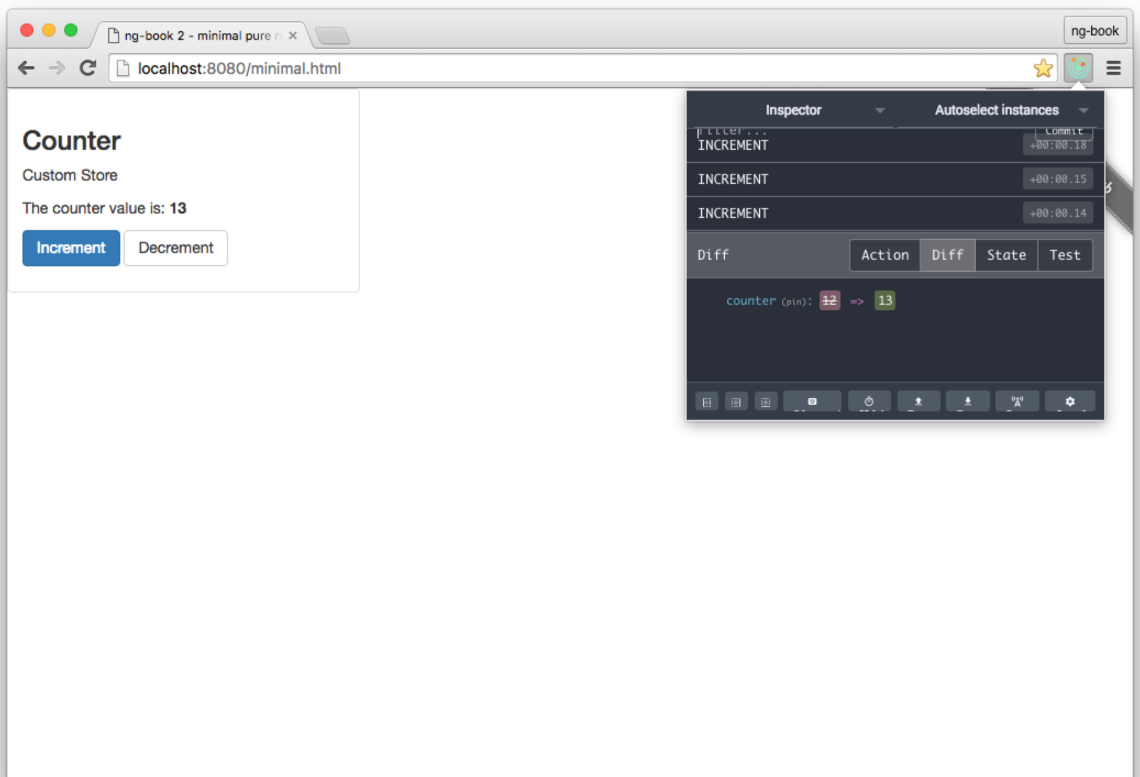
The method `readState` reads from our store and updates `this.counter` to the current value. Because `this.counter` is a property on this class and bound in the view, Angular will detect when it changes and re-render this component.

We define two helper methods: `increment` and `decrement`, each of which dispatch their respective actions to the store.

Putting It All Together

Try it out!

```
cd code/redux/redux-chat/redux-counter
npm install
npm start
open http://localhost:4200
```



Working Counter App

Congratulations! You've created your first Angular and Redux app!

What's Next

Now that we've built a basic app using Redux and Angular, we should try building a more complicated app. When we build bigger apps we encounter new challenges like:

- How do we combine reducers?
- How do we extract data from different branches of the state?
- How should we organize our Redux code?

In the next chapter, we'll build a chat app which will tackle all of these questions!

References

If you want to learn more about Redux, here are some good resources:

- Official Redux Website[115]
- This Video Tutorial by Redux's Creator[116]
- Real World Redux[117] (presentation slides)
- The power of higher-order reducers[118]

To learn more about Redux and Angular checkout:

- angular2-redux[119]
- ng2-redux[120]
- ngrx/store[121]

Onward!

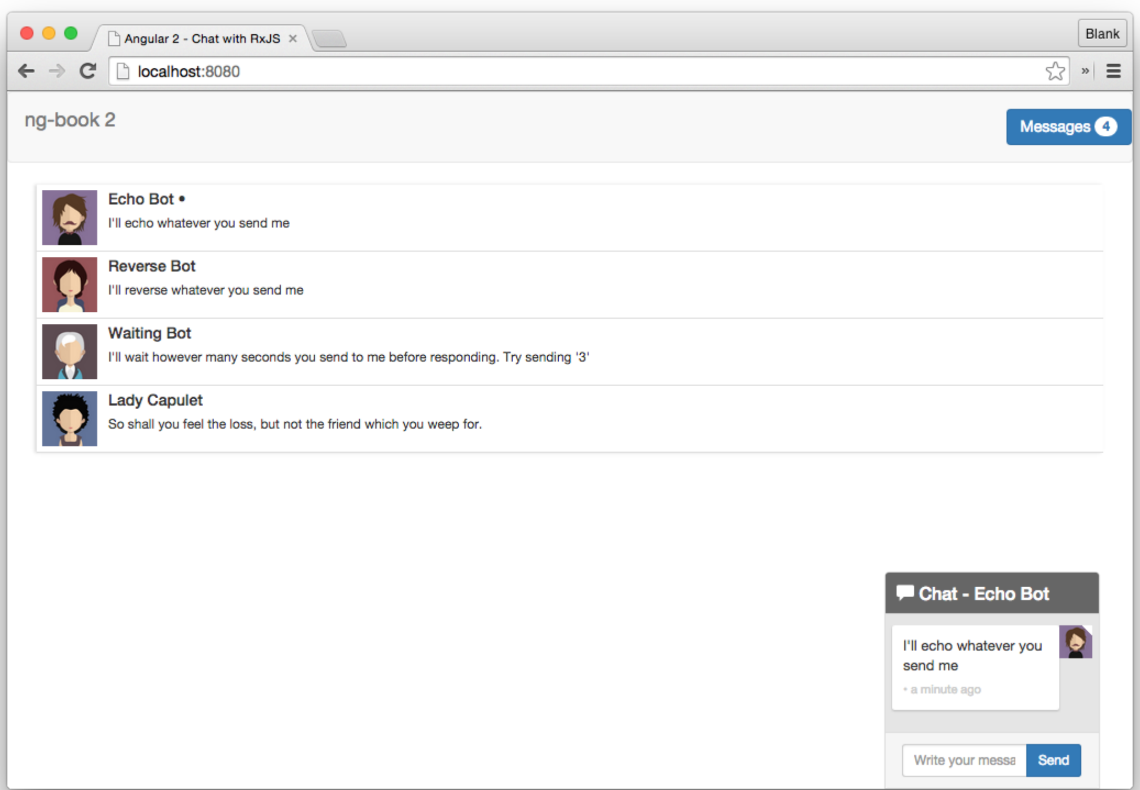
[115] http://redux.js.org/
[116] https://egghead.io/courses/getting-started-with-redux
[117] https://speakerdeck.com/chrisui/real-world-redux
[118] http://slides.com/omnidan/hor
[119] https://github.com/InfomediaLtd/angular2-redux
[120] https://github.com/angular-redux/ng2-redux
[121] https://github.com/ngrx/store

Intermediate Redux in Angular

In the last chapter we learned about Redux, the popular and elegant data architecture. In that chapter, we built an extremely basic app that tied our Angular components and the Redux store together.

In this chapter we're going to take on those ideas and build on them to create a more sophisticated chat app.

Here's a screenshot of the app we're going to build:



Completed Chat Application

Context For This Chapter

Earlier in this book we built a chat app using RxJS. We're going to be building that same app again only this time with Redux. The point is for you to be able to compare and contrast how the same app works with different data architecture strategies.

You are not required to have read the RxJS chapter in order to work through this one. This chapter stands on its own with regard to the RxJS chapters. If you have read that chapter, you'll be able to skim through some of the sections here where the code is largely the same (for instance, the data models themselves don't change much).

We *do* expect that you've read through the previous Redux chapter or at least have some familiarity with Redux.

Chat App Overview

In this application we've provided a few bots you can chat with. Open up the code and try it out:

```
cd code/redux/redux-chat
npm install
npm start
```

Now open your browser to `http://localhost:4200`.

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

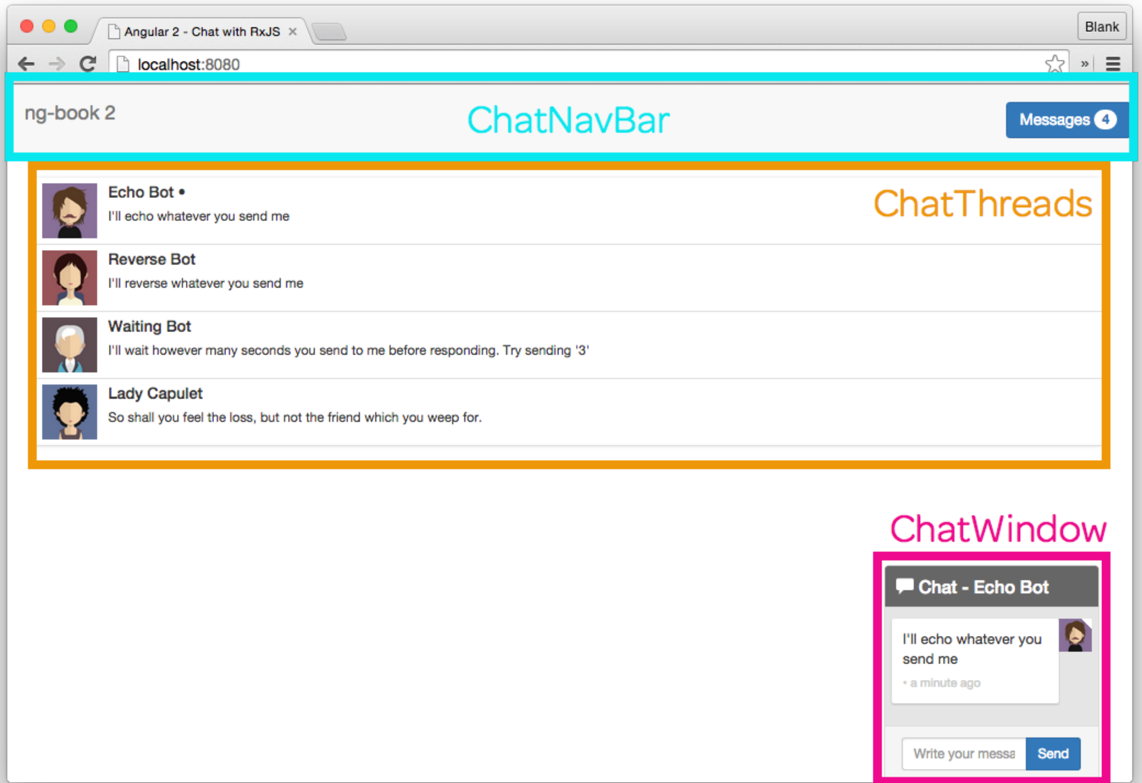
Let's look at an overview of how this app is constructed. We have

- 3 top-level Angular Components
- 3 models
- and 2 reducers, with their respective action creators

Let's look at them one at a time.

Components

The page is broken down into three top-level components:

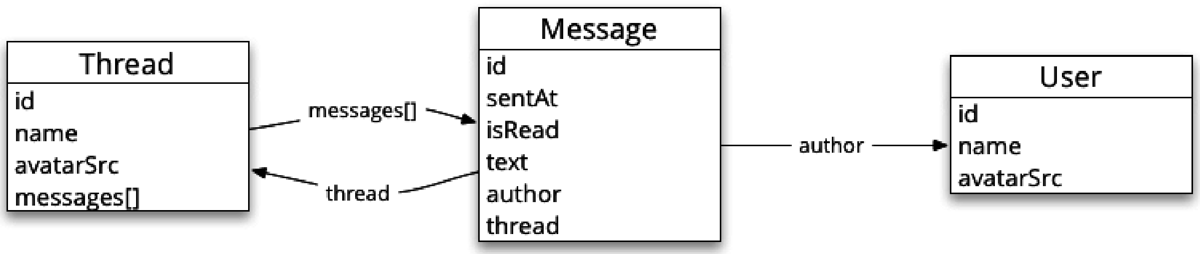


Redux Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count
- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - shows the messages in the current thread with an input box to send new messages

Models

This application also has three models:



Redux 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

Reducers

In this app, we have two reducers:

- `UsersReducer` - handles information about the current user
- `ThreadsReducer` - handles threads and their messages

Summary

At a high level our data architecture looks like this:

- All information about the users and threads (which hold messages) are contained in our central store
- Components subscribe to changes in that store and display the appropriate data (unread count, list of threads, the messages themselves
- When the user sends a message, our components dispatch an action to the store

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular and Redux. We'll start by implementing our models, then look at how we create our app state and reducers, and then finally we'll implement the Components.

Implementing the Models

Let's start with the easy stuff and take a look at the models.

We're going to be specifying each of our model definitions as `interfaces`. This isn't a requirement and you're free to use more elaborate objects if you wish. That said, objects with methods that mutate their internal state can break the functional model that we're striving for.

That is, all mutations to our app state should only be made by the reducers - the objects in the state should be immutable themselves.

So by defining an `interface` for our models,

1. we're able to ensure that the objects we're working with conform to an expected format at compile time and
2. we don't run the risk of someone accidentally adding a method to the model object that would work in an unexpected way.

`User`

Our `User` interface has an `id`, `name`, and `avatarSrc`.

code/redux/redux-chat/src/app/user/user.model.ts

```
/**
 * A User represents an agent that sends messages
 */
export interface User {
  id: string;
  name: string;
  avatarSrc: string;
  isClient?: boolean;
}
```

We also have a boolean `isClient` (the question mark indicates that this field is optional). We will set this value to `true` for the `User` that represents the client, the person using the app.

`Thread`

Similarly, `Thread` is also a TypeScript interface:

code/redux/redux-chat/src/app/thread/thread.model.ts

```
import { Message } from '../message/message.model';

/**
 * Thread represents a group of Users exchanging Messages
 */
export interface Thread {
  id: string;
  name: string;
  avatarSrc: string;
  messages: Message[];
}
```

We store the `id` of the `Thread`, the `name`, and the current `avatarSrc`. We also expect an array of `Messages` in the `messages` field.

Message

`Message` is our third and final model `interface`:

code/redux/redux-chat/src/app/message/message.model.ts

```
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';

/**
 * Message represents one message being sent in a Thread
 */
export interface Message {
  id?: string;
  sentAt?: Date;
  isRead?: boolean;
  thread?: Thread;
  author: User;
  text: string;
}
```

Each message has:

- `id` - the id of the message
- `sentAt` - when the message was sent
- `isRead` - a boolean indicating that the message was read
- `author` - the `User` who wrote this message
- `text` - the text of the message
- `thread` - a reference to the containing `Thread`

App State

Now that we have our models, let's talk about the shape of our central state. In the previous chapter, our central state was a single object with the key `counter` which had the value of a `number`. This app, however, is more complicated.

Here's the first part of our app state:

code/redux/redux-chat/src/app/app.reducer.ts

```
export interface AppState {
 users: UsersState;
 threads: ThreadsState;
}
```

Our `AppState` is also an `interface` and it has two top level keys: `users` and `threads` - these are defined by two more interfaces `UsersState` and `ThreadsState`, which are defined in their respective reducers.

A Word on Code Layout

This is a common pattern we use in Redux apps: the top level state has a top-level key for each reducer. In our app we're going to keep this top-level reducer in `app.reducer.ts`.

Each reducer will have it's own file. In that file we'll store:

- The `interface` that describes that branch of the state tree
- The value of the initial state, for that branch of the state tree
- The reducer itself
- Any *selectors* that query that branch of the state tree - we haven't talked about *selectors* yet, but we will soon.

The reason we keep all of these different things together is because they all deal with the structure of this branch of the state tree. By putting these things in the same file it's very easy to refactor everything at the same time.

You're free to have multiple layers of nesting, if you so desire. It's a nice way to break up large modules in your app.

The Root Reducer

Since we're talking about how to split up reducers, let's look at our root reducer now:

code/redux/redux-chat/src/app/app.reducer.ts

```
export interface AppState {
 users: UsersState;
 threads: ThreadsState;
}

const rootReducer: Reducer<AppState> = combineReducers<AppState>({
 users: UsersReducer,
 threads: ThreadsReducer
});

export default rootReducer;
```

Notice the symmetry here - our `UsersReducer` will operate on the `users` key, which is of type `UsersState` and our `ThreadsReducer` will operate on the `threads` key, which is of type `ThreadsState`.

This is made possible by the `combineReducers` function which takes a map of keys and reducers and returns a new reducer that operates appropriately on those keys.

Of course we haven't finished looking at the structure of our `AppState` yet, so let's do that now.

The `UsersState`

Our `UsersState` holds a reference to the `currentUser`.

code/redux/redux-chat/src/app/user/users.reducer.ts

```
export interface UsersState {
  currentUser: User;
};

const initialState: UsersState = {
  currentUser: null
};
```

You could imagine that this branch of the state tree could hold information about all of the users, when they were last seen, their idle time, etc. But for now this will suffice.

We'll use `initialState` in our reducer when we define it below, but for now we're just going to set the current user to `null`.

The `ThreadsState`

Let's look at the `ThreadsState`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export interface ThreadsEntities {
  [id: string]: Thread;
}

export interface ThreadsState {
  ids: string[];
  entities: ThreadsEntities;
  currentThreadId?: string;
};

const initialState: ThreadsState = {
  ids: [],
  currentThreadId: null,
  entities: {}
};
```

We start by defining an interface called `ThreadsEntities` which is a map of thread `ids` to `Threads`. The idea is that we'll be able to look up any thread by id in this map.

In the `ThreadsState` we're also storing an array of the `ids`. This will store the list of possible ids that we might find in `entities`.

> This strategy is used by the commonly-used library normalizr[122]. The idea is that when we standardize how we store entities in our Redux state, we're able to build helper libraries and it's clearer to work with. Instead of wondering what the format is for each tree of the state, when we use `normalizr` a lot of the choices have been made for us and we're able to work more quickly.
>
> I've opted not to teach `normalizr` in this chapter because we're learning so many other things. That said, I would be very likely to use `normalizr` in my production applications.
>
> That said, `normalizr` is totally optional - nothing major changes in our app by not using it.
>
> If you'd like to learn how to use `normalizr`, checkout the official docs[123], this blog post[124], and the thread referenced by Redux creator Dan Abramov here[125]

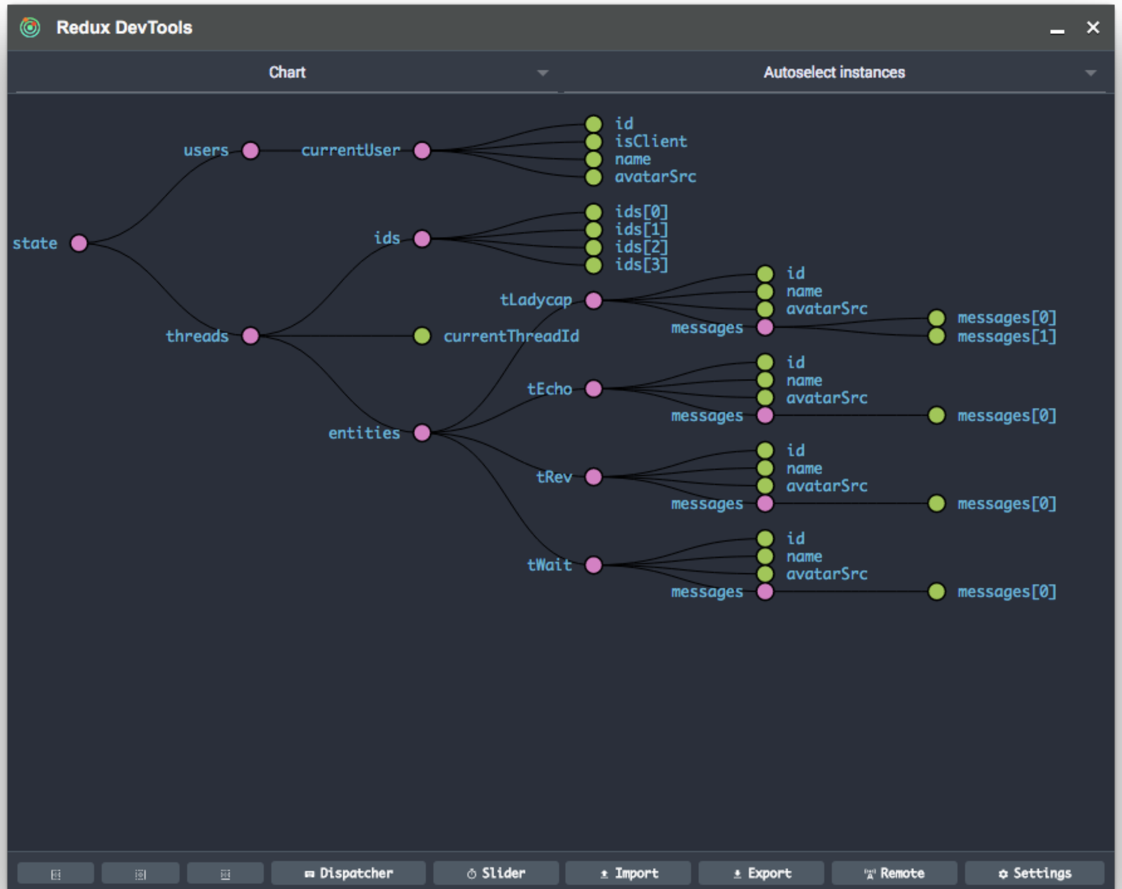
We store the currently viewed thread in `currentThreadId` - the idea here is that we want to know which thread the user is currently looking at.

We set our `initialState` to "empty" values.

[122] https://github.com/paularmstrong/normalizr
[123] https://github.com/paularmstrong/normalizr
[124] https://medium.com/@mcowpercoles/using-normalizr-js-in-a-redux-store-96ab33991369#.l8ur7ipu6
[125] https://twitter.com/dan_abramov/status/663032263702106112

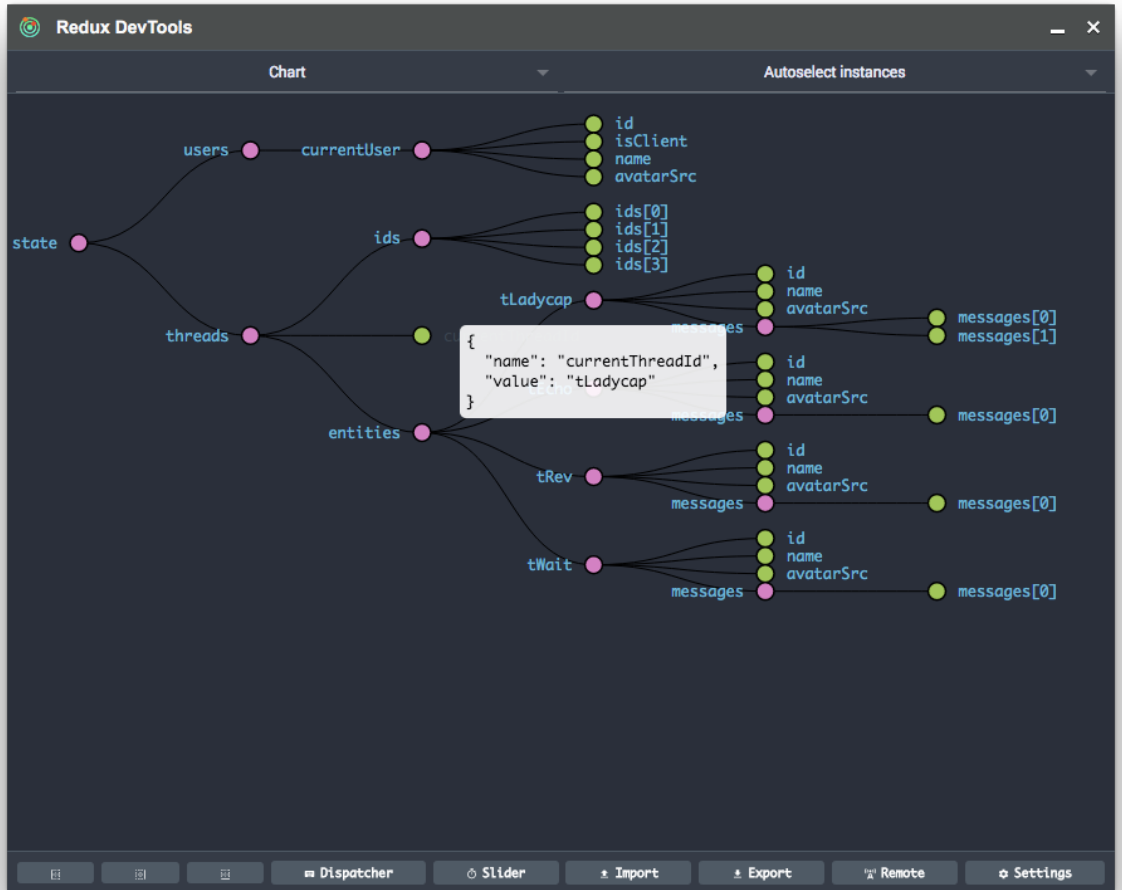
Visualizing Our `AppState`

Redux Devtools provides us with a "Chart" view that lets us inspect the state of our app. Here's what mine looks like after being booted with all of the demo data:



Redux Chat State Chart

What's neat is that we can hover over an individual node and see the attributes of that piece of data:



Inspecting the current thread

Building the Reducers (and Action Creators)

Now that we have our central state, we can start changing it using our reducers!

Since reducers handle actions, we need to know the format of our actions in our reducer. So let's build our action creators at the same time we build our reducers

Set Current User Action Creators

The `UsersState` stores the current user. This means we need an action to set the current user. We're going to keep our actions in the `actions` folder and name the actions to match their corresponding reducer, in this case `UserActions`.

code/redux/redux-chat/src/app/user/user.actions.ts

```
export const SET_CURRENT_USER = '[User] Set Current';
export interface SetCurrentUserAction extends Action {
  user: User;
}
export const setCurrentUser: ActionCreator<SetCurrentUserAction> =
  (user) => ({
    type: SET_CURRENT_USER,
    user: user
  });
```

Here we define the `const SET_CURRENT_USER`, which we'll use to `switch` on in our reducer.

We also define a new subinterface `SetCurrentUserAction` which extends `Action` to add a `user` property. We'll use the `user` property to indicate *which user* we want to make the current user.

The function `setCurrentUser` is our proper action creator function. It takes `user` as an argument, and returns a `SetCurrentUserAction` which we can give to our reducer.

`UsersReducer` - Set Current User

Now we turn our attention to our `UsersReducer`:

code/redux/redux-chat/src/app/user/users.reducer.ts

```
export const UsersReducer =
  function(state: UsersState = initialState, action: Action): UsersState {
  switch (action.type) {
    case UserActions.SET_CURRENT_USER:
    const user: User = (<UserActions.SetCurrentUserAction>action).user;
      return {
        currentUser: user
      };
    default:
      return state;
  }
};
```

Our `UsersReducer` takes a `UsersState` as the first argument. Notice that this isn't the `AppState`! Our "child reducer" only works with it's branch of the state tree.

Our `UsersReducer`, like all reducers, returns a new state, in this case it is of type `UsersState`.

Next we `switch` on the `action.type` and we handle the `UserActions.SET_CURRENT_USER`.

In order to set the current user, we need to get the `user` from the incoming action. To do this, we first cast the `action` to `UserActions.SetCurrentUserAction` and then we read the `.user` field.

> It might seem a little weird that we originally created a `SetCurrentUserAction` but then now we switch on a `type` string instead of using the type directly.
>
> Indeed, we are fighting TypeScript a little here. We lose interface metadata when the TypeScript is compiled to JavaScript. We could instead try some sort of reflection (through decorator metadata, or looking at a constructor etc.).
>
> While down-casting our `SetCurrentUserAction` to an `Action` on `dispatch` and then re-casting is a bit ugly, it's a straightforward and portable way to handle this "polymorphic dispatch" for this app.

We need to return a new `UsersState`. Since `UsersState` only has one key, we return an object with the `currentUser` set to the incoming action's `user`.

Thread and Messages Overview

The core of our application is messages in threads. There are three actions we need to support:

1. Adding a new thread to the state
2. Adding messages to a thread
3. Selecting a thread

Let's start by creating a new thread

Adding a New Thread Action Creators

Here's the action creator for adding a new `Thread` to our state:

code/redux/redux-chat/src/app/thread/thread.actions.ts

```
export const ADD_THREAD = '[Thread] Add';
export interface AddThreadAction extends Action {
  thread: Thread;
}
export const addThread: ActionCreator<AddThreadAction> =
  (thread) => ({
    type: ADD_THREAD,
    thread: thread
  });
```

Notice that this is structurally very similar to our previous action creator. We define a `const ADD_-THREAD` that we can switch on, a custom `Action`, and an action creator `addThread` which generates the `Action`.

Notice that we don't initialize the `Thread` itself here - the `Thread` is accepted as an argument.

Adding a New Thread Reducer

Now let's start our `ThreadsReducer` by handling `ADD_THREAD`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const ThreadsReducer =
  function(state: ThreadsState = initialState, action: Action): ThreadsState {
  switch (action.type) {

    // Adds a new Thread to the list of entities
    case ThreadActions.ADD_THREAD: {
      const thread = (<ThreadActions.AddThreadAction>action).thread;

      if (state.ids.includes(thread.id)) {
        return state;
      }

      return {
        ids: [ ...state.ids, thread.id ],
        currentThreadId: state.currentThreadId,
        entities: Object.assign({}, state.entities, {
          [thread.id]: thread
        })
      };
    }

    // Adds a new Message to a particular Thread
```

Our `ThreadsReducer` handles the `ThreadsState`. When we handle the `ADD_THREAD` action, we cast the `action` object back into a `ThreadActions.AddThreadAction` and then pull the `Thread` out.

Next we check to see if this new `thread.id` already appears in the list of `state.ids`. If it does, then we don't make any changes, but instead return the current `state`.

However if this `thread` is new, then we need to add it to our current state.

Remember when we create a new `ThreadsState` we need to take care to not mutate our old state. This looks more complicated than any state we've done so far, but it's not very different in principle.

We start by adding our `thread.id` to the `ids` array. Here we're using the ES6 spread operator (...) to indicate that we want to put all of the existing `state.ids` into this new array and then append `thread.id` to the end.

`currentThreadId` does not change when we add a new thread, so we return the *old* `state.currentThreadId` for this field.

For `entities`, remember that it is an object where the key is the `string` id of each thread and the value is the thread itself. We're using `Object.assign` here to create a new object that merges the old `state.entities` with our newly added `thread` into a new object.

> You might be kind of tired of meticulously copying these objects when we need to make changes. That's a common response! In fact, it's easy to make mutations here by accident.
>
> This is why Immutable.js[126] was written. Immutable.js is often used with Redux for this purpose. When we use Immutable, these careful updates are handled for us.
>
> I'd encourage you to take a look at Immutable.js and see if it is a good fit for your reducers.

Now we can add new threads to our central state!

Adding New Messages Action Creators

Now that we have threads we can start adding messages to them.

Let's define a new action for adding messages:

code/redux/redux-chat/src/app/thread/thread.actions.ts

```
export const ADD_MESSAGE = '[Thread] Add Message';
export interface AddMessageAction extends Action {
  thread: Thread;
  message: Message;
}
```

The `AddMessageAction` adds a `Message` to a `Thread`.

Here's the action creator for adding a message:

code/redux/redux-chat/src/app/thread/thread.actions.ts

```
export const addMessage: ActionCreator<AddMessageAction> =
  (thread: Thread, messageArgs: Message): AddMessageAction => {
    const defaults = {
      id: uuid(),
      sentAt: new Date(),
      isRead: false,
      thread: thread
    };
    const message: Message = Object.assign({}, defaults, messageArgs);

    return {
```

[126] https://facebook.github.io/immutable-js/

```
      type: ADD_MESSAGE,
      thread: thread,
      message: message
    };
  };
```

The addMessage action creator accepts a thread and an object we use for crafting the message. Notice here that we keep a list of defaults. The idea here is that we want to encapsulate creating an id, setting the timestamp, and setting the isRead status. Someone who wants to send a message shouldn't have to worry about how the UUIDs are formed, for instance.

That said, maybe the client using this library crafted the message beforehand and if they send a message with an existing id, we want to keep it. To enable this default behavior we merge the messageArgs into the defaults and copy those values to a new object.

Lastly we return the ADD_MESSAGE action with the thread and new message.

Adding A New Message Reducer

Now we will add our ADD_MESSAGE handler to our ThreadsReducer. When a new message is added, we need to take the thread and add the message to it.

There is one tricky thing we need to handle that may not be obvious at this point: if the thread is the "current thread" we need to *mark this message as read*.

The user will always have one thread that is the "current thread" that they're looking at. We're going to say that if a new message is added to the current thread, then it's automatically marked as read.

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
    case ThreadActions.ADD_MESSAGE: {
      const thread = (<ThreadActions.AddMessageAction>action).thread;
      const message = (<ThreadActions.AddMessageAction>action).message;

      // special case: if the message being added is in the current thread, then
      // mark it as read
      const isRead = message.thread.id === state.currentThreadId ?
                      true : message.isRead;
      const newMessage = Object.assign({}, message, { isRead: isRead });

      // grab the old thread from entities
      const oldThread = state.entities[thread.id];

      // create a new thread which has our newMessage
      const newThread = Object.assign({}, oldThread, {
```

```
        messages: [...oldThread.messages, newMessage]
      });

      return {
        ids: state.ids, // unchanged
        currentThreadId: state.currentThreadId, // unchanged
        entities: Object.assign({}, state.entities, {
          [thread.id]: newThread
        })
      };
    }

    // Select a particular thread in the UI
```

The code is a bit long because we're being careful not to mutate the original thread, but it is not much different than what we've done so far in principle.

We start by extracting the `thread` and `message`.

Next we mark the message as read, if its part of the "current thread" (we'll look at how to set the current thread next).

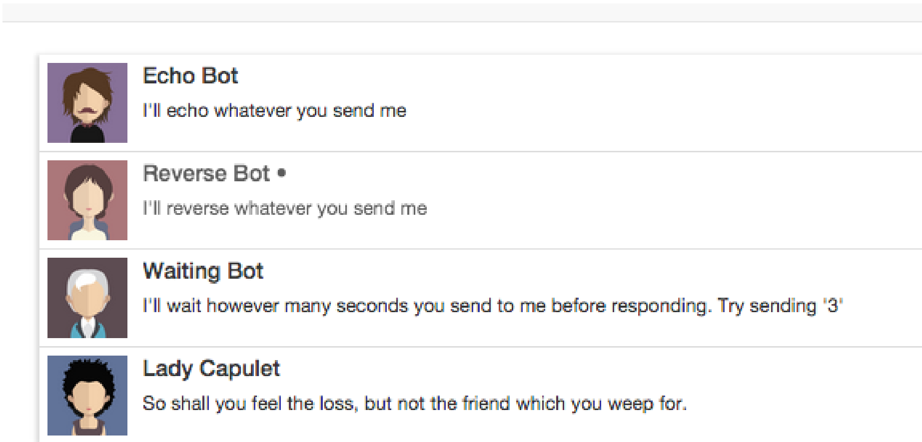
Then we grab the `oldThread` and create a `newThread` which has the `newMessage` appended on to the old `messages`.

Finally we return the new `ThreadsState`. The current list of thread `ids` and the `currentThreadId` are unchanged by adding a message, so we pass the old values here. The only thing we change is that we update `entities` with our `newThread`.

Now let's implement the last part of our data backbone: selecting a thread.

Selecting A Thread Action Creators

Our user can have multiple chat sessions in progress at the same time. However, we only have one chat window (where the user can read and send messages). When the user clicks on a thread, we want to show that thread's messages in the chat window.



Selecting A Thread

We need to keep track of which thread is the currently selected thread. To do that, we'll use the `currentThreadId` property in the `ThreadsState`.

Let's create the actions for this:

code/redux/redux-chat/src/app/thread/thread.actions.ts

```
export const SELECT_THREAD = '[Thread] Select';
export interface SelectThreadAction extends Action {
  thread: Thread;
}
export const selectThread: ActionCreator<SelectThreadAction> =
  (thread) => ({
    type: SELECT_THREAD,
    thread: thread
  });
```

There's nothing conceptually new in this action: we've got a new type of `SELECT_THREAD` and we pass the `Thread` that we're selecting as an argument.

Selecting A Thread Reducer

To select a `thread` we need to do two things:

1. set `currentThreadId` to the selected `thread`'s `id`
2. mark all messages in that `thread` as read

Here's the code for that reducer:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
    case ThreadActions.SELECT_THREAD: {
      const thread = (<ThreadActions.SelectThreadAction>action).thread;
      const oldThread = state.entities[thread.id];

      // mark the messages as read
      const newMessages = oldThread.messages.map(
        (message) => Object.assign({}, message, { isRead: true }));

      // give them to this new thread
      const newThread = Object.assign({}, oldThread, {
        messages: newMessages
      });

      return {
        ids: state.ids,
        currentThreadId: thread.id,
        entities: Object.assign({}, state.entities, {
          [thread.id]: newThread
        })
      };
    }

    default:
      return state;
  }
};
```

We start by getting the `thread`-to-select and then using that `thread.id` to get the current `Thread` that exists in `state` to get the values.

> This maneuver is a bit defensive. Why not just use the `thread` that is passed in? That might be the right design decision for some apps. In this case we protect against some external mutation of `thread` by reading the last known values of that thread in `state.entities`.

Next we create a copy of all of the old messages and set them as `isRead: true`. Then we assign those new read messages to `newThread`.

Finally we return our new `ThreadsState`.

Reducers Summary

We did it! Above is everything we need for the backbone of our data architecture.

To recap, we have a `UsersReducer` which maintains the current user. We have a `ThreadsReducer` which manages:

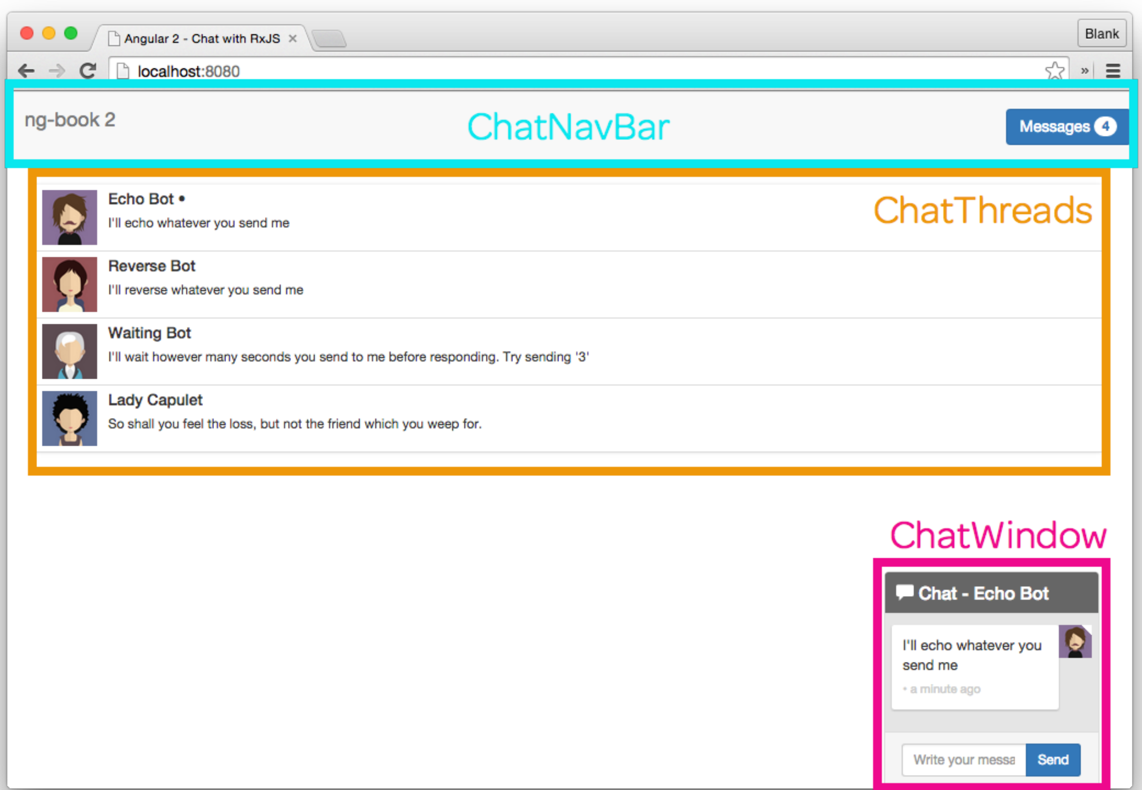
- The list of threads
- The messages in those threads
- The currently selected thread

We can derive everything else that we need (e.g. the unread count) from these pieces of data.

Now we need to hook them up to our components!

Building the Angular Chat App

As we mentioned earlier in the chapter, the page is broken down into three top-level components:



Redux Chat Top-Level Components

- `ChatNavBarComponent` - contains the unread messages count

- `ChatThreadsComponent` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindowComponent` - shows the messages in the current thread with an input box to send new messages

We're going to bootstrap our app much like we did in the last chapter. We're going to initialize our Redux store at the top of the app and provide it via Angular's dependency injection system (take a look at the previous chapter if this looks unfamiliar):

code/redux/redux-chat/src/app/app.store.ts

```
import { InjectionToken } from '@angular/core';
import {
  createStore,
  Store,
  compose,
  StoreEnhancer
} from 'redux';

import {
  AppState,
  default as reducer
} from './app.reducer';

export const AppStore = new InjectionToken('App.store');

const devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;

export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}

export const appStoreProviders = [
   { provide: AppStore, useFactory: createAppStore }
];
```

The top-level `AppComponent`

Our `AppComponent` component is the top-level component. It doesn't do much other than render the `ChatPage`.

code/redux/redux-chat/src/app/app.component.ts

```
import { Component, Inject } from '@angular/core';
import * as Redux from 'redux';

import { AppStore } from './app.store';
import { AppState } from './app.reducer';
import { ChatExampleData } from './data/chat-example-data';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(@Inject(AppStore) private store: Redux.Store<AppState>) {
    ChatExampleData(store);
  }
}
```

and the template:

code/redux/redux-chat/src/app/app.component.html

```
<div>
  <chat-page></chat-page>
</div>
```

For this app the bots operate on data on the client and are not connected to a server. The function `ChatExampleData()` sets up the initial data for the app. We won't be covering this code in detail in the book, so feel free to look at the code on disk if you want to learn more about how it works.

We're not using a router in this app, but if we were, we would put it here at the top level of the app. For now, we're going to create a `ChatPage` which will render the bulk of our app.

We don't have any other pages in this app, but it's a good idea to give each page it's own component in case we add some in the future.

