



ng-book 2

The Complete Book on AngularJS 2



FULLSTACK.io



Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

ng-book 2

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

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

Contents

Book Revision	1
Prerelease	1
Bug Reports	1
Chat With The Community!	1
Writing your First Angular 2 Web Application	1
Simple Reddit Clone	1
Getting started	4
Our First TypeScript	8
Running The App	13
Adding Data to the Component	16
Working with arrays	18
Expanding our Application	21
Rendering Multiple Rows	37
Adding New Articles	47
Finishing Touches	48
Full Code Listing	49
Wrapping Up	53
Getting Help	53
TypeScript	54
Angular 2 is built in TypeScript	54
What do we get with TypeScript?	55
Types	56
Built-in types	58
Classes	60
Utilities	66
Wrapping up	69
How Angular Works	70
Application	70
Product Model	74
Components	75
Component Decorator	77
The ProductsList Component	85

CONTENTS

The ProductRow Component	97
The ProductImage Component	100
The PriceDisplay Component	101
The ProductDepartment Component	102
The Completed Project	103
A Word on Data Architecture	104
Built-in Components	106
Introduction	106
NgIf	106
NgSwitch	106
NgStyle	109
NgClass	111
NgFor	114
NgNonBindable	120
Conclusion	120
Forms in Angular 2	121
Forms are Crucial, Forms are Complex	121
Controls and Control Groups	121
Our First Form	123
Using FormBuilder	130
Adding Validations	134
Watching For Changes	145
ngModel	147
Wrapping Up	149
Data Architecture in Angular 2	150
An Overview of Data Architecture	150
Data Architecture with Observables - Part 1: Services	152
Observables and RxJS	152
Chat App Overview	154
Implementing the Models	157
Implementing UserService	159
The MessagesService	162
The ThreadsService	175
Data Model Summary	186
Data Architecture with Observables - Part 2: View Components	187
Building Our Views: The ChatApp Top-Level Component	187
The ChatThreads Component	189
The Single ChatThread Component	193
The ChatWindow Component	197

CONTENTS

The ChatMessage Component	207
The ChatNavBar Component	212
Summary	216
Next Steps	218
HTTP	219
Introduction	219
Using angular2/http	220
A Basic Request	221
Writing a YouTubeSearchComponent	226
angular/http API	244
Routing	248
Why routing?	248
How client-side routing works	249
Writing our first routes	251
Components of Angular 2 routing	251
Putting it all together	254
Routing strategies	260
Path location strategy	262
Running the application	262
Route Parameters	264
Music Search App	266
Router Lifecycle Hooks	283
Nested routes	294
Summary	299
Changelog	300

Book Revision

Revision 14 - Covers up to Angular 2 (2.0.0-beta.0, 2015-12-23)

Prerelease

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

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: us@fullstack.io¹.

Chat With The Community!

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

¹<mailto:us@fullstack.io?Subject=ng-book%20feedback>

²<https://gitter.im/ng-book/ng-book>

Writing your First Angular 2 Web Application

Simple Reddit Clone

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

You can think of this app as the beginnings of a site like [Reddit³](#) or [Product Hunt⁴](#).

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

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

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

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

³<http://reddit.com>