The `ChatPage`

Our chat page renders our three main components:

- `ChatNavBarComponent`
- `ChatThreadsComponent` and
- `ChatWindowComponent`

Here it is in code:

code/redux/redux-chat/src/app/chat-page/chat-page.component.ts

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

@Component({
  selector: 'chat-page',
  templateUrl: './chat-page.component.html',
  styleUrls: ['./chat-page.component.css']
})
export class ChatPageComponent implements OnInit {
  constructor() { }
  ngOnInit() { }
}
```

and the template:

code/redux/redux-chat/src/app/chat-page/chat-page.component.html

```
<div>
  <chat-nav-bar></chat-nav-bar>
  <div class="container">
    <chat-threads></chat-threads>
    <chat-window></chat-window>
  </div>
</div>
```

For this app we are using a design pattern called *container components* and these three components are all container components. Let's talk about what that means.

Container vs. Presentational Components

It is hard to reason about our apps if there is data spread throughout all of our components. However, our apps are dynamic - they need to be populated with runtime data and they need to be responsive to user interaction.

One of the patterns that has emerged in managing this tension is the idea of presentational vs. container components. The idea is this:

1. You want to minimize the number of components which interact with outside data sources. (e.g. APIs, the Redux Store, Cookies etc.)
2. Therefore deliberately put data access into "container" components and
3. Require purely 'functional' presentation components to have all of their properties (inputs and outputs) managed by container components.

The great thing about this design is that presentational components are predictable. They're reusable because they don't make assumptions about your overall data-architecture, they only give requirements for their own use.

But even beyond reuse, they're predictable. Given the same inputs, they always return the same outputs (e.g. render the same way).

> If you squint, you can see that the philosophy that requires reducers to be pure functions is the same that requires presentational components be 'pure components'

It would be great if our entire app could be all presentational components, but of course, the real world has messy, changing data. So we try to put this complexity of adapting our real-world data into our container components.

> If you're an advanced programmer you may see that there is a loose analogy between MVC and container/presentation components. That is, the presentational component is sort of a "view" of data that is passed in. A container component is sort of a "controller" in that it takes the "model" (the data from the rest of the app) and adapts it for the presentational components.
>
> That said, if you haven't been programming very long, take this analogy with a grain of salt as Angular components are already a view and a controller themselves.

In our app the container components are going to be the components which interact with the store. This means our container components will be anything that:

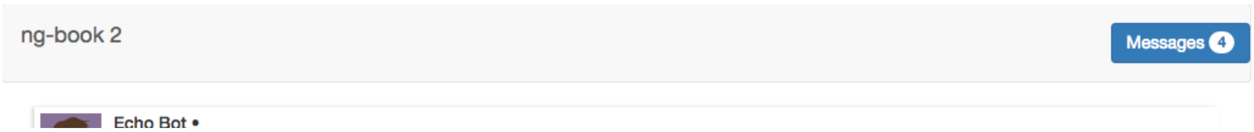
1. Reads data from the store
2. Subscribes to the store for changes
3. Dispatches actions to the store

Our three main components are container components and anything below them will be presentational (i.e. functional / pure / not interact with the store).

Let's build our first container component, the nav bar.

Building the `ChatNavBarComponent`

In the nav bar we'll show an unread messages count to the user.



The Unread Count in the `ChatNavBarComponent`

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.

Let's look at the component code first:

code/redux/redux-chat/src/app/chat-nav-bar/chat-nav-bar.component.ts

```
import { Component, Inject } from '@angular/core';
import { AppStore } from '../app.store';
import * as Redux from 'redux';
import {
  AppState,
  getUnreadMessagesCount
} from '../app.reducer';

@Component({
  selector: 'chat-nav-bar',
  templateUrl: './chat-nav-bar.component.html',
  styleUrls: ['./chat-nav-bar.component.css']
})
export class ChatNavBarComponent {
  unreadMessagesCount: number;

  constructor(@Inject(AppStore) private store: Redux.Store<AppState>) {
    store.subscribe(() => this.updateState());
    this.updateState();
  }

  updateState() {
    this.unreadMessagesCount = getUnreadMessagesCount(this.store.getState());
  }
}
```

and the template:

code/redux/redux-chat/src/app/chat-nav-bar/chat-nav-bar.component.html

```html
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="https://ng-book.com/2">
        <img src="assets/images/logos/ng-book-2-minibook.png"/>
         ng-book 2
      </a>
    </div>
    <p class="navbar-text navbar-right">
      <button class="btn btn-primary" type="button">
        Messages <span class="badge">{{ unreadMessagesCount }}</span>
      </button>
    </p>
  </div>
</nav>
```

Our template gives us the DOM structure and CSS necessary for rending a nav bar (these CSS-classes come from the CSS framework Bootstrap).

The only variable we're showing in this template is `unreadMessagesCount`.

Our `ChatNavBarComponent` has `unreadMessagesCount` as an instance variable. This number will be set to the sum of unread messages in all threads.

Notice in our `constructor` we do three things:

1. `Inject` our store
2. Subscribe to any changes in the store
3. Call `this.updateState()`

We call `this.updateState()` after `subscribe` because we want to make sure this component is initialized with the most recent data. `subscribe` will only be called if something changes *after* this component is initialized.

`updateState()` is the most interesting function - we set `unreadMessagesCount` to the value of the function `getUnreadMessagesCount`. What is `getUnreadMessagesCount` and where did it come from?

`getUnreadMessagesCount` is a new concept called *selectors*.

Redux Selectors

Thinking about our `AppState`, how might we go about getting the unread messages count? How about something like this:

```
// get the state
let state = this.store.getState();

// get the threads state
let threadsState = state.threads;

// get the entities from the threads
let threadsEntities = threadsState.entities;

// get all of the threads from state
let allThreads = Object.keys(threadsEntities)
               .map((threadId) => entities[threadId]);

// iterate over all threads and ...
let unreadCount = allThreads.reduce(
      (unreadCount: number, thread: Thread) => {
        // foreach message in that thread
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            // if it's unread, increment unread count
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0);
```

Should we put this logic in the `ChatNavBarComponent`? There's two problems with that approach:

1. This chunk of code reaches deep into our `AppState`. A better approach would be to co-locate this logic next to where the state itself is written.
2. What if we need the unread count somewhere else in the app? How could we share this logic?

Solving these problems is the idea behind *selectors*.

Selectors are functions that take a part of the state and return a value.

Let's take a look at how to make a few selectors.

Threads Selectors

Let's start with an easy one. Say we have our `AppState` and we want to get the `ThreadsState`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getThreadsState = (state): ThreadsState => state.threads;
```

Pretty easy, right? Here we're saying, given the top-level `AppState`, we can find the `ThreadsState` at `state.threads`.

Let's say that we want to get the current thread. We could do it like this:

```
const getCurrentThread = (state: AppState): Thread => {
  let currentThreadId = state.threads.currentThreadId;
  return state.threads.entities[currentThreadId];
}
```

For this small example, this selector works fine. But it's worth thinking about how we can make our selectors maintainable as the app grows. It would be nice if we could use selectors to query other selectors. It also would be nice to be able to specify a selector that has multiple selectors as a dependency.

This is what the reselect[127] library provides. With `reselect` we can create small, focused selectors and then combine them together into bigger functionality.

Let's look at how we will get the current thread using `createSelector` from `reselect`.

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getThreadsEntities = createSelector(
  getThreadsState,
  ( state: ThreadsState ) => state.entities );
```

We start by writing `getThreadsEntities`. `getThreadsEntities` uses `createSelector` and passes two arguments:

1. `getThreadsState`, the selector we defined above and
2. A callback function which will receive *the value of the selector in #1* and return the value we want to select.

This might seem like a lot of overhead to call `state.entities`, but it sets us up for a much more maintainable selectors down the line. Let's look at `getCurrentThread` using `createSelector`:

[127] https://github.com/reactjs/reselect#createselectorinputselectors--inputselectors-resultfunc

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getCurrentThread = createSelector(
  getThreadsEntities,
  getThreadsState,
  ( entities: ThreadsEntities, state: ThreadsState ) =>
    entities[state.currentThreadId] );
```

Notice here that we're citing **two** selectors as dependencies: `getThreadsEntities` and `getThreadsState` - when these selectors resolve they become the arguments to the callback function. We can then combine them together to return the selected thread.

Unread Messages Count Selector

Now that we understand how selectors work, let's create a selector that will get the number of unread messages. If you look at our first attempt at unread messages above, we can see that each variable could instead become it's own selector (`getThreadsState`, `getThreadsEntities`, etc.)

Here's a selector that will get all `Threads`:

code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getAllThreads = createSelector(
  getThreadsEntities,
  ( entities: ThreadsEntities ) => Object.keys(entities)
                         .map((threadId) => entities[threadId]));
```

And then given all of the threads, we can get the sum of the unread messages over all threads:

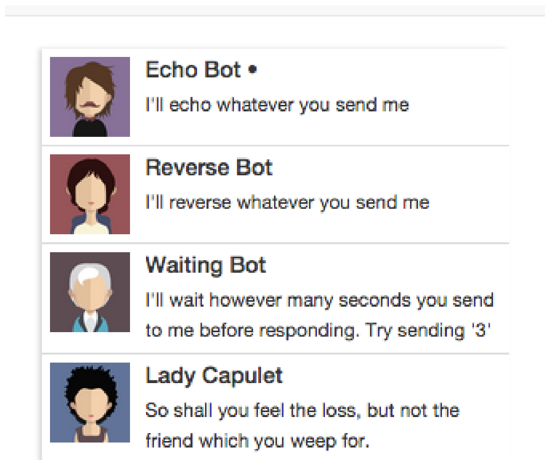
code/redux/redux-chat/src/app/thread/threads.reducer.ts

```
export const getUnreadMessagesCount = createSelector(
  getAllThreads,
  ( threads: Thread[] ) => threads.reduce(
      (unreadCount: number, thread: Thread) => {
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0));
```

Now that we have this selector, we can use it to get the number of unread messages in our `ChatNavBarComponent` (and anywhere else in our app where we might need it).

Building the `ChatThreadsComponent`

Next let's build our thread list in the `ChatThreadsComponent`.



Time Ordered List of Threads

`ChatThreadsComponent` Controller

Let's take a look at our component controller `ChatThreadsComponent` before we look at the template:

code/redux/redux-chat/src/app/chat-threads/chat-threads.component.ts

```
import {
  Component,
  OnInit,
  Inject
} from '@angular/core';
import { AppStore } from '../app.store';
import * as Redux from 'redux';
import {
  Thread
} from '../thread/thread.model';
import * as ThreadActions from '../thread/thread.actions';
import {
  AppState,
  getCurrentThread,
  getAllThreads
} from '../app.reducer';

@Component({
```

```
  selector: 'chat-threads',
  templateUrl: './chat-threads.component.html',
  styleUrls: ['./chat-threads.component.css']
})
export class ChatThreadsComponent {
  threads: Thread[];
  currentThreadId: string;

  constructor(@Inject(AppStore) private store: Redux.Store<AppState>) {
    store.subscribe(() => this.updateState());
    this.updateState();
  }

  updateState() {
    const state = this.store.getState();

    // Store the threads list
    this.threads = getAllThreads(state);

    // We want to mark the current thread as selected,
    // so we store the currentThreadId as a value
    this.currentThreadId = getCurrentThread(state).id;
  }

  handleThreadClicked(thread: Thread) {
    this.store.dispatch(ThreadActions.selectThread(thread));
  }
}
```

We're storing two instance variables on this component:

- `threads` - the list of `Threads`
- `currentThreadId` - the current thread (conversation) that the user is participating in

In our `constructor` we keep a reference to the Redux store and subscribe to updates. When the store changes, we call `updateState()`.

`updateState()` keeps our instance variables in sync with the Redux store. Notice that we're using two selectors:

- `getAllThreads` and
- `getCurrentThread`

which keep their respective instance variables up to date.

The one new idea we've added is an event handler: `handleThreadClicked`. `handleThreadClicked` will dispatch the `selectThread` action. The idea here is that when a thread is clicked on, we'll tell our store to set this new thread as the selected thread and the rest of the application should update in turn.

`ChatThreadsComponent` template

Let's look at the `ChatThreadsComponent` template and its configuration:

code/redux/redux-chat/src/app/chat-threads/chat-threads.component.html

```
<!-- conversations -->
<div class="row">
  <div class="conversation-wrap">
    <chat-thread
         *ngFor="let thread of threads"
         [thread]="thread"
         [selected]="thread.id === currentThreadId"
         (onThreadSelected)="handleThreadClicked($event)">
    </chat-thread>
  </div>
</div>
```

In our template we're using `ngFor` to iterate over our `threads`. We're using a new directive to render the individual threads called `ChatThreadComponent`.

`ChatThreadComponent` is a *presentational* component. We **won't** be able to access the store in `ChatThreadComponent`, neither for fetching data nor dispatching actions. Instead, we're going to pass everything we need to this component through `inputs` and handle any interaction through `outputs`.

We'll look at the implementation of `ChatThreadComponent` next, but look at the inputs and outputs we have in this template first.

- We're sending the input `[thread]` with the individual `thread`
- On the input `[selected]` we're passing a *boolean* which indicates if this thread (`thread.id`) is the "current" thread (`currentThreadId`)
- If the thread is clicked, we will emit the output event `(onThreadSelected)` - when this happens we'll call `handleThreadClicked()` (which dispatches a thread selected event to the store).

Let's dig in to the `ChatThreadComponent`.

The Single `ChatThreadComponent`

The `ChatThreadComponent` will be used to display a **single thread** in the list of threads. Remember that `ChatThreadComponent` is a *presentational component* - it doesn't manipulate any data that isn't given to it directly.

Here's the component controller code:

code/redux/redux-chat/src/app/chat-thread/chat-thread.component.ts

```
import {
  Component,
  OnInit,
  Input,
  Output,
  EventEmitter
} from '@angular/core';
import { Thread } from '../thread/thread.model';

@Component({
  selector: 'chat-thread',
  templateUrl: './chat-thread.component.html',
  styleUrls: ['./chat-thread.component.css']
})
export class ChatThreadComponent implements OnInit {
  @Input() thread: Thread;
  @Input() selected: boolean;
  @Output() onThreadSelected: EventEmitter<Thread>;

  constructor() {
    this.onThreadSelected = new EventEmitter<Thread>();
  }

  ngOnInit() { }

  clicked(event: any): void {
    this.onThreadSelected.emit(this.thread);
    event.preventDefault();
  }
}
```

The main thing to look at here is the `onThreadSelected` `EventEmitter`. If you haven't used `EventEmitters` much, the idea is that it's an implementation of the observer pattern. We use it as the "output channel" for this component - when we want to send data we call `onThreadSelected.emit` and pass whatever data we want along with it.

In this case, we want to emit the current thread as the argument to the `EventEmitter`. When this element is clicked, we will call `onThreadSelected.emit(this.thread)` which will trigger the callback in our parent (`ChatThreadsComponent`) component.

Here is where we specify our `@Input()`s of `thread` and `selected`, as well as the `@Output()` of `onThreadSelected`.

`ChatThreadComponent` template

Here's the code for our `@Component` decorator and `template`:

code/redux/redux-chat/src/app/chat-thread/chat-thread.component.html

```
<div class="media conversation">
  <div class="pull-left">
    <img class="media-object avatar"
         src="{{thread.avatarSrc}}">
  </div>
  <div class="media-body">
    <h5 class="media-heading contact-name">{{thread.name}}
      <span *ngIf="selected">&bull;</span>
    </h5>
    <small class="message-preview">
      {{thread.messages[thread.messages.length - 1].text}}
    </small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
```

Notice that in our view we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`. In the `message-preview` tag we've got the following:

```
{{ thread.messages[thread.messages.length - 1].text }}
```

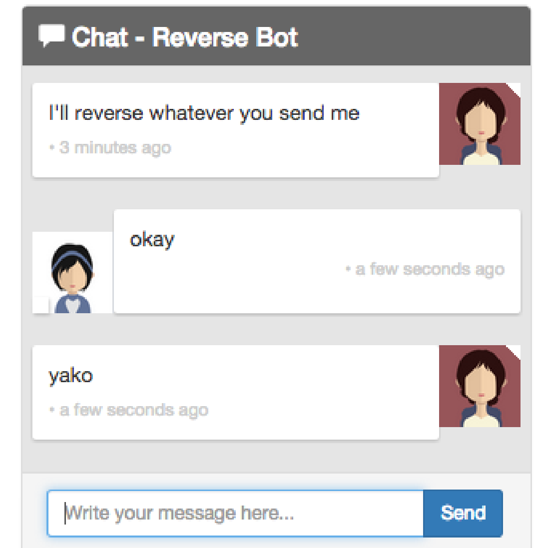
This gets the last message in the thread and displays the text of that message. The idea is we are showing a preview of the most recent message in that thread.

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.

Building the `ChatWindowComponent`

The `ChatWindowComponent` is the most complicated component in our app. Let's take it one section at a time:



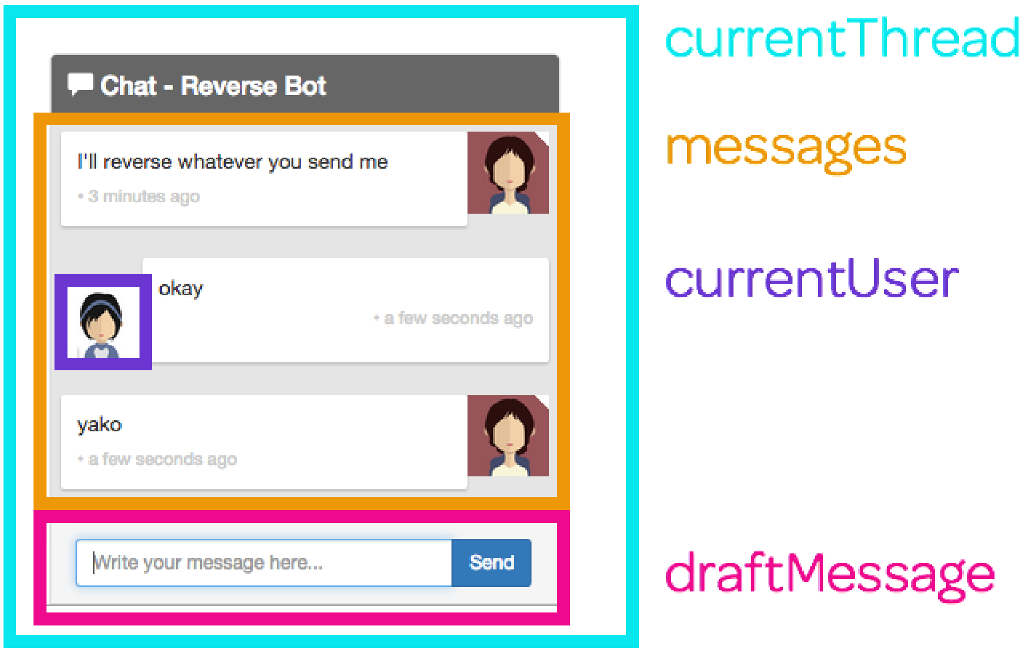
The Chat Window

Our `ChatWindowComponent` class has three properties: `currentThread` (which holds a `Thread` (that contains `Message[]` as a property), `draftMessage`, and `currentUser`:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
export class ChatWindowComponent {
  currentThread: Thread;
  draftMessage: { text: string };
  currentUser: User;
```

Here's a diagram of where each one is used:



Chat Window Properties

In our `constructor` we're going to inject two things:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
constructor(@Inject(AppStore) private store: Redux.Store<AppState>,
            private el: ElementRef) {
  store.subscribe(() => this.updateState() );
  this.updateState();
  this.draftMessage = { text: '' };
}
```

The first is our Redux Store. The second, `el` 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.

In our constructor we subscribe to our store, as we have in our other container components.

The next thing we do is to set a default `draftMessage` with an empty string for the `text`. We'll use the `draftMessage` to keep track of the input box as the user is typing their message.

`ChatWindowComponent updateState()`

When the store changes we will update the instance variables for this component:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
  updateState() {
    const state = this.store.getState();
    this.currentThread = getCurrentThread(state);
    this.currentUser = getCurrentUser(state);
    this.scrollToBottom();
  }
```

Here we store the current thread and the current user. If a new message comes in, we also want to scroll to the bottom of the window. It's a bit coarse to call `scrollToBottom` here, but it's a simple way to make sure that the user doesn't have to scroll manually each time there is a new message (or they switch to a new thread).

`ChatWindowComponent scrollToBottom()`

To scroll to the bottom of the chat window, we're going to use the `ElementRef el` that we saved in the constructor. To make this element scroll, we're going to set the `scrollTop` property of our host element:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
  scrollToBottom(): void {
    const scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    if (scrollPane) {
      setTimeout(() => scrollPane.scrollTop = scrollPane.scrollHeight);
    }
  }
```

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.

`ChatWindowComponent sendMessage`

When we want to send a new message, we'll do it by taking:

- The current thread
- The current user
- The draft message text

And then dispatching a new `addMessage` action on the store. Here's what it looks like in code:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
  sendMessage(): void {
    this.store.dispatch(ThreadActions.addMessage(
      this.currentThread,
      {
        author: this.currentUser,
        isRead: true,
        text: this.draftMessage.text
      }
    ));
    this.draftMessage = { text: '' };
  }
```

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.

After we dispatch the message, we create a new `Message`** and set that new `Message` to `this.draftMessage`. This will clear the search box, and by creating a new object we ensure we don't mutate the message that was sent to the store.

`ChatWindowComponent` `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 both events:

code/redux/redux-chat/src/app/chat-window/chat-window.component.ts

```
  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }
```

> We create this `onEnter` event handler as a separate function from `sendMessage` because `onEnter` will accept an `event` as an argument and then call `event.preventDefault()`. This way we *could* call `sendMessage` in scenarios other than in response to a browser event. In this case, we're not really calling `sendMessage` in any other situation, but I find that it's nice to separate the event handler from the function that 'does the work'.
>
> That is, a `sendMessage` function that also 1. requires an event to be passed to it and 2. handles that event feels like a function that may be handling too many concerns.

Now that we've handled the controller code, let's look at the template

`ChatWindowComponent` template

We start our template by opening the panel tags: and showing the chat name in the header:

code/redux/redux-chat/src/app/chat-window/chat-window.component.html

```
<div class="chat-window-container">
  <div class="chat-window">
    <div class="panel-container">
      <div class="panel panel-default">

        <div class="panel-heading top-bar">
          <div class="panel-title-container">
            <h3 class="panel-title">
              <span class="glyphicon glyphicon-comment"></span>
              Chat - {{currentThread.name}}
            </h3>
          </div>
          <div class="panel-buttons-container"  >
            <!-- you could put minimize or close buttons here -->
          </div>
        </div>

        <div class="panel-body msg-container-base">
          <chat-message
               *ngFor="let message of currentThread.messages"
               [message]="message">
          </chat-message>
        </div>

        <div class="panel-footer">
          <div class="input-group">
            <input type="text"
                   class="chat-input"
                   placeholder="Write your message here..."
                   (keydown.enter)="onEnter($event)"
                   [(ngModel)]="draftMessage.text" />
            <span class="input-group-btn">
              <button class="btn-chat"
                 (click)="onEnter($event)"
                 >Send</button>
            </span>
```

```
        </div>
      </div>

    </div>
  </div>
</div>
</div>
```

Next we show the list of messages. Here we use `ngFor` to iterate over our list of messages. We'll describe the individual `chat-message` component in a minute.

code/redux/redux-chat/src/app/chat-window/chat-window.component.html

```
      <div class="panel-body msg-container-base">
        <chat-message
             *ngFor="let message of currentThread.messages"
             [message]="message">
        </chat-message>
      </div>
```

Lastly we have the message input box and closing tags:

code/redux/redux-chat/src/app/chat-window/chat-window.component.html

```
      <div class="panel-footer">
        <div class="input-group">
          <input type="text"
                 class="chat-input"
                 placeholder="Write your message here..."
                 (keydown.enter)="onEnter($event)"
                 [(ngModel)]="draftMessage.text" />
          <span class="input-group-btn">
            <button class="btn-chat"
               (click)="onEnter($event)"
               >Send</button>
          </span>
        </div>
      </div>

    </div>
  </div>
</div>
```

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/redux/redux-chat/src/app/chat-window/chat-window.component.html

```
            <input type="text"
                   class="chat-input"
                   placeholder="Write your message here..."
                   (keydown.enter)="onEnter($event)"
                   [(ngModel)]="draftMessage.text" />
```

Using `ngModel`

As we've talked about before, we don't generally use two-way data binding as the crux of our data architecture (like we might have in Angular 1). This is particularly true when we're using Redux which is strictly a one-way data flow.

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.

Clicking "Send"

On our "Send" button we bind the `(click)` property to the `onEnter` function of our component:

code/redux/redux-chat/src/app/chat-window/chat-window.component.html

```
              <button class="btn-chat"
                 (click)="onEnter($event)"
                 >Send</button>
```

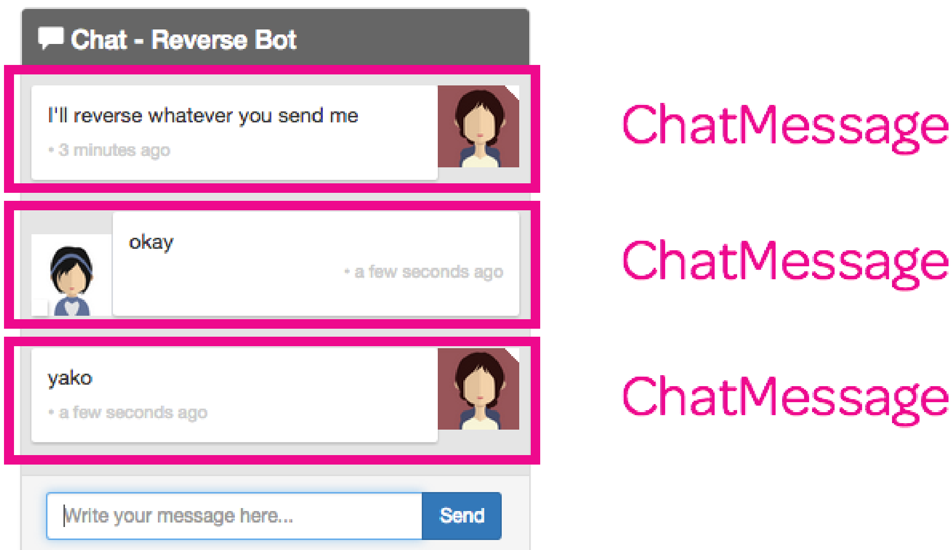
We're using the same `onEnter` function to handle the events which should send the draft message for both the button and hitting the enter button.

The `ChatMessageComponent`

Instead of putting the rendering code for each individual message in this component, instead we're going to create another *presentational component* `ChatMessageComponent`.

Tip: If you're using `ngFor` that's a good indication you should create a new component.

Each `Message` is rendered by the `ChatMessageComponent`.



The `ChatMessageComponent`

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`.

Setting `incoming`

Remember that each `ChatMessageComponent` belongs to one `Message`. So in `ngOnInit` we will set `incoming` depending on if this `Message` was written by the current user:

code/redux/redux-chat/src/app/chat-message/chat-message.component.ts

```
import {
  Component,
  OnInit,
  Input
} from '@angular/core';
import { Message } from '../message/message.model';

@Component({
  selector: 'chat-message',
  templateUrl: './chat-message.component.html',
  styleUrls: ['./chat-message.component.css']
})
export class ChatMessageComponent implements OnInit {
  @Input() message: Message;
  incoming: boolean;

  ngOnInit(): void {
    this.incoming = !this.message.author.isClient;
  }
}
```

The `ChatMessageComponent` template

In our template we have two interesting ideas:

1. the `FromNowPipe`
2. `[ngClass]`

First, here's the code:

code/redux/redux-chat/src/app/chat-message/chat-message.component.html

```
<div class="msg-container"
     [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

  <div class="avatar"
       *ngIf="!incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>

```

```
  <div class="messages"
    [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
    <p>{{message.text}}</p>
    <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
  </div>

  <div class="avatar"
       *ngIf="incoming">
    <img src="{{message.author.avatarSrc}}">
  </div>
</div>
```

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`[128] library. You can read the source of the `FromNowPipe` in `code/redux/redux-chat/src/app/pipes/from-now.pipe.ts`

We also make extensive use of `ngClass` in this view. The idea is, when we say:

```
[ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

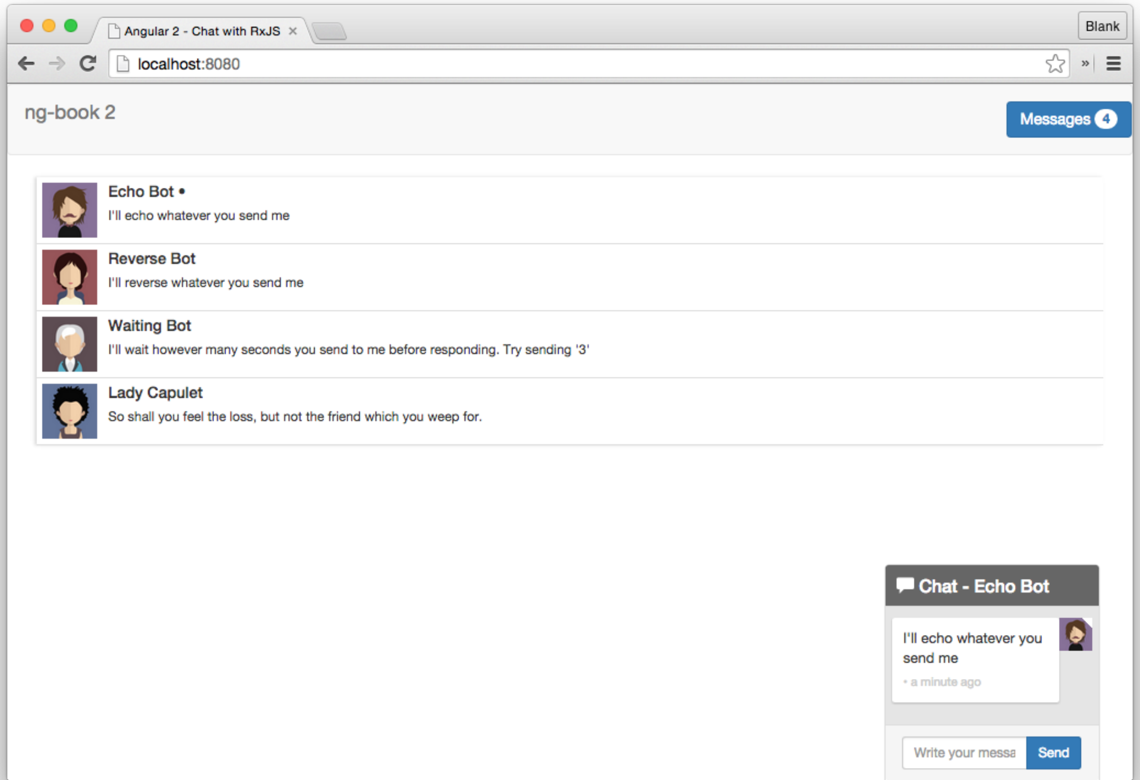
We're asking Angular to apply the `msg-receive` class if `incoming` is truthy (and apply `msg-sent` if `incoming` is falsey).

By using the `incoming` property, we're able to display incoming and outgoing messages differently.

Summary

There we go, if we put them all together we've got a fully functional chat app!

[128] http://momentjs.com/



Completed Chat Application

If you checkout `code/redux/redux-chat/src/app/data/chat-example-data.ts` you'll see we've written a handful of bots for you that you can chat with. Checkout the code and try writing a few bots of your own!

Advanced Components

Throughout this book, we've learned how to use Angular's built-in directives 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 *content projection*
- Accessing neighbor directives
- Using lifecycle hooks
- Detecting changes

How to Use This Chapter

This chapter gives a tour of advanced Angular APIs. It's assumed the reader is familiar with the basics of creating components, using built-in directives, and organizing component files.

As this is an intermediate/advanced level chapter, it's assumed the reader is able to fill in some of the basics (such as importing dependencies).

This chapter comes with runnable code, found in the `advanced-components` folder. If at any time you feel you're lacking context, checkout the example code for this chapter.

To run the demos in this chapter, change into the project folder and run:

```
npm install
npm start
```

Then open your browser to http://localhost:4200

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` decorator.

Let's write a component that uses inline styles:

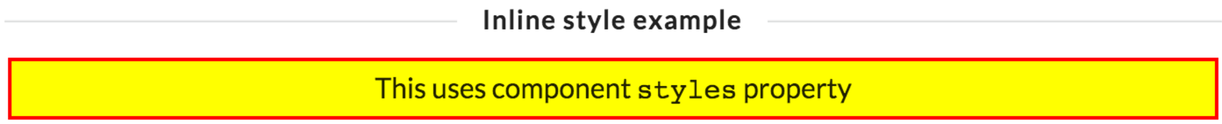
code/advanced-components/src/app/styling/inline-style/inline-style.component.ts

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

@Component({
  selector: 'app-inline-style',
  styles: [`
  .highlight {
    border: 2px solid red;
    background-color: yellow;
    text-align: center;
    margin-bottom: 20px;
  }
  `],
  template: `
  <h4 class="ui horizontal divider header">
    Inline style example
  </h4>

  <div class="highlight">
    This uses component <code>styles</code>
    property
  </div>
  `
})
export class InlineStyleComponent {
}
```

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:



Example of component using styles

Another way to declare CSS classes is to use the `styleUrls` property. This allows us to declare our CSS in 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/src/app/styling/external-style/external-style.component.css

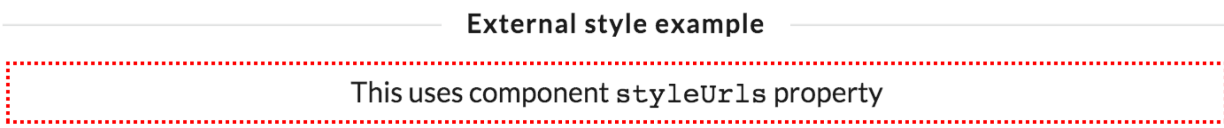
```
.highlight {
  border: 2px dotted red;
  text-align: center;
  margin-bottom: 20px;
}
```

Then we can write the code that references it:

code/advanced-components/src/app/styling/external-style/external-style.component.ts

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

@Component({
  selector: 'app-external-style',
  styleUrls: ['./external-style.component.css'],
  template: `
<h4 class="ui horizontal divider header">
  External style example
</h4>

<div class="highlight">
  This uses component <code>styleUrls</code>
  property
</div>
`
})
export class ExternalStyleComponent {
}
```

And when we load the page, we see our div with a dotted border:

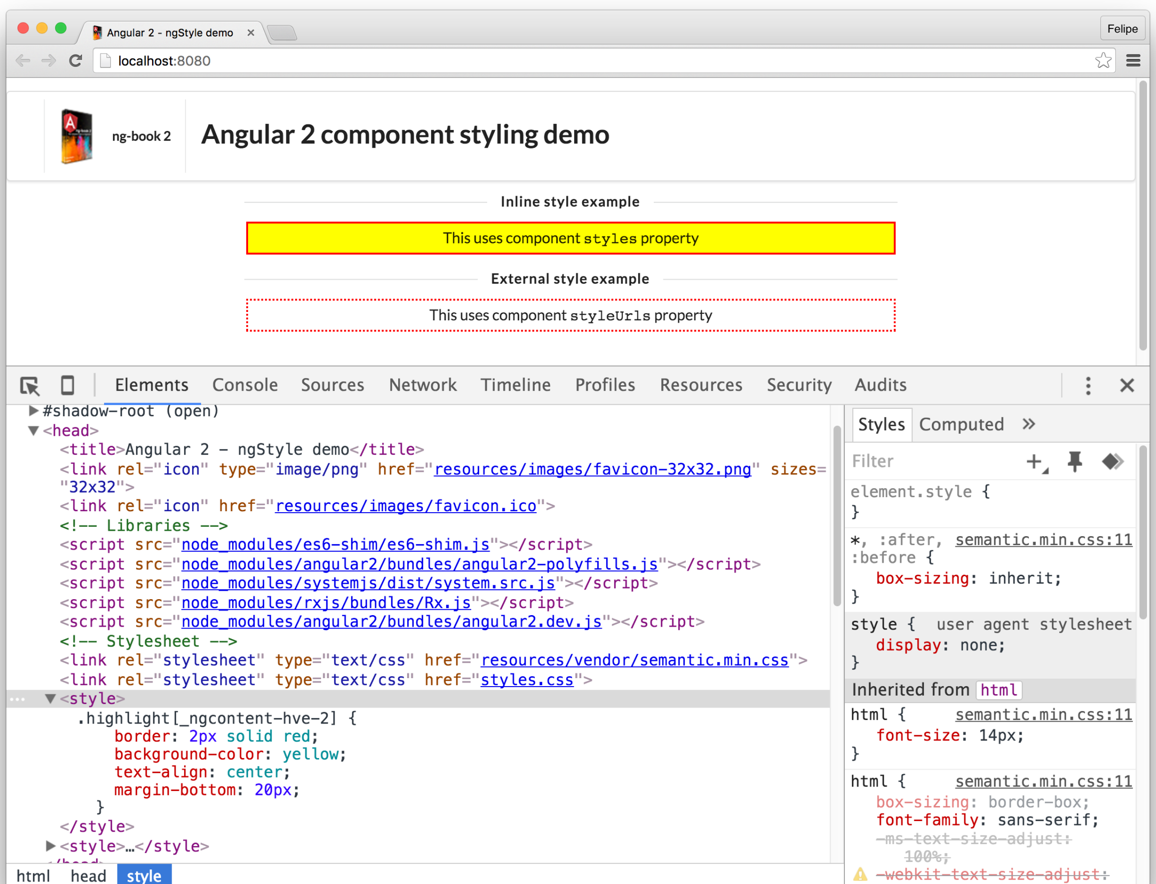


Example of component using styleUrls

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:

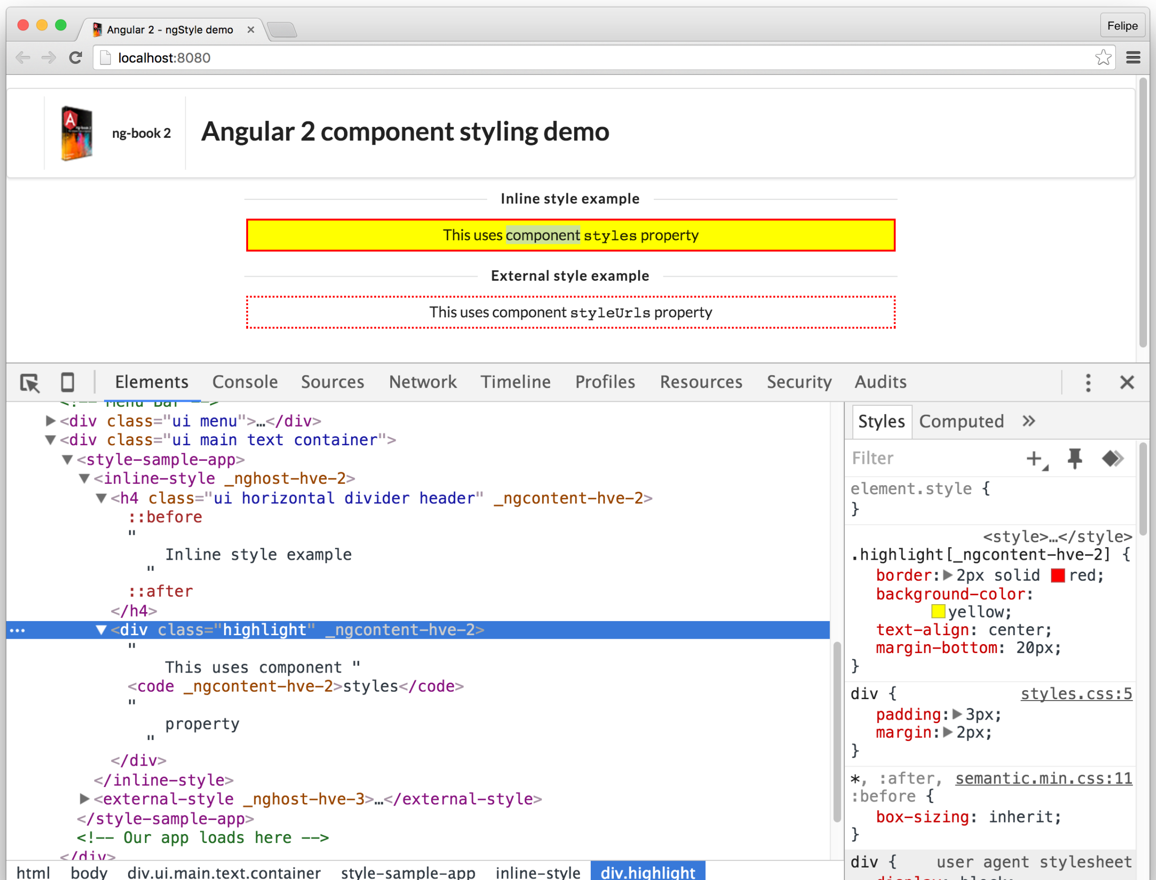


Injected style

You'll also notice that the CSS class has been scoped with `_ngcontent-hve-2`:

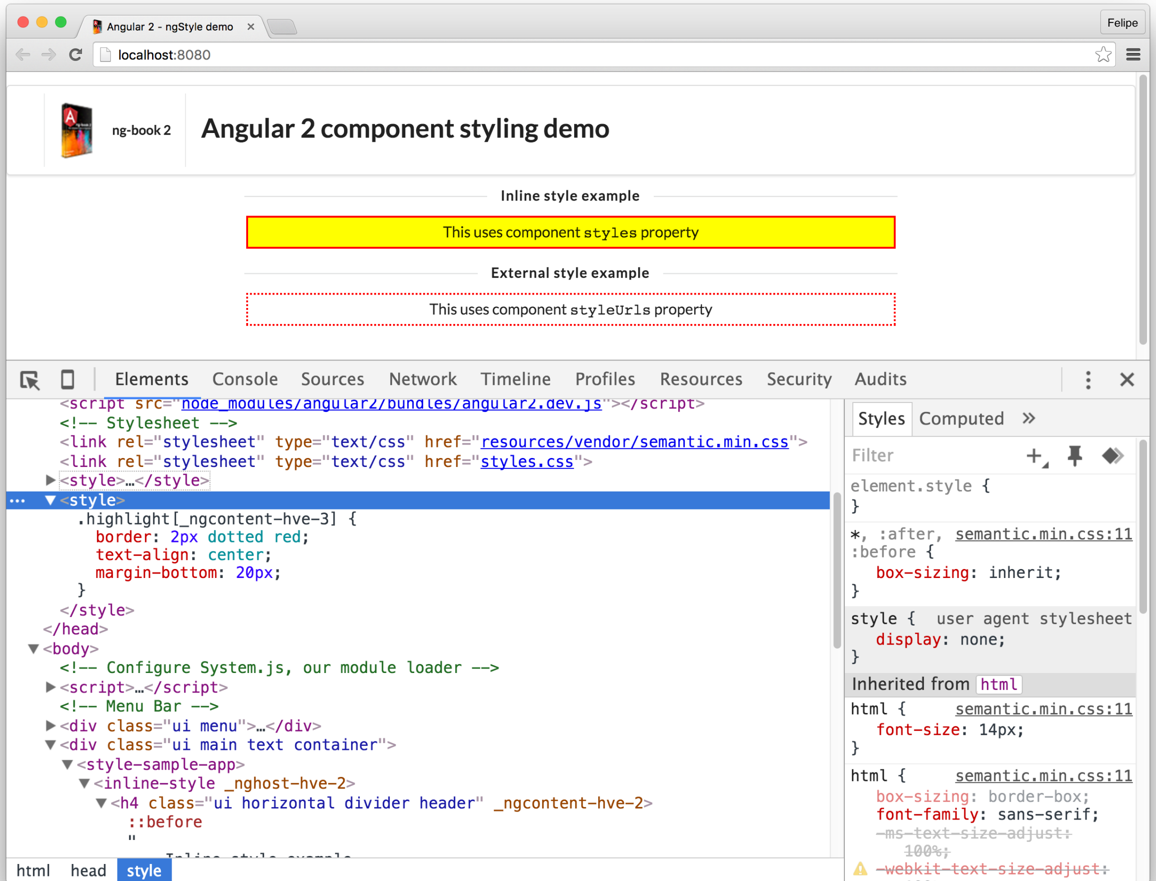
```
.highlight[\_ngcontent-hve-2] {
  border: 2px solid red;
  background-color: yellow;
  text-align: center;
  margin-bottom: 20px;
}
```

And if we check how our <div> is rendered, you'll find that _ng-content-hve-2 was added:



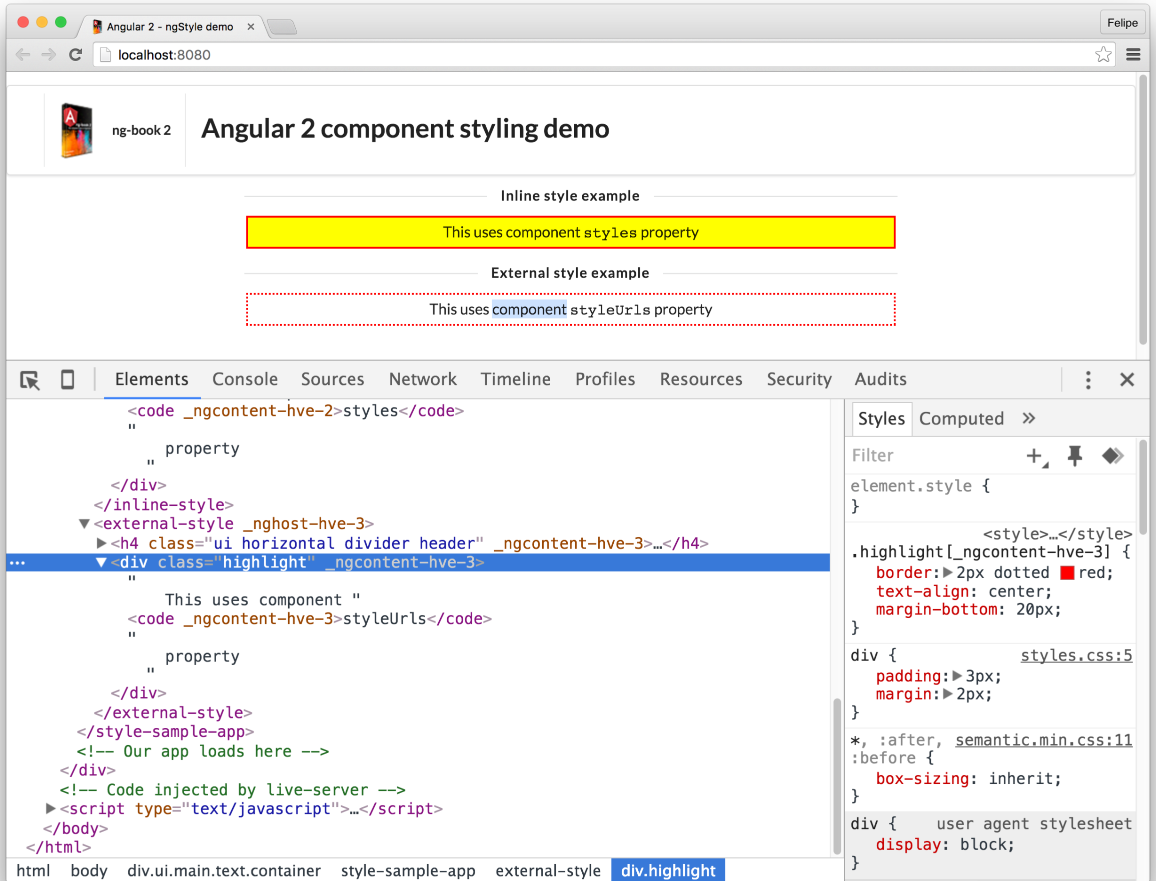
Injected style

The same thing happens for our external style:



External style

and:



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 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[129].

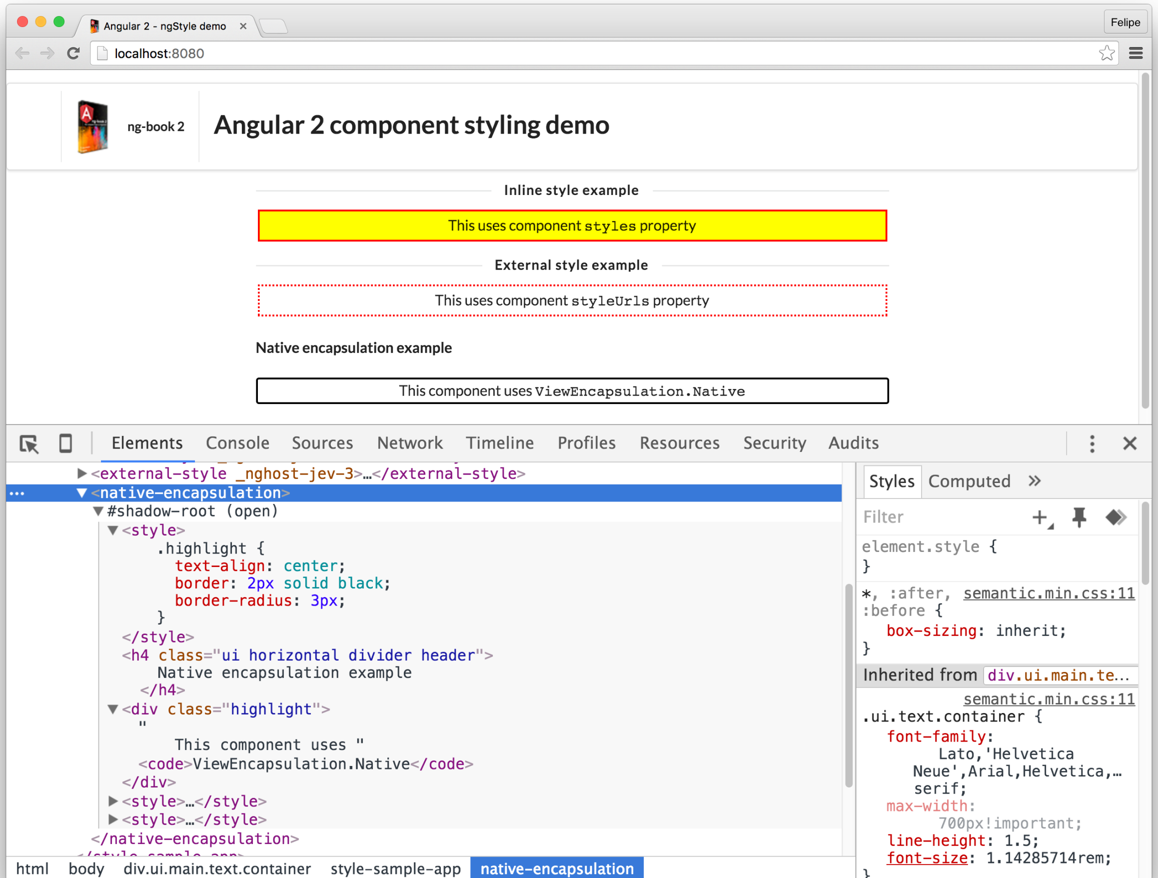
Let's create another component that uses the **Native** encapsulation (Shadow DOM) to understand how this works:

code/advanced-components/src/app/styling/native-encapsulation/native-encapsulation.component.ts

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

@Component({
  selector: 'app-native-encapsulation',
  styles: [`
  .highlight {
    text-align: center;
    border: 2px solid black;
    border-radius: 3px;
    margin-botton: 20px;
  }`],
  template: `
  <h4 class="ui horizontal divider header">
    Native encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.Native</code>
  </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
export class NativeEncapsulationComponent {
}
```

In this case, if we inspect the source code, we'll see:

[129] http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/



Native encapsulation

Everything inside the `#shadow-root` element has been encapsulated and isolated from the rest of the page.

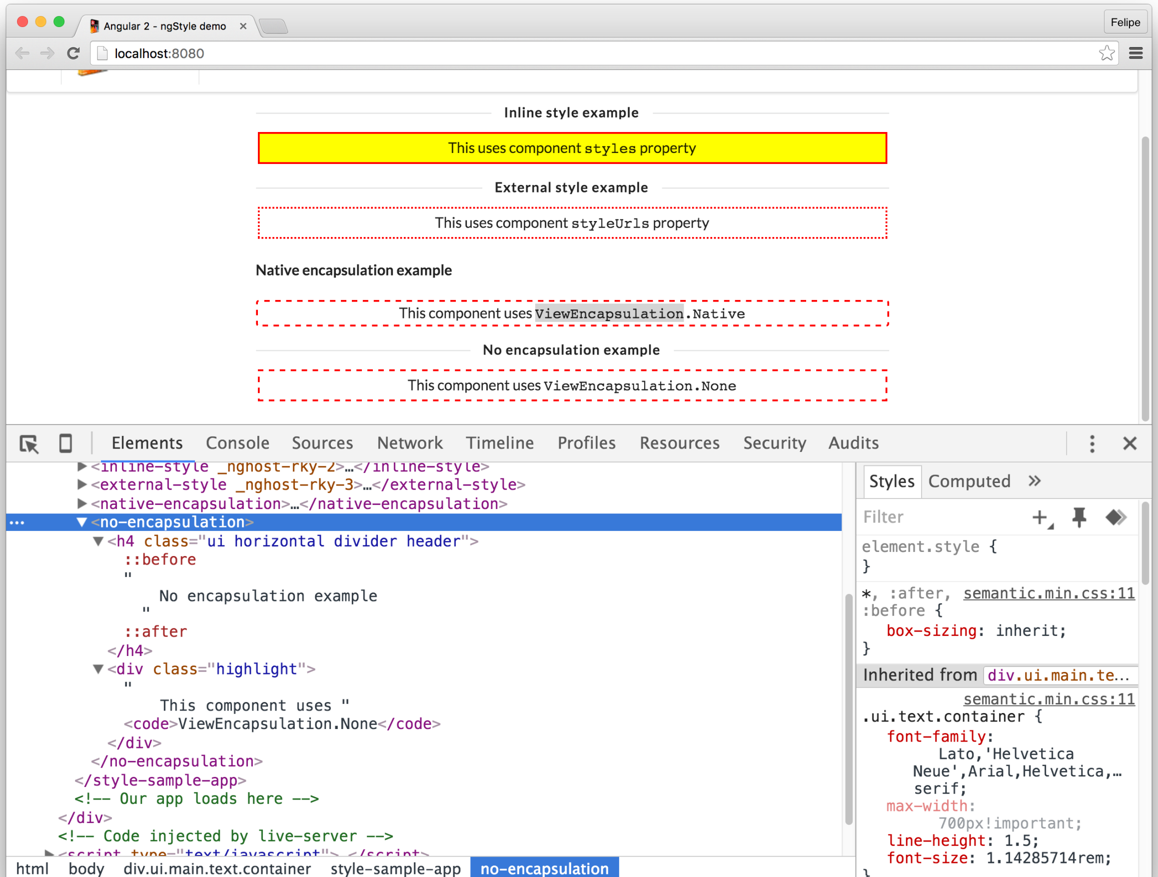
No Encapsulation

Finally, if we create a component that specifies `ViewEncapsulation.None`, no style encapsulation will be added:

code/advanced-components/src/app/styling/no-encapsulation/no-encapsulation.component.ts

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

@Component({
  selector: 'app-no-encapsulation',
  styles: [`
  .highlight {
    border: 2px dashed red;
    text-align: center;
    margin-bottom: 20px;
  }
  `],
  template: `
  <h4 class="ui horizontal divider header">
    No encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.None</code>
  </div>
  `,
  encapsulation: ViewEncapsulation.None
})
export class NoEncapsulationComponent {
}
```

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:

```
.highlight {
  border: 2px dashed red;
  text-align: center;
  margin-bottom: 20px;
}
```

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 `<app-no-encapsulation></app-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. Sometimes we have a 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

Now let's 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:

```
<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/src/app/host/popup-demo/steps/host-1.ts

```
@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  constructor() {
    console.log('Directive bound');
  }
}
```

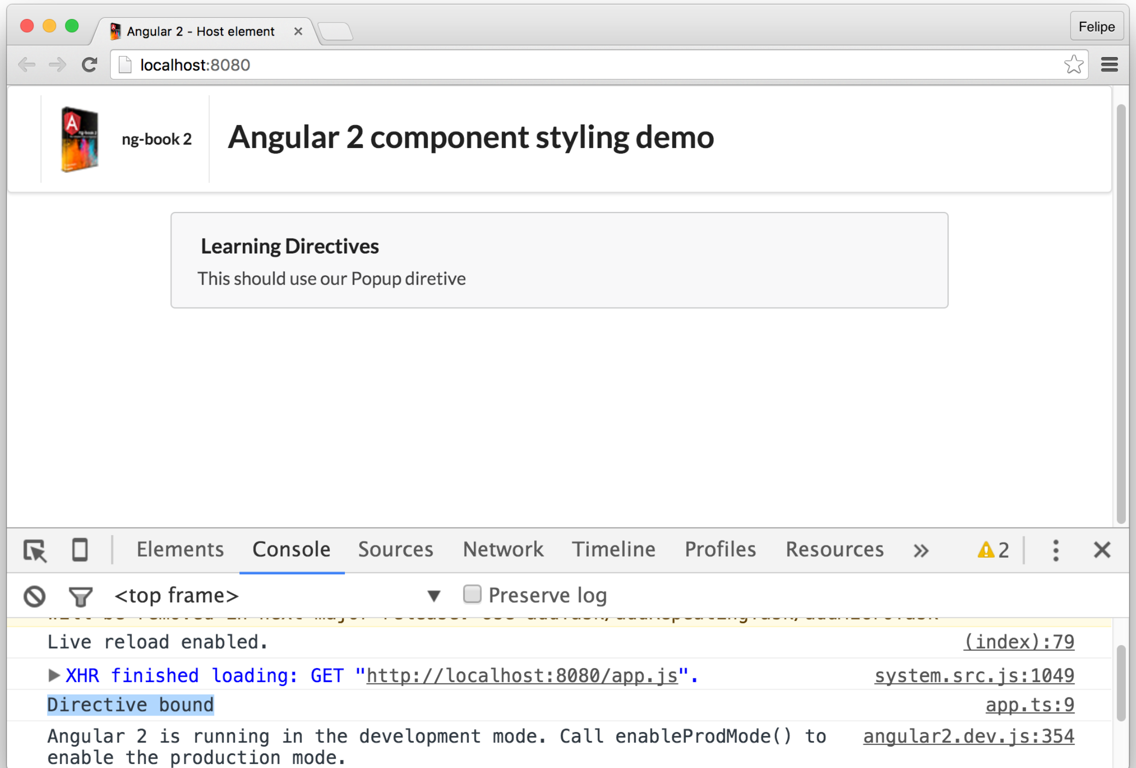
We use the `Directive` decorator and set the `selector` option to `[popup]`. This will make this directive bind to any elements that define the popup attribute.

Now let's create an app that has an element that has the `popup` attribute:

code/advanced-components/src/app/host/popup-demo/steps/host-1.ts

```
@Component({
  selector: 'app-popup-demo',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>
  `
})
export class PopupDemoComponent1 {
}
```

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:



Binding to host element

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/src/app/host/popup-demo/steps/host-2.ts

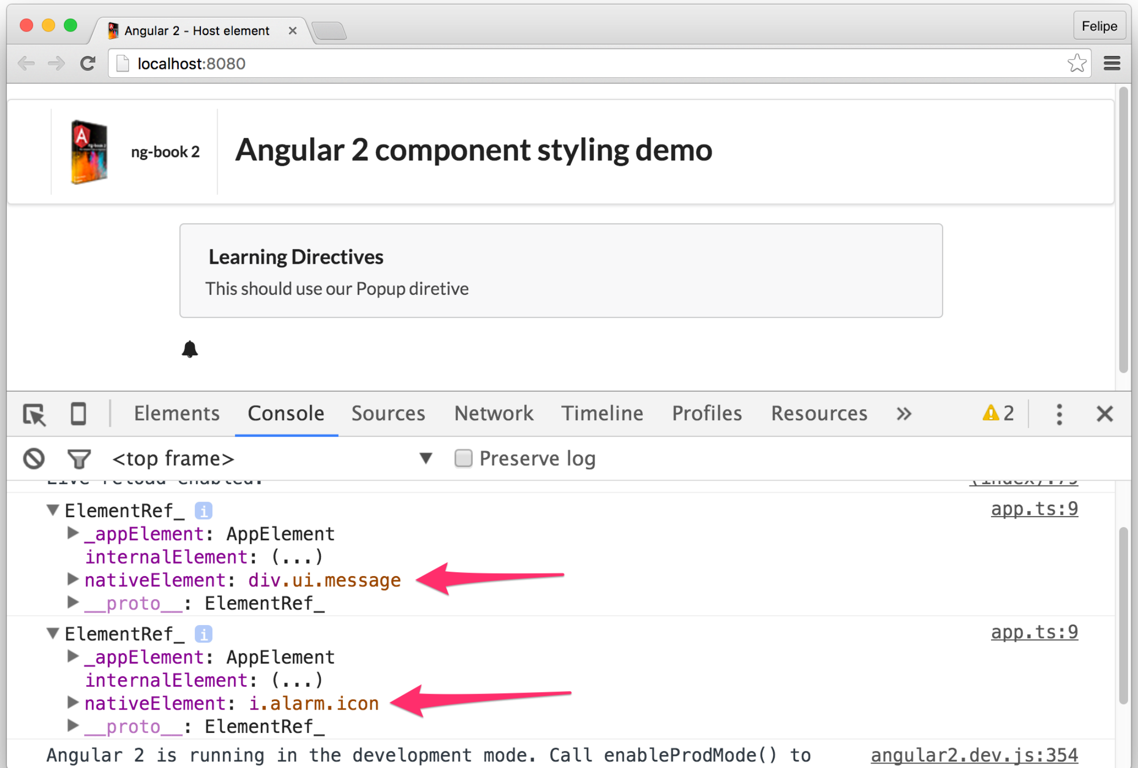
```
import { Component, Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }
}
```

We can also add a second element to the page that uses our directive, so we can see two different `ElementRefs` logged to the console:

code/advanced-components/src/app/host/popup-demo/steps/host-2.ts

```
@Component({
  selector: 'app-pop-demo',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup></i>
  `
})
export class PopupDemoComponent2 {
}
```

When we run our app now, we can see two different `ElementRefs`: 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'll do something very similar, but the syntax is different. In order to bind the directive to a host's `click` event, we're going to use the decorator `HostListener`.

The `HostListener` decorator allows a directive to listen to events on its host element.

We'll do this by decorating a function on the component with the `@HostListener()` decoration.

We also want the host element to define what message will pop up when the element is clicked, using the `message` attribute.

First, let's add an `inputs` attribute to the directive. We'll do this by importing `Input` and using the `@Input` decorator with the property we will use for this input:

```
import { Component, Input } from '@angular/core';
...
class Popup {
  @Input() message: String;
  ...
}
```

We're saying that we're having a property with the name `message` and expect to receive an input with the same name.

Then, let's add the `HostListener` decoration. We'll do this by adding `@HostLisener('click')` on the function we want to call when the host is clicked:

code/advanced-components/src/app/host/popup-demo/steps/host-3.ts

```
  HostListener
} from '@angular/core';

@Directive({
  selector: '[popup]'
})
export class PopupDirective {
  @Input() message: String;

  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }

  @HostListener('click') displayMessage(): void {
    alert(this.message);
  }
}
```

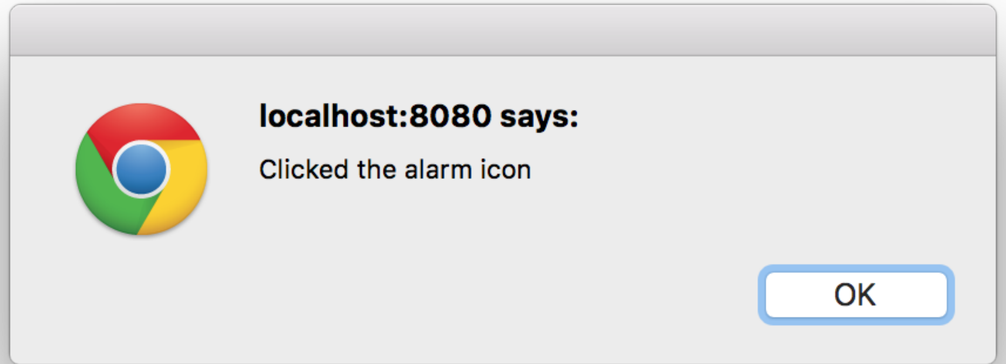
Then when the host element is clicked we'll call the directive's `displayMessage` method, which will display the message the host element defines.

And finally, we need to change our app template a bit to add the message we want displayed for each element:

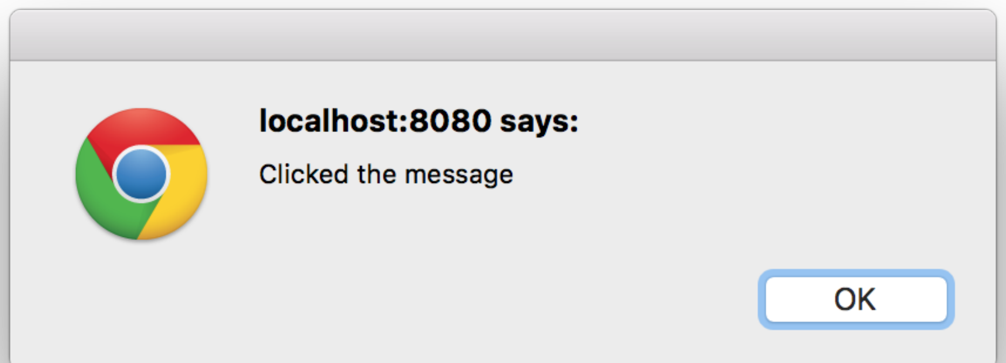
code/advanced-components/src/app/host/popup-demo/steps/host-3.ts

```
@Component({
  selector: 'app-popup-demo',
  template: `
  <div class="ui message" popup
       message="Clicked the message">
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup
     message="Clicked the alarm icon"></i>
  `
})
export class PopupDemoComponent3 {
}
```

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 reference variable**. 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/src/app/host/popup-demo/steps/host-4.ts

```
@Directive({
  selector: '[popup]',
  exportAs: 'popup',
})
export class PopupDirective {
  @Input() message: String;

  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }

  @HostListener('click') displayMessage(): void {
    alert(this.message);
  }
}
```

And now we need to change the two elements to export the template reference:

code/advanced-components/src/app/host/popup-demo/steps/host-4.ts

```
  template: `
  <div class="ui message" popup #popup1="popup"
       message="Clicked the message">
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup #popup2="popup"
     message="Clicked the alarm icon"></i>
```

See that we used the template var `#popup1` for the `div.message` and `#popup2` for the `icon`.

Now let's add two buttons, one to trigger each popup:

code/advanced-components/src/app/host/popup-demo/steps/host-4.ts

```
  <div style="margin-top: 20px;">
    <button (click)="popup1.displayMessage()" class="ui button">
      Display popup for message element
    </button>

    <button (click)="popup2.displayMessage()" class="ui button">
      Display popup for alarm icon
    </button>
  </div>
```

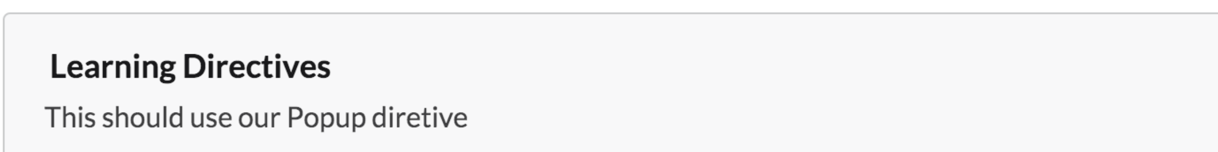
Now reload the page and click each of the buttons and each message will appear as expected.

Creating a Message Pane with Content Projection

Sometimes when we are creating components we want to pass inner markup as an argument to the component. This technique is called *content projection*. The idea is that it lets us specify a bit of markup that will be expanded into a bigger template.

Angular 1 dug deep in the dictionary and called this *transclusion*.

Let's create a new directive that will render a nicely styled message like this:



Popup 1

Our goal is to write markup like this:

```
<div message header="My Message">
  This is the content of the message
</div>
```

Which will render into the more complicated HTML like:

```
<div class="ui message">
  <div class="header">
    My Message
  </div>

  <p>
    This is the content of the message
  </p>
</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's CSS

To add attributes to the host element, we use a new decorator, similar to when we listened to events on the host: the `HostBinding` decorator. But now, instead of specifying the event name we want to listen for, we'll define the attribute name we want to 'bind' to. In this component, it looks like this:

```
@HostBinding('attr.class') cssClass = 'ui message';
```

This decoration tells angular that we want the value of `cssClass` to be kept in sync with the host's attribute `class`.

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/src/app/content-projection/content-projection-demo/messageo.component.ts

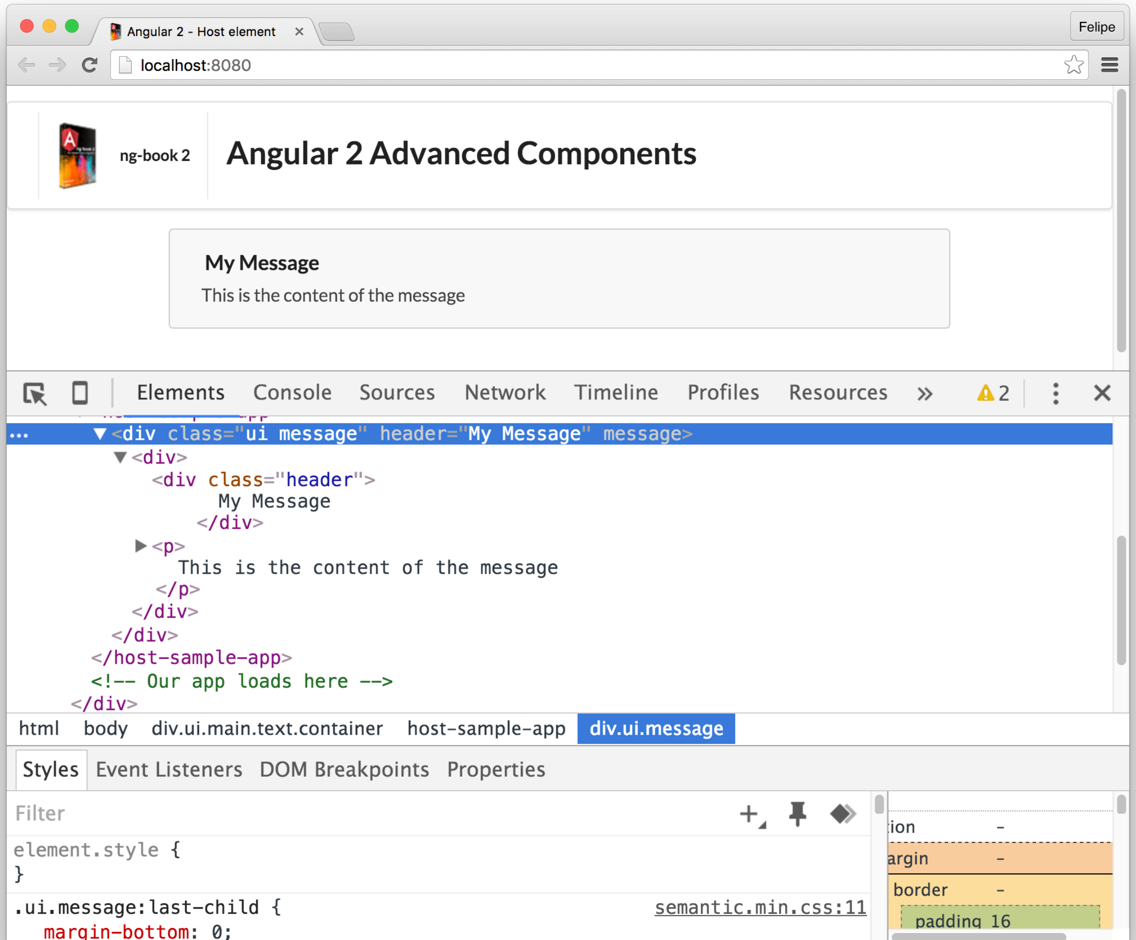
```
/* tslint:disable:component-selector */
import {
  Component,
  OnInit,
  Input,
  HostBinding
 } from '@angular/core';

@Component({
  selector: '[app-message]',
```

```
  template: `
    <div class="header">
      {{ header }}
    </div>
    <p>
      <ng-content></ng-content>
    </p>
  `
})
export class MessageComponent implements OnInit {
  @Input() header: string;
  @HostBinding('attr.class') cssClass = 'ui message';

  ngOnInit(): void {
    console.log('header', this.header);
  }
}
```

A few highlights:

- We use the `@Input` decorator to indicate we want to receive a `header` 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 project 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:



projected 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 *tabset*, 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 in a way that we'll be able to use the following markup:

```
<tabset>
  <tab title="Tab 1">Tab 1</tab>
  <tab title="Tab 2">Tab 2</tab>
  ...
</tabset>
```

We're going to use Semantic UI Tab styles[130] to render the tabs.

ContentTabComponent

Let's start by writing the `ContentTabComponent`

code/advanced-components/src/app/tabs/content-tabs-demo/content-tab.component.ts

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

@Component({
  selector: 'tab',
  templateUrl: './content-tab.component.html'
})
export class ContentTabComponent implements OnInit {
  @Input() title: string;
  active = false;
  name: string;

  constructor() { }

  ngOnInit() { }
}
```

and the template:

[130] http://semantic-ui.com/modules/tab.html#/examples

code/advanced-components/src/app/tabs/content-tabs-demo/content-tab.component.html

```
<div class="ui bottom attached tab segment"
  [class.active]="active">

  <ng-content></ng-content>

</div>
```

There are not many new concepts here. We're declaring a component that will use the `ContentTabComponent` selector, and it will allow a `title` input to be set.

Then we're rendering a `<div>` and using the content projection concept we learned in 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')` decorator we added to the `title` property. This decorator is a way to ask Angular to automatically bind the value of the *input* `title` into the *property* `title`.

`ContentTabsetComponent` Component

Now let's move on to the `ContentTabsetComponent` component that will be used to wrap the tabs:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabset.component.ts

```
import {
  Component,
  AfterContentInit,
  QueryList,
  ContentChildren
} from '@angular/core';

import { ContentTabComponent } from './content-tab.component';

@Component({
  selector: 'tabset',
  templateUrl: './content-tabset.component.html'
})
export class ContentTabsetComponent implements AfterContentInit {
  @ContentChildren(ContentTabComponent) tabs: QueryList<ContentTabComponent>;

  ngAfterContentInit(): void {
    this.tabs.toArray()[0].active = true;
  }

```

```
  setActive(tab: ContentTabComponent): void {
    this.tabs.toArray().forEach((t) => t.active = false);
    tab.active = true;
  }

  constructor() { }
}
```

and the template:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabset.component.html

```
<div class="ui top attached tabular menu">
  <a *ngFor="let tab of tabs"
     class="item"
     [class.active]="tab.active"
     (click)="setActive(tab)">

    {{ tab.title }}

  </a>
</div>
<ng-content></ng-content>
```

Let's break down the implementation so we can learn about the new concepts it introduces.

ContentTabsetComponent @Component Decorator

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.

ContentTabsetComponent class

Now let's turn our attention to the ContentTabsetComponent class. The first new idea we see here is that the ContentTabsetComponent 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 have been initialized.

`ContentTabsetComponent` `ContentChildren` and `QueryList`

Next thing we do is declare the `tabs` property that will hold every `ContentTabComponent` component we declare inside the `ContentTabsetComponent`. Notice that instead of declaring this list as an array of `ContentTabComponents`, we use the class `QueryList`, passing a generic of `ContentTabComponent`. Why is this?

`QueryList` is a class provided by Angular and when we use `QueryList` with a `ContentChildren` Angular populates this with the **components that match the query** and then **keeps the items up to date** if the state of the application changes.

However, `QueryList` requires a `ContentChildren` to populate it, so let's take a look at that now.

On the `tabs` instance variable, we add the `@ContentChildren(Tab)` decorator. This decorator will tell Angular to inject all the direct child directives (of the `ContentTabComponent` 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 `ContentTabComponent` components**.

Initializing the `ContentTabsetComponent`

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 Angular's `QueryList` into a native TypeScript array.

`ContentTabsetComponent` `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 to active.

Using the `ContentTabsetComponent`

Now the next step is to code the application component that makes use of both of the components we created. Here's how we write the component:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabs-demo.component.ts

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

@Component({
  selector: 'app-content-tabs-demo',
  templateUrl: './content-tabs-demo.component.html'
})
export class ContentTabsDemoComponent implements OnInit {
  tabs: any;
```

```

  constructor() { }

  ngOnInit() {
    this.tabs = [
      { title: 'About', content: 'This is the About tab' },
      { title: 'Blog', content: 'This is our blog' },
      { title: 'Contact us', content: 'Contact us here' },
    ];
  }

}
```

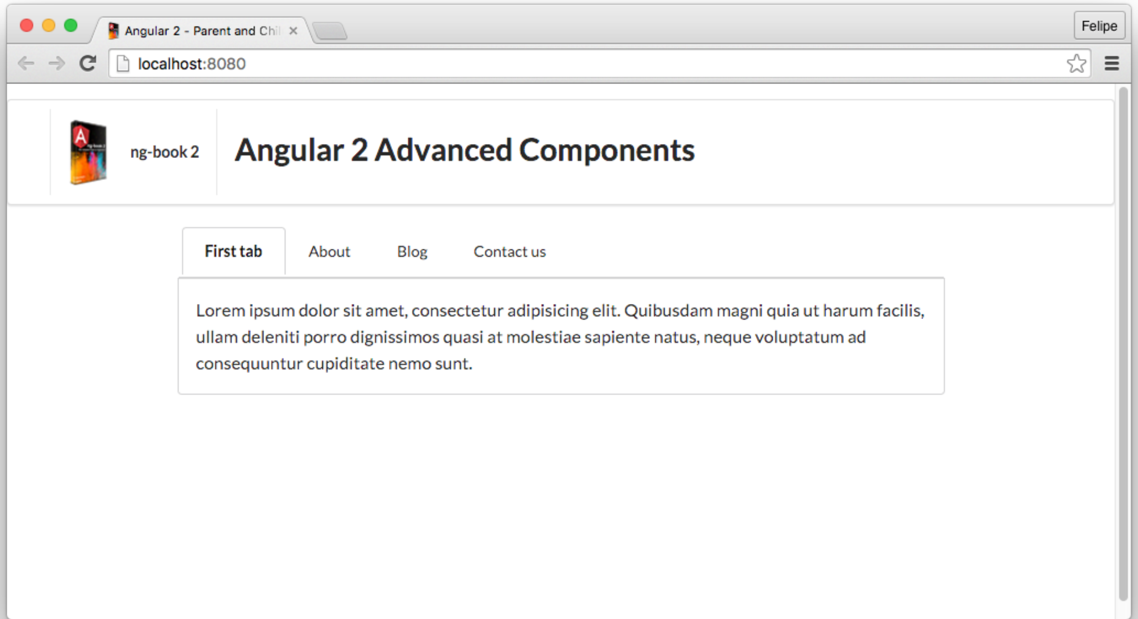
and template:

code/advanced-components/src/app/tabs/content-tabs-demo/content-tabs-demo.component.html

```
<tabset>
  <tab title="First tab">
    Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    Quibusdam magni quia ut harum facilis, ullam deleniti porro
    dignissimos quasi at molestiae sapiente natus, neque voluptatum
    ad consequuntur cupiditate nemo sunt.
  </tab>

  <tab
     *ngFor="let tab of tabs"
     [title]="tab.title">
    {{ tab.content }}
  </tab>
</tabset>
```

We're declaring that we're using `tabs-sample-app` as our component's selector and using the `ContentTabsetComponent` and `ContentTabComponent` components.

On the template we then create a `ContentTabsetComponent` 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 or component 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 or component 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.

> It is actually not mandatory for the class to implement the interface, one could just create the method of the hook. But it is considered good practice[131] and has benefits from strong typing and editor tooling.

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/src/app/lifecycle/on-init/on-init.component.ts

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

@Component({
  selector: 'app-on-init',
  template: `
  <div class="ui label">
    <i class="cubes icon"></i> Init/Destroy
  </div>
  `
})
export class OnInitComponent implements OnInit, OnDestroy {
  constructor() { }

```

[131] https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

```
  ngOnInit(): void {
    console.log('On init');
  }

  ngOnDestroy(): void {
    console.log('On destroy');
  }
}
```

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/src/app/lifecycle/on-init/on-init-demo.component.ts

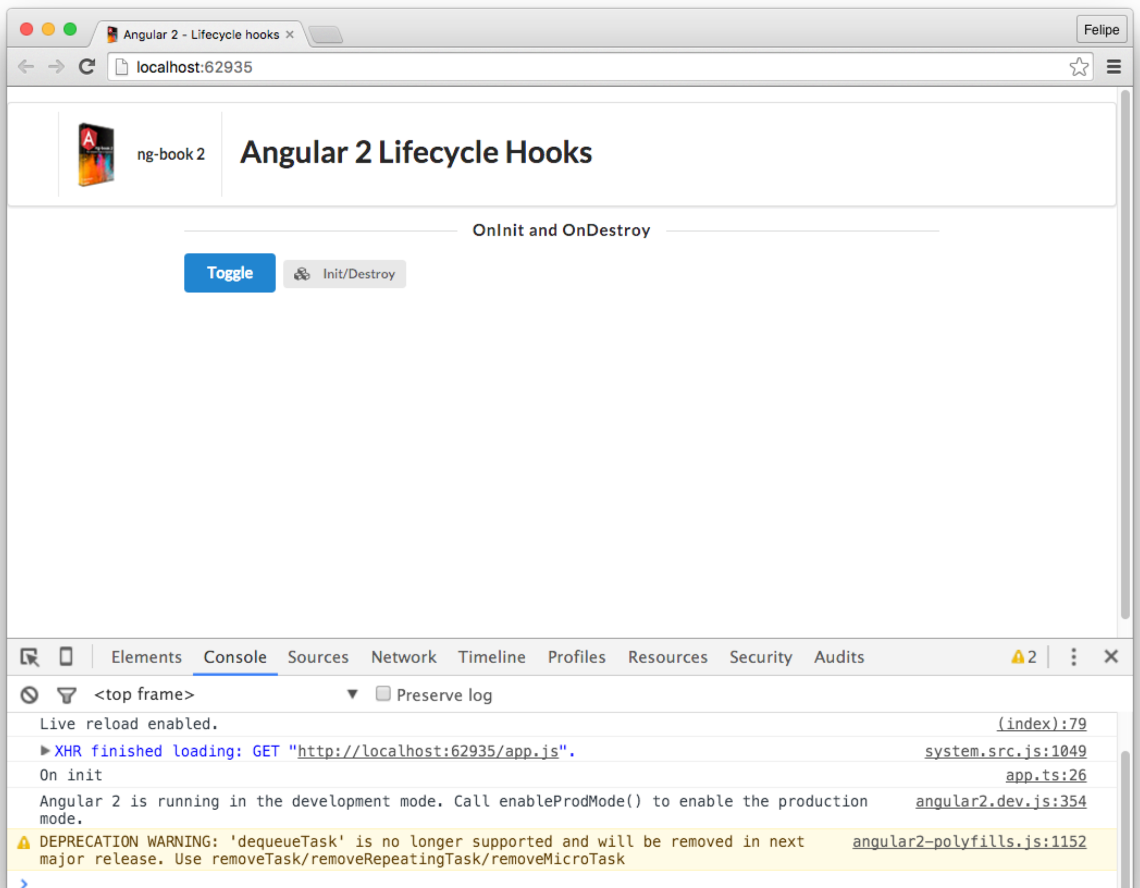
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-on-init-demo',
  templateUrl: './on-init-demo.component.html'
})
export class OnInitDemoComponent {
  display: boolean;

  constructor() {
    this.display = true;
  }

  toggle(): void {
    this.display = !this.display;
  }
}
```

and the template:

code/advanced-components/src/app/lifecycle/on-init/on-init-demo.component.html

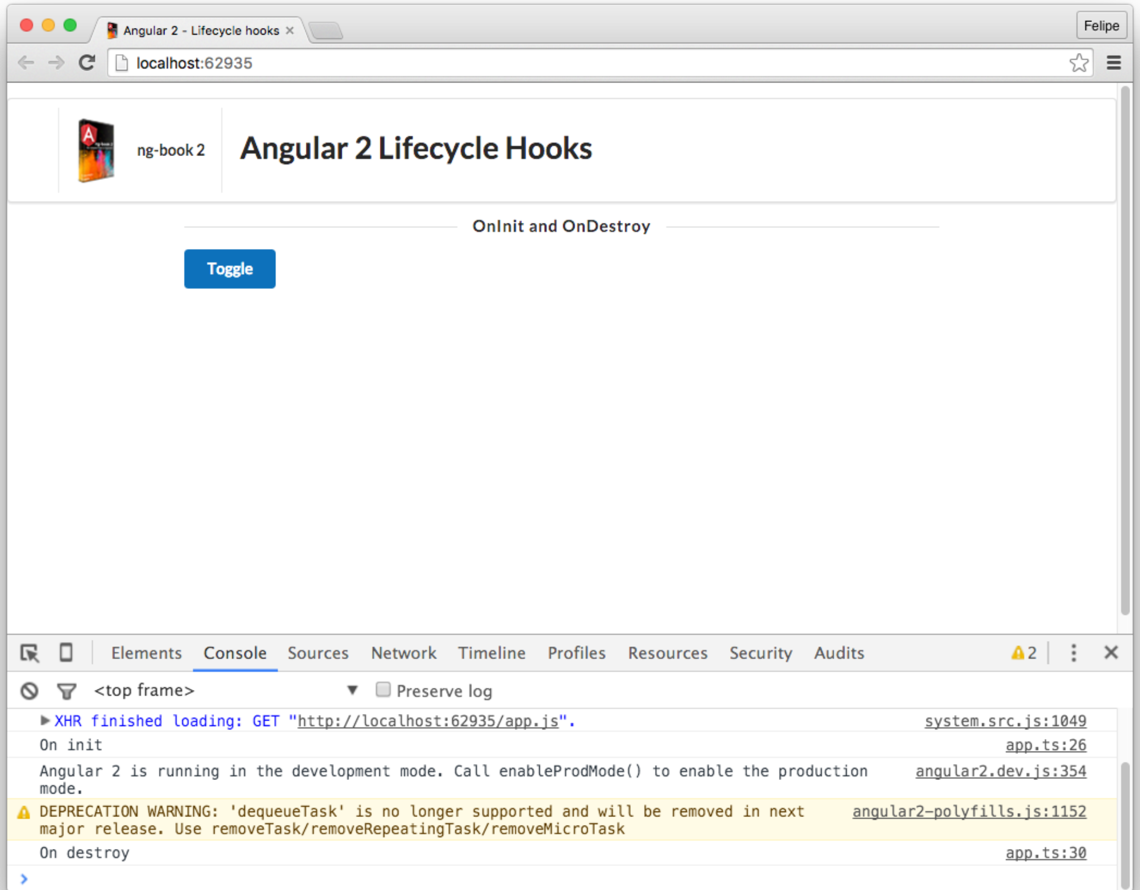
```
<h4 class="ui horizontal divider header">
  OnInit and OnDestroy
</h4>

<button class="ui primary button" (click)="toggle()">
  Toggle
</button>
<app-on-init *ngIf="display"></app-on-init>
```

When we first run the application, we can see that the `OnInit` hook was called when the component was first instantiated:



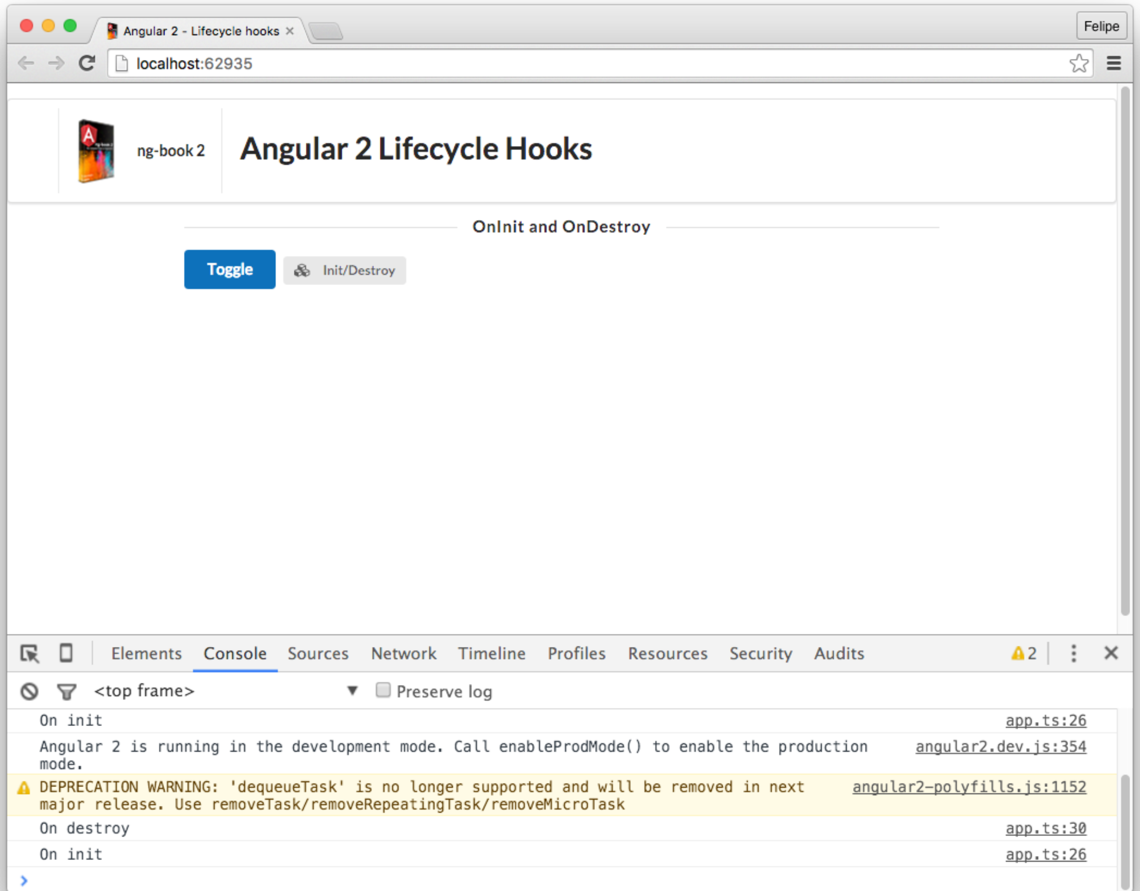
Initial state of our component

When I click the `Toggle` button for the first time, the component is destroyed and the hook is called as expected:



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 has two inputs: *name* and *comment*:

code/advanced-components/src/app/lifecycle/on-changes/on-changes.component.ts

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

@Component({
  selector: 'app-on-changes',
  templateUrl: './on-changes.component.html'
})
export class OnChangesComponent implements OnChanges {
  @Input('name') name: string;
  @Input('comment') comment: string;

  ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
    console.log('Changes', changes);
  }
}
```

and template:

code/advanced-components/src/app/lifecycle/on-changes/on-changes.component.html

```
<div class="ui comments">
  <div class="comment">
    <a class="avatar">
      <img src="assets/images/avatars/matt.jpg">
    </a>
    <div class="content">
      <a class="author">{{name}}</a>
      <div class="text">
        {{comment}}
      </div>
    </div>
  </div>
</div>
```

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/src/app/lifecycle/on-changes/on-changes.component.ts

```
  ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
    console.log('Changes', changes);
  }
```

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 `SimpleChange` 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:

```
{
  name: {
    currentValue: 'new name value',
    previousValue: 'old name value'
  },
  comment: {
    currentValue: 'new comment value',
    previousValue: 'old comment value'
  }
}
```

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/src/app/lifecycle/on-changes/on-changes-demo.component.ts

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

@Component({
  selector: 'app-on-changes-demo',
  templateUrl: './on-changes-demo.component.html',
  styles: []
})
export class OnChangesDemoComponent implements OnInit {
  display: boolean;
  name: string;
  comment: string;

  constructor() { }

  ngOnInit() {
```

```
    this.display = true;
    this.name = 'Felipe Coury';
    this.comment = 'I am learning so much!';
  }

  setValues(namefld, commentfld): void {
    this.name = namefld.value;
    this.comment = commentfld.value;
  }

  toggle(): void {
    this.display = !this.display;
  }

  }
```

and template:

code/advanced-components/src/app/lifecycle/on-changes/on-changes-demo.component.html

```
<h4 class="ui horizontal divider header">
  OnChanges
</h4>

<div class="ui form">
  <div class="field">
    <label>Name</label>
    <input
      type="text"
      #namefld
      value="{{name}}"
     (keyup)="setValues(namefld, commentfld)">
  </div>

  <div class="field">
    <label>Comment</label>
    <textarea
      #commentfld
      (keyup)="setValues(namefld, commentfld)"
      rows="2">{{comment}}</textarea>
  </div>
</div>

<app-on-changes
```

```
  [name]="name"
  [comment]="comment"
  ></app-on-changes>
```

Note the important pieces we added to the template. We declare a new form with name and comment fields.

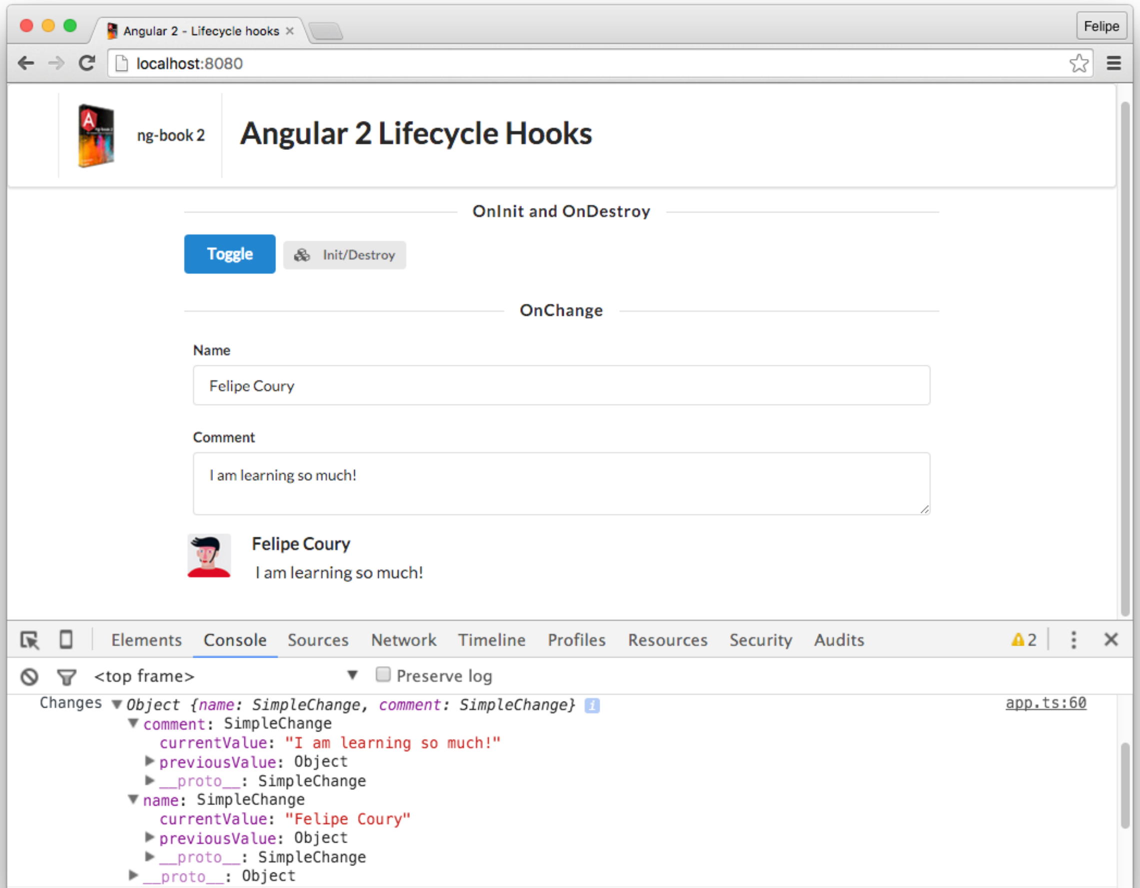
When the `keyup` event is fired for either the name or comment fields, we are calling `setValues` with the template references `namefld` and `commentfld` that represent the `input` and `textarea`.

This method just takes the value from those fields and updates the name and comment properties accordingly:

code/advanced-components/src/app/lifecycle/on-changes/on-changes-demo.component.ts

```
  setValues(namefld, commentfld): void {
    this.name = namefld.value;
    this.comment = commentfld.value;
  }
```

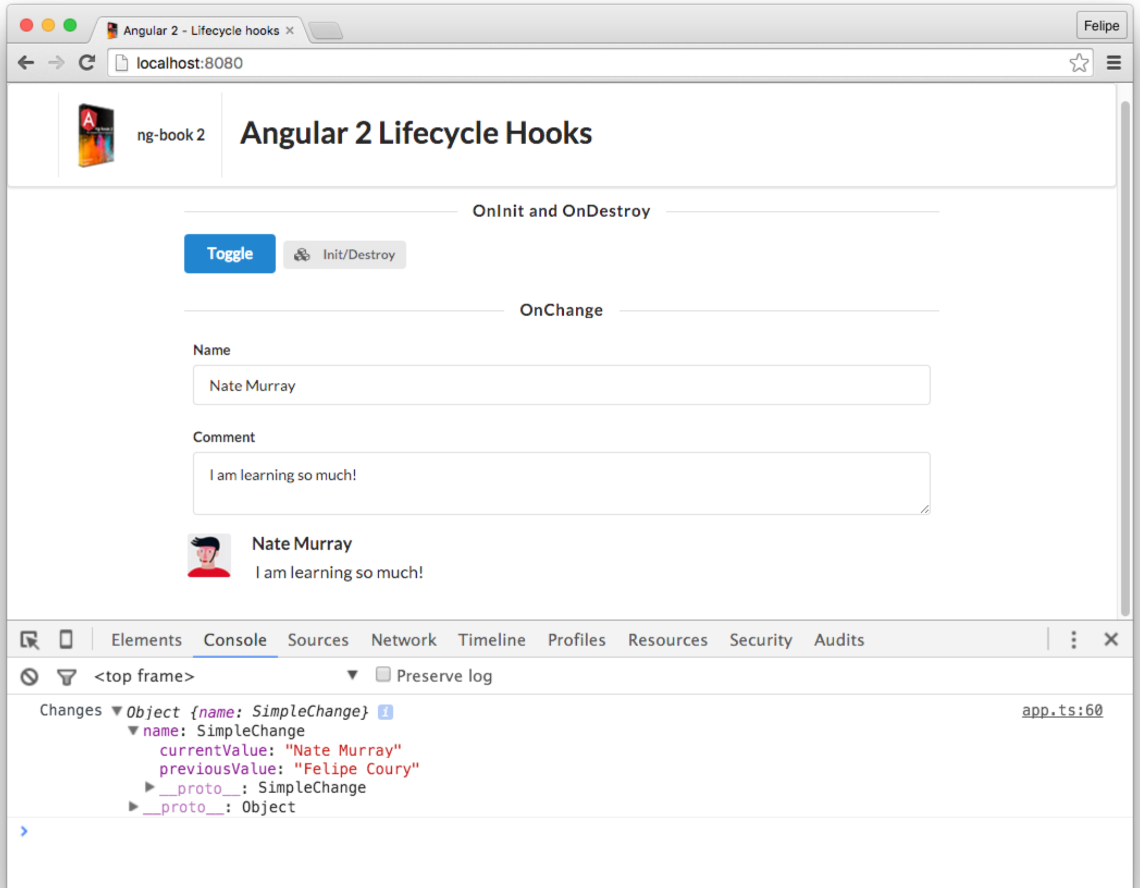
So now, the first time we open the app, we can see that our `OnChanges` hook is called:



OnChanges

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:



OnChanges

DoCheck

The default notification system implemented by `OnChanges` 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, especially 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 `OnChanges` hook gets overridden by `DoCheck` so if we implement both, `OnChanges` 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

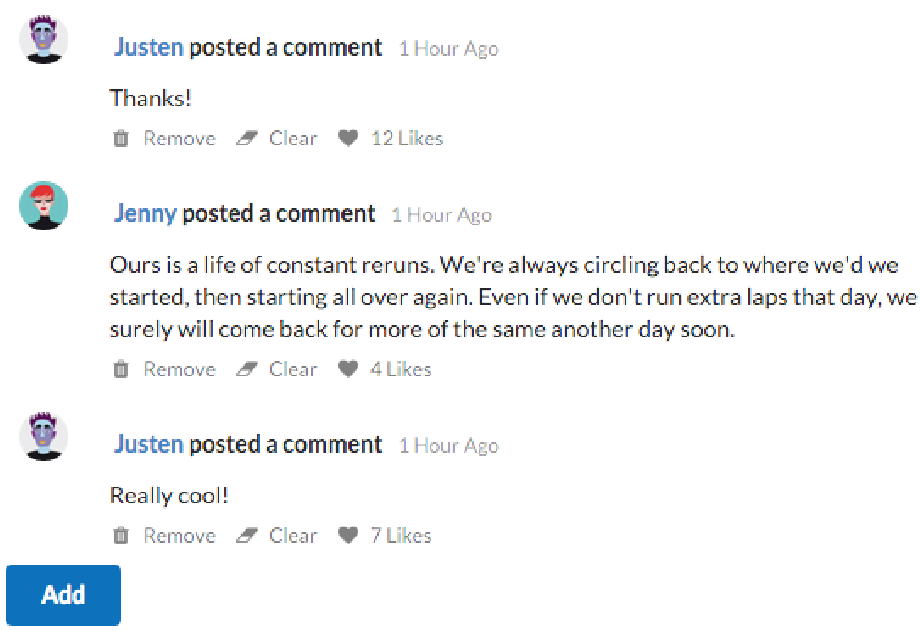
Iterable differs should be used when we have a list-like structure and we're only interested in knowing things that were added or removed from that list.

Key-value differs

Key-value differs should be used for dictionary-like structures, and work 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 `DoCheck`

To illustrate these concepts, let's build a component that renders a stream of comments, like below:



`DoCheck` example

Let's write a component that will render one individual comment. First, the template:

code/advanced-components/src/app/lifecycle/differs/comment.component.html

```
<div class="ui feed">
  <div class="event">
    <div class="label" *ngIf="comment.author">
      <img src="assets/images/avatars/{{comment.author.toLowerCase()}}.jpg">
    </div>
    <div class="content">
      <div class="summary">
        <a class="user">
          {{comment.author}}
        </a> posted a comment
        <div class="date">
          1 Hour Ago
        </div>
      </div>
      <div class="extra text">
        {{comment.comment}}
      </div>
      <div class="meta">
        <a class="trash" (click)="remove()">
          <i class="trash icon"></i> Remove
        </a>
        <a class="trash" (click)="clear()">
          <i class="eraser icon"></i> Clear
        </a>
        <a class="like" (click)="like()">
          <i class="like icon"></i> {{comment.likes}} Likes
        </a>
      </div>
    </div>
  </div>
</div>
```

and in the component:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

```
import {
  Component,
  Input,
  Output,
  EventEmitter,
  KeyValueDiffers,
  DoCheck
} from '@angular/core';

@Component({
  selector: 'app-comment',
  templateUrl: './comment.component.html'
})
export class CommentComponent implements DoCheck {
  @Input() comment: any;
  @Output() onRemove: EventEmitter<any>;
  differ: any;
```

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.

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/src/app/lifecycle/differs/comment.component.ts

```
  constructor(differs: KeyValueDiffers) {
    this.differ = differs.find([]).create();
    this.onRemove = new EventEmitter();
  }
```

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/src/app/lifecycle/differs/comment.component.ts

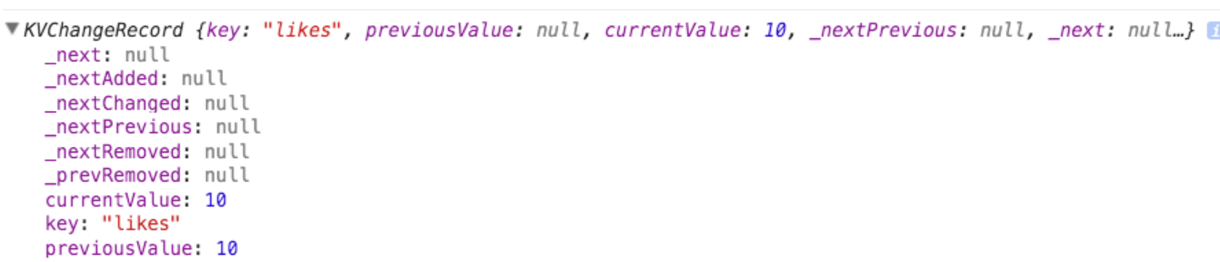
```
  ngDoCheck(): void {
    const changes = this.differ.diff(this.comment);

    if (changes) {
      changes.forEachAddedItem(r =>
        this.logChange('added', r)
      );
      changes.forEachRemovedItem(r =>
        this.logChange('removed', r)
      );
      changes.forEachChangedItem(r =>
        this.logChange('changed', r)
      );
    }
  }
```

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.



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/src/app/lifecycle/differs/comment.component.ts

```
  logChange(action, r) {
    if (action === 'changed') {
      console.log(
        r.key,
        action,
        'from',
        r.previousValue,
        'to',
        r.currentValue
      );
    }
    if (action === 'added') {
      console.log(action, r.key, 'with', r.currentValue);
    }
    if (action === 'removed') {
      console.log(
        action,
        r.key,
        '(was ' + r.previousValue + ')'
      );
    }
  }
```

Finally, let's write the methods that will help us change things on our component, to trigger our `DoCheck` hook:

code/advanced-components/src/app/lifecycle/differs/comment.component.ts

```
  remove(): void {
    this.onRemove.emit(this.comment);
  }

  clear(): void {
    delete this.comment.comment;
  }

  like(): void {
    this.comment.likes += 1;
  }
```

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 the like counter for the comment.

Rendering a list of comments with `CommentsListComponent`

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. First the template:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.html

```
<app-comment
  *ngFor="let comment of comments"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>

<button
  class="ui primary button"
  (click)="addComment()">
  Add
</button>
```

The component template is 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 `CommentsListComponent`:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
/* tslint:disable:max-line-length,quotemark */
import {
  Component,
  IterableDiffers,
  DoCheck
} from '@angular/core';

@Component({
  selector: 'app-comments-list',
  templateUrl: './comments-list.component.html'
})
export class CommentsListComponent implements DoCheck {
  comments: any[];
  iterable: boolean;
  authors: string[];
  texts: string[];
  differ: any;
```

Here we declare the variables we'll use: `comments`, `iterable`, `authors`, and `texts`.

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
  constructor(differs: IterableDiffers) {
    this.differ = differs.find([]).create(null);
    this.comments = [];

    this.authors = ['Elliot', 'Helen', 'Jenny', 'Joe', 'Justen', 'Matt'];
    this.texts = [
      "Ours is a life of constant reruns. We're always circling back to where we'd w\
e started, then starting all over again. Even if we don't run extra laps that day, w\
e surely will come back for more of the same another day soon.",
      'Really cool!',
      'Thanks!'
    ];

    this.addComment();
  }
```

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 `IterableDiffers`. 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/src/app/lifecycle/differs/comments-list.component.ts

```
  getRandomInt(max: number): number {
    return Math.floor(Math.random() * (max + 1));
  }

  getRandomItem(array: string[]): string {
    const pos: number = this.getRandomInt(array.length - 1);
    return array[pos];
  }

  addComment(): void {
    this.comments.push({
      author: this.getRandomItem(this.authors),
      comment: this.getRandomItem(this.texts),
      likes: this.getRandomInt(20)
    });
```

```
  }

  removeComment(comment) {
    const pos = this.comments.indexOf(comment);
    this.comments.splice(pos, 1);
  }
```

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/src/app/lifecycle/differs/comments-list.component.ts

```
  removeComment(comment) {
    const pos = this.comments.indexOf(comment);
    this.comments.splice(pos, 1);
  }
```

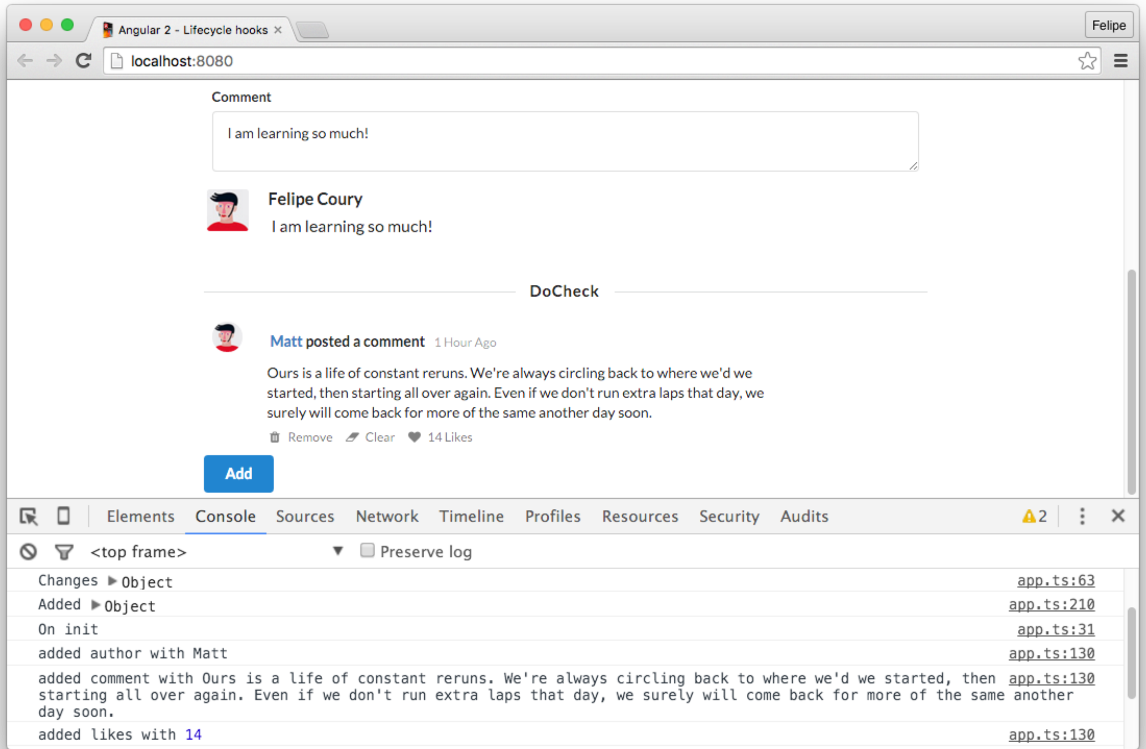
And finally we declare our change detection method `ngDoCheck()`:

code/advanced-components/src/app/lifecycle/differs/comments-list.component.ts

```
  ngDoCheck(): void {
    const changes = this.differ.diff(this.comments);

    if (changes) {
      changes.forEachAddedItem(r => console.log('Added', r.item));
      changes.forEachRemovedItem(r => console.log('Removed', r.item));
    }
  }
```

The iterable differ behaves the same way as the key-value differ but it only provides methods for items that were added or removed.

When we run the app now, we get the list of comments with one comment:

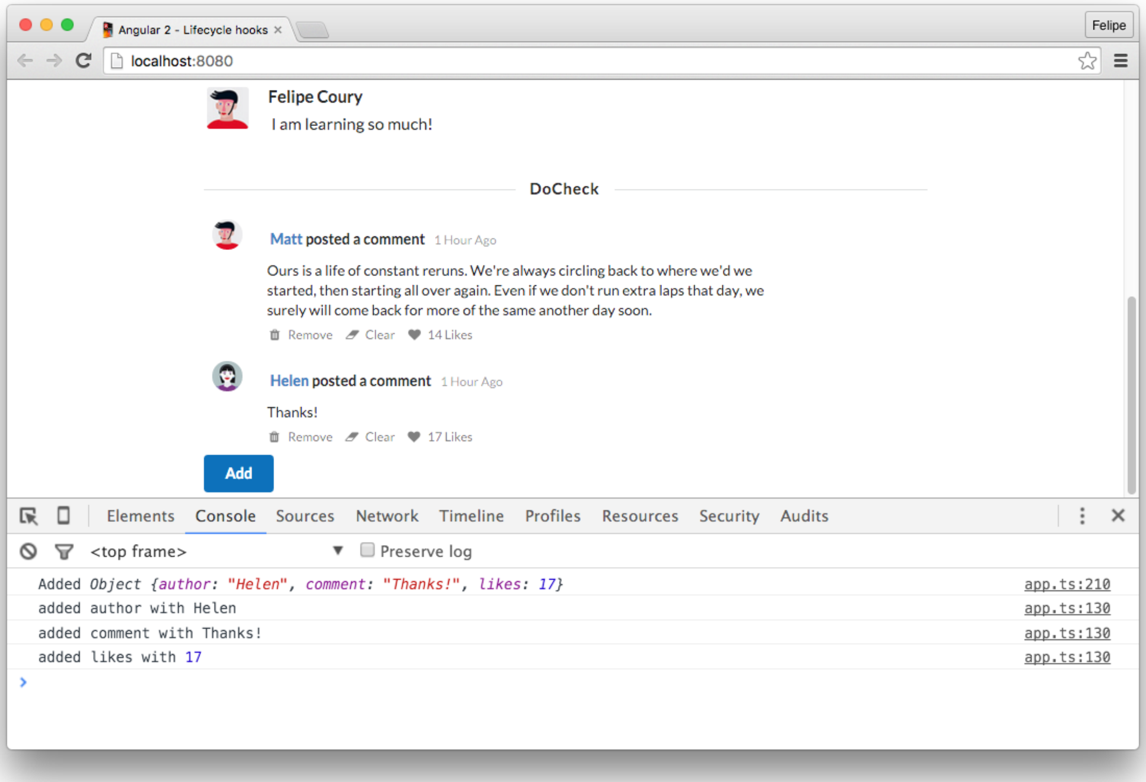


Initial state

We can also see that a few things were logged to the console, like:

```
added author with Matt
...
added likes with 14
```

Let's see what happens when we add a new comment to the list by clicking the `Add` button:



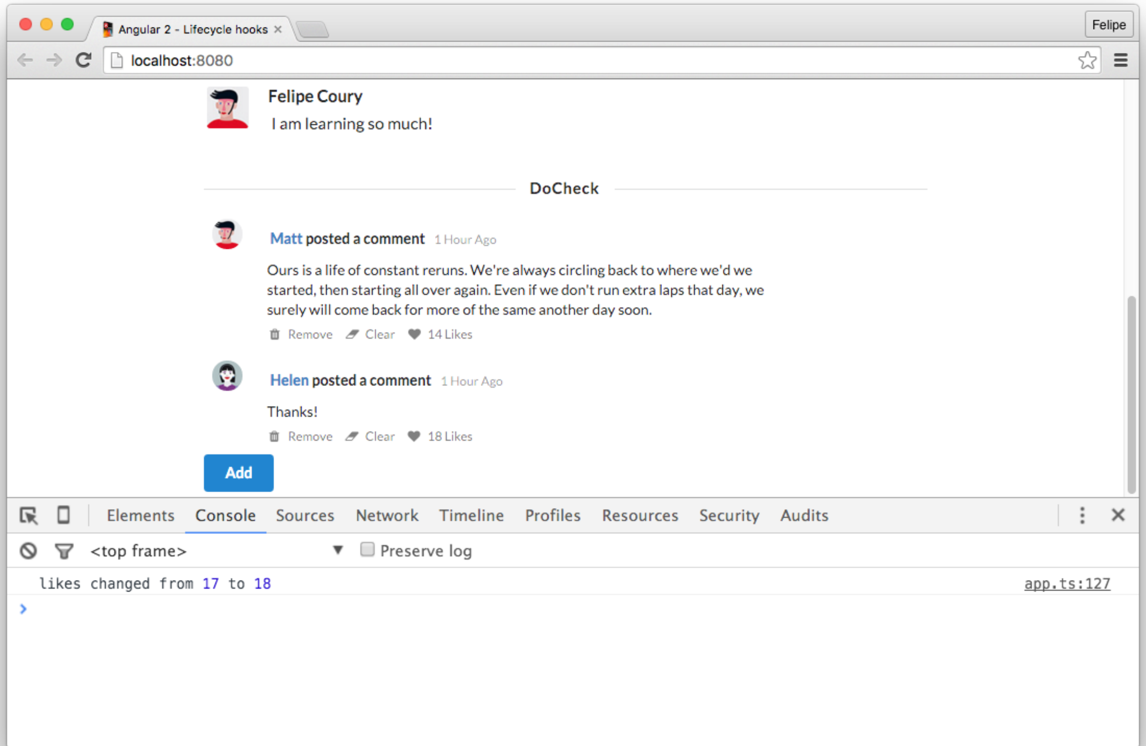
Comment added

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:

```
added author with Helen
added comment with Thanks!
added likes with 17
```

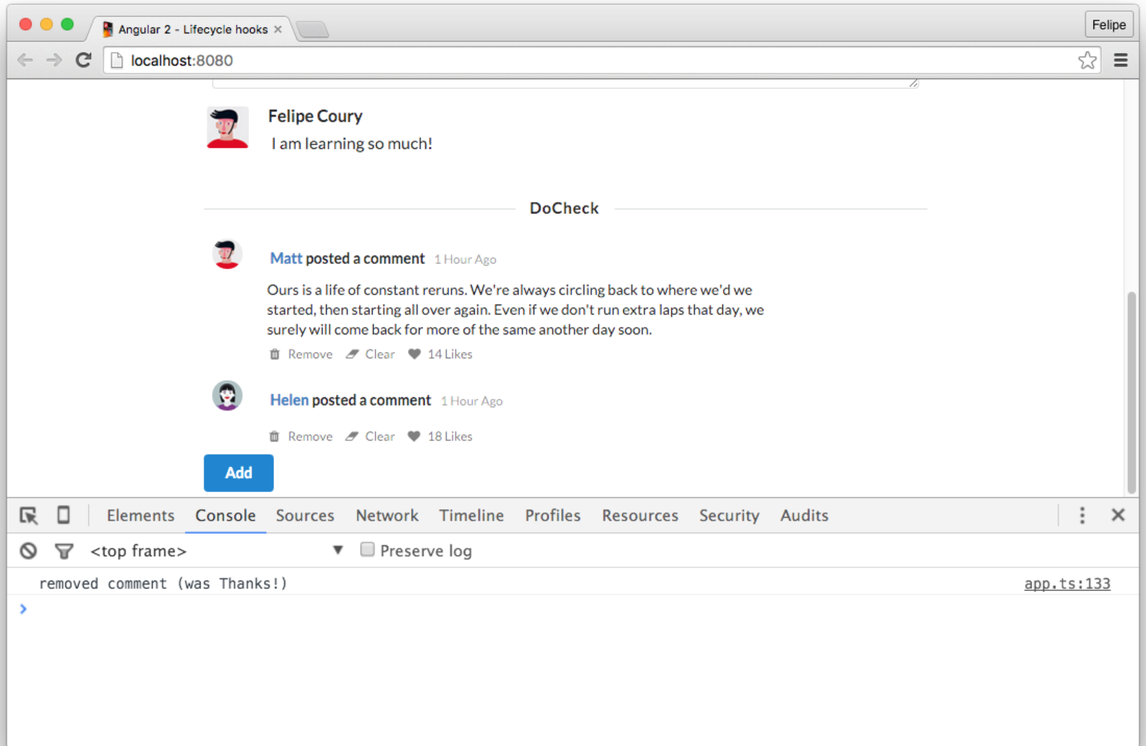
Now we can click the like button for this new comment:



Number of likes changed

And now only the like change was detected.

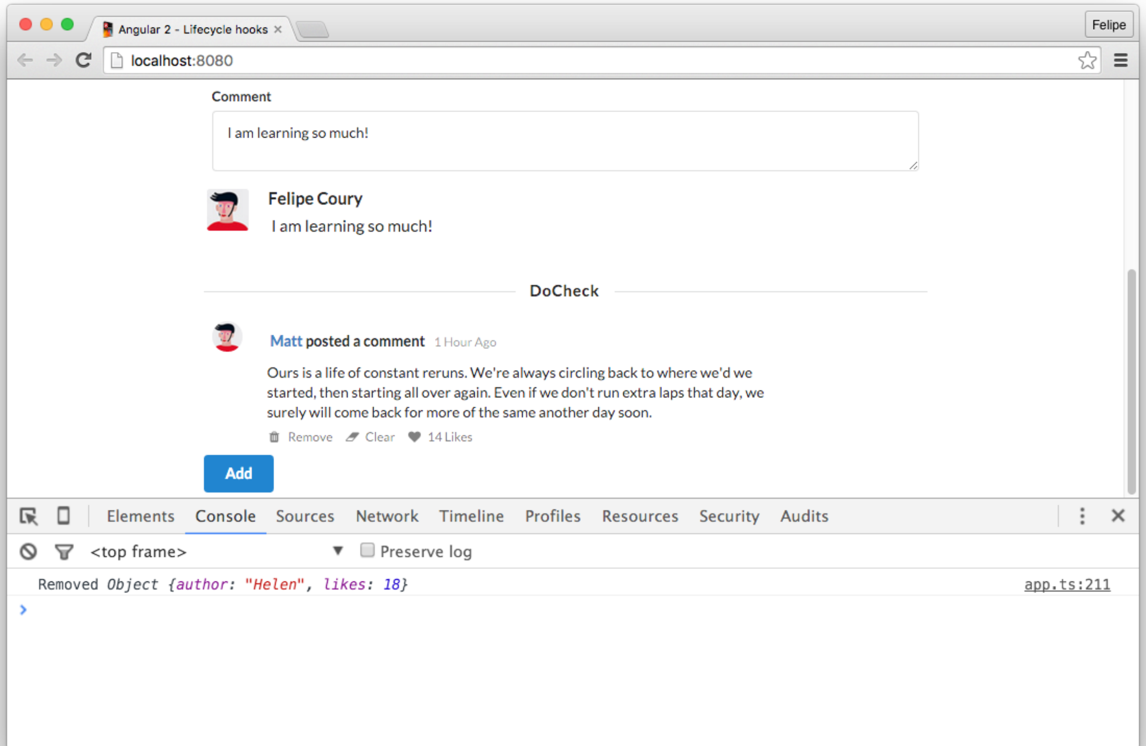
If we click the *Clear* icon, it will remove the `comment` key from the comment object:



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/src/app/lifecycle/all-hooks/all-hooks.component.ts

```
import {
  Component,
  OnInit,
  OnDestroy,
  DoCheck,
  OnChanges,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked
} from '@angular/core';

@Component({
  selector: 'app-all-hooks',
  templateUrl: './all-hooks.component.html'
})
export class AllHooksComponent implements OnInit,
  OnDestroy, DoCheck,
  OnChanges, AfterContentInit,
  AfterContentChecked, AfterViewInit,
  AfterViewChecked {
  counter: number;

  constructor() {
    console.log('AllHooksComponent --------- [constructor]');
    this.counter = 1;
  }
  inc() {
    console.log('AllHooksComponent --------- [counter]');
    this.counter += 1;
  }
  ngOnInit() {
    console.log('AllHooksComponent - OnInit');
  }
  ngOnDestroy() {
    console.log('AllHooksComponent - OnDestroy');
  }
  ngDoCheck() {
    console.log('AllHooksComponent - DoCheck');
  }
  ngOnChanges() {
    console.log('AllHooksComponent - OnChanges');
```

```
  }
  ngAfterContentInit() {
    console.log('AllHooksComponent - AfterContentInit');
  }
  ngAfterContentChecked() {
    console.log('AllHooksComponent - AfterContentChecked');
  }
  ngAfterViewInit() {
    console.log('AllHooksComponent - AfterViewInit');
  }
  ngAfterViewChecked() {
    console.log('AllHooksComponent - AfterViewChecked');
  }

}
```

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/src/app/lifecycle/all-hooks/all-hooks-demo.component.html

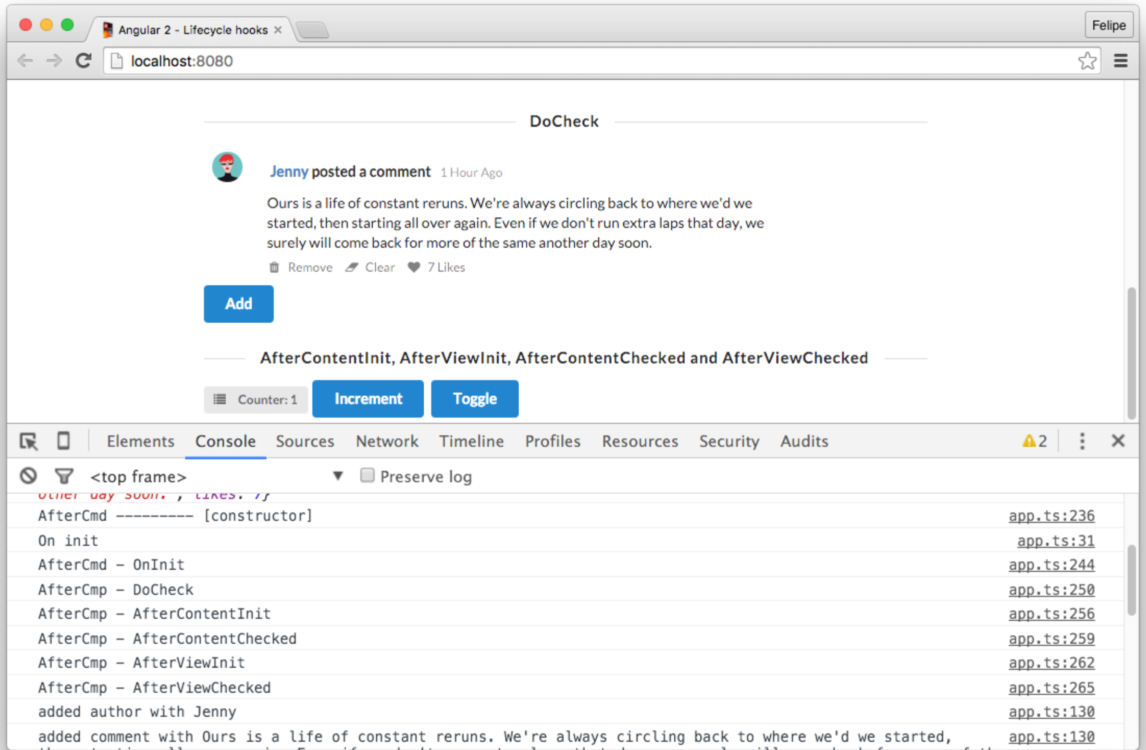
```
<h4 class="ui horizontal divider header">
  AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked
</h4>

<app-all-hooks
  *ngIf="displayAfters"
  ></app-all-hooks>

<button class="ui primary button" (click)="toggleAfters()">
  Toggle
</button>
```

The final implementation for the app demo component now will look like this:

code/advanced-components/src/app/lifecycle/all-hooks/all-hooks-demo.component.ts

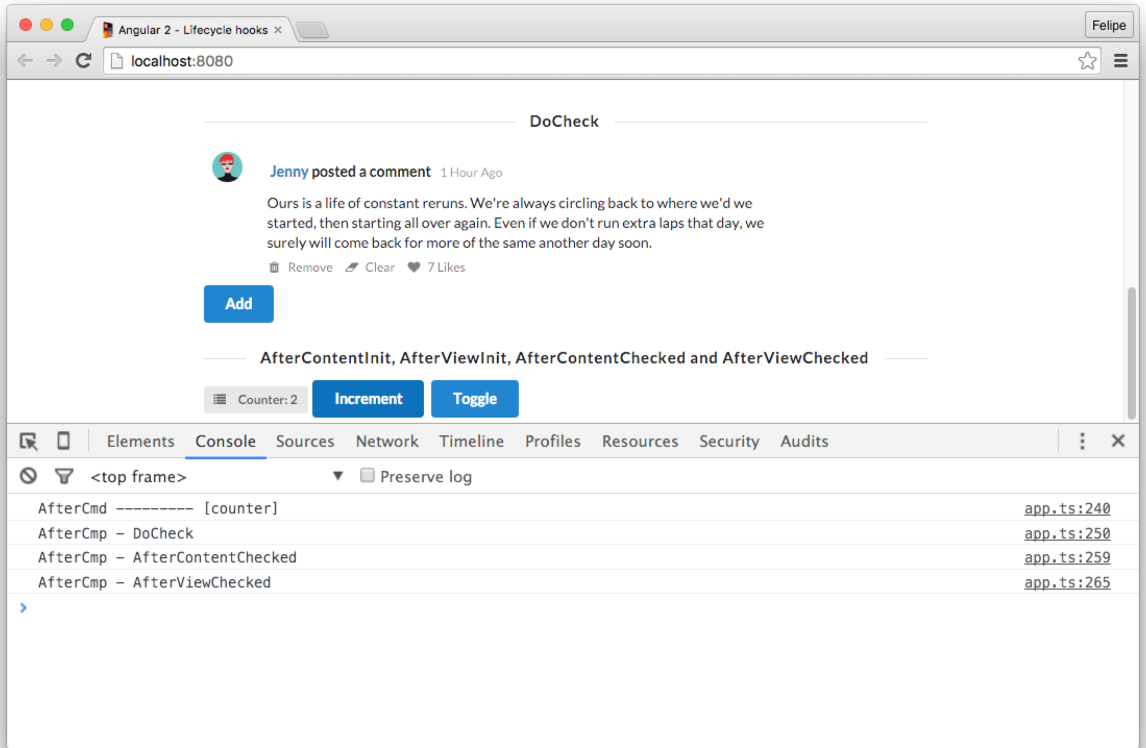
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-all-hooks-demo',
  templateUrl: './all-hooks-demo.component.html',
  styles: []
})
export class AllHooksDemoComponent implements OnInit {
  displayAfters = true;

  constructor() { }

  ngOnInit() { }

  toggleAfters(): void {
    this.displayAfters = !this.displayAfters;
  }
}
```

When the application starts, we can see each hook is logged:



App started

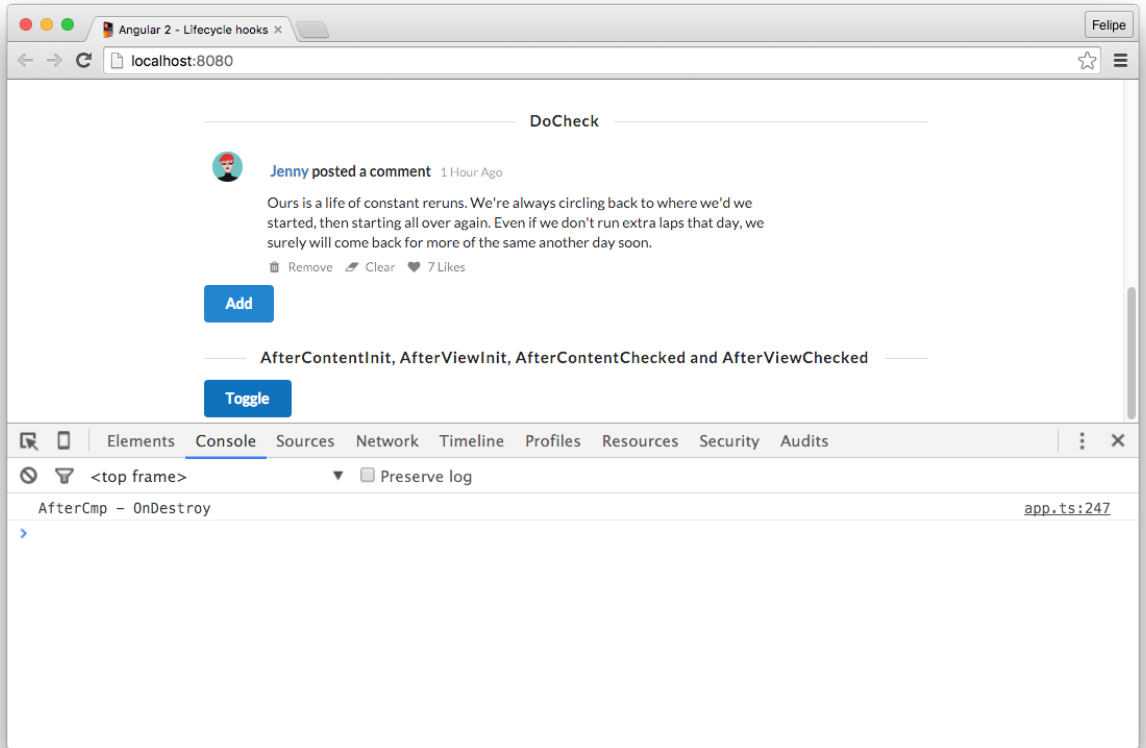
Now let's clear the console and click the `Increment` button:



After counter increment

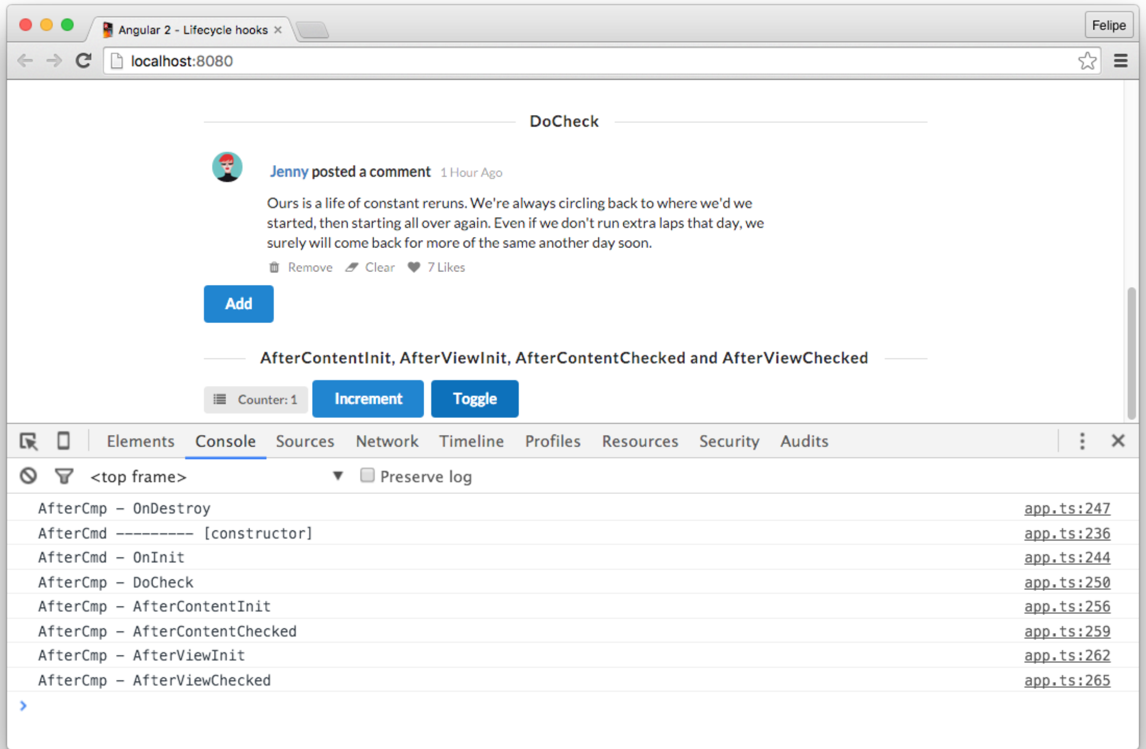
You can see that now only the `DoCheck`, `AfterContentCheck` and `AfterViewCheck` hooks were triggered.

Sure enough, if we click the `Toggle` button:



App started

And click it again:



App started

All the hooks are triggered.

Advanced Templates

Template 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:

```
<app-comment
  *ngFor="let comment of comments"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>
```

This gets converted into:

```
<app-comment
  template="ngFor let comment of comments; #i=index"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</app-comment>
```

Which then gets converted into:

```
<template
  ngFor
  [ngForOf]="comments"
  let-comment="$implicit"
  let-index="i">
  <app-comment
    [comment]="comment"
    (onRemove)="removeComment($event)">
  </app-comment>
</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` decorator for our class:

```
@Directive({
  selector: '[ngBookIf]'
})
```

We're using `[ngBookIf]` as the selector because, as we learned above, when we use `*ngBookIf="condition"`, it will be converted to:

```
<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 job 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/src/app/templates/ng-book-if/ng-book-if.directive.ts

```
export class NgBookIfDirective {
  constructor(private viewContainer: ViewContainerRef,
              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 and also specify that this is an `input` using the `Input()` decorator:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if.directive.ts

```
  @Input() set ngBookIf(condition) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.template);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

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 demo component:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if-demo.component.ts

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

@Component({
  selector: 'app-ng-book-if',
  templateUrl: './ng-book-if-demo.component.html',
})
export class NgBookIfDemoComponent {
  display: boolean;

  constructor() {
    this.display = true;
  }

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

and template:

code/advanced-components/src/app/templates/ng-book-if/ng-book-if-demo.component.html

```
<button class="ui primary button" (click)="toggle()">
  Toggle
</button>

<div *ngBookIf="display">
  The message is displayed
</div>
```

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` - `NgBookFor`

Now let's write a simplified version of the `ngFor` directive that Angular provides to handle repetition of templates for a given collection.

`NgBookFor` template deconstruction

This directive will be used with the `*NgBookFor="let var of collection"` notation.

Like we did for the previous directive, we need to declare the selector as being `[NgBookFor]`. However the input parameter, in this case, won't be `NgBookFor` 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:

```
<template something [somethingOf]="collection" let-var="$implicit">
  <!-- ... -->
</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.

In the 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`.

`NgBookFor @Directive`

Time to write the directive.

code/advanced-components/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
import {
  Directive,
  IterableDiffer,
  IterableDiffers,
  ViewRef,
  ViewContainerRef,
  TemplateRef,
  ChangeDetectorRef,
  DoCheck,
  Input
} from '@angular/core';

@Directive({
  selector: '[ngBookFor]'
})
export class NgBookForDirective implements DoCheck {
  private items: any;
  private differ: IterableDiffer<any>;
  private views: Map<any, ViewRef> = new Map<
    any,
    ViewRef
  >();
```

```
  constructor(
    private viewContainer: ViewContainerRef,
    private template: TemplateRef<any>,
    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 in change detection in 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 change.

The next step is to write code that will trigger when we set the `ngBookForOf` input:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
  set ngBookForOf(items) {
    this.items = items;
    if (this.items && !this.differ) {
      this.differ = this.differs.find(items).create();
    }
  }
```

When we set this attribute, we're keeping the collection on the directive's `items` 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 in 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/src/app/templates/ng-book-for/ng-book-for.directive.ts

```
  ngDoCheck(): void {
    if (this.differ) {
      const changes = this.differ.diff(this.items);
      if (changes) {
        changes.forEachAddedItem(change => {
          const view = this.viewContainer.createEmbeddedView(
            this.template,
            { $implicit: change.item }
          );
          this.views.set(change.item, view);
        });
        changes.forEachRemovedItem(change => {
          const view = this.views.get(change.item);
          const idx = this.viewContainer.indexOf(view);
          this.viewContainer.remove(idx);
          this.views.delete(change.item);
        });
      }
    }
  }
```

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 items 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.

```
let view = this.viewContainer.createEmbeddedView(this.template, {'$implicit': change\
.item});
```

The second argument to `createEmbeddedView` is the *view context*. In this case, we're setting the `$implicit` local variable to `change.item`. This will allow us to reference the variable we declared back on the `*NgBookFor="let var of collection"` as `var` on that view. That is, the `var` in `let var` is the `$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 connect the item with 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 remove 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/src/app/templates/ng-book-for/ng-book-for-demo.component.ts

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

@Component({
  selector: 'app-ng-book-for-demo',
  templateUrl: './ng-book-for-demo.component.html'
})
export class NgBookForDemoComponent implements OnInit {
  people: any[];

  constructor() { }

  ngOnInit() {
    this.people = [
      {name: 'Joe', age: 10},
      {name: 'Patrick', age: 21},
      {name: 'Melissa', age: 12},
      {name: 'Kate', age: 19}
    ];
  }

  remove(p) {
    const idx: number = this.people.indexOf(p);
    this.people.splice(idx, 1);
    return false;
  }

  add(name, age) {
    this.people.push({name: name.value, age: age.value});
    name.value = '';
    age.value = '';
  }
}
```

and template:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<ul>
  <li *ngBookFor="let p of people">
    {{ p.name }} is {{ p.age }}
    <a href (click)="remove(p)">Remove</a>
  </li>
</ul>

<div class="ui form">
  <div class="fields">
    <div class="field">
      <label>Name</label>
      <input type="text" #name placeholder="Name">
    </div>
    <div class="field">
      <label>Age</label>
      <input type="text" #age placeholder="Age">
    </div>
  </div>
</div>
<div class="ui submit button"
     (click)="add(name, age)">
  Add
</div>
```

We're using our directive to iterate through a list of people:

code/advanced-components/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<ul>
  <li *ngBookFor="let p of people">
    {{ p.name }} is {{ p.age }}
    <a href (click)="remove(p)">Remove</a>
  </li>
</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/src/app/templates/ng-book-for/ng-book-for-demo.component.html

```
<div class="ui form">
  <div class="fields">
    <div class="field">
      <label>Name</label>
      <input type="text" #name placeholder="Name">
    </div>
    <div class="field">
      <label>Age</label>
      <input type="text" #age placeholder="Age">
    </div>
  </div>
</div>
<div class="ui submit button"
     (click)="add(name, age)">
  Add
</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 to 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:

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  <button (click)="changeName()">Change!</button>
  `
})
class MyComponent {
  name: string;
  constructor() {
    this.name = 'Felipe';
  }
```

```

  changeName() {
    this.name = 'Nate';
  }
}
```

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:

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  `
})
class MyComponent {
  name: string;
  constructor(private http: Http) {
    this.http.get('/names/1')
      .map(res => res.json())
      .subscribe(data => this.name = data.name);
  }
}
```

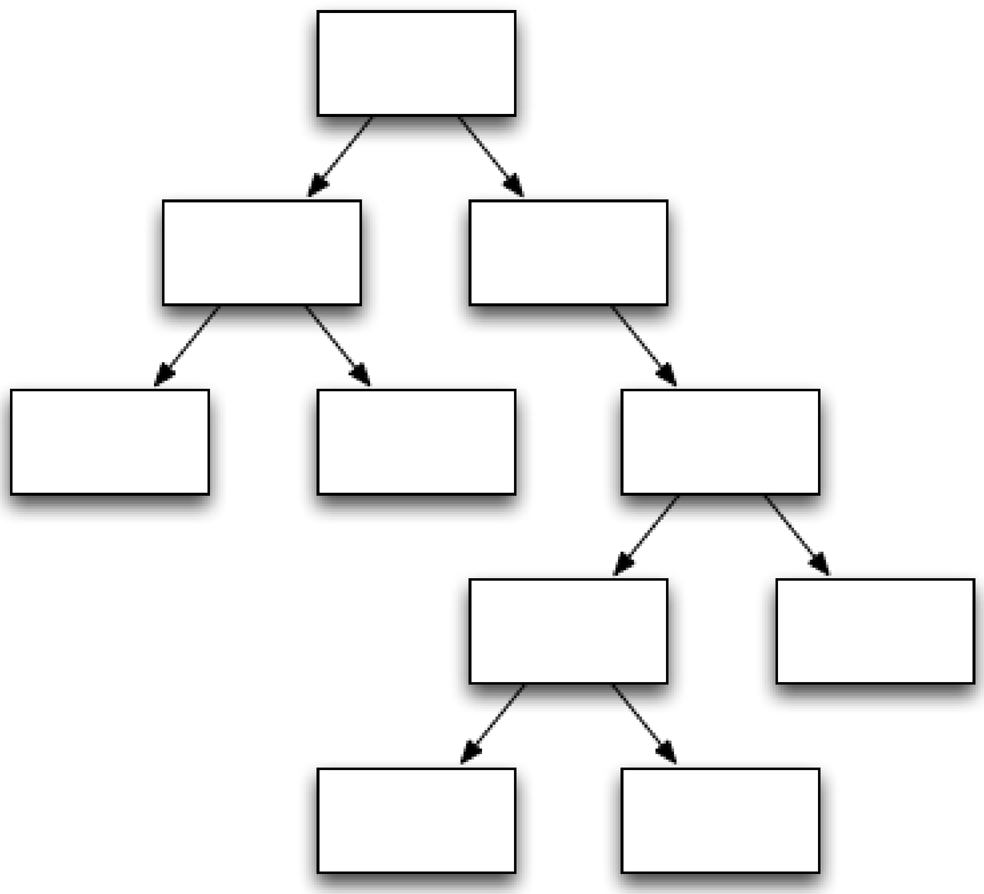
And finally, we could have a timer that would trigger the change:

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  `
})
class MyComponent {
  name: string;
  constructor() {
    setTimeout(() => this.name = 'Felipe', 2000);
  }
}
```

But how does Angular become aware of these changes?

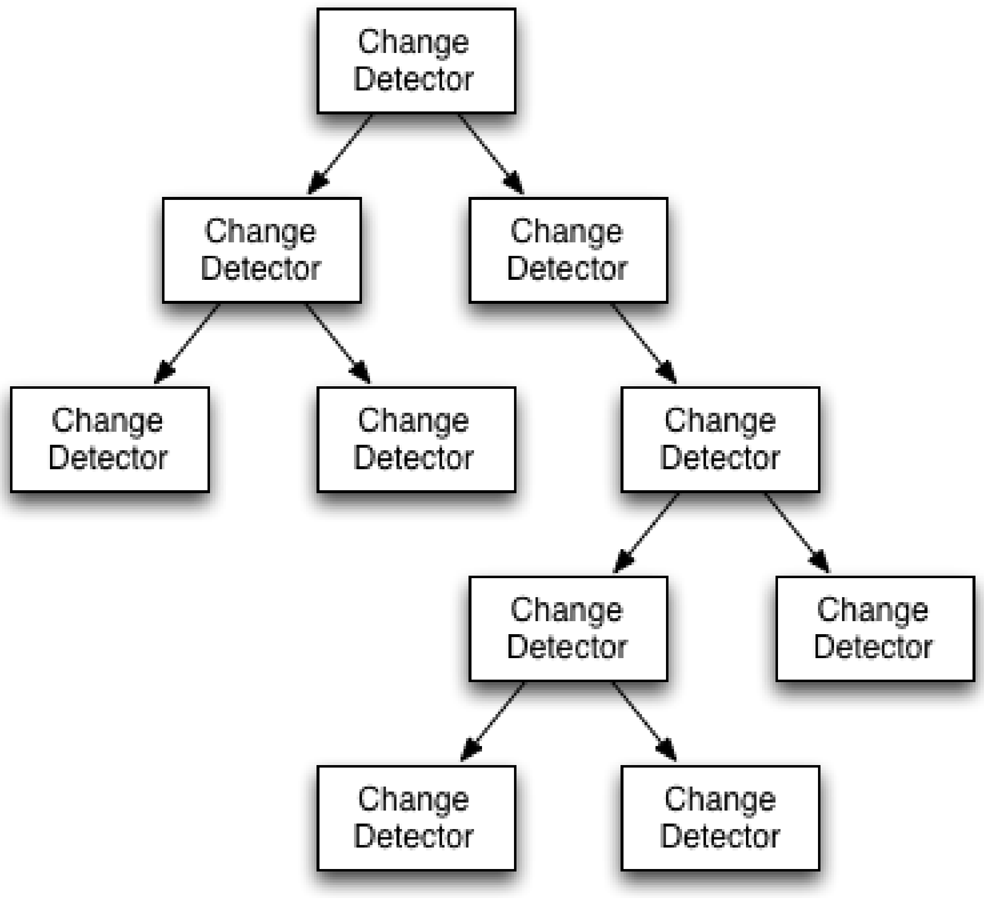
The first thing to know is that each component gets a change detector.

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:



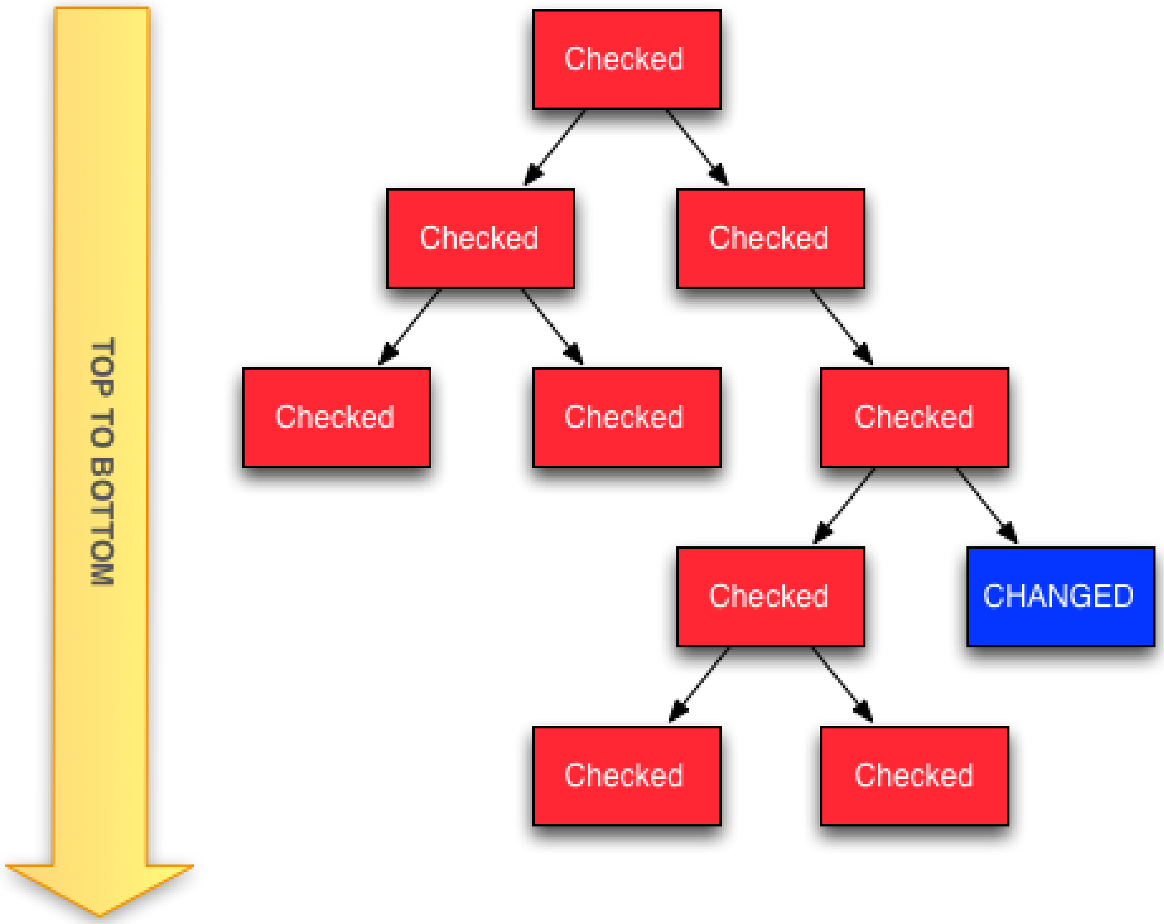
Component tree

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 components change, no matter where it is in the tree, 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 so that you get very fast performance.

The first way to change the change detector behavior is by telling a component that it should only be checked if one of its *input values* change.

To recap, an input value is an attribute your component receives from the outside world. For instance, in this code:

```
class Person {
  constructor(public name: string, public age: string) {}
}

@Component({
  selector: 'mycomp',
  template: `
    <div>
      <span class="name">{{ person.name }}</span>
      is {{ person.age }} years old.
    </div>
  `
})
class MyComp {
  @Input() person: Person;
}
```

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/src/app/change-detection/on-push-demo/profile.model.ts

```
/**
 * User Profile object, stores the first and
 * last name as well as a function that gives the time
 **/
export class Profile {
  constructor(public first: string, public last: string) {}

  lastChanged() {
    return new Date();
```

```
  }
}
```

So we start with some imports and we declare a `Profile` class that will be used as the input in both of our components. Notice that we also created a method called `lastChange()` on the `Profile` class. It will help us 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 `DefaultChangeDetectionComponent` that will use the default change detection strategy:

code/advanced-components/src/app/change-detection/on-push-demo/default-change-detection.component.ts

```
import {
  Component,
  Input
} from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-default-change-detection',
  templateUrl: './default-change-detection.component.html'
})
export class DefaultChangeDetectionComponent {
  @Input() profile: Profile;
}
```

and template:

code/advanced-components/src/app/change-detection/on-push-demo/default-change-detection.component.html

```
<h4 class="ui horizontal divider header">
  Default Strategy
</h4>

<form class="ui form">
  <div class="field">
    <label>First Name</label>
    <input
    type="text"
    [(ngModel)]="profile.first"
    name="first"
    placeholder="First Name">
```

```
  </div>
  <div class="field">
    <label>Last Name</label>
    <input
    type="text"
    [(ngModel)]="profile.last"
    name="last"
    placeholder="Last Name">
  </div>
</form>

<h5><em>Updates if either changes (e.g. more often)</em></h5>
<div>
  {{profile.lastChanged() | date:'medium'}}
</div>
```

And a second component using `OnPush` strategy:

code/advanced-components/src/app/change-detection/on-push-demo/on-push-change-detection.component.ts

```
import {
  Component,
  Input,
  ChangeDetectionStrategy
} from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-on-push-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  templateUrl: './on-push-change-detection.component.html'
})
export class OnPushChangeDetectionComponent {
  @Input() profile: Profile;
}
```

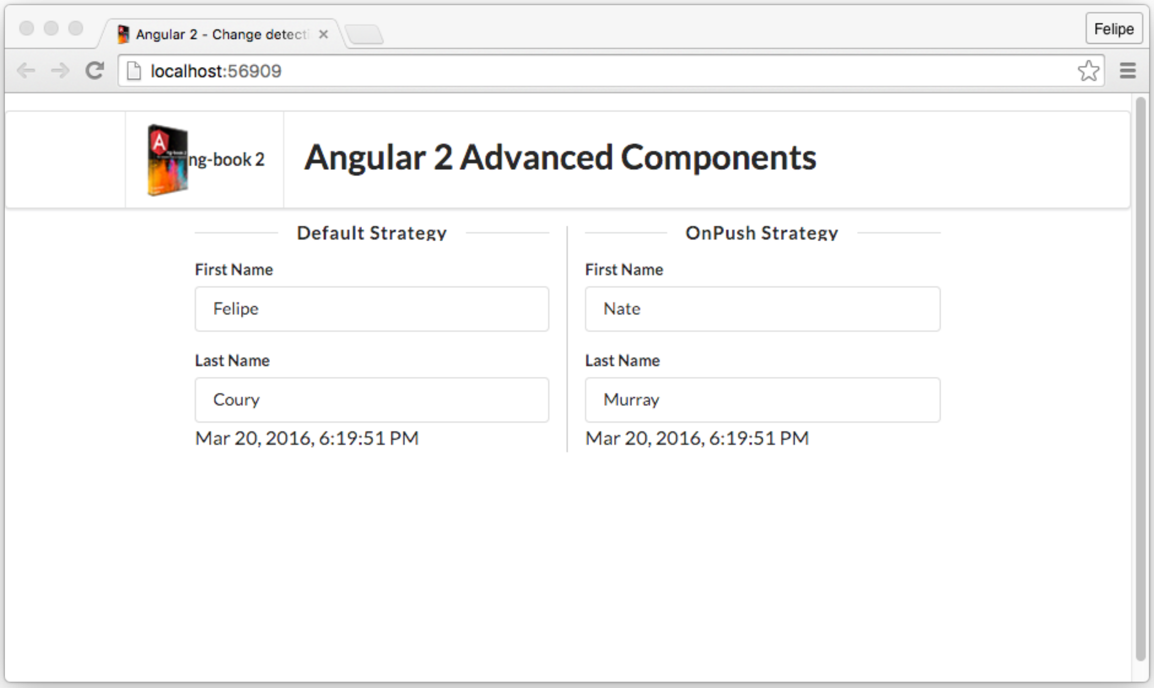
As we can see, both components use the same template. The only thing that is different is the header.

Finally, let's add the component that will render both components side by side:

code/advanced-components/src/app/change-detection/on-push-demo/on-push-demo.component.ts

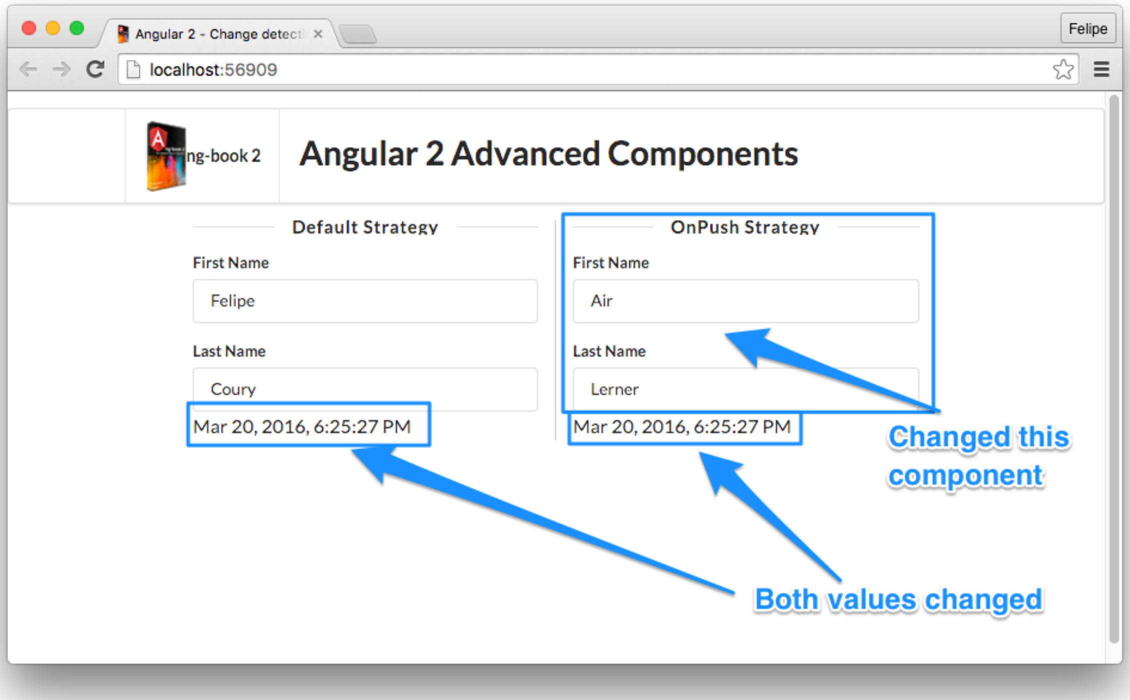
```
import { Component } from '@angular/core';
import { Profile } from './profile.model';

@Component({
  selector: 'app-on-push-demo',
  template: `
  <div class="ui page grid">
    <div class="two column row">
      <div class="column area">
        <app-default-change-detection
          [profile]="profile1">
        </app-default-change-detection>
      </div>
      <div class="column area">
        <app-on-push-change-detection
          [profile]="profile2">
        </app-on-push-change-detection>
      </div>
    </div>
  </div>
  `
})
export class OnPushDemoComponent {
  profile1: Profile = new Profile('Felipe', 'Coury');
  profile2: Profile = new Profile('Nate', 'Murray');
}
```

When we run this application, we should see both components rendered like below:



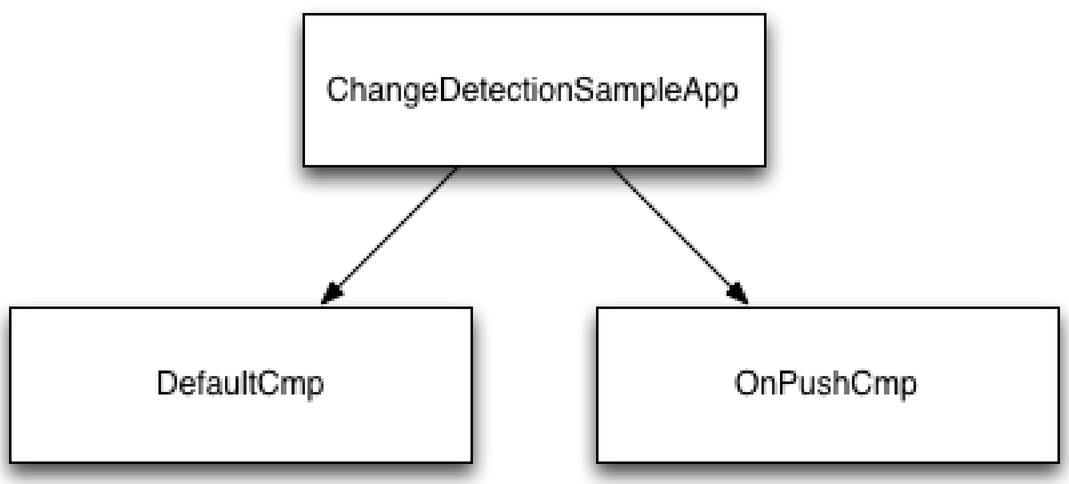
Default vs. OnPush strategies

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

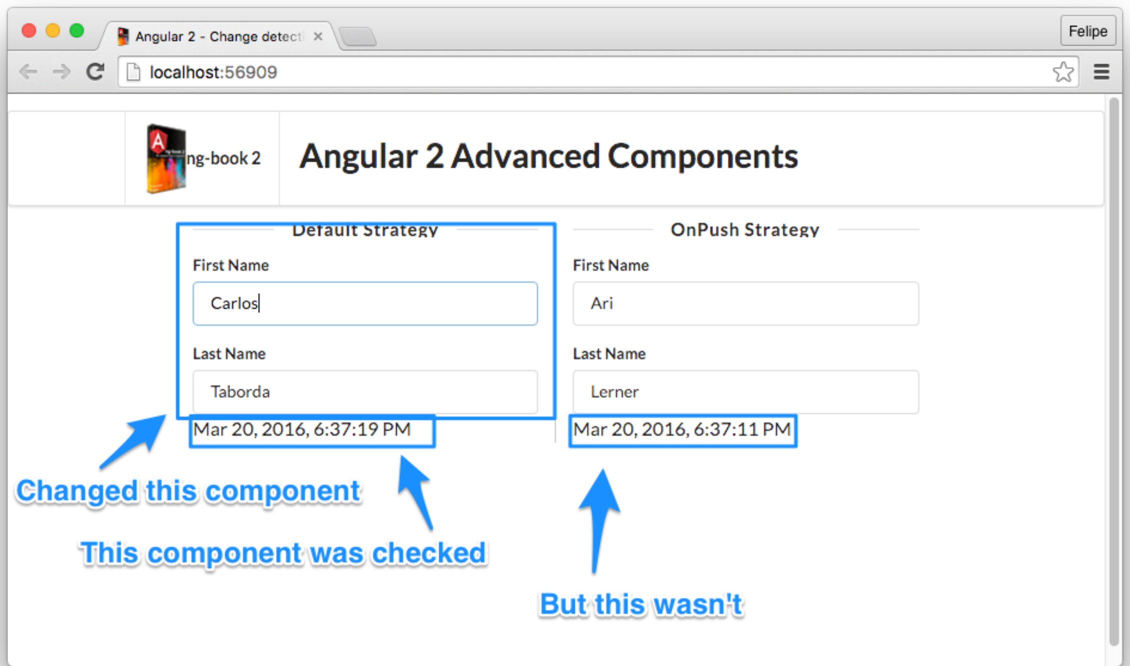
To understand why this happened, let's check this new tree of components:



Tree of components

Angular checks for changes from the top to the bottom, so it queried first `OnPushDemoComponent`, then `DefaultChangeDetectionComponent` and finally `OnPushChangeDetectionComponent`. When it inferred that `OnPushChangeDetectionComponent` changed, it updates all the components of the tree, from top to bottom, making the `DefaultChangeDetectionComponent` 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 `DefaultChangeDetectionComponent` component was checked but `OnPushChangeDetectionComponent` wasn't. This happened because when we set the `OnPush` strategy for this component, it made the change detection 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 changes 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 triggered (*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 are out of the Zones control, would be:

- using a third party library that runs asynchronously
- immutable data
- Observables

these are perfect candidates for 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 have this component 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/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
import {
  Component,
  OnInit,
  Input,
  ChangeDetectionStrategy,
  ChangeDetectorRef
 } from '@angular/core';
import { Observable } from 'rxjs/Rx';

@Component({
  selector: 'app-observable-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
```

```
export class ObservableChangeDetectionComponent implements OnInit {
  @Input() items: Observable<number>;
  counter = 0;

  constructor(private changeDetector: ChangeDetectorRef) {
  }

  ngOnInit() {
    this.items.subscribe((v) => {
      console.log('got value', v);
      this.counter++;
      if (this.counter % 5 === 0) {
        this.changeDetector.markForCheck();
      }
    },
    null,
    () => {
      this.changeDetector.markForCheck();
    });
  }
}
```

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 to use the `OnPush` detection strategy:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
@Component({
  selector: 'app-observable-change-detection',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
```

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/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
export class ObservableChangeDetectionComponent implements OnInit {
  @Input() items: Observable<number>;
  counter = 0;
```

Then we use the constructor to get hold of the component's change detector:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
  constructor(private changeDetector: ChangeDetectorRef) {
  }
```

Then, during the component initialization, on the `ngOnInit` hook:

code/advanced-components/src/app/change-detection/observables-demo/observable-change-detection.component.ts

```
  ngOnInit() {
    this.items.subscribe((v) => {
      console.log('got value', v);
      this.counter++;
      if (this.counter % 5 === 0) {
        this.changeDetector.markForCheck();
      }
    },
    null,
    () => {
      this.changeDetector.markForCheck();
    });
  }
```

We're subscribing to the Observable. The `subscribe` method takes three callbacks as arguments: `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 has been 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/src/app/change-detection/observables-demo/observables-demo.component.ts

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

@Component({
  selector: 'app-observables-demo',
  template: `
  <app-observable-change-detection
    [items]="itemObservable">
    </app-observable-change-detection>
  `
})
export class ObservablesDemoComponent implements OnInit {
  itemObservable: Observable<number>;

  constructor() { }

  ngOnInit() {
    this.itemObservable = Observable.timer(100, 100).take(101);
  }

}
```

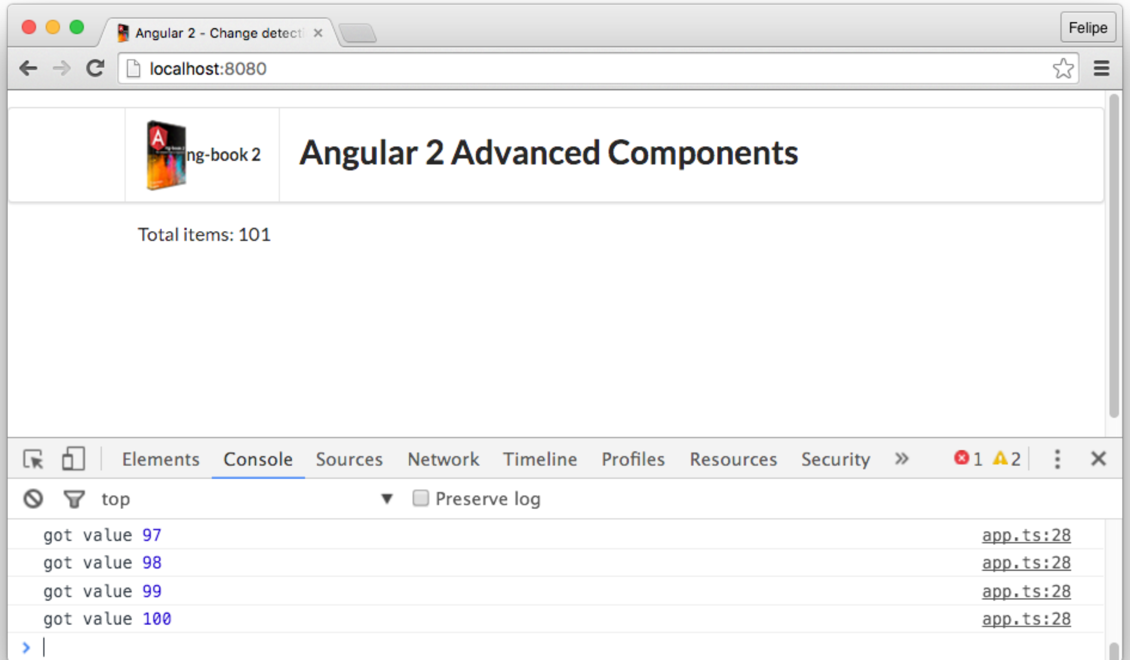
The important line here is the following:

```
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 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.

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 your 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[132]. 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[133] 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:

```
describe('Calculator', () => {
  it('sums 1 and 1 to 2', () => {
    var calc = new Calculator();
    expect(calc.sum(1, 1)).toEqual(2);
  });
});
```

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.

[132] https://angular.github.io/protractor/#/
[133] http://jasmine.github.io/2.4/introduction.html

Normally we use `describe` for each logical unit we're testing and inside that 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[134].

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.)

[134] http://jasmine.github.io/2.4/introduction.html

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:

Since we're using Angular CLI, this `karma.conf.js` file is already created for us! However, if your project does not use Angular CLI, you may need to setup Karma on your own.

code/routes/music/karma.conf.js

```
// Karma configuration file, see link for more information
// https://karma-runner.github.io/1.0/config/configuration-file.html

module.exports = function(config) {
  let configuration = {
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular-devkit/build-angular/plugins/karma')
    ],
```

```
    client: {
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      dir: require('path').join(__dirname, '../coverage'),
      reports: ['html', 'lcovonly'],
      fixWebpackSourcePaths: true
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  };

  if (process.env.TRAVIS) {
    configuration.customLaunchers = {
      Chrome_travis_ci: {
        base: 'Chrome',
        flags: ['--no-sandbox']
      }
    };
    configuration.browsers = ['Chrome_travis_ci'];
  }

  config.set(configuration);
};
```

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.

```
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 `spotify.service.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/src/app/spotify.service.spec.ts

```
import {
  inject,
  fakeAsync,
  tick,
  TestBed
} from '@angular/core/testing';
```

Next, we'll import a couple more classes:

code/routes/music/src/app/spotify.service.spec.ts

```
import {MockBackend} from '@angular/http/testing';
import {
  Http,
  ConnectionBackend,
  BaseRequestOptions,
  Response,
  ResponseOptions
} 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/src/app/spotify.service.spec.ts

```
import { SpotifyService } from './spotify.service';
```

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[135]

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:

[135] http://martinfowler.com/articles/mocksArentStubs.html

```
describe('Speedtrap', function() {
  it('tickets a car at more than 60mph', function() {
    var stubCar = { getSpeed: function() { return 61; } };
    var speedTrap = new SpeedTrap(stubCar);
    speedTrap.ticketCount = 0;
    speedTrap.checkSpeed();
    expect(speedTrap.ticketCount).toEqual(1);
  });
});
```

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:

```
class MockCar {
  startCallCount: number = 0;

  start() {
    this.startCallCount++;
  }
}
```

And it would be used to write another test like this:

```
describe('CarRemote', function() {
  it('starts the car when the start key is held', function() {
    var car = new MockCar();
    var remote = new CarRemote();
    remote.holdButton('start');
    expect(car.startCallCount).toEqual(1);
  });
});
```

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.

`TestBed.configureTestingModule` and Providers

When we test our Angular apps we need to make sure we configure the top-level `NgModule` that we will use for this test. When we do this, we can configure providers, declare components, and import other modules: just like you would when using `NgModules` generally.

Sometimes when testing Angular code, we *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 `TestBed.configureTestingModule` 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/src/app/spotify.service.spec.ts

```
describe('SpotifyService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        BaseRequestOptions,
        MockBackend,
        SpotifyService,
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] },
      ]
    });
  });
```

Notice that `TestBed.configureTestingModule` accepts an **array of providers** in the `providers` key 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/src/app/spotify.service.spec.ts

```
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] },
      ]
```

This code uses `provide` with `useFactory` 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 `BaseRequestOption` 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/src/app/spotify.service.ts

```
  getTrack(id: string): Observable<any[]> {
    return this.query(`/tracks/${id}`);
  }
```

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/src/app/spotify.service.ts

```
  query(
    URL: string,
    params?: Array<string>
  ): Observable<any[]> {
    let queryURL = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join('&')}`;
    }
    const apiKey = environment.spotifyApiKey;
    const headers = new Headers({
      Authorization: `Bearer ${apiKey}`
    });
    const options = new RequestOptions({
      headers: headers
    });

    return this.http
      .request(queryURL, options)
      .map((res: any) => res.json());
  }
```

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:

```
describe('getTrack', () => {
  it('retrieves using the track ID',
    inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend) =>\
 {
      var res;
      mockBackend.connections.subscribe(c => {
        expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
        let response = new ResponseOptions({body: '{"name": "felipe"}'});
        c.mockRespond(new Response(response));
      });
      spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});
```

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:

```
inject([Class1,    /* ..., */ ClassN],
       (instance1, /* ..., */ instanceN) => {
  // ... testing code ...
})
```

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 Promises are fulfilled and Observables are notified immediately after you call `tick()`.

So in this code:

```
inject([SpotifyService, MockBackend],
       fakeAsync((spotifyService, mockBackend) => {
  // ... testing code ...
}));
```

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`:

```
var res;
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'`:

```
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.

```
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:

```
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:

```
spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
  res = _res;
});
```

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:

```
tick();
```

We now perform one final check just to make sure our response we setup is the one we received:

```
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/src/app/spotify.service.spec.ts

```
  function expectURL(backend: MockBackend, url: string) {
    backend.connections.subscribe(c => {
      expect(c.request.url).toBe(url);
      const response = new ResponseOptions({body: '{"name": "felipe"}'});
      c.mockRespond(new Response(response));
    });
  }
```

Following the same lines, it should be very simple to create similar tests for `getArtist` and `getAlbum` methods:

code/routes/music/src/app/spotify.service.spec.ts

```
  describe('getArtist', () => {
    it('retrieves using the artist ID',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/artists/ARTIST_ID');
        svc.getArtist('ARTIST_ID').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
```

```
      }))
    );
  });

  describe('getAlbum', () => {
    it('retrieves using the album ID',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/albums/ALBUM_ID');
        svc.getAlbum('ALBUM_ID').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
      }))
    );
  });
```

Now searchTrack is slightly different: instead of calling query, this method uses the search method:

code/routes/music/src/app/spotify.service.ts

```
  searchTrack(query: string): Observable<any[]> {
    return this.search(query, 'track');
  }
```

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/src/app/spotify.service.ts

```
  search(query: string, type: string): Observable<any[]> {
    return this.query(`/search`, [
      `q=${query}`,
      `type=${type}`
    ]);
  }
```

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/src/app/spotify.service.spec.ts

```
  describe('searchTrack', () => {
    it('searches type and term',
      inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
        let res;
        expectURL(backend, 'https://api.spotify.com/v1/search?q=TERM&type=track');
        svc.searchTrack('TERM').subscribe((_res) => {
          res = _res;
        });
        tick();
        expect(res.name).toBe('felipe');
      }))
    );
  });
```

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/src/app/artist/artist.component.ts

```
  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
```

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 write tests it's often convenient to use `beforeEach` with `TestBed.configureTestingModule` 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/src/app/artist/artist.component.spec.ts

```
describe('ArtistComponent', () => {
  beforeEach(async(() => {
    configureMusicTests();
  }));
```

We define `configureMusicTests` in the helper file `MusicTestHelpers.ts`. Let's look at that now.

Here's the implementation of `configureMusicTests`. Don't worry, we'll explain each part:

code/routes/music/src/app/test/test.module.ts

```
export function configureMusicTests() {
  const mockSpotifyService: MockSpotifyService = new MockSpotifyService();

  TestBed.configureTestingModule({
    imports: [
      { // TODO RouterTestingModule.withRoutes coming soon
        ngModule: RouterTestingModule,
        providers: [provideRoutes(routerConfig)]
      },
      TestModule
    ],
    providers: [
      mockSpotifyService.getProviders(),
      {
        provide: ActivatedRoute,
        useFactory: (r: Router) => r.routerState.root, deps: [ Router ]
      }
    ]
  });
}
```

We start by creating an instance of `MockSpotifyService` that we will use to mock the real implementation of `SpotifyService`.

Next we use a class called `TestBed` and call `configureTestingModule`. `TestBed` is a helper library that ships with Angular to help make testing easier.

In this case, `TestBed.configureTestingModule` is used to configure the `NgModule` used for testing. You can see that we provide an `NgModule` configuration as the argument which has:

- `imports` and
- `providers`

In our `imports` we're importing

- The `RouterTestingModule` and configuring it with our `routerConfig` - this configures the routes for testing
- The `TestModule` - which is the `NgModule` which declares all of the components we will test (see `MusicTestHelpers.ts` for the full details)

In `providers`

- We provide the `MockSpotifyService` (via `mockSpotifyService.getProviders()`)
- and the `ActivatedRoute`

Let's take a closer look at these starting with the `Router`.

Router

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/src/app/test/test.module.ts

```
@Component({
  selector: 'blank-cmp',
  template: ``
})
export class BlankCmp {
}

@Component({
  selector: 'root-cmp',
  template: `<router-outlet></router-outlet>`
})
export class RootCmp {
}

export const routerConfig: Routes = [
  { path: '', component: BlankCmp },
  { path: 'search', component: SearchComponent },
  { path: 'artists/:id', component: ArtistComponent },
  { path: 'tracks/:id', component: TrackComponent },
  { path: 'albums/:id', component: AlbumComponent }
];
```

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 this out, 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/src/app/test/spotify.service.mock.ts

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  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/src/app/test/spotify.service.mock.ts

```
  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
    this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
    this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
    this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
  }
```

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 it 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`[136]

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:

[136] https://github.com/angular/angular/blob/b0cebdba6b65c1e9e7eb5bf801ea42dc7c4a7f25/modules/angular2/src/testing/testing_internal.ts#L205

code/routes/music/src/app/artist/artist.component.ts

```
  ngOnInit(): void {
    this.spotify
      .getArtist(this.id)
      .subscribe((res: any) => this.renderArtist(res));
  }
```

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/src/app/test/spotify.service.mock.ts

```
  subscribe(callback) {
    callback(this.fakeResponse);
  }
```

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/src/app/test/spotify.service.mock.ts

```
  setResponse(json: any): void {
    this.fakeResponse = json;
  }
```

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/src/app/test/spotify.service.mock.ts

```
  getProviders(): Array<any> {
    return [{ provide: SpotifyService, useValue: this }];
  }
```

This last method is a helper method to be used in `TestBed.configureTestingModule` `providers` like we'll see later when we get back to writing component tests.

Here's what our `MockSpotifyService` looks like altogether:

code/routes/music/src/app/test/spotify.service.mock.ts

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  fakeResponse;

  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
    this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
    this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
    this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
  }

  subscribe(callback) {
    callback(this.fakeResponse);
  }

  setResponse(json: any): void {
    this.fakeResponse = json;
  }

  getProviders(): Array<any> {
    return [{ provide: SpotifyService, useValue: this }];
  }
}
```

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/src/app/artist/artist.component.spec.ts

```
import {
  async,
  ComponentFixture,
  TestBed,
  inject,
  fakeAsync,
} from '@angular/core/testing';
import { Router } from '@angular/router';
import { Location } from '@angular/common';
import {
  advance,
  createRoot,
  RootCmp,
  configureMusicTests
} from '../test/test.module';

import { MockSpotifyService } from '../test/spotify.service.mock';
import { SpotifyService } from '../spotify.service';
import { ArtistComponent } from './artist.component';
```

Next, before we can start to describe our tests `configureMusicTests` to ensure we can access our `musicTestProviders` in each test:

code/routes/music/src/app/artist/artist.component.spec.ts

```
describe('ArtistComponent', () => {
  beforeEach(async(() => {
    configureMusicTests();
  }));
```

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/src/app/artist/artist.component.ts

```
export class ArtistComponent implements OnInit {
  id: string;
  artist: Object;

  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }

  ngOnInit(): void {
    this.spotify
      .getArtist(this.id)
      .subscribe((res: any) => this.renderArtist(res));
  }
```

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/src/app/artist/artist.component.spec.ts

```
  describe('initialization', () => {
    it('retrieves the artist', fakeAsync(
      inject([Router, SpotifyService],
             (router: Router,
              mockSpotifyService: MockSpotifyService) => {
        const fixture = createRoot(router, RootCmp);

        router.navigateByUrl('/artists/2');
        advance(fixture);

        expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
      })));
  });
```

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/src/app/test/test.module.ts

```
export function advance(fixture: ComponentFixture<any>): void {
  tick();
  fixture.detectChanges();
}
```

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 it's 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 it's 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 `TestBed.configureTestingModule` `providers`.

Notice that we're injecting:

1. `Router`
2. `SpotifyService`

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.

Testing `ArtistComponent`'s Initialization

Let's review the contents of our actual test:

code/routes/music/src/app/artist/artist.component.spec.ts

```
      const fixture = createRoot(router, RootCmp);

      router.navigateByUrl('/artists/2');
      advance(fixture);

      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/src/app/test/test.module.ts

```
export function createRoot(router: Router,
                           componentType: any): ComponentFixture<any> {
  const f = TestBed.createComponent(componentType);
  advance(f);
  (<any>router).initialNavigation();
  advance(f);
  return f;
}
```

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 it's `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.

Notice that we're using the `TestBed` library again to call `TestBed.createComponent`. This function creates a component of the appropriate type.

> `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/src/app/artist/artist.component.ts

```
  back(): void {
    this.location.back();
  }
```

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/src/app/artist/artist.component.spec.ts

```
  describe('back', () => {
    it('returns to the previous location', fakeAsync(
      inject([Router, Location],
             (router: Router, location: Location) => {
        const fixture = createRoot(router, RootCmp);
        expect(location.path()).toEqual('/');

        router.navigateByUrl('/artists/2');
        advance(fixture);
        expect(location.path()).toEqual('/artists/2');

        const artist = fixture.debugElement.children[1].componentInstance;
        artist.back();
        advance(fixture);

        expect(location.path()).toEqual('/');
      })));
  });
```

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/src/app/artist/artist.component.html

```html
<div *ngIf="artist">
  <h1>{{ artist.name }}</h1>

  <p>
    <img src="{{ artist.images[0].url }}">
  </p>

  <p><a href (click)="back()">Back</a></p>
</div>
```

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:

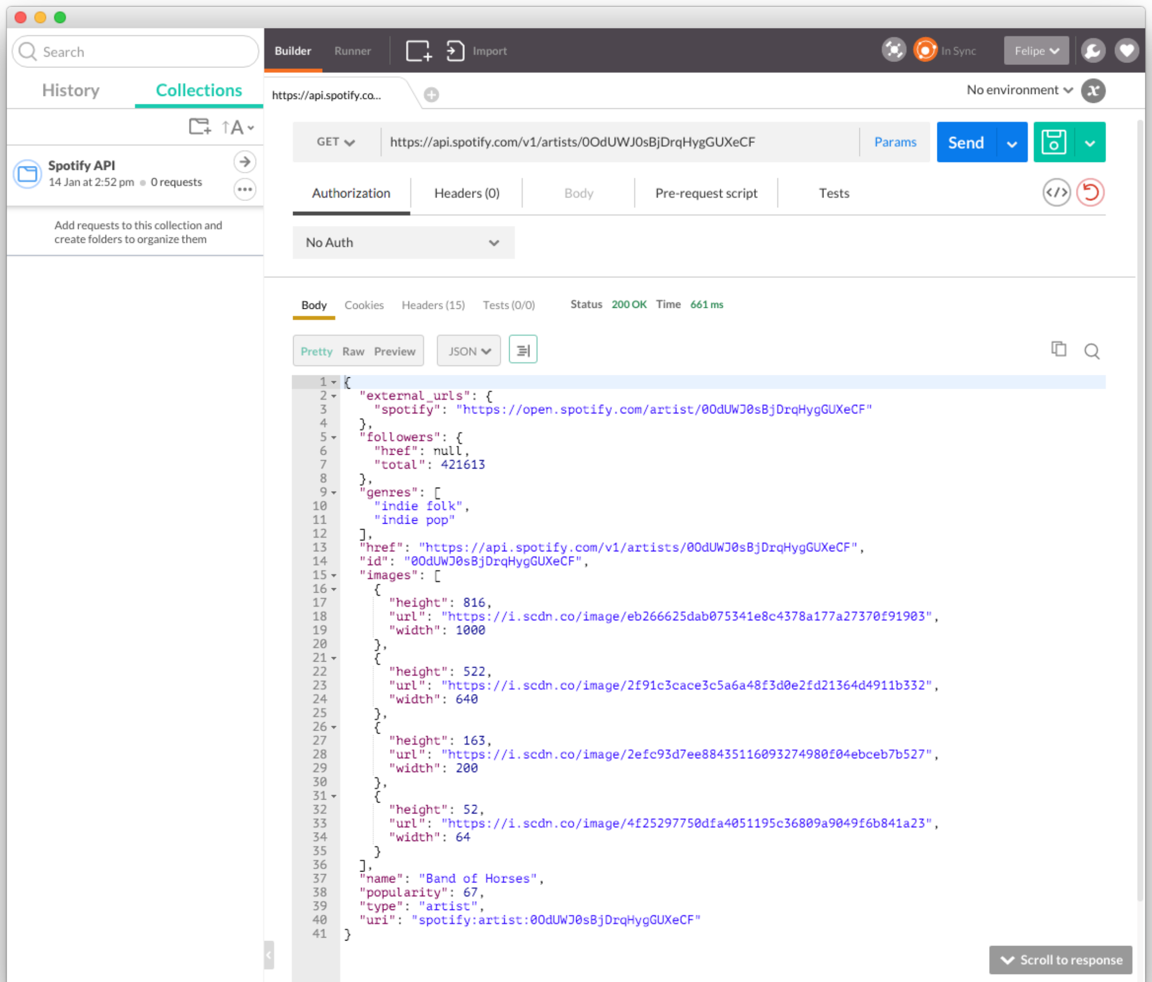
code/routes/music/src/app/artist/artist.component.spec.ts

```typescript
  describe('renderArtist', () => {
    it('renders album info', fakeAsync(
      inject([Router, SpotifyService],
             (router: Router,
              mockSpotifyService: MockSpotifyService) => {
        const fixture = createRoot(router, RootCmp);

        const artist = {name: 'ARTIST NAME', images: [{url: 'IMAGE_1'}]};
        mockSpotifyService.setResponse(artist);

        router.navigateByUrl('/artists/2');
        advance(fixture);

        const compiled = fixture.debugElement.nativeElement;

        expect(compiled.querySelector('h1').innerHTML).toContain('ARTIST NAME');
        expect(compiled.querySelector('img').src).toContain('IMAGE_1');
      })));
  });
```

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.nativeE`

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 `DemoFormWithEventsComponent` 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

As a reminder, here's the full code for that class:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.ts

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

@Component({
  selector: 'app-demo-form-with-events',
  templateUrl: './demo-form-with-events.component.html',
  styles: []
})
export class DemoFormWithEventsComponent implements OnInit {
  myForm: FormGroup;
  sku: AbstractControl;

  ngOnInit() {
  }
```

```

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];

    this.sku.valueChanges.subscribe(
      (value: string) => {
        console.log('sku changed to:', value);
      }
    );

    this.myForm.valueChanges.subscribe(
      (form: any) => {
        console.log('form changed to:', form);
      }
    );

  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form.sku);
  }

}
```

And the template:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with events</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form">

    <div class="field"
         [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
      <input type="text"
             class="form-control"
             id="skuInput"
```

```
            placeholder="SKU"
            [formControl]="sku">
      <div *ngIf="!sku.valid"
        class="ui error message">SKU is invalid</div>
      <div *ngIf="sku.hasError('required')"
        class="ui error message">SKU is required</div>
    </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

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/src/app/utils.ts

```
export class ConsoleSpy {
  public logs: string[] = [];
  log(...args) {
    this.logs.push(args.join(' '));
  }
  warn(...args) {
    this.log(...args);
  }
}
```

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*[137].
>
> 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[138] installed:

```
$ node --harmony
> var test = (a, b, ...theArgs) => console.log('a=',a,'b=',b,'theArgs=',theArgs);
undefined
> test(1,2,3,4,5);
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:

```
console.log('First message', 'is', 123);
console.log('Second message');
console.log('Third message');
```

We expect the `_logs` field to have an array of `['First message is 123', 'Second message', 'Third message']`.

[137] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/rest_parameters
[138] https://nodejs.org/en/

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/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
describe('DemoFormWithEventsComponent', () => {
  let component: DemoFormWithEventsComponent;
  let fixture: ComponentFixture<DemoFormWithEventsComponent>;

  let originalConsole, fakeConsole;
  let el, input, form;
```

And then we can install the fake console and specify our providers:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  beforeEach(async(() => {
    // replace the real window.console with our spy
    fakeConsole = new ConsoleSpy();
    originalConsole = window.console;
    (<any>window).console = fakeConsole;

    TestBed.configureTestingModule({
      imports: [ FormsModule, ReactiveFormsModule ],
      declarations: [ DemoFormWithEventsComponent ]
    })
    .compileComponents();
  }));
```

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/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  // restores the real console
  afterAll(() => (<any>window).console = originalConsole);
```

Configuring the Testing Module

Notice that in the `beforeEach` we call `TestBed.configureTestingModule` - remember that `configureTestingModule` sets up the root `NgModule` for our tests.

In this case we're importing the two forms modules and declaring the `DemoFormWithEvents` component.

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/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
  it('validates and triggers events', fakeAsync( () => {
    fixture = TestBed.createComponent(DemoFormWithEventsComponent);
    component = fixture.componentInstance;
    el = fixture.debugElement.nativeElement;
    input = fixture.debugElement.query(By.css('input')).nativeElement;
    form = fixture.debugElement.query(By.css('form')).nativeElement;
    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/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    input.value = '';
    dispatchEvent(input, 'input');
    fixture.detectChanges();
    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 `el` variable) for all child elements that are error messages and then making sure we have both error messages displayed:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    let msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs[0].innerHTML).toContain('SKU is invalid');
    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/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    fixture.detectChanges();
    tick();
```

And make sure all the error messages are gone:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs.length).toEqual(0);
```

Finally, we will trigger the submit event of the form:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    fixture.detectChanges();
    dispatchEvent(form, 'submit');
    tick();
```

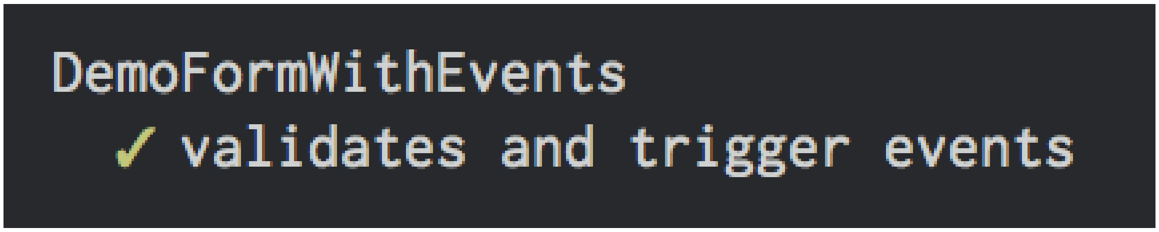
And finally we make sure the event was kicked by checking that the message we log to the console when the form is submitted is there:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts

```
    // checks for the form submitted message
    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/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
    fixture = TestBed.createComponent(DemoFormWithEventsComponent);
```

The `createComponent` code is pretty straightforward: Creates the component with `TestBed.createComponent`, retrieves all the elements we need and calls `detectChanges`.

Now the first thing we want to test is that given an empty SKU field, we should see two error messages:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  it('displays errors with no sku', fakeAsync( () => {
    input.value = '';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    // no value on sku field, all error messages are displayed
    const msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs[0].innerHTML).toContain('SKU is invalid');
    expect(msgs[1].innerHTML).toContain('SKU is required');
  }));
```

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/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  it('displays no errors when sku has a value', fakeAsync( () => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    const msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs.length).toEqual(0);
  }));
```

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/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  it('handles sku value changes', fakeAsync( () => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    tick();

    expect(fakeConsole.logs).toContain('sku changed to: XYZ');
  }));
```

We can write similar code for both the form change…

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

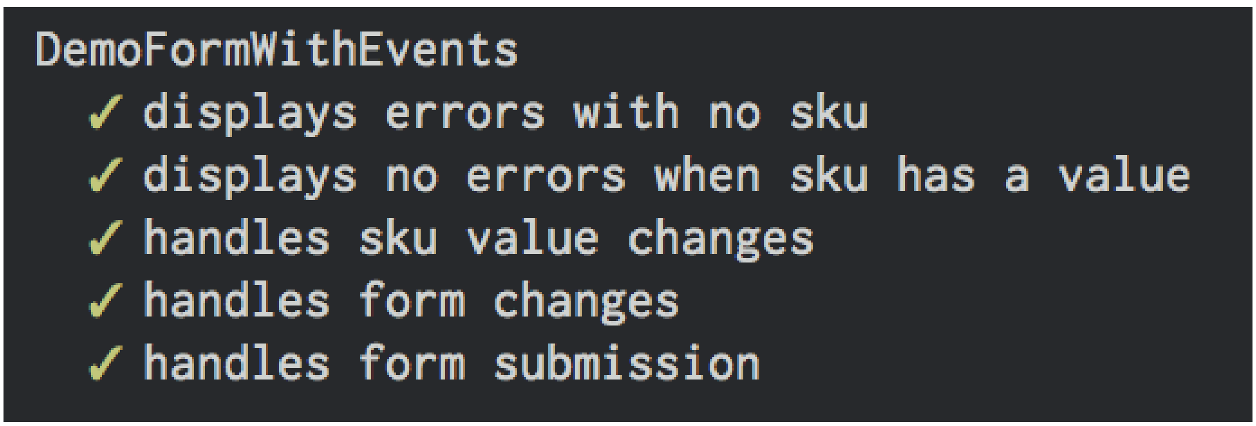
```
  it('handles form changes', fakeAsync(() => {
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    tick();

    expect(fakeConsole.logs).toContain('form changed to: [object Object]');
  }));
```

… and the form submission events:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts

```
  it('handles form submission', fakeAsync((tcb) => {
    input.value = 'ABC';
    dispatchEvent(input, 'input');
    tick();

    fixture.detectChanges();
    dispatchEvent(form, 'submit');
    tick();

    expect(fakeConsole.logs).toContain('you submitted value: ABC');
  }));
```

When we run the tests now, we get a much nicer output:

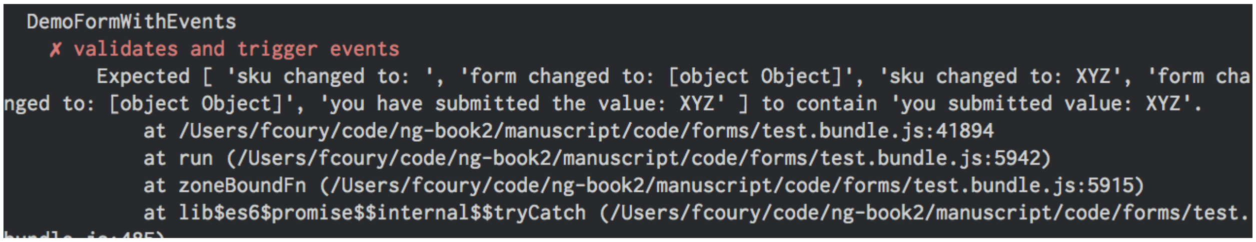


DemoFormWithEvents test output after refactoring

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:

```
onSubmit(form: any): void {
  console.log('you have submitted the value:', form.sku);
}
```

If we ran the previous version of the test, here's what would happen:



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 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` or `HttpClient` 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: `HttpTestingController`.

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 `HttpClient`.

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/src/app/more-http-requests/more-http-requests.component.ts

```
  makePost(): void {
    this.loading = true;
    this.http
      .post(
        'https://jsonplaceholder.typicode.com/posts',
        JSON.stringify({
          body: 'bar',
          title: 'foo',
          userId: 1
        })
      )
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
```

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:

First, we'll need to setup our tests to use the `HttpClientTestingModule` and `HttpTestingController`:

code/src/app/more-http-requests/more-http-requests.component.spec.ts

```
import {
  async,
  inject,
  ComponentFixture,
  TestBed
} from '@angular/core/testing';

import { HttpClient, HttpRequest, HttpHeaders } from '@angular/common/http';
import {
  HttpTestingController,
```

```
  HttpClientTestingModule
} from '@angular/common/http/testing';

import { MoreHttpRequestsComponent } from './more-http-requests.component';

describe('MoreHttpRequestsComponent', () => {
  let component: MoreHttpRequestsComponent;
  let fixture: ComponentFixture<MoreHttpRequestsComponent>;
  let httpMock: HttpTestingController;

  beforeEach(
    async(() => {
      TestBed.configureTestingModule({
        declarations: [MoreHttpRequestsComponent],
        imports: [HttpClientTestingModule]
      });
    })
  );

  beforeEach(
    async(
      inject([HttpTestingController], _httpMock => {
        fixture = TestBed.createComponent(MoreHttpRequestsComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
        httpMock = _httpMock;
      })
    )
  );

  afterEach(
    inject([HttpTestingController], (httpMock: HttpTestingController) => {
      httpMock.verify();
    })
  );
```

Above, we configure our testing module to import the `HttpClientTestingModule`. Then we inject `HttpTestingController` and store it as a variable in `httpMock`.

Now we're ready to write our test:

code/src/app/more-http-requests/more-http-requests.component.spec.ts

```
  it(
    'performs a POST',
    async(() => {
      component.makePost();

      const req = httpMock.expectOne(
        'https://jsonplaceholder.typicode.com/posts'
      );
      expect(req.request.method).toEqual('POST');
      req.flush({ response: 'OK' });
      expect(component.data).toEqual({ response: 'OK' });

      httpMock.verify();
    })
  );
```

We start by call the `makePost()` function directly on the component. This might look odd because we don't typically call functions directly on our components. But what we're trying to do here is cause the HTTP request to be made, that way we can test expectations on it.

Next we use the instance variable `httpMock` and expect that one request was made to `jsonplaceholder` by using the `expectOne` function.

The line `req.flush` will send a "mock" response to that HTTP request, and then we `expect` that the `component.data` matches that response.

Lastly, we call `httpMock.verify()` to finalize any remaining expectations.

Now that we understand how this works, adding a second test for DELETE method is straightforward.

Testing `DELETE`

Here's how the `makeDelete` method is implemented:

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makeDelete(): void {
    this.loading = true;
    this.http
      .delete('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
```

And this is the code we use to test it:

src/app/more-http-requests/more-http-requests.component.spec.ts

```
  it(
    'performs a DELETE',
    async(() => {
      component.makeDelete();

      const req = httpMock.expectOne(
        'https://jsonplaceholder.typicode.com/posts/1'
      );

      expect(req.request.method).toEqual('DELETE');
      req.flush({ response: 'OK' });
      expect(component.data).toEqual({ response: 'OK' });

      httpMock.verify();
    })
  );
```

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/src/app/more-http-requests/more-http-requests.component.ts

```
  makeHeaders(): void {
    const headers: HttpHeaders = new HttpHeaders({
      'X-API-TOKEN': 'ng-book'
    });

    const req = new HttpRequest(
      'GET',
      'https://jsonplaceholder.typicode.com/posts/1',
      {
        headers: headers
      }
    );

    this.http.request(req).subscribe(data => {
      this.data = data['body'];
    });
  }
```

In this case, what our test should focus on is making sure the header `X-API-TOKEN` is being properly set to `ng-book`:

src/app/more-http-requests/more-http-requests.component.spec.ts

```
  it(
    'sends correct headers',
    async(() => {
      component.makeHeaders();

      const req = httpMock.expectOne(
        req =>
          req.headers.has('X-API-TOKEN') &&
          req.headers.get('X-API-TOKEN') == 'ng-book'
      );

      req.flush({ response: 'OK' });
      expect(component.data).toEqual({ response: 'OK' });

      httpMock.verify();
    })
  );
```

The `req.headers` attribute returns the headers and we're using two methods to perform two different assertions:

- the `has` method to check whether a given header was set, ignoring it's 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`.

Now let's move to a more complex example.

Testing `YouTubeSearchService`

Back in the HTTP chapter we also built a YouTube video search. The HTTP interaction for that example takes place on a service called `YouTubeSearchService`:

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(
    private http: HttpClient,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string
  ) {}

  search(query: string): Observable<SearchResult[]> {
    const params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl).map(response => {
      return <any>response['items'].map(item => {
        // console.log("raw item", item); // uncomment if you want to debug
        return new SearchResult({
          id: item.id.videoId,
          title: item.snippet.title,
          description: item.snippet.description,
          thumbnailUrl: item.snippet.thumbnails.high.url
```

```
      });
    });
  });
 }
}
```

It uses the YouTube API to search for videos and parse the results into a `SearchResult` instance:

code/http/src/app/you-tube-search/search-result.model.ts

```
export class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id            || null;
    this.title           = obj && obj.title         || null;
    this.description     = obj && obj.description   || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl  || null;
    this.videoUrl        = obj && obj.videoUrl      ||
                         `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

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/src/app/you-tube-search/you-tube-search.component.before.spec.ts

```
describe('YouTubeSearchComponent (before)', () => {
  let component: YouTubeSearchComponent;
  let fixture: ComponentFixture<YouTubeSearchComponent>;

  beforeEach(
    async(() => {
      TestBed.configureTestingModule({
        declarations: [
          YouTubeSearchComponent,
          SearchResultComponent,
          SearchBoxComponent
        ],
        imports: [HttpClientTestingModule],
        providers: [
          YouTubeSearchService,
          { provide: YOUTUBE_API_KEY, useValue: 'YOUTUBE_API_KEY' },
          { provide: YOUTUBE_API_URL, useValue: 'YOUTUBE_API_URL' }
        ]
      });
    })
  );
```

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 `YouTubeSearchService` instance, but setting fake values for `YOUTUBE_API_KEY` and `YOUTUBE_API_URL` constants. We're also importing the `HttpClientTestingModule`.

Now, let's begin to write our first test case:

code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts

```
  describe('search', () => {
    it(
      'parses YouTube response',
      inject(
        [YouTubeSearchService, HttpTestingController],
        fakeAsync((service, httpMock) => {
          let res;

          service.search('hey').subscribe(_res => {
            res = _res;
          });

          const req = httpMock.expectOne({ method: 'GET' });
```

```
        req.flush({
          items: [
            {
              id: { videoId: 'VIDEO_ID' },
              snippet: {
                title: 'TITLE',
                description: 'DESCRIPTION',
                thumbnails: {
                  high: { url: 'THUMBNAIL_URL' }
                }
              }
            }
          ]
        });

        tick();

        const video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
        expect(video.title).toEqual('TITLE');
        expect(video.description).toEqual('DESCRIPTION');
        expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');

        httpMock.verify();
      })
    )
  );
});
```

Here 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.

Here we are telling `HttpClient` 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 `req.flush` method of the connection.

code/http/src/app/you-tube-search/you-tube-search.component.before.spec.ts

```
        req.flush({
          items: [
            {
              id: { videoId: 'VIDEO_ID' },
              snippet: {
                title: 'TITLE',
                description: 'DESCRIPTION',
                thumbnails: {
                  high: { url: 'THUMBNAIL_URL' }
                }
              }
            }
          ]
        });
```

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/src/app/you-tube-search/you-tube-search.component.before.spec.ts

```
        const video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
        expect(video.title).toEqual('TITLE');
        expect(video.description).toEqual('DESCRIPTION');
        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/src/app/you-tube-search/you-tube-search.component.spec.ts

```
  function search(term: string, response: any, callback) {
    return inject(
      [YouTubeSearchService, HttpTestingController],
      fakeAsync((service, httpMock) => {
        let res;

        // search
        service.search(term).subscribe(_res => {
          res = _res;
        });

        const req = httpMock.expectOne({ method: 'GET' });
        req.flush(response);
        tick();

        callback(req.request, res);
      })
    );
  }
```

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/src/app/you-tube-search/you-tube-search.component.spec.ts

```
  it(
    'parses YouTube video id',
    search('hey', defaultResponse, (req, res) => {
      const video = res[0];
      expect(video.id).toEqual('VIDEO_ID');
    })
  );

  it(
    'parses YouTube video title',
    search('hey', defaultResponse, (req, res) => {
      const video = res[0];
```

```
        expect(video.title).toEqual('TITLE');
      })
    );

    it(
      'parses YouTube video description',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.description).toEqual('DESCRIPTION');
      })
    );

    it(
      'parses YouTube video thumbnail',
      search('hey', defaultResponse, (req, res) => {
        const video = res[0];
        expect(video.description).toEqual('DESCRIPTION');
      })
    );
```

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/src/app/you-tube-search/you-tube-search.component.spec.ts

```
    it(
      'sends the query',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toContain('q=term');
      })
    );

    it(
      'sends the API key',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toContain('key=YOUTUBE_API_KEY');
      })
    );

    it(
      'uses the provided YouTube URL',
      search('term', defaultResponse, (req, res) => {
        expect(req.url).toMatch(/^YOUTUBE_API_URL\?/);
```

```
    })
  );
```

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.

Converting an AngularJS 1.x App to Angular

If you've been using Angular for a while, then you probably already have production AngularJS 1 apps. Angular is great, but there's no way we can drop everything and rewrite our entire production apps in Angular. What we need is a way to *incrementally* upgrade our AngularJS 1 app. Thankfully, Angular has a fantastic way to do that.

The interoperability of AngularJS 1 (ng1) and Angular (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 AngularJS 1 and Angular, there's a lot of peripheral concepts. For instance:

Mapping AngularJS 1 Concepts to Angular: 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: AngularJS 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[139]) 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:

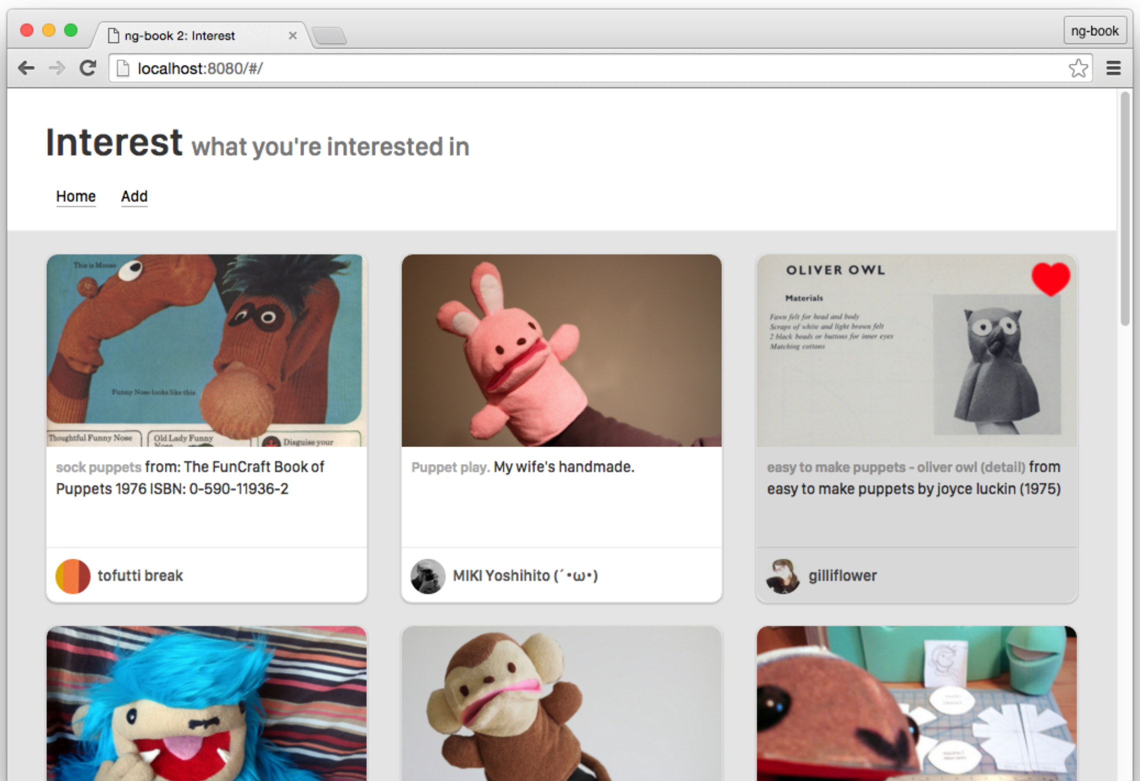
Writing hybrid ng1/ng2 apps: ng2 provides a way to bootstrap your ng1 app and then write ng2 components and services. You can write ng2 components that will mix with ng1 components and it "just works". Furthermore, the dependency injection system supports passing between ng1 and ng2 (both directions), so you can write services which will run in either ng1 or ng2.

[139] http://teropa.info/blog/2014/10/24/how-ive-improved-my-angular-apps-by-banning-ng-controller.html

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/upgrade/ng1` and `code/conversion/hybrid`

The hybrid app is written using Angular CLI. In order to run it, change into the directory and type:

```
npm install
npm start
```

Before we dive in, let's set the stage for interoperability between ng1 and ng2

Mapping AngularJS 1 to Angular

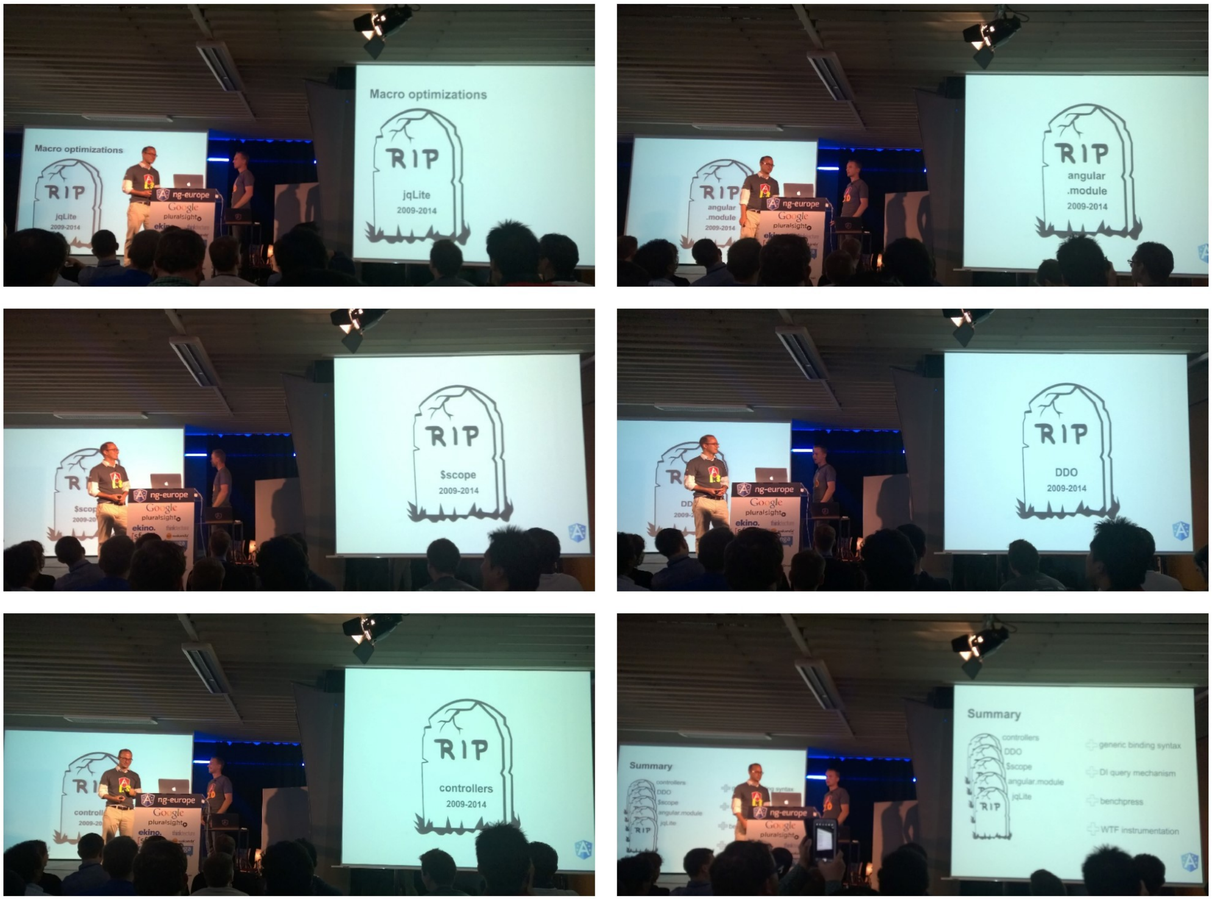
From a high level, the five main parts of AngularJS 1 are:

- Directives
- Controllers
- Scopes
- Services
- Dependency Injection

Angular 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 AngularJS 1 (video here[140]). Specifically, they announced that Angular was killing off:

- `$scope` (& two-way binding by default)
- Directive Definition Objects
- Controllers
- `angular.module`

[140] https://www.youtube.com/watch?v=gNmWybAyBHI



Igor and Tobias killing off many APIs from 1.x. at ngEurope 2014. Photo Credit: Michael Bromley (used with permission)

As someone who's built AngularJS 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 is, it turns out, a lot of the same ideas are still with us and, in fact, Angular provides just as much functionality but with **a much simpler model**.

At a high-level Angular 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 Components in AngularJS 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 AngularJS 1 Components in Angular**: 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 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 AngularJS 1 App

To set the stage, let's go over the AngularJS 1 version of our app.

> This chapter assumes some knowledge of AngularJS 1 and ui-router[141]. If you're not comfortable with AngularJS 1 yet, check out ng-book 1[142].
>
> We won't be diving too deeply into explaining each AngularJS 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.

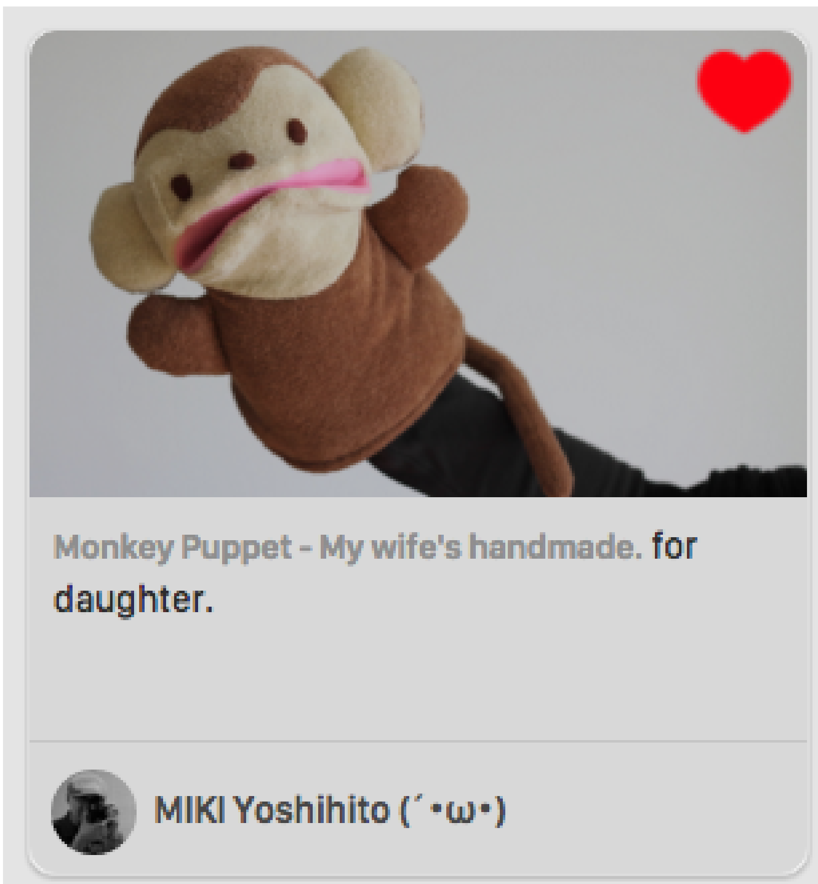
[141] https://github.com/angular-ui/ui-router
[142] http://ng-book.com

```
cd code/upgrade/ng1  # change directories
npm install            # install dependencies
npm run go             # run the app
```

If your browser doesn't open automatically, open the url: http://localhost:8080[143].

> Note that the AngularJS 1 app in `ng1` will run on port `8080` whereas the hybrid app (discussed below) will run on port `4200`.

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.

[143] http://localhost:8080

The ng1-app HTML

The `index.html` in our ng1 app uses a common structure:

code/upgrade/ng1/index.html

```
<!DOCTYPE html>
<html ng-app='interestApp'>
<head>
  <meta charset="utf-8">
  <title>Interest</title>
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/sf.css">
  <link rel="stylesheet" href="css/interest.css">
</head>
<body class="container-fullwidth">

  <div class="page-header">
    <div class="container">
      <h1>Interest <small>what you're interested in</small></h1>

      <div class="navLinks">
        <a ui-sref='home' id="navLinkHome">Home</a>
        <a ui-sref='add' id="navLinkAdd">Add</a>
      </div>
    </div>
  </div>

  <div id="content">
    <div ui-view=''></div>
  </div>

  <script src="js/vendor/lodash.js"></script>
  <script src="js/vendor/angular.js"></script>
  <script src="js/vendor/angular-ui-router.js"></script>
  <script src="js/app.js"></script>
</body>
</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/upgrade/ng1/js/app.js

```
angular.module('interestApp', ['ui.router'])
.service('PinsService', function($http, $q) {
  this._pins = null;

  this.pins = function() {
    var self = this;
    if(self._pins == null) {
      // initialize with sample data
      return $http.get("/js/data/sample-data.json").then(
        function(response) {
          self._pins = response.data;
          return self._pins;
        })
    } else {
      return $q.when(self._pins);
    }
  }

  this.addPin = function(newPin) {
```

```
    // adding would normally be an API request so lets mock async
    return $q.when(
      this._pins.unshift(newPin)
    );
  }
})
```

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/upgrade/ng1/js/data/sample-data.json


```
[
  {
    "title": "sock puppets",
    "description": "from:\nThe FunCraft Book of Puppets\n1976\nISBN: 0-590-11936-2",
    "user_name": "tofutti break",
    "avatar_src": "images/avatars/42826303@N00.jpg",
    "src": "images/pins/106033588_167d811702_o.jpg",
    "url": "https://www.flickr.com/photos/tofuttibreak/106033588/",
    "faved": false,
    "id": "106033588"
  },
  {
    "title": "Puppet play.",
    "description": "My wife's handmade.",
    "user_name": "MIKI Yoshihito (´□ω□)",
    "avatar_src": "images/avatars/7940758@N07.jpg",
    "src": "images/pins/4422575066_7d5c4c41e7_o.jpg",
    "url": "https://www.flickr.com/photos/mujitra/4422575066/",
    "faved": false,
    "id": "4422575066"
  },
  {
    "title": "easy to make puppets - oliver owl (detail)",
    "description": "from easy to make puppets by joyce luckin (1975)",
    "user_name": "gilliflower",
    "avatar_src": "images/avatars/26265986@N00.jpg",
    "src": "images/pins/6819859061_25d05ef2e1_o.jpg",
    "url": "https://www.flickr.com/photos/gilliflower/6819859061/",
    "faved": false,
    "id": "6819859061"
  },
```

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[144].

As we mentioned, we're going to have two routes:

code/upgrade/ng1/js/app.js

```
.config(function($stateProvider, $urlRouterProvider) {
  $stateProvider
    .state('home', {
      templateUrl: '/templates/home.html',
      controller: 'HomeController as ctrl',
      url: '/',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })
    .state('add', {
      templateUrl: '/templates/add.html',
      controller: 'AddController as ctrl',
      url: '/add',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })

    $urlRouterProvider.when('', '/');
})
```

[144] https://github.com/angular-ui/ui-router/wiki

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/upgrade/ng1/js/app.js

```
.controller('HomeController', function(pins) {
  this.pins = pins;
})
```

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/upgrade/ng1/templates/home.html

```
<div class="container">
  <div class="row">
    <pin item="pin" ng-repeat="pin in ctrl.pins">
    </pin>
  </div>
</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/upgrade/ng1/js/app.js

```
})
.directive('pin', function() {
  return {
    restrict: 'E',
    templateUrl: '/templates/pin.html',
    scope: {
      'pin': "=item"
    },
    link: function(scope, elem, attrs) {
      scope.toggleFav = function() {
        scope.pin.faved = !scope.pin.faved;
      }
    }
  }
})
```

This directive shouldn't be taken as an example of directive using the current best-practices. For instance, if I was writing this component anew (in ng1) I would probably use the new `.component` directive available in AngularJS 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/upgrade/ng1/templates/pin.html

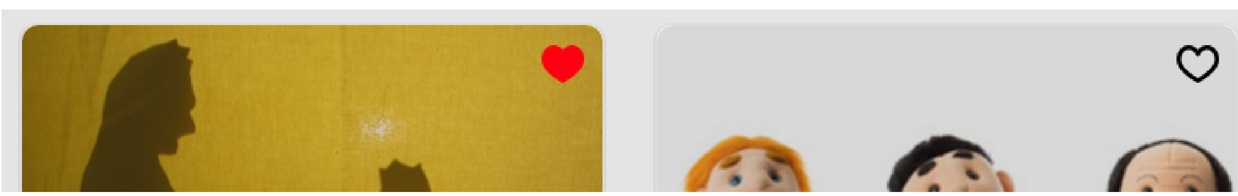
```
<div class="col-sm-6 col-md-4">
  <div class="thumbnail">
    <div class="content">
      <img ng-src="{{pin.src}}" class="img-responsive">
      <div class="caption">
        <h3>{{pin.title}}</h3>
        <p>{{pin.description | truncate:100}}</p>
      </div>
      <div class="attribution">
        <img ng-src="{{pin.avatar_src}}" class="img-circle">
        <h4>{{pin.user_name}}</h4>
```

```
      </div>
    </div>
    <div class="overlay">
      <div class="controls">
        <div class="heart">
          <a ng-click="toggleFav()">
            <img src="/images/icons/Heart-Empty.png" ng-if="!pin.faved"></img>
            <img src="/images/icons/Heart-Red.png"   ng-if="pin.faved"></img>
          </a>
        </div>
      </div>
    </div>
  </div>
</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/upgrade/ng1/js/app.js

```
.controller('AddController', function($state, PinsService, $timeout) {
  var ctrl = this;
  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/upgrade/ng1/js/app.js

```
  var makeNewPin = function() {
    return {
      "title": "Steampunk Cat",
      "description": "A cat wearing goggles",
      "user_name": "me",
      "avatar_src": "images/avatars/me.jpg",
      "src": "/images/pins/cat.jpg",
      "url": "http://cats.com",
      "faved": false,
      "id": Math.floor(Math.random() * 10000).toString()
    }
  }

  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/upgrade/ng1/js/app.js

```
  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
})
```

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 `PinsService.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`:

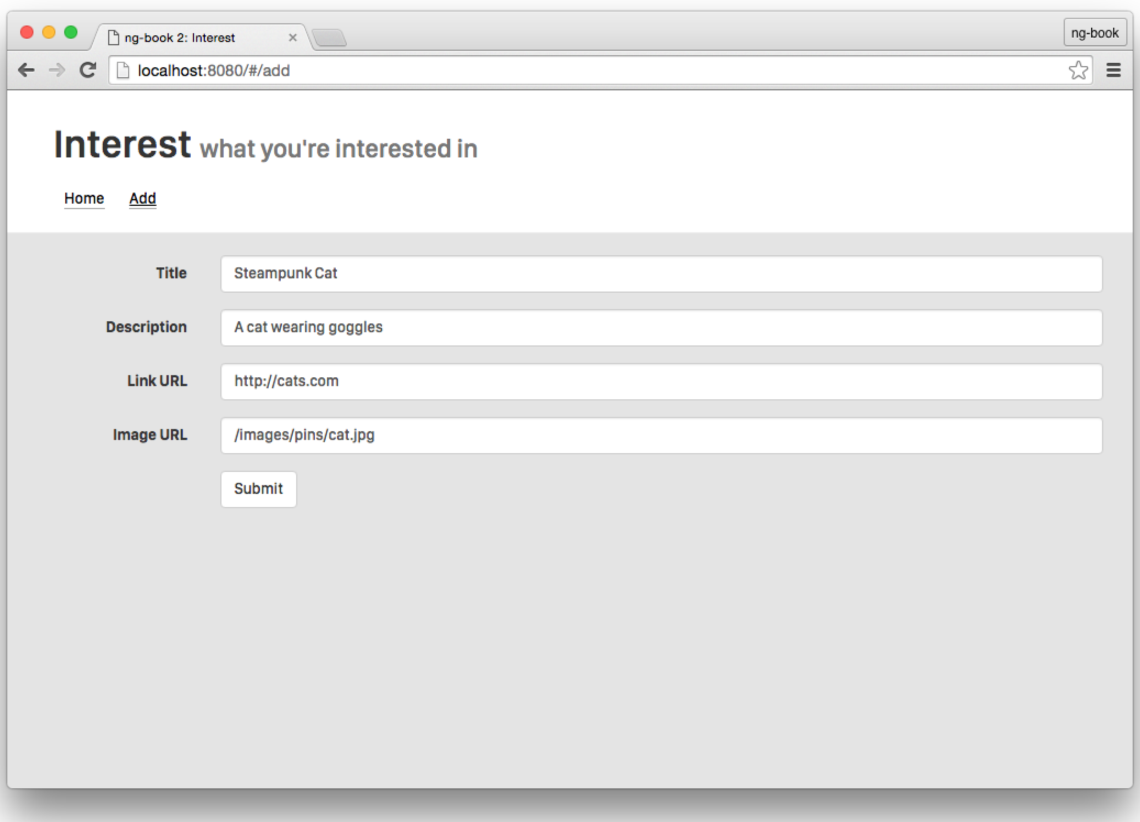
code/upgrade/ng1/js/app.js

```
.controller('AddController', function($state, PinsService, $timeout) {
  var ctrl = this;
  ctrl.saving = false;

  var makeNewPin = function() {
    return {
      "title": "Steampunk Cat",
      "description": "A cat wearing goggles",
      "user_name": "me",
      "avatar_src": "images/avatars/me.jpg",
      "src": "/images/pins/cat.jpg",
      "url": "http://cats.com",
      "faved": false,
      "id": Math.floor(Math.random() * 10000).toString()
    }
  }

  ctrl.newPin = makeNewPin();

  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
```

```
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
})
```

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/upgrade/ng1/templates/add.html

```
<div class="container">
  <div class="row">

    <form class="form-horizontal">

      <div class="form-group">
        <label for="title"
               class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="title"
                 placeholder="Title"
                 ng-model="ctrl.newPin.title">
        </div>
      </div>

      <div class="form-group">
        <label for="description"
               class="col-sm-2 control-label">Description</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="description"
                 placeholder="Description"
                 ng-model="ctrl.newPin.description">
        </div>
      </div>

      <div class="form-group">
        <label for="url"
               class="col-sm-2 control-label">Link URL</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="url"
                 placeholder="Link URL"
                 ng-model="ctrl.newPin.url">
        </div>
      </div>

      <div class="form-group">
```

```
        <label for="url"
               class="col-sm-2 control-label">Image URL</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="url"
                 placeholder="Image URL"
                 ng-model="ctrl.newPin.src">
        </div>
      </div>

      <div class="form-group">
        <div class="col-sm-offset-2 col-sm-10">
          <button type="submit"
                  class="btn btn-default"
                  ng-click="ctrl.submitPin()">Submit</button>
        </div>
      </div>
      <div ng-if="ctrl.saving">
        Saving...
      </div>
    </form>

  </div>
</div>
```

ng1: Summary

There we have it. This app has just the right amount of complexity that we can start porting it to Angular.

Building A Hybrid

Now we're ready to start putting some Angular in our AngularJS 1 app.

Before we start using Angular 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`.

To run it, run:

```
npm install
npm start
```

Then open your browser to http://localhost:4200 – note that this is a **different URL** than the pure-AngularJS 1 app above.

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 AngularJS 1 libraries (in `js/vendor`) and we're loading the Angular libraries from `npm`.

In your project, you might want to vendor them both, use bower[145], etc. However, using `npm` is very convenient for Angular, and so we suggest using `npm` to install Angular.

One of the first challenges we face when making a hybrid app is ensuring our build-process can support both JavaScript and TypeScript files, as well as resolving our assets, type-definitions, and so on.

Here we're using Angular CLI (which is based on Webpack) in order to build this app. We'll describe the specific steps necessary to get our app running within Angular CLI, but if you have an existing build process, it might take some additional work to get it in order.

Dependencies with `package.json`

You install dependencies with `npm` using the `package.json` file. Here's our `package.json` for the hybrid example:

code/upgrade/hybrid/package.json

```
{
  "name": "hybrid",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
```

[145] http://bower.io/

```
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "7.2.0",
    "@angular/compiler": "7.2.0",
    "@angular/core": "7.2.0",
    "@angular/forms": "7.2.0",
    "@angular/http": "7.2.0",
    "@angular/platform-browser": "7.2.0",
    "@angular/platform-browser-dynamic": "7.2.0",
    "@angular/router": "7.2.0",
    "@angular/upgrade": "6.0.0",
    "@types/jasmine": "2.8.8",
    "@types/jasminewd2": "2.0.3",
    "core-js": "2.4.1",
    "reflect-metadata": "0.1.3",
    "rxjs": "6.3.0",
    "ts-node": "7.0.1",
    "zone.js": "0.8.26",
    "@angular/animations": "7.2.0"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "0.12.0-rc.0",
    "@angular-devkit/build-optimizer": "0.5.7",
    "@angular/cli": "7.2.0-rc.0",
    "@angular/compiler-cli": "7.2.0",
    "@types/angular-ui-router": "1.1.40",
    "@types/jasmine": "2.5.38",
    "@types/node": "10.9.4",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "0.2.0",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "0.2.2",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
```

```
    "tslint": "~4.4.2",
    "typescript": "~3.2.2",
    "@angular/language-service": "7.2.0"
  }
}
```

> 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.

Notice that we've included the `@angular/upgrade` package. This module contains the tools necessary for booting a hybrid app.

Compiling our code

We're going to be using TypeScript in this example alongside our JavaScript AngularJS 1 code. To do this, we're going to put all of our "old" JavaScript code in the folder `js/`.

We also want to load AngularJS, as well as `angular-ui-router` and our AngularJS 1 app. Here, to do this we're going to include them in the `scripts` tag of our `.angular-cli.json`

```
{
  "apps": [
    {
      // ...
      "scripts": [
        "js/vendor/angular.js",
        "js/vendor/angular-ui-router.js",
        "js/app.js"
      ],
    }
  ]
}
```

> This step may vary depending on your build process. For instance, if you have an existing AngularJS app you may have an existing build process that builds that app into one or a few files (e.g. using Gulp or another build system). In that case, if you want to bring that build into your Angular CLI project, you could have a separate step that would build those files and import them into "scripts" here.
>
> In the case that you want a more unified workflow, you'll need to run `ng eject` and modify the generated Webpack file from there.
>
> That said, building custom Webpack configurations is beyond the scope of this book.

When we write hybrid ng2 apps **the Angular code becomes the entry point**. This makes sense because **it's Angular that's providing the backwards compatibility with AngularJS 1**. Let's take a closer look at the bootstrapping process.

Bootstrapping our Hybrid App

Now that we have our project structure in place, let's bootstrap the app.

If you recall, with AngularJS 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:

```
// code/upgrade/hybrid/src/app/app.module.ts
import {
  NgModule,
  forwardRef
} from '@angular/core';
import { CommonModule } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';

import { UpgradeAdapter } from '@angular/upgrade';
declare var angular: any;

/*
 * Create our upgradeAdapter
 */
const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter(
  forwardRef(() => MyAppModule)); // <-- notice forward reference

// ...
// upgrade and downgrade components in here
// ...

/*
 * Create our app's entry NgModule
 */
```

```
@NgModule({
  declarations: [ MyNg2Component, ... ],
  imports: [
    CommonModule,
    BrowserModule
  ],
  providers: [ MyNg2Services, ... ]
})
class MyAppModule { }

/*
 * Bootstrap the App
 */
upgradeAdapter.bootstrap(document.body, ['interestApp']);
```

We start by importing the `UpgradeAdapter` and then we create an instance of it: `upgradeAdapter`.

However, the constructor of `UpgradeAdapter` requires an `NgModule` that we'll be using for our Angular up - but we haven't defined it yet! To get around this we use the `forwardRef` function which allows us to take a 'forward reference' to our `NgModule` which we declare below.

When we define our `NgModule` `MyAppModule` (or specifically in this app it will be `InterestAppModule`), we define it like we would any other Angular `NgModule`: we put in our declarations, imports, providers, etc.

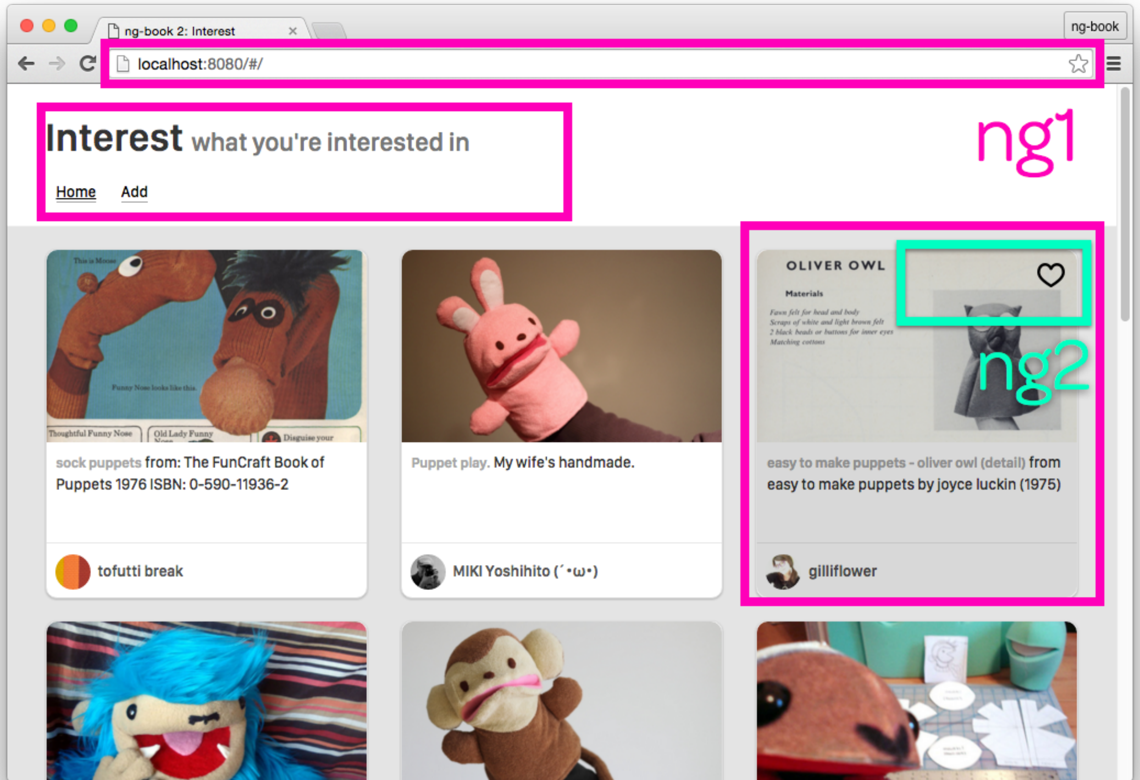
Lastly, we tell the `upgradeAdapter` to `bootstrap` our app on the element `document.body` and we specify the module name of our **AngularJS 1 app**.

This will bootstrap our AngularJS 1 app within our Angular app! Now we can start replacing pieces with Angular.

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



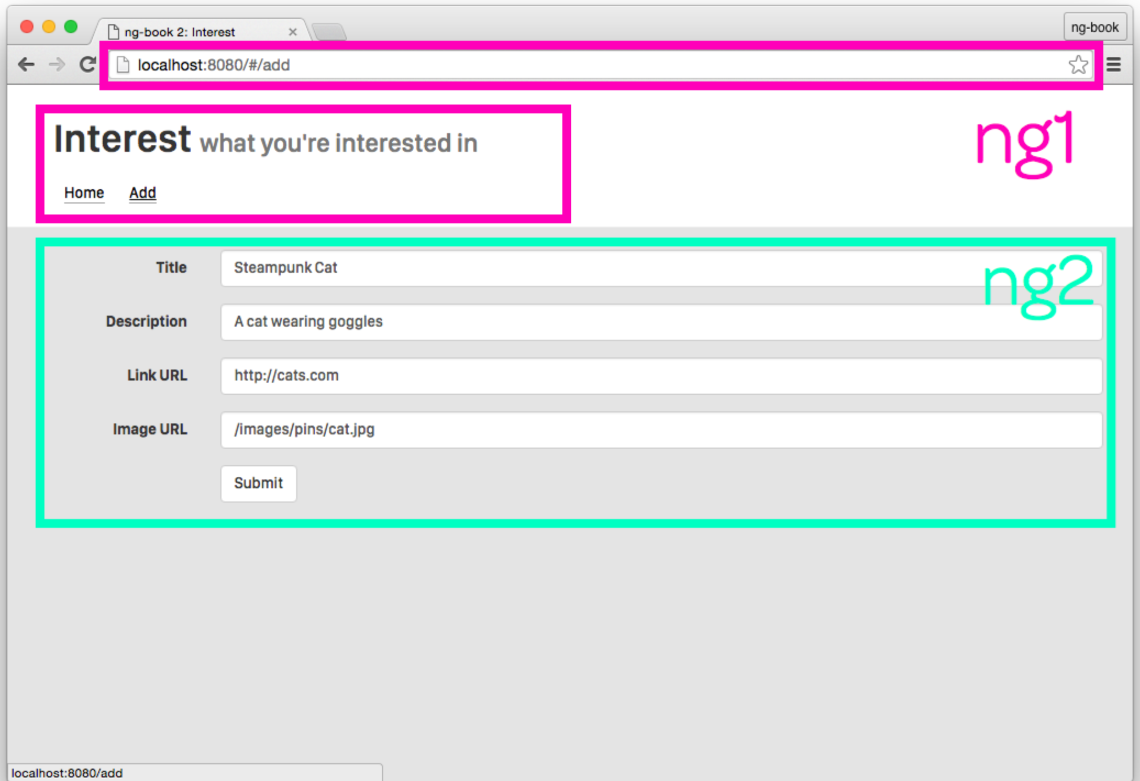
Homepage ng1 and ng2 Components

The first thing to notice is that we're going to continue to manage routing with ng1. Of course, Angular 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 AngularJS 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 decorators*. 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/upgrade/ng1/js/app.js

```
var makeNewPin = function() {
  return {
    "title": "Steampunk Cat",
    "description": "A cat wearing goggles",
    "user_name": "me",
    "avatar_src": "images/avatars/me.jpg",
    "src": "/images/pins/cat.jpg",
    "url": "http://cats.com",
    "faved": false,
    "id": Math.floor(Math.random() * 10000).toString()
  }
}

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 / 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/upgrade/hybrid/src/js/app.d.ts

```
interface Pin {
  title: string;
  description: string;
  user_name: string;
  avatar_src: string;
  src: string;
  url: string;
  faved: boolean;
  id: string;
}
```

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 adding a `reference` to `typings.d.ts`. For instance in `typings.d.ts` we'll add this:

```
/// <reference path="./js/app.d.ts"/>
```

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 `@types`

Typescript allows for loading third-party types via NPM.

We're going to use `angular-ui-router` with our app, so let's install the typings for `angular-ui-router`. To get this setup, all we have to do is install the `@types/angular-ui-router` package.

```
npm install @types/angular-ui-router --save
```

Now, by default, TypeScript will read types from the `node_modules/@types/` directory. We'll look at how we uses these types in our code in a moment.

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/upgrade/hybrid/src/js/app.d.ts

```
interface Pin {
  title: string;
  description: string;
  user_name: string;
  avatar_src: string;
  src: string;
  url: string;
  faved: boolean;
  id: string;
}

interface PinsService {
  pins(): Promise<Pin[]>;
  addPin(pin: Pin): Promise<any>;
}
```

Here we're 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[146]
- TypeScript Handbook: Writing definition files[147]
- Quick tip: Typescript declare keyword[148]

Now that we have this file setup, TypeScript will know about the `Pin` and `PinsService` types in our code.

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.

Next, let's write our component:

[146]http://www.typescriptlang.org/Handbook#modules-working-with-other-javascript-libraries
[147]https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Writing%20Definition%20Files.md
[148]http://blogs.microsoft.co.il/gilf/2013/07/22/quick-tip-typescript-declare-keyword/

code/upgrade/hybrid/src/app/pin-controls/pin-controls.component.ts

```
import {
  Component,
  Input,
  Output,
  EventEmitter
} from '@angular/core';

@Component({
  selector: 'pin-controls',
  templateUrl: './pin-controls.component.html',
  styleUrls: ['./pin-controls.component.css']
})
export class PinControlsComponent {
  @Input() pin: Pin;
  @Output() faved: EventEmitter<Pin> = new EventEmitter<Pin>();

  toggleFav(): void {
    this.faved.emit(this.pin);
  }
}
```

Notice here that we'll match the element `pin-controls`.

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/upgrade/hybrid/src/app/pin-controls/pin-controls.component.html

```
<div class="controls">
  <div class="heart">
    <a (click)="toggleFav()">
      <img src="/assets/images/icons/Heart-Empty.png" *ngIf="!pin.faved" />
      <img src="/assets/images/icons/Heart-Red.png"   *ngIf="pin.faved" />
    </a>
  </div>
</div>
```

Notice that instead of specifying `inputs` and `outputs` in the `@Component` decorator, in this case we're annotating the properties on the class directly with the `@Input` and `@Output` decorators. 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 now use it in a **AngularJS 1** template. Here's what our `pin.html` template looks like now:

code/upgrade/hybrid/src/assets/templates/pin.html

```
<div class="col-sm-6 col-md-4">
  <div class="thumbnail">
    <div class="content">
      <img ng-src="{{pin.src}}" class="img-responsive">
      <div class="caption">
        <h3>{{pin.title}}</h3>
        <p>{{pin.description | truncate:100}}</p>
      </div>
      <div class="attribution">
        <img ng-src="{{pin.avatar_src}}" class="img-circle">
        <h4>{{pin.user_name}}</h4>
      </div>
    </div>
    <div class="overlay">
      <pin-controls [pin]="pin"
                    (faved)="toggleFav($event)"></pin-controls>
    </div>
  </div>
</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:

```
<pin-controls [pin]="pin"
              (faved)="toggleFav($event)"></pin-controls>
```

What's interesting here is that we're using the ng2 input bracket syntax `[pin]` and the ng2 output parentheses 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.module.ts` file

First we need to import the necessary libraries and declare the `angular` variable:

code/upgrade/hybrid/src/app/app.module.ts

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

import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { AddPinComponent } from './add-pin/add-pin.component';
import { PinControlsComponent } from './pin-controls/pin-controls.component';
import { AnalyticsService } from './analytics.service';

declare var angular: any;
```

Then we create a `.directive` in (almost) the normal ng1 way:

code/upgrade/hybrid/src/app/app.module.ts

```
declare var angular: any;

/*
 * Create our upgradeAdapter
 */
export const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter(
  forwardRef(() => AppModule));

  /*
 * Expose our ng2 content to ng1
 */
angular.module('interestApp')
  .directive('pinControls',
             upgradeAdapter.downgradeNg2Component(PinControlsComponent))
```

Remember that our ng1 app 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`.

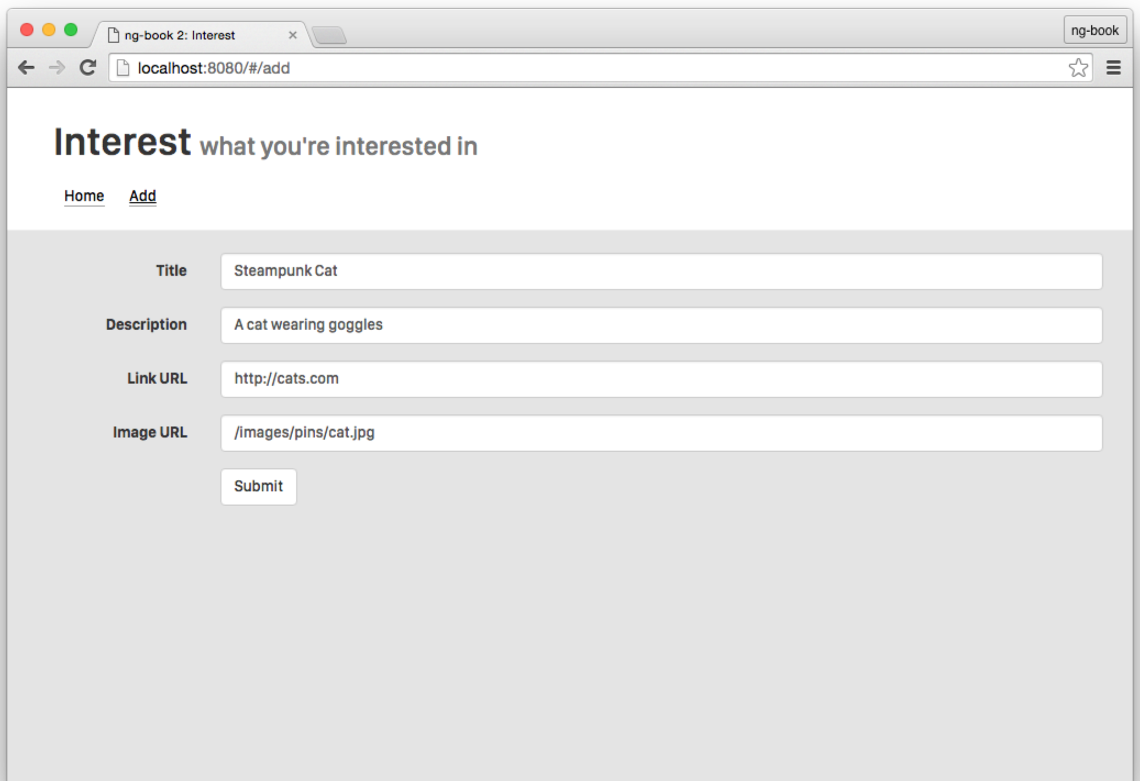
`downgradeNg2Component` will convert our `PinControlsComponent` into an ng1-compatible directive. Pretty neat.

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.



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 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/upgrade/hybrid/src/app/app.module.ts

```
/*
 * Expose our ng1 content to ng2
 */
upgradeAdapter.upgradeNg1Provider('PinsService');
upgradeAdapter.upgradeNg1Provider('$state');
```

And that's it. Now we can `@Inject` our ng1 services into ng2 components like so:

```
// angular.ui.IStateService is available because we've
// installed @types/angular-ui-router in our package.json
type IStateService = angular.ui.IStateService;

class AddPinComponent {
  constructor(@Inject('PinsService') public pinsService: PinsService,
              @Inject('$state') public uiState: IStateService) {
  }
  // ...
  // now you can use this.pinsService
  // or this.uiState
  // ...
}
```

In this constructor, there's a few things to look at:

The `@Inject` decorator, 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/upgrade/hybrid/src/js/app.d.ts

```
interface PinsService {
  pins(): Promise<Pin[]>;
  addPin(pin: Pin): Promise<any>;
}
```

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/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
declare var angular: any;
import {
  Component,
  Inject
} from '@angular/core';
// angular.ui.IStateService is available because we've
// installed @types/angular-ui-router in our package.json
type IStateService = angular.ui.IStateService;
```

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` is straightforward:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
@Component({
  selector: 'add-pin',
  templateUrl: './add-pin.component.html',
  styleUrls: ['./add-pin.component.css']
})
```

`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 forms work, checkout the forms chapter, where we describe forms in depth.

code/upgrade/hybrid/src/app/add-pin/add-pin.component.html

```
<div class="container">
  <div class="row">

    <form (ngSubmit)="onSubmit()"
          class="form-horizontal">

      <div class="form-group">
        <label for="title"
              class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text"
                class="form-control"
                id="title"
                name="title"
                placeholder="Title"
                [(ngModel)]="newPin.title">
        </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/upgrade/hybrid/src/app/add-pin/add-pin.component.html

```
<div class="container">
  <div class="row">

    <form (ngSubmit)="onSubmit()"
          class="form-horizontal">

      <div class="form-group">
        <label for="title"
               class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="title"
                 name="title"
                 placeholder="Title"
                 [(ngModel)]="newPin.title">
        </div>
      </div>

      <div class="form-group">
        <label for="description"
               class="col-sm-2 control-label">Description</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="description"
                 name="description"
                 placeholder="Description"
                 [(ngModel)]="newPin.description">
        </div>
      </div>

      <div class="form-group">
        <label for="url"
               class="col-sm-2 control-label">Link URL</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="url"
                 name="url"
                 placeholder="Link URL"
                 [(ngModel)]="newPin.url">
```

```
        </div>
      </div>

      <div class="form-group">
        <label for="url"
               class="col-sm-2 control-label">Image URL</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="url"
                 name="url"
                 placeholder="Image URL"
                 [(ngModel)]="newPin.src">
        </div>
      </div>

      <div class="form-group">
        <div class="col-sm-offset-2 col-sm-10">
          <button type="submit"
                  class="btn btn-default"
                  >Submit</button>
        </div>
      </div>
      <div *ngIf="saving">
        Saving...
      </div>
    </form>
```

`AddPinComponent` Controller

Now we can define `AddPinComponent`. We start by setting up two instance variables:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
export class AddPinComponent {
  saving = false;
  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/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
  constructor(@Inject('PinsService') private pinsService: PinsService,
              @Inject('$state') private uiState: IStateService) {
    this.newPin = this.makeNewPin();
  }
```

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/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
  makeNewPin(): Pin {
    return {
      title: 'Steampunk Cat',
      description: 'A cat wearing goggles',
      user_name: 'me',
      avatar_src: '/assets/images/avatars/me.jpg',
      src: '/assets/images/pins/cat.jpg',
      url: 'http://cats.com',
      faved: false,
      id: Math.floor(Math.random() * 10000).toString()
    };
  }
```

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/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```
  onSubmit(): void {
    this.saving = true;
    console.log('submitted', this.newPin);
    setTimeout(() => {
      this.pinsService.addPin(this.newPin).then(() => {
        this.newPin = this.makeNewPin();
        this.saving = false;
        this.uiState.go('home');
      });
    }, 2000);
  }
```

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/upgrade/ng1/js/app.js

```
  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
```

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:

```
this.pinsService.addPin(this.newPin).then(() => {
  // ...
});
```

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/upgrade/hybrid/src/js/app.d.ts

```
  pins(): Promise<Pin[]>;
  addPin(pin: Pin): Promise<any>;
}
```

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`

Now let's use the `AddPinComponent`.

Downgrade ng2 `AddPinComponent`

To use `AddPinComponent` we need to downgrade it:

code/upgrade/hybrid/src/app/app.module.ts

```
angular.module('interestApp')
  .directive('pinControls',
             upgradeAdapter.downgradeNg2Component(PinControlsComponent))
  .directive('addPin',
             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/upgrade/hybrid/src/js/app.js

```
    .state('add', {
      template: "<add-pin></add-pin>",
      url: '/add',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })
```

Exposing an ng2 service to ng1

So far we've downgraded ng2 components to be used 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/upgrade/hybrid/src/app/analytics.service.ts

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

/**
 * Analytics Service records metrics about what the user is doing
 */
@Injectable()
export class AnalyticsService {
  events: string[] = [];

  public recordEvent(event: string): void {
    console.log(`Event: ${event}`);
    this.events.push(event);
  }
}
```

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.

> The `@Injectable` decorator really means that other dependencies can be injected into this service, but it's recommended to add it to all services, even those that don't have dependencies. Read more about `@Injectable` in the chapter on dependency injection

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 make sure `AnayticsService` is in the list of `providers` for our `NgModule`:

code/upgrade/hybrid/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    AddPinComponent,
    PinControlsComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [
    AnalyticsService
  ]
})
export class AppModule { }
```

Then we can use `downgradeNg2Provider`:

code/upgrade/hybrid/src/app/app.module.ts

```
angular.module('interestApp')
  .factory('AnalyticsService',
           upgradeAdapter.downgradeNg2Provider(AnalyticsService));
```

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/upgrade/hybrid/src/js/app.js

```
.controller('HomeController', function(pins, AnalyticsService) {
  AnalyticsService.recordEvent('HomeControllerVisited');
  this.pins = pins;
})
```

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/upgrade/hybrid/src/js/app.js

```
.directive('pin', function(AnalyticsService) {
  return {
    restrict: 'E',
    templateUrl: '/assets/templates/pin.html',
    scope: {
      'pin': "=item"
    },
    link: function(scope, elem, attrs) {
      scope.toggleFav = function() {
        AnalyticsService.recordEvent('PinFaved');
        scope.pin.faved = !scope.pin.faved;
      }
    }
  }
})
```

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[149]
- The Angular2 Upgrade Spec Test[150]
- The Angular2 Source for `DowngradeNg2ComponentAdapter`[151]

---

[149] https://angular.io/docs/ts/latest/guide/upgrade.html
[150] https://github.com/angular/angular/blob/master/modules/angular2/test/upgrade/upgrade_spec.ts
[151] https://github.com/angular/angular/blob/master/modules/angular2/src/upgrade/downgrade_ng2_adapter.ts

NativeScript: Mobile Applications for the Angular Developer

> In this chapter, we're going to walk through how to build your first NativeScript app. NativeScript is a huge topic that could warrant it's own book.
>
> Here we're going to explain NativeScript for the Angular Developer. By the end of this chapter you'll understand the differences between NativeScript and a 'regular' Angular web-app, and have the foundation to be creating your own native apps using NativeScript and Angular.

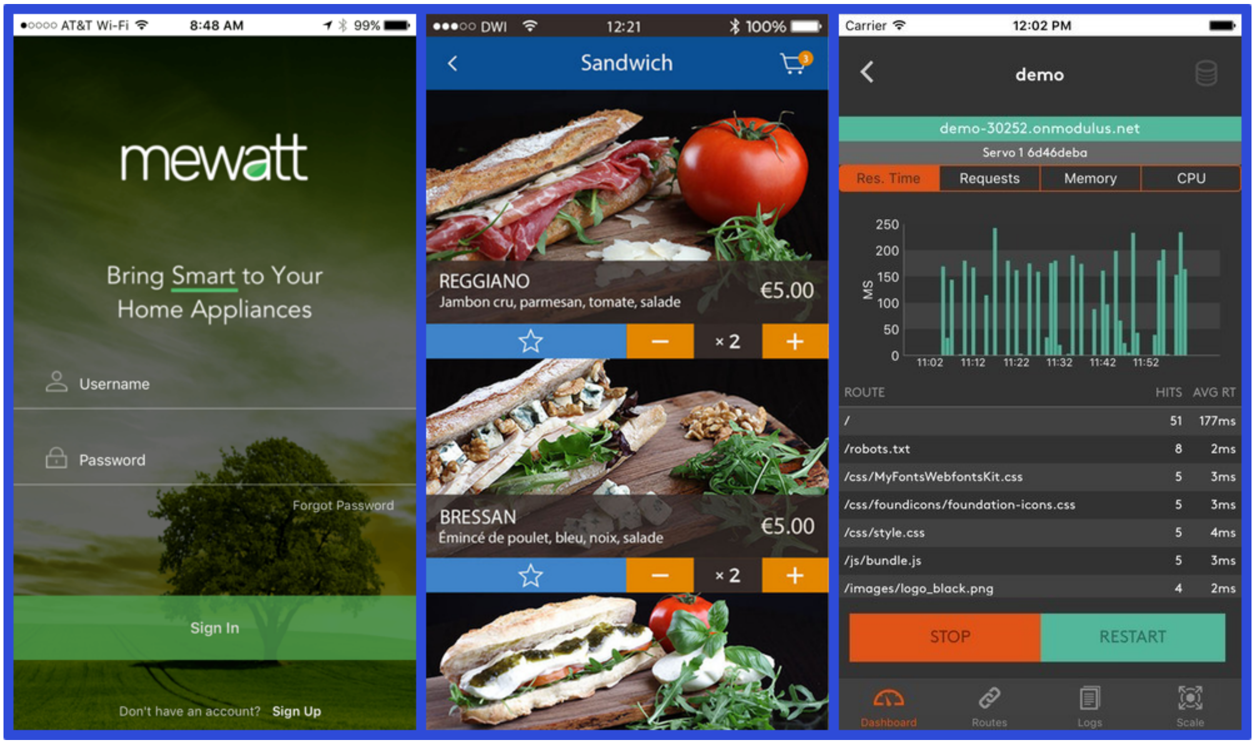
Being that Angular was designed to be unspecific to any particular deployment platform, you can take much of your web application code and reuse it beyond just the web.

It is the norm for businesses to have not only a fully functional web application, but a mobile application to compliment it as well. A few years back, companies would need to spend countless dollars to fund a team of iOS and Android developers to accomplish the same task of creating a mobile application.

With Angular, mobile development becomes not only cheaper, but more maintainable and efficient.

What is NativeScript?

NativeScript is a cross platform mobile development framework that leverages technologies you already know: JavaScript, CSS, and of course, Angular.



NativeScript Showcase

With NativeScript, developers can build native iOS and Android applications using a single shared code base.

Where NativeScript Differs from Other Popular Frameworks

NativeScript isn't the first or only framework to make it easy to develop Android and iOS applications using a single code base. Mobile development frameworks can be separated into two: *hybrid* mobile and *native* mobile.

Hybrid Mobile Applications

Hybrid mobile frameworks are those such as Ionic Framework[152], PhoneGap[153], Apache Cordova[154], and Onsen UI[155]. These are frameworks that allow you to develop mobile applications using web technologies, but render these mobile applications in what's called a *web view*. A web view is essentially a web browser and it allows you to use HTML with full DOM support for all your component rendering.

The conveniences of a web view is not without limitation. The number one flaw in using a web view to render mobile applications comes down to performance. Not all mobile devices are treated

[152] https://ionicframework.com/
[153] http://phonegap.com/
[154] https://cordova.apache.org/
[155] https://onsen.io/

as equal even if they have the same version of Android or iOS. There are thousands of different mobile handsets in existence all with varying hardware and processing power, not to mention all the custom flavors of Android. Because of this diversity, the consistency in web view performance is very poor, leaving some people with an amazing user experience and some with hardly useable applications.

Native Mobile Applications

Native mobile applications built with frameworks such as NativeScript[156], React Native[157], and Xamarin[158] **do not render in a web view**. These are applications that use the **native UI components** that Google and Apple made available to developers and as a result don't suffer from performance instability.

So how does one choose between the available native mobile frameworks? The simple answer is to choose between each of their underlying development technologies. React Native uses ReactJS, a common JavaScript framework for web developers, and Xamarin uses C#, a common development language for .NET developers. NativeScript of course uses Angular.

As an Angular developer, it makes sense to go the NativeScript route because we'll get fantastic native performance while keeping our familiar Angular development experience.

What are the System and Development Requirements for NativeScript?

NativeScript doesn't have any system requirements beyond what you'd need when developing Objective-C based iOS applications or Java based Android applications.

For example, let's say you wanted to build and deploy an Android application developed with NativeScript. You would need at least the following:

- Windows, Linux, or Mac
- Java Development Kit (JDK) 8+
- 4GB of hard drive space
- 4GB of RAM

The above system and software requirements are what's necessary for installing and using the Android SDK.

If you wanted to build and deploy and iOS application with NativeScript, the requirements are a bit different:

- Mac

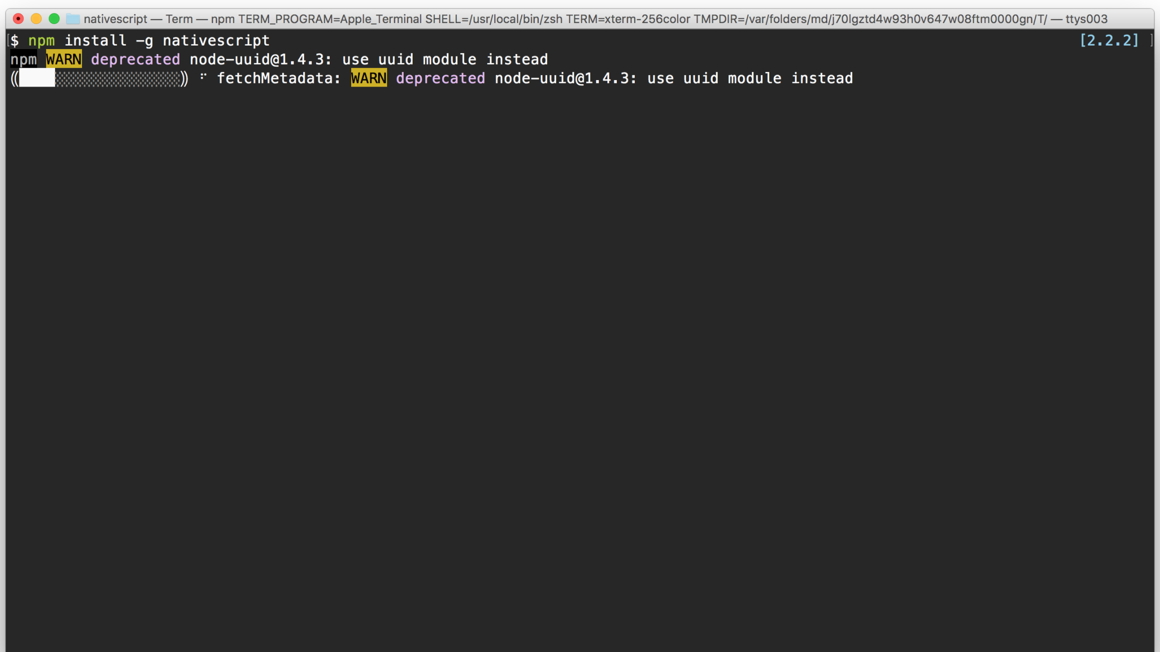
[156] https://www.nativescript.org/
[157] https://facebook.github.io/react-native/
[158] https://www.xamarin.com/

- Xcode 7+
- 5GB of hard drive space
- 4GB of RAM

Notice the main difference here is that a Mac is required. While you can develop Android and iOS applications with NativeScript, you cannot actually build and deploy iOS applications unless you're using a Mac. This is a limitation that exists because of Apple.

From a development perspective, NativeScript uses the Node Package Manager (NPM) a tool that is part of Node.js and something you probably already have installed as an Angular developer. With NPM, the NativeScript CLI can be installed using the following command:

```
npm install -g nativescript
```



Installing NativeScript

A list of available commands can be found by running `tns --help` or `tns help` if you wish to view them in a web browser rather than the Command Prompt or Terminal.

For more information on installing NativeScript for Mac, Windows, and Linux, visit the NativeScript installation documentation[159].

[159]https://docs.nativescript.org/start/quick-setup.html

There are a significant number of tools to be installed to do native app development. Once everything is installed properly, NativeScript development is relatively painless, but make sure you visit the URL above if you run into any trouble getting the NativeScript build tools installed.

With the NativeScript CLI, native mobile applications can be developed with Angular.

Creating your First Mobile Application with NativeScript and Angular

To be successful in developing NativeScript applications with Angular, you should already have the NativeScript CLI tool installed and either Xcode or the Android SDK installed, or both.

The goal here is to become familiar with the mobile application creation process and some of the UX and UI differences between an Angular web application and an Angular NativeScript application.

Using the Command Prompt (Windows) or Terminal (Mac and Linux), execute the following:

```
tns create NgProject --ng
```

The above command will create a project directory, `NgProject`, wherever your command line's active directory is located. The `--ng` flag indicates that we want to create an Angular with TypeScript project. It is necessary to use the `--ng` flag because NativeScript doesn't require Angular to build mobile applications. It is an option, one that we're going to take full advantage of.

Adding Build Platforms for Cross Platform Deployment

While a project has been created and can be actively developed, there are no build platforms such as Android or iOS enabled for building and deployment.

To build for a specific platform, it must first be added. Using the NativeScript CLI, execute the following:

```
tns platform add [platform]
```

Just swap out `[platform]` with either `android` or `ios` depending on which you wish to add, remembering that iOS requires a Mac with Xcode installed.

Building and Testing for Android and iOS

When the application is ready for testing or deployment to the app stores, we can make use of a few NativeScript CLI commands. Before deployment, you'll probably want to test the application on your device or emulator. Using the command line, execute the following to emulate the application:

```
tns emulate [platform]
```

Swapping `[platform]` with `android` or `ios` will launch the application in the specified emulator. To test the application on a device, swap out `emulate` with the word `run` while your device is connected to your development machine.

```
tns run [platform]
```

The emulation process can often take a bit of time because a lot of recompilation happens in the process. To make development more efficient, the NativeScript CLI offers live-reload functionality called live-sync. We can utilize this feature by executing the following command in our terminal:

```
tns livesync [platform] --emulator --watch
```

After swapping `[platform]` with either `android` or `ios`, changes made to TypeScript, CSS, or HTML files will be automatically deployed to the Android or iOS simulator, much faster than if you were to strictly emulate the application.

When it comes to deploying our app to the app store, we can use the following command:

```
tns build [platform]
```

After replacing `[platform]` with the appropriate platform, the binaries and build packages will be created.

Installing JavaScript, Android, and iOS Plugins and Packages

Like with any Angular web application, there are external components available to make the development process easier. This applies to NativeScript applications as well.

Most JavaScript packages will work in a NativeScript application as long as there isn't a dependency on the DOM. As previously mentioned, NativeScript being a native framework, doesn't use a web view and has no concept of a DOM. JavaScript libraries can be included via NPM, for example:

```
npm install jssha --save
```

The above would install the JavaScript hashing library, jsSHA, to your Angular NativeScript project.

There are native plugins available strictly for NativeScript as well. These are typically plugins that make use of native device features or interface with Android or iOS directly in some fashion.

Take, for example, the NativeScript SQLite plugin:

```
tns plugin add nativescript-sqlite
```

The above command will install SQLite functionality for both Android and iOS.

Understanding the Web to NativeScript UI and UX Differences

As a web developer you're probably very familiar with HTML and common design practices for building attractive, responsive, and overall great web applications. With NativeScript we're using Angular and CSS, but we're not using HTML. Instead we are using XML which won't have the same markup tags that you'd find in HTML.

So how do you take your UI and UX skills to mobile?

There are a few things that need to be taken into consideration when designing your mobile application. You need to worry about the screen layout and the screen components.

Planning the NativeScript Page Layout

When designing a web application, common layout components include `<div>` tags and `<table>` tags. Generally if you want a grid of rows and columns you'd use a table and if you wanted a stack of components you'd use a div because it acted as a container.

In NativeScript, you don't have the `<div>` and `<table>` tags, but you have something similar. Instead you have the `<StackLayout>` and `<GridLayout>` tags.

So let's compare web and NativeScript.

Let's say we wanted to contain a bunch of HTML components on a website. You might do something like the following:

```
<div>
    <span>Nic Raboy was here</span>
    <span>https://www.thepolyglotdeveloper.com</span>
</div>
```

To accomplish the same in a NativeScript application, you'd do the following:

```
<StackLayout>
    <Label text="Nic Raboy was here"></Label>
    <Label text="https://www.thepolyglotdeveloper.com"></Label>
</StackLayout>
```

In both the web and NativeScript scenarios you can nest the `<div>` and `<StackLayout>` tags as appropriate to create more component groupings.

The use of grids in NativeScript and on the web are a bit different in structure, but the same in concept. Take the following HTML:

```
<table>
    <tr>
        <td>Nic</td>
        <td>Raboy</td>
    </tr>
    <tr>
        <td>Burke</td>
        <td>Holland</td>
    </tr>
</table>
```

In NativeScript, instead of defining rows and columns with `<tr>` and `<td>` tags something a little different happens:

```
<GridLayout rows="auto, auto" columns="*, *">
    <Label text="Nic" row="0" col="0"></Label>
    <Label text="Raboy" row="0" col="1"></Label>
    <Label text="Burke" row="1" col="0"></Label>
    <Label text="Holland" row="1" col="1"></Label>
</GridLayout>
```

In the above `<GridLayout>` we define that we want two rows that take the height of their child components and two columns that stretch evenly to fill the screen.

But what about a flexbox, commonly found on the web?

When building websites, there is the opportunity to set `<div>` tags, or any other container, to have a CSS property of `display: flex`. This allows websites to behave appropriately for different screen sizes. Nearly the same can be used in NativeScript using the `<FlexboxLayout>` as a container, which is nearly the same as the web's implementation.

Adding UI Components to the Page

When it comes to NativeScript there are many UI components available, each accomplishing something different. For example we already saw how to display static text on the screen through the use of the `<Label>` component, but what other options are available?

There are too many components to name, but some of the common components include buttons, images, lists, and inputs. These are all components that are common to what you'd find in a web application as well.

To add a button to our application, we'd add the following to one of our layouts:

```
<Button text="Submit Me" (tap)="myFunc()"></Button>
```

Notice the use of the `(tap)` attribute. This is not specific to the UI component, but more a mixture of Angular and NativeScript. In a web application these events are better known as `(click)` events, however, they both accomplish the same.

To include an image, local or remote, within an application, we can use the `<Image>` tag like so (similar to the `<img />` tag on the web):

```
<Image src="https://placehold.it/350x150"></Image>
```

Many mobile applications, like web applications, collect data from users. This data is collected through forms composed of text input fields. To accept text input in a NativeScript application, make use of the `<TextField>` tag like the following:

```
<TextField
    text="First Name"
    [(ngModel)]="firstname"></TextField>
```

The `[(ngModel)]` attribute seen above is identical to that which is found in an Angular web application. It allows the binding of data between the UI and the TypeScript paired to it.

It is often necessary to list large amounts of data within a mobile application. This data is presented in what is called a `<ListView>`. These lists are populated from arrays of strings or objects that are defined within the application TypeScript.

```
<ListView [items]="people">
    <Template let-person="item">
        <Label [text]="person.firstname"></Label>
    </Template>
</ListView>
```

The above snippet will create a list from an array of objects called `people`. Each object in the array will be called `person` and the `firstname` of each `person` will be displayed in a list row.

Again, there are many other components available, some not heard of in the land of web development. However, they are all similar by design.

Just like with web components, NativeScript UI components don't look attractive in their vanilla state. They need to be themed and styled with some artistic flair.

Styling Components with CSS

There are a few options available when it comes to giving a NativeScript application a boost in the attractiveness department, just as there is in web design.

NativeScript allows UI components to be styled with a CSS subset. To be clear, most web CSS will work in NativeScript, but not everything. To change the font color of a `<Label>` component, the following is an option:

```
.title {
    color: #cc0000;
}
```

The class name can then be applied to the UI component in the same fashion as with HTML.

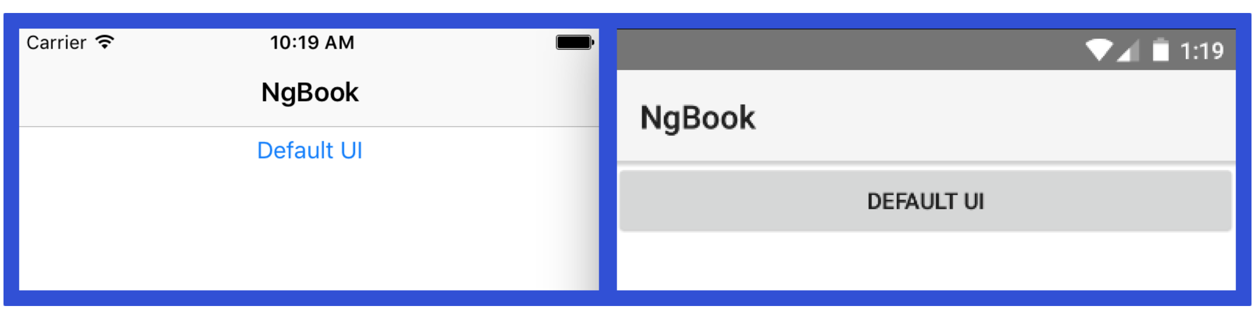
Creating a custom stylesheet isn't the only solution when it comes to making a NativeScript application more attractive. When building a website, there are frameworks such as Bootstrap that were designed to make life easier. We can translate this same concept with NativeScript.

There is what is called NativeScript Theme, which is a package of CSS styles designed to be easily added to any application.

Take the following action bar with button example:

```
<ActionBar title="NgBook"></ActionBar>
<StackLayout>
    <Button text="Default UI"></Button>
</StackLayout>
```

The above code would generate a native, but very plain looking action bar with a very plain looking button. On Android and iOS, it would look like the following:

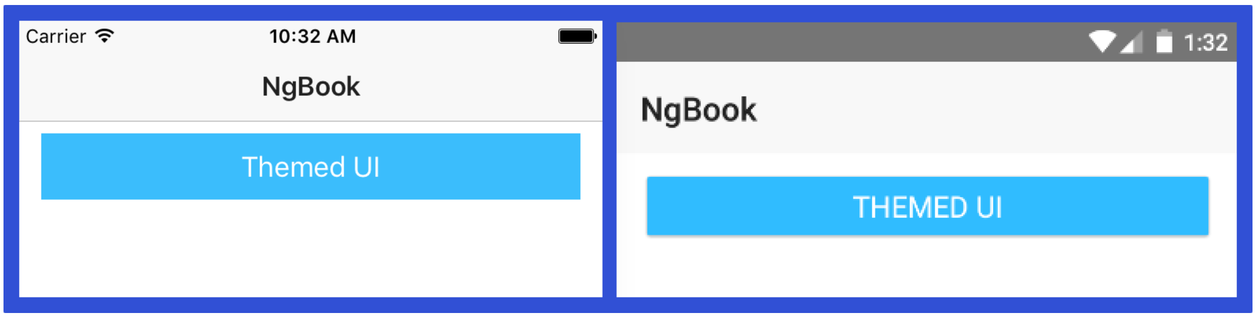


NativeScript Basic CSS

This simple UI can be significantly improved by using NativeScript Theme. For example, take the minor revisions to the code snippet found below:

```
<ActionBar title="NgBook" class="action-bar"></ActionBar>
<StackLayout>
    <Button text="Themed UI" class="btn btn-primary"></Button>
</StackLayout>
```

A few class names were applied to the components giving them a much more pleasant look and feel as demonstrated in the image below:



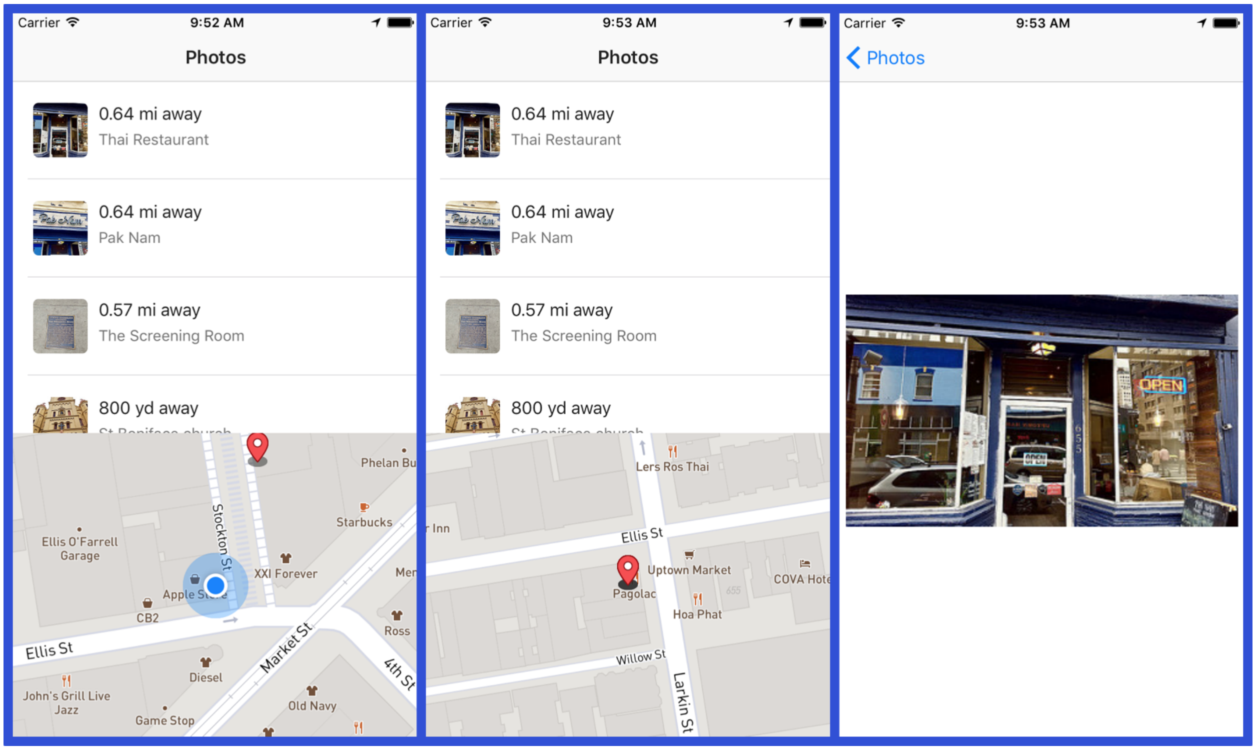
NativeScript Theme CSS

The naming conventions for the theme classes have a similar naming convention to those found in the popular web frameworks.

Developing a Geolocation Based Photo Application

Taking what we know about Angular, web development, and the NativeScript mobile framework, we can apply it towards creating a native and functional mobile application for both iOS and Android.

Much of what comes next will be a review of the Angular skills you already have, but in a mobile example. The example application will use geolocation and the Flickr API to show images that were captured near you.



NativeScript Photos Near Me

The application will have two pages that act as a master-detail interface meaning the first page will list data and the second page will show more information about the data selected from the first page.

> The completed project can be found in the sample code under `code/nativescript/photos-near-me`.

Creating a Fresh NativeScript Project

To get the most out of this demo, it would be good to start with a new project. As a review to what was mentioned previously, a project can be created by executing the following:

```
tns create GeoPhotoProject --ng
cd GeoPhotoProject
tns platform add android
tns platform add ios
```

The above commands will create an Angular NativeScript project called **GeoPhotoProject** with the Android and iOS build platforms. To be able to build iOS applications we must be using a Mac with Xcode installed.

The default project template will be a single page application, so we'll have to add more pages and configure the Angular Router.

Creating a Multiple Page Master-Detail Interface

The default project template uses the project's **app/app.component.html** file as the default page. This file will still be valuable in this project, but we're going to create two new pages.

Let's create a few of the new components we'll use by executing the following commands to create necessary files and directories:

```
mkdir -p app/components/image-component
mkdir -p app/components/imagesList-component
touch app/components/image-component/image.component.ts
touch app/components/image-component/image.component.html
touch app/components/imagesList-component/imagesList.component.ts
touch app/components/imagesList-component/imagesList.component.html
```

We can also create these directories manually in our Explorer window, if the `mkdir` or `touch` commands are not available in our command-line (or if we just feel more comfortable in the UI).

The first page in our application flow will be the **imagesList.component** page to display all the list of photos.

Let's open the project's **app/components/imagesList-component/imagesList-component.ts** file and include the following basic class code:

```
import { Component, NgZone } from "@angular/core";
import { Router } from "@angular/router";

@Component({
    selector: "ImagesListComponent",
    templateUrl: "components/imagesList-component/imagesList.component.html"
})
export class ImagesListComponent {

    public constructor(private zone: NgZone, private router: Router) { }

}
```

In the above code the `ImagesListComponent` class is being defined and various Angular components are being imported and injected into the `constructor` method in the usual method.

The UI that goes with the `ImagesListComponent` class is found in the **app/components/imageList-component/imagesList-component.html** file. For now, let's update the file to contain following HTML markup:

```
<ActionBar title="Photos" class="action-bar"></ActionBar>
<StackLayout>
</StackLayout>
```

Before we add useful functionality to the first page of our application, let's lay the foundation to the second page and link them together.

Open the project's **app/components/image-component/image-component.ts** file and include the following TypeScript code:

```
import { Component, OnInit } from "@angular/core";
import { ActivatedRoute } from "@angular/router";

@Component({
    templateUrl: "components/image-component/image.component.html"
})
export class ImageComponent implements OnInit {

    public constructor(private activatedRoute: ActivatedRoute) { }

    public ngOnInit() { }

}
```

In the above code the `ImageComponent` class is created and various Angular components are imported and injected in the `constructor` method. The core difference here, as of now, is the `ngOnInit` method which is going to be used to load data after the page loads.

The UI that goes with the TypeScript code is found in the **app/components/image-component/image-component.html** file and it will contain, for now, the following HTML markup:

```
<ActionBar></ActionBar>
<StackLayout>
</StackLayout>
```

With the pages available, they need to be brought together for Angular routing. This requires two things to happen. First, the routes need to be defined and second they need to be included in the project's `@NgModule` block.

Let's create an **app/app.routing.ts** file in our project and include the following routing configuration code:

code/nativescript/photos-near-me/app/app.routing.ts

```
import { ImagesListComponent } from "./components/imagesList-component/imagesList.co\
mponent";
import { ImageComponent } from "./components/image-component/image.component";

export const routes = [
    { path: "", component: ImagesListComponent },
    { path: "image-component/:photo_id", component: ImageComponent },
];

export const navigatableComponents = [
    ImagesListComponent,
    ImageComponent
];
```

In the above code, both the `ImagesListComponent` and `ImageComponent` classes were imported. The routes define how to navigate to each of the classes and what data can be passed. The `ImagesListComponent` has an empty path which represents the default, or first page that loads when the application starts. The `ImageComponent` has a path with one URL parameter which represent a piece of data that can be passed from the `ImagesListComponent` page to the `ImageComponent` page.

Without getting too far ahead of ourselves, the `photo_id` represents the photo we wish to load in the second page. This is a piece to the Flickr API.

The **app/app.routing.ts** file needs to be imported and added to the project's `@NgModule` block. In our project's **app/app.module.ts** file and include the following TypeScript code:

```
import { NativeScriptModule } from "nativescript-angular/platform";
import { NgModule } from "@angular/core";
import { NativeScriptFormsModule } from "nativescript-angular/forms";
import { NativeScriptHttpModule } from "nativescript-angular/http";
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { registerElement } from "nativescript-angular/element-registry";

import { AppComponent } from "./app.component";
import { routes, navigatableComponents } from "./app.routing";

@NgModule({
    imports: [
        NativeScriptModule,
        NativeScriptFormsModule,
        NativeScriptHttpModule,
        NativeScriptRouterModule,
```

```
        NativeScriptRouterModule.forRoot(routes)
    ],
    declarations: [
        AppComponent,
        ...navigatableComponents,
    ],
    bootstrap: [AppComponent],
    providers: []
})
export class AppModule {}
```

There is more setup in this file than what you'll find in the default. To save us some time we're importing the `NativeScriptFormsModule`, `NativeScriptHttpModule`, and `NativeScriptRouterModule` along with the `routes` and `navigatableComponents` variables that were defined in the previous file.

Each module is added to the `imports` array of the `@NgModule` block and the two page classes found in the `navigatableComponents` variable are added to the `declarations` array.

Even though the application doesn't do much at the moment, it is linked together and ready to go. Adding UI components and functionality will be explored later on.

Finally, we'll need to add a place for our pages to render via our routes. In our main app component in `app/app.component.html`, let's add the `<page-router-outlet/>` markup to tell Angular where to render our subroutes. Since we don't have any common views between views, can replace all of the content with this markup:

```
<page-router-outlet></page-router-outlet>
```

Creating a Flickr Service for Obtaining Photos and Data

Flickr will be a critical part of this application. Instead of calling the Flickr API directly in each of the pages we wish to use it, the better approach would be to create an Angular service, also known as a provider.

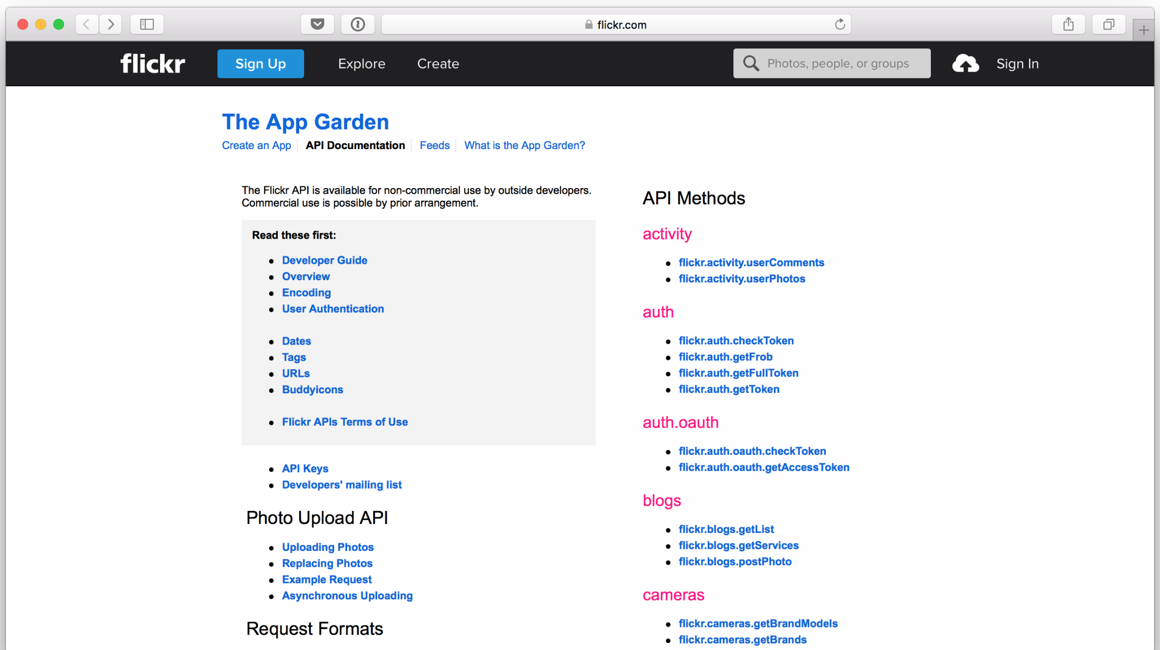
In a Flickr provider we can add logic to query for photos based on latitude and longitude information as well as get information about particular photos.

Before designing this provider it is a good idea to create a global configuration file for the application. This will prevent hard coded URL values, amongst other things, in the application.

Let's create a **app/config.ts** file and include the following:

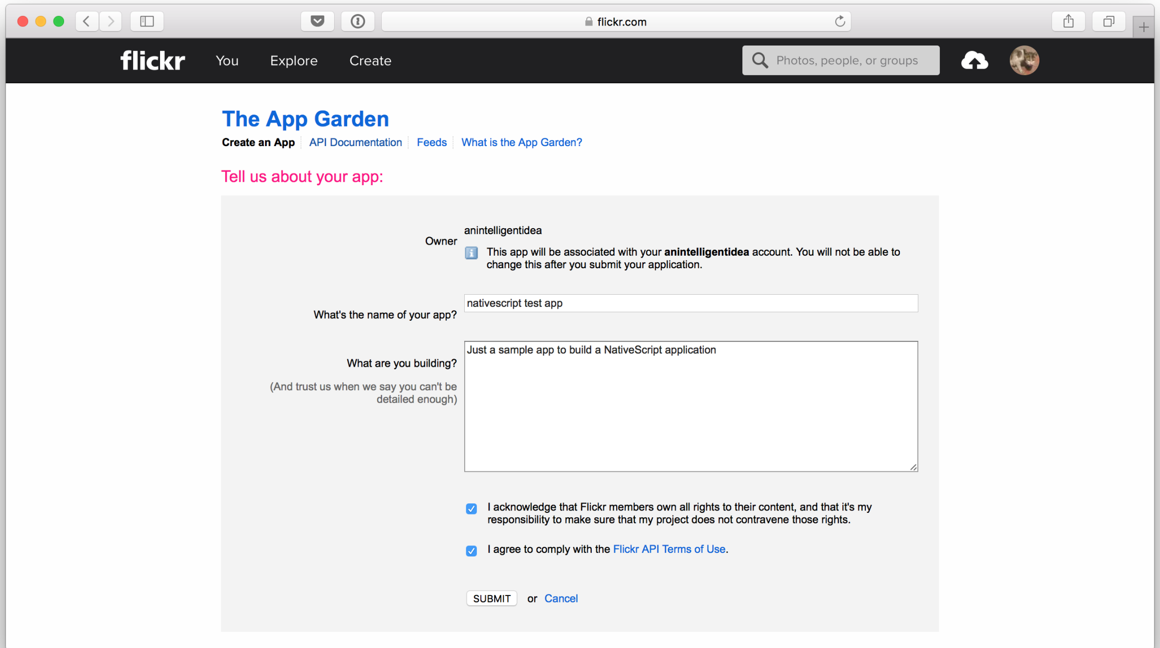
```
export const Config = {
    Flickr: {
        CLIENT_ID: "FLICKR_CLIENT_ID_HERE",
        API_URL: "https://api.flickr.com/services/rest/?"
    }
};
```

Before using the Flickr API, an account needs to be created to obtain a client id. We'll head to https://www.flickr.com/services/api/[160] and create an account.



Flickr will create a `client_id` that will be unique to our application. The value of the `FLICKR_-CLIENT_ID_HERE` in our `app/config.ts` file.

[160] https://www.flickr.com/services/api/



With the configuration file created, we need to define a data model for the Flickr responses. While not absolutely necessary, it does create a more maintainable TypeScript application.

Let's create the `app/models` directory, if it does not already exist:

```
mkdir app/models
```

Create an **app/models/getInfoResponse.ts** file and include the following TypeScript code:

code/nativescript/photos-near-me/app/models/getInfoResponse.ts

```
interface Owner {
    username: string;
    realname: string;
}

export class GetInfoResponse {
    owner: Owner;
    farm: number;
    server: number;
    secret: string;
    id: number;
    url: string;
}
```

The above represents the data that is returned from the Flickr `flickr.photos.getInfo` RESTful endpoint. The data makes it possible to obtain an image file along with holding information about that image file.

The second model we need is for Flickr search data. Create an **app/models/photosSearchResponse.ts** file with the following TypeScript code:

code/nativescript/photos-near-me/app/models/photosSearchResponse.ts

```
export class PhotosSearchResponse {
    id: string;
    owner: string;
    secret: string;
    server: number;
    title: string;
    latitude: string;
    longitude: string;
    datetaken: string;
    url_t: string;
    url_m: string;
    url_q: string;
    url_n: string;
    distance: string;

    constructor() {
        this.url_n = " ";
    }
}
```

The above model holds useful information such as the photo id, the owner, and geolocation information all useful when discovering images and displaying them on the second page of the application.

With the data models created, we can now create the Flickr service. Let's create a file at **app/services/flickr.service.ts** in the project.

```
mkdir app/services
touch app/services/flickr.service.ts
```

We'll start with this foundation, in the `flickr.service.ts` file:

```
import { Component, Injectable } from "@angular/core";
import { Http, Response } from "@angular/http";
import { Observable } from "rxjs/Rx";
import { Config } from "../app.config";
import { PhotosSearchResponse } from "../models/photosSearchResponse";
import { GetInfoResponse } from "../models/getInfoResponse";
import "rxjs/add/operator/map";

@Injectable()
export class FlickrService {

    public constructor(private http: Http) { }

    public photosSearch(lat: number, lon: number): Observable<PhotosSearchResponse[]\
> { }

    public getPhotoInfo(photoId: number): Observable<GetInfoResponse> { }

}
```

Both the `photosSearch` and `getPhotoInfo` functions return observables which are streams of data obtained by HTTP requests to the Flicker API.

The `photosSearch` function will take a latitude and longitude and apply it towards Flickr's API like follows:

code/nativescript/photos-near-me/app/services/flickr.service.ts

```
    public photosSearch(lat: number, lon: number): Observable<PhotosSearchResponse[]\
> {
        let url = `${Config.Flickr.API_URL}method=flickr.photos.search&api_key=${Con\
fig.Flickr.CLIENT_ID}&content_type=1&lat=${lat}&lon=${lon}&extras=url_q,geo&format=j\
son&nojsoncallback=1`;

        return this.http.get(url)
            .map(response => response.json().photos.photo)
            .catch(error => Observable.throw(error));
    }
```

An HTTP request is made per the Flickr API documentation. Using RxJS, the response of the request is transformed using the `map` operator to be of type `PhotosSearchResponse`. If there is an error in the response, it will be caught through the normal http promise error chain. Just like normal Angular, our HTTP request won't execute until the observable is subscribed.

The `getPhotoInfo` method will take a photo id, probably from the result returned in the previous `photosSearch` function:

code/nativescript/photos-near-me/app/services/flickr.service.ts

```
    public getPhotoInfo(photoId: number): Observable<GetInfoResponse> {
        let url = `${Config.Flickr.API_URL}method=flickr.photos.getInfo&api_key=${Co\
nfig.Flickr.CLIENT_ID}&photo_id=${photoId}&format=json&nojsoncallback=1`;

        return this.http.get(url)
            .map(response => response.json().photo)
            .catch(error => Observable.throw(error));
    }
```

Like with the `photosSearch` function, the `getPhotoInfo` function makes a HTTP request against the Flickr API and parse the response using RxJS.

Before the Flickr provider can be used throughout the application, it must be added to the `@NgModule` block similarly to how the application pages were added.

Inside the project's **app/app.module.ts** file, we need to import the Flickr service must be imported and then add it to the `providers` array in the `@NgModule` block:

```
import { FlickrService } from "./services/flickr.service";

@NgModule({
  // ...
  providers: [FlickrService]
})
...
```

The Flickr provider can now be used in the various pages of the application.

Creating a Service for Calculating Device Location and Distance

Up until now, all the TypeScript has been general to Angular and with nothing to do with NativeScript. This geolocation application will have dependence on the location of the Android or iOS device so NativeScript must be used to natively interface with the GPS components.

Because GPS will be used throughout the application, it is a good idea to create an Angular provider for it. This will keep the code clean and maintainable.

Before creating the provider, a JavaScript library must be installed into the project.

```
npm install humanize-distance --save
```

The `humanize-distance` library allows us to calculate the distance between two latitude and longitude locations. This will be particularly useful when checking our user's device location versus that of a photo returned from Flickr.

We'll also need to include a nativescript library called `nativescript-geolocation` using the `tns` plugin command:

```
tns plugin add nativescript-geolocation
```

Let's create another service called the `geolocation.service`:

```
touch app/services/geolocation.service.ts
```

In this new file, let's include the following foundation code:

```
import { Injectable } from "@angular/core";
import * as geolocation from "nativescript-geolocation";
var humanizeDistance = require("humanize-distance");

@Injectable()
export class GeolocationService {

    public latitude: number;
    public longitude: number;

    public getLocation(): Promise<any> { }

    public getDistanceFrom(latitude: number, longitude: number): string { }

    private _getCurrentLocation(): Promise<any> { }

}
```

This provider will be injectable into the application pages. It will use the `nativescript-geolocation` plugin which interfaces with native Android and iOS GPS code. The `humanize-distance` library is imported differently because it is JavaScript rather than TypeScript.

code/nativescript/photos-near-me/app/services/geolocation.service.ts

```
    private _getCurrentLocation(): Promise<any> {
        return new Promise(
            (resolve, reject) => {
                geolocation.getCurrentLocation({
                    desiredAccuracy: Accuracy.high,
                    timeout: 20000
                })
                .then(location => {

                    this.latitude = location.latitude;
                    this.longitude = location.longitude;

                    resolve();
                })
                .catch(error => {
                    reject(error);
                })
            }
        );
    }
```

Using the geolocation plugin we can get the current longitude and latitude of the device GPS. This is an asynchronous request and must be added to a JavaScript promise or observable. The result of `_getCurrentLocation` will be a promise of any data.

Not all devices have GPS hardware and both Android and iOS require permissions to use location services. Because of this a few checks must be put into place.

code/nativescript/photos-near-me/app/services/geolocation.service.ts

```
    public getLocation(): Promise<any> {
        return new Promise(
            (resolve, reject) => {
                if (!geolocation.isEnabled()) {
                    geolocation.enableLocationRequest(true).then(() => {
                        this._getCurrentLocation()
                            .then(resolve)
                            .catch(reject);
                    });
                }
                else {
                    this._getCurrentLocation()
                        .then(resolve)
```

```
                            .catch(reject);
                    }
                }
            );
    }
```

Using the `getLocation` method a check to see if the geolocation service is enabled is made. If it is not enabled, a request to enable it will be made. Provided that everything checks out, a call to the other `_getCurrentLocation` function will be made. This also applies if the geolocation service is enabled already.

With the device location in hand, a distance can be calculated from a different location, more than likely the picture distance.

code/nativescript/photos-near-me/app/services/geolocation.service.ts

```
    public getDistanceFrom(latitude: number, longitude: number): string {
        return humanizeDistance({ latitude: latitude, longitude: longitude }, { lati\
tude: this.latitude, longitude: this.longitude }, 'en-US', 'us');
    }
```

The `getDistanceFrom` method will use the `humanize-distance` library to get us a better distance format like kilometers, miles, etc.

Like with the Flickr provider, the geolocation provider needs to be added the project's `@NgModule` block. Let's open our project's **app/app.module.ts** file and include the following lines:

```
import { GeolocationService } from "./services/geolocation.service";

@NgModule({
  // ...
  providers: [FlickrService, GeolocationService]
})
...
```

Essentially, we're importing the provider and adding it to the `providers` array of the `@NgModule` block. At this point the geolocation provider can be used throughout the application.

Including Mapbox Functionality in the NativeScript Application

As of right now neither of the two application routes have any functionality that is particularly useful. The application has two very useful providers, but they aren't being used yet.

Since geolocation will be used, it makes sense to present a map. There are many options when it comes to maps. Two popular map solutions are Mapbox and Google Maps. For this example Mapbox renders itself the most convenient.

To install Mapbox in a NativeScript application, execute the following:

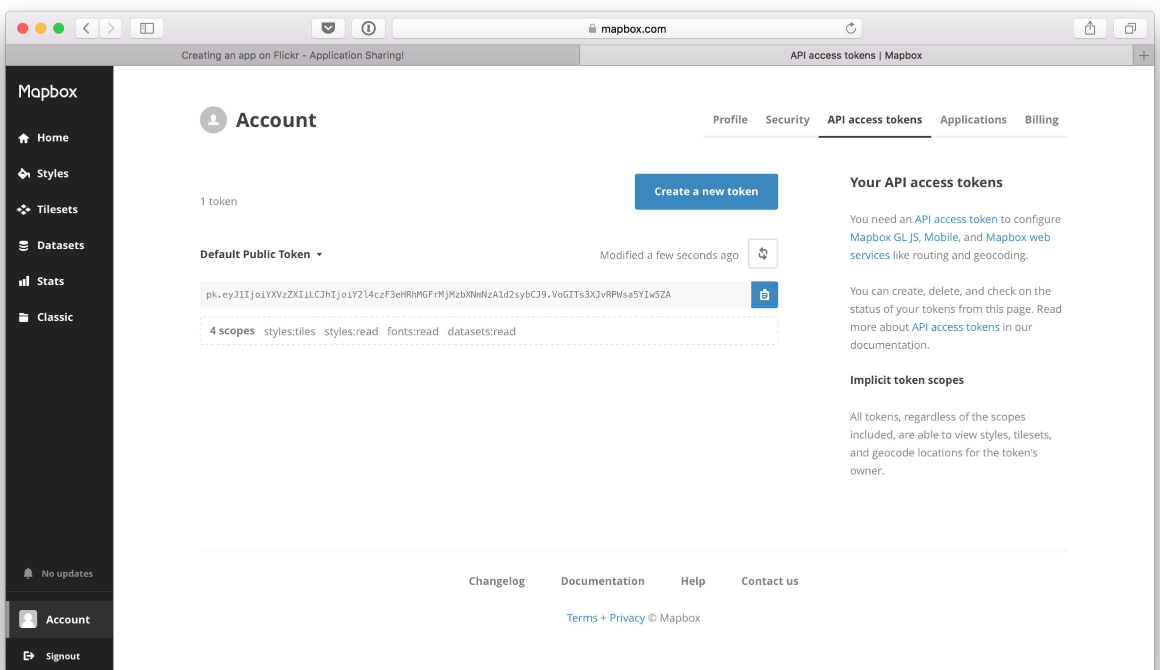
```
tns plugin add nativescript-mapbox
```

The Mapbox plugin for NativeScript has its own set of available HTML markup tags. To expose these tags in an Angular application, they must be registered in the project's **app/app.module.ts** file like so:

```
import { registerElement } from "nativescript-angular/element-registry";

var map = require("nativescript-mapbox");
registerElement("Mapbox", () => map.Mapbox);
```

Once registered, the `<Mapbox>` tag can be used within HTML files. However, Mapbox requires a valid API token in order to be used.

Register for an API token via the Mapbox Developers Page[161].



[161] https://www.mapbox.com/developers/

Let's store the value of the mapbox access_token open the project's **app/config.js** file. This is the same file where we added the Flickr API information. Modify this file to look like the following:

```
export const Config = {
    Flickr: {
        CLIENT_ID: "FLICKR_CLIENT_ID_HERE",
        API_URL: "https://api.flickr.com/services/rest/?"
    },
    MapBox: {
        ACCESS_TOKEN: "MAPBOX_ACCESS_TOKEN_HERE"
    }
};
```

The Mapbox API token will be obtained similarly to how the Flickr API token was obtained within the application. While Mapbox hasn't been added to the UI or the page logic, it is not able to be added.

Implementing the First Page of the Geolocation Application

There was a lot of preparation that went into this project so far, but each of the pages are now ready to be crafted.

Open the **app/components/imageList-component/imageList.component.ts** file that was created earlier. We added foundation, but now it is time to finish it with functional logic.

```
import { Component, NgZone } from "@angular/core";
import { FlickrService } from "../../services/flickr.service";
import { PhotosSearchResponse } from "../../models/photosSearchResponse";
import { Router } from "@angular/router";
import { GeolocationService } from "../../services/geolocation.service";
import { Config } from "../../app.config";

@Component({
    selector: "ImagesListComponent",
    templateUrl: "components/imagesList-component/imagesList.component.html"
})
export class ImagesListComponent {

    private mapbox: any;
    public mapboxKey: string;
    public photos: PhotosSearchResponse[];

    public constructor(private flickrService: FlickrService, private geolocationServ\
```

```
ice: GeolocationService, private zone: NgZone, private router: Router) { }

    public onMapReady(args) { }

    public dropMarkers() { }

    public centerMap(args: any) { }

    public showPhoto(args: any) { }

    public loadPhotos() { }

}
```

In the above TypeScript file, each of the services and models that were previously created are now being imported into the page. The `ImagesListComponent` has a private variable which will hold the Mapbox and several public variables that will be bound to the UI.

In the `constructor` method each of the two providers are injected so they can be used throughout the current page of the application.

In a typical Angular application an `OnInit` would be used after the `constructor` method has executed. To prevent a race condition, this page will not make use of it. Instead, an `onMapReady` method will be created and used via the HTML markup. In other words, when the Mapbox thinks it's ready, this `onMapReady` method will trigger.

code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts

```
    public onMapReady(args) {
        this.mapbox = args.map;
        this.geolocationService.getLocation().then(() => {
            this.loadPhotos().subscribe(
                photos => {
                    this.photos = photos.map((photo) => {
                        photo.distance = this.geolocationService.getDistanceFrom(
                            parseFloat(photo.latitude),
                            parseFloat(photo.longitude));
                        return photo;
                    });
                    this.dropMarkers();
                    this.mapbox.setCenter({
                        lat: this.geolocationService.latitude,
                        lng: this.geolocationService.longitude,
                        animated: true
                    });
```

```
                },
                error => console.log(error));
            });
    }
```

Once triggered, the `mapbox` variable will be set with the current Mapbox. Using the geolocation service, the device GPS location is obtained and Flickr photos near the location are queried. A humanized distance is calculated for each of the photos retrieved from the API call.

The photos obtained from the Flickr API are stored in the `photos` array at which point they are placed as markers on the map using the `dropMarkers` method. At the end of the initialization period the map is centered on the devices location.

The `dropMarkers` method called from the `onMapReady` looks like the following:

code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts

```
    public dropMarkers() {
        let markers = this.photos.map((photo: PhotosSearchResponse, index: number) =\
> {
            return {
                lat: photo.latitude,
                lng: photo.longitude,
                onTap: () => {
                    this.zone.run(() => {
                        this.showPhoto({ index: index });
                    });
                }
            }
        });
        this.mapbox.addMarkers(markers);
    }
```

In the above method, the `photos` array is recreated through a JavaScript `map` and stored as `markers`. The new objects found in the array include the longitude and latitude of the photo and a tap event, `showPhoto`, which will navigate to the next page. To keep everything in sync, the `showPhoto` method must be added within the Angular zone.

The `markers` array is added to the Mapbox for display on the soon to be created map component.

code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts

```
    public showPhoto(args: any) {
        let photo = this.photos[args.index];
        this.router.navigate(["/image-component", photo.id]);
    }
```

The route to the second page of the application requires a photo id. This information is obtained from a specific photo that was selected. Remember, the photo information was added within the `dropMarkers` method.

code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts

```
    public loadPhotos() {
      return this.flickrService.photosSearch(
        this.geolocationService.latitude,
        this.geolocationService.longitude);
    }
```

The `loadPhotos` method was used in the `onMapReady` method for subscribing to the Flickr observable. It was created to make the lines of the file shorter and easier to read.

The final method of the first application page, `centerMap`, will center the map on a particular photo:

code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts

```
    public centerMap(args: any) {
        let photo = this.photos[args.index];
        this.mapbox.setCenter({
            lat: parseFloat(photo.latitude),
            lng: parseFloat(photo.longitude),
            animated: false
        });
    }
```

So what does the UI markup look like for the TypeScript logic that was just implemented? Open the project's **app/components/imageList-component/imageList.component.html** file. The UI is composed of two vertical sections, a list of pictures which resides on the upper level and a map which resides on the lower level.

```
<ActionBar title="Photos" class="action-bar"></ActionBar>
<StackLayout>
    <GridLayout columns="*" rows="*, 280">
        <ListView [items]="photos" row="0" col="0" class="list-group" (itemTap)="cen\
terMap($event)">
        </ListView>
        <ContentView row="1" col="0">
        </ContentView>
    </GridLayout>
</StackLayout>
```

In the above markup a `GridLayout` will allow for vertical sections, hence the two row values. The asterisk in the `columns` means that each row will take the full width of the screen. Since there is an asterisk and numeric value in the `rows`, the bottom row will have a height of `280` and the top row will take all remaining part of the screen.

The `ListView` is setup to iterate over each element in the public `photos` array. When a row is tapped, the `centerMap` method is called to position the map over the photo that was clicked. The `ContentView` is the second row and it will hold the map.

The `ListView` is incomplete though. It should really look like the following:

```
<ListView [items]="photos" row="0" col="0" class="list-group" (itemTap)="centerMap($\
event)">
    <template let-item="item">
        <GridLayout columns="auto, *" rows="auto" class="list-group-item">
            <Image [src]="item.url_q" width="50" height="50" col="0" class="thumb im\
g-rounded"></Image>
            <StackLayout row="0" col="1">
                <Label [text]="item.distance + ' away'" class="list-group-item-headi\
ng"></Label>
                <Label [text]="item.title" class="list-group-item-text" textWrap="tr\
ue"></Label>
            </StackLayout>
        </GridLayout>
    </template>
</ListView>
```

Each row of the `ListView` will have two columns and an automatically sized row height. The first column of the list row will be the image returned from the Flickr API and the second column will have stacked text which includes the photo title and the humanized distance.

```
<ContentView row="1" col="0">
    <Mapbox
        accessToken="{{ mapboxKey }}"
        mapStyle="streets"
        zoomLevel="17"
        hideLogo="true"
        showUserLocation="true"
        (mapReady)="onMapReady($event)">
    </Mapbox>
</ContentView>
```

The Mapbox calls the `onMapReady` function and uses the `mapboxKey` found in the configuration file. Other default properties are used as well.

The first, and default, page of the application is now complete. However, a page for showing the picture still needs to be completed. This is the page navigated to after tapping a marker on the map.

Implementing the Second Page of the Geolocation Application

The second and final page of the application will show an image based on what was selected in the previous page. Open the project's **app/components/image-component/image.component.ts** file and include the following TypeScript code:

```
import { Component, OnInit } from "@angular/core";
import { ActivatedRoute } from "@angular/router";
import { FlickrService } from "../../services/flickr.service";

@Component({
    templateUrl: "components/image-component/image.component.html"
})
export class ImageComponent implements OnInit {

    public url: string;

    public constructor(private activatedRoute: ActivatedRoute, private flickrService\
: FlickrService) { }

    public ngOnInit() { }

    public getPhoto(photoId: number) { }

}
```

The Flickr provider was imported to what was created previously and it is injected into the `constructor` method. The `url` variable will hold the image URL that will be bound to and presented in the UI.

code/nativescript/photos-near-me/app/components/image-component/image.component.ts

```
    public ngOnInit() {
        this.activatedRoute.params.subscribe(params => {
            let photoId = params["photo_id"];
            this.getPhoto(photoId);
        });
    }
```

When this page is initialized, the `ngOnInit` method will obtain the URL parameter and pass it to the `getPhoto` message.

code/nativescript/photos-near-me/app/components/image-component/image.component.ts

```
    public getPhoto(photoId: number) {
        this.flickrService.getPhotoInfo(photoId).subscribe(
            photo => {
                this.url = `https://farm${photo.farm}.staticflickr.com/${photo.serve\
r}/${photo.id}_${photo.secret}_n.jpg`;
            },
            error => console.log(error)
        );
    }
```

After making a request to the Flickr API with the Flickr provider, the public `url` variable will be filled.

With the logic in place, open the project's **app/components/image-component/image-component.html** file and include the following markup:

```
<ActionBar></ActionBar>
<StackLayout>
    <Image [src]="url" width="360" height="360"></Image>
</StackLayout>
```

The `Image` tag will present an image based on the `url` that was populated in the TypeScript code. Within the action bar, there will be a back button to navigate to the previous page.

Try it out!

Now that we have the basic structure for our app in place, try running:

```
tns livesync android --emulator --watch
# or
tns livesync ios --emulator --watch
```

When you're ready to create a build call:

```
tns build android
# or
tns build ios
```

NativeScript for Angular Developers

NativeScript makes it very easy for Angular developers to develop native mobile applications that use native device features, SDKs, and concepts. As technology evolves for the best, the need to know Java or Objective-C is dwindling in favor of these cross platform mobile development frameworks.

Obviously there is a lot more to learn about using NativeScript than we can cover in just this first chapter. Checkout these resources:

- Official NativeScript Site[162]
- Official NativeScript Docs[163]
- NativeScript App Examples[164]
- NativeScript on StackOverflow[165]

[162]https://www.nativescript.org/
[163]https://docs.nativescript.org/
[164]https://www.nativescript.org/app-samples-with-code
[165]http://stackoverflow.com/questions/tagged/nativescript

Changelog

This document highlights the changes for each version of ng-book. You can find this document on the web at: https://www.ng-book.com/2/p/Changelog/[166].

Be sure to check there to ensure that you have the latest revision.

Revision 73 - `2018-01-08`

Updates the book and code to Angular 7 version `angular-7.2.0`.

Revision 72 - `2018-12-12`

Updates the book and code to Angular 7 version `angular-7.1.2`.

Revision 71 - `2018-10-23`

Updates the book and code to Angular 7 version `angular-7.0.0`.

- Chapter "First App"
  - Various bug fixes
- Chapter: "Forms"
  - Overhaul to be consistent with version 7

Revision 70 - `2018-09-13`

Restores the "HTTP" Chapter

Revision 69 - `2018-09-08`

Updates the book and code to Angular 6 version `angular-6.0.5`.

[166] https://www.ng-book.com/2/p/Changelog/

Bug Fixes

- Chapter: "First App"
  - Previous versions instructed gave the command to build as: `ng build --target=production --base-href /` however, the build command was changed to: `ng build --prod`
  - Fixed the `app.component.html` `input` fields to have the `id` attribute for better rendering. Thanks, P. Colagrosso!
  - Fixed some confusing language about titles. Thanks, Brother B!
- Chapter: "How Angular Works"
  - Removed a note about using selectors in a `div` tag, as it requires extra configuration. Thanks, VC!
  - Adds `export` to the early `AppComponent` examples because it's easy to miss later. Thanks, Brother B!
- Chapter: "Routing"
  - Fixed some typos reported by A. Smith. Thanks, A!

Revision 68 - `2018-05-08`

Updates the book and code to Angular 6 version `angular-6.0.0`.

Revision 67 - `2018-01-17`

Updates the book and code to Angular 5 version `angular-5.2.0`.

Revision 66 - `2017-11-14`

- "HTTP" - Updates the chapter to use `HttpClient` instead of `Http`
- "Testing" - Updates the chapter to use `HttpClient`, matching the "HTTP" chapter

Revision 65 - `2017-11-01`

Updates the book and code to Angular 5 version `angular-5.0.0`.

Revision 64 - `2017-09-15`

Updates the book and code to Angular 5 version `angular-5.0.0-beta.6`.

- "Routing" - Fixes Spotify API key issue

Revision 63 - `2017-08-02`

Updates the book and code to Angular 4 version `angular-4.3.2`.

Revision 62 - `2017-06-23`

- "How to Read This Book" - Fixed typos via Travas N.
- "First App" - Fixed typos via Travas N.
- "NativeScript" - Fixed typos via Travas N.
- "Advanced Components" - Fixed typos via Travas N.
- "Redux" - Fixed typos via Travas N.
- "Testing" - Fixed typos via Travas N.

Revision 61 - `2017-05-24`

- "How to Read This Book" - Fixed typo reported by Daniel R.
- "First App" - Fixed typos reported by Daniel R.
- "TypeScript" - Fixed typo reported by Daniel R.
- "Advanced Components" - Fixed typos reported by Daniel R.
- "RxJS" - Fixed dead link to RxMarbles reported by Daniel R., fixed typos by Travas N.
- "Redux" - Fixed typos via Travas N.
- "Routing" - Fixed typos via Travas N.

Revision 60 - `2017-04-27`

Updates the book and code to Angular 4 version `angular-4.1.0`.

Revision 59 - `2017-04-07`

- "First App" - Reddit example, added `ids` to the `input` tags and other typos - Reported by Alexey A., Damien W.
- "How Angular Works" - Fixed typos reported by Richard M.
- "Built-in Components" - Fixed typos reported by Oleksij L.
- "Advanced Components" - Typos reported by Tom G. and Arshaan B.
- "HTTP" - Typos reported by Arshaan B.
- "Dependency Injection" - Fixes wrong code example reported by Emin L.
- "Redux" - Fixes old reference to `OpaqueToken` (now `InjectionToken`), Fixes typos reported by Arshaan B.
- "Routing" - Fixed URLs that pointed to localhost:8080 instead of port 4200 - Reported by Arshaan B.
- "RxJS" - Fixes a confusing hypothetical subscription reported by Arshaan B.

- "Forms" - Fixes the demo which was using the same component and typos - reported by Christopher S., Richard M.
- Fixed remaining `next()` -> `emit()` for `EventEmitter`
- Replaced "parenthesis" with "parentheses" when it was intended. Special thanks to Richard M. for pointing this out
- Use consistent casing for "JavaScript" everywhere - reported by Damien W.

Revision 58 - `2017-03-24`

Updates the book and code to Angular 4 version `angular-4.0.0`.

Revision 57 - `2017-03-23`

Updates the book to Angular 4 version `angular-4.0.0.rc6`.

- Fixes the `SpyObject` in the music/routing test.

Revision 56 - `2017-03-22`

Updated the **entire book** to use Angular CLI, the new styleguide for folder layout, and pass linting.

Updated code to `angular-2.4.10`.

This includes updates to:

- "Forms"
- "Advanced Components"
- "HTTP"
- "Routing"
- "Testing"
- "RxJS Intro"
- "RxJS Chat"
- "Redux Intro"
- "Redux Chat" " "NgUpgrade Conversion"

Revision 55 - `2017-03-17`

- Rewrote "Dependency Injection" Chapter and updated it to use `angular-cli` and conform to style guide. Moved it earlier in the book.
- Updated "Built-in Directives" Chapter to use `angular-cli` and conform to style guide
- "Writing your First Angular Web Application", clarified thanks to input from Zach S., Blair A., Leandro A.

Revision 54 - 2017-03-10

- Book updated to `angular-2.4.9`
- Changed the use of the word `annotation` to `decorator`, across the board
- "Dependency Injection" added `@Injectable` description and fixed typos.
- Webpack CSS fix in many chapters reported by Daniel W.
- Pointed out where Bootstrap is used: HTTP, Routing, Introduction to Redux with TypeScript, Data Architecture with Observables
- Added dot-notation example as recommended by Luis M. T. L.
- Changed link to `AbstractControl`
- Added link regarding two-way binding as recommended by Tom G.
- "Writing your First Angular 2 Web Application", clarified as reported by Brother Bill, Terry W., Rob D., Robert S., and Aaron K.
- "Built-in Directives", clarified as reported by Brother Bill
- "Forms", typo reported by Robert S. and Andrew B.
- "HTTP", typo reported by Brother Bill
- "Routing", clarified as reported by Brother Bill and Daniel F.
- "TypeScript", bug reported by Willemhein T. and Shane G.
- Changed `EventEmitter`'s depreciated next() to emit(), reported by Adam Beck

Revision 53 - 2017-03-01

- Added a section on deployment to the first chapter
- Updated "How Angular Works" to use `@angular/cli`
- Updated `@angular/cli` to version `1.0.0-rc.0`
- Added a note about how to run the examples in "Built-in Directives"
- Updated "How to Read This Book" with a note about each project's `README.md`

Revision 52 - 2017-02-22

- Added "How to Read This Book"
- Updated `angular-cli` to use `@angular/cli` package
- Clarity updates to the first chapter
- Book updated to `angular-2.4.8`

Revision 51 - 2017-02-14

Fixes code formatting bugs in the first chapter

Revision 50 - 2017-02-10

Book updated to `angular-2.4.7`

Revision 49 - `2017-01-18`

Minor fixes

Revision 48 - `2017-01-13`

Added chapter on building native mobile apps with NativeScript and Angular

Revision 47 - `2017-01-06`

Fixes missing images in Built-in Directives

Revision 46 - `2017-01-03`

Book up to date with `angular-2.4.0`

- Added Protractor E2E tests for every project
- "Routing", Fixed a child route `pathMatch` ambiguity

Revision 45 - `2016-12-05`

Book up to date with `angular-2.3.0-rc.0`

Revision 44 - `2016-11-17`

Fixed typos in chapters:

- "Writing your First Angular 2 Web Application", reported by Mike B., Steve A., Terry W., Alessandro C., Andrew Blair
- "TypeScript", reported by Kevin D.
- "How Angular Works", reported by Kevin D. and Jason T.
- "Forms", reported by Kevin D.
- "HTTP", reported by Kevin D.
- "Routing", reported by Kevin D.
- "Advanced Components", reported by Kevin D.
- "Built-in Directives", reported by Jason T. and Farooq A.
- "Dependency Injection", reported by Kevin D.
- "Testing", reported by Kevin D.
- "Converting an Angular 1 App to Angular 2", reported by Kevin D.

Revision 43 - `2016-11-08`

Book up to date with `angular-2.2.0-rc.0`

Revision 42 - `2016-10-14`

Entire book up to date with `angular-2.1.0` Bonus video content and sample app completed (premium package users)

- Chapter “Built-in Components” renamed to “Built-in Directives”
- Service dependencies made private, reported by Jamie B.

Fixed typos and clarified in chapters:

- “How Angular Works”, reported by kbiesbrock
- “Converting to ng2”, reported by Dilip S.
- “Built-in Directives”, reported by Pieris C.
- “Dependency Injection”, reported by Tim P.
- “Routing”, reported by Kashyap M
- “Advanced Components”, reported by Kashyap M., by Justin B. and many by Németh T.

Revision 41 - `2016-09-28`

Rewrote the first chapter to use `ng-cli` and the new styleguide.

- “First App” Chapter:
  - Split files into style-guide friendly templates and components
  - Fixed a bunch of typos reported by David S., and Luis H., Jan L., Aaron Spilman
- “HTTP” Chapter - fixed typos - Thanks Jim H.!

Revision 40 - `2016-09-20`

Entire book up to date with `angular-2.0.0` final!

Revision 39 - `2016-09-03`

Entire book up to date with `angular-2.0.0-rc.6`

Revision 38 - `2016-08-29`

Entire book up to date with `angular-2.0.0-rc.5`

- Entire book changes:
  - Upgraded every example to use `NgModules`
  - Upgraded tests to use `TestBuilder`

Revision 37 - `2016-08-02`

New chapter: Intermediate Redux in Angular 2!

Bugfixes:

- `ts-cli` -> `ts-node` - Thanks Tim. P

Revision 36 - `2016-07-20`

New chapter: Redux in TypeScript and Angular 2!

- Re-ordered chapters

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[167]

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

[167] https://github.com/angular/angular/blob/master/CHANGELOG.md#200-beta5-2016-02-10

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[168] styles
- "Built-in Components" Chapter
  - Updated built-in directives 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.

[168] http://semantic-ui.com

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 - `2015-08-01`

- Finished built-in components chapter

Revision 4 - `2015-07-30`

- 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 - `2015-07-21`

- Added forms chapter

Revision 2 - `2015-07-15`

- 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 - `2015-07-01`

Initial version of the book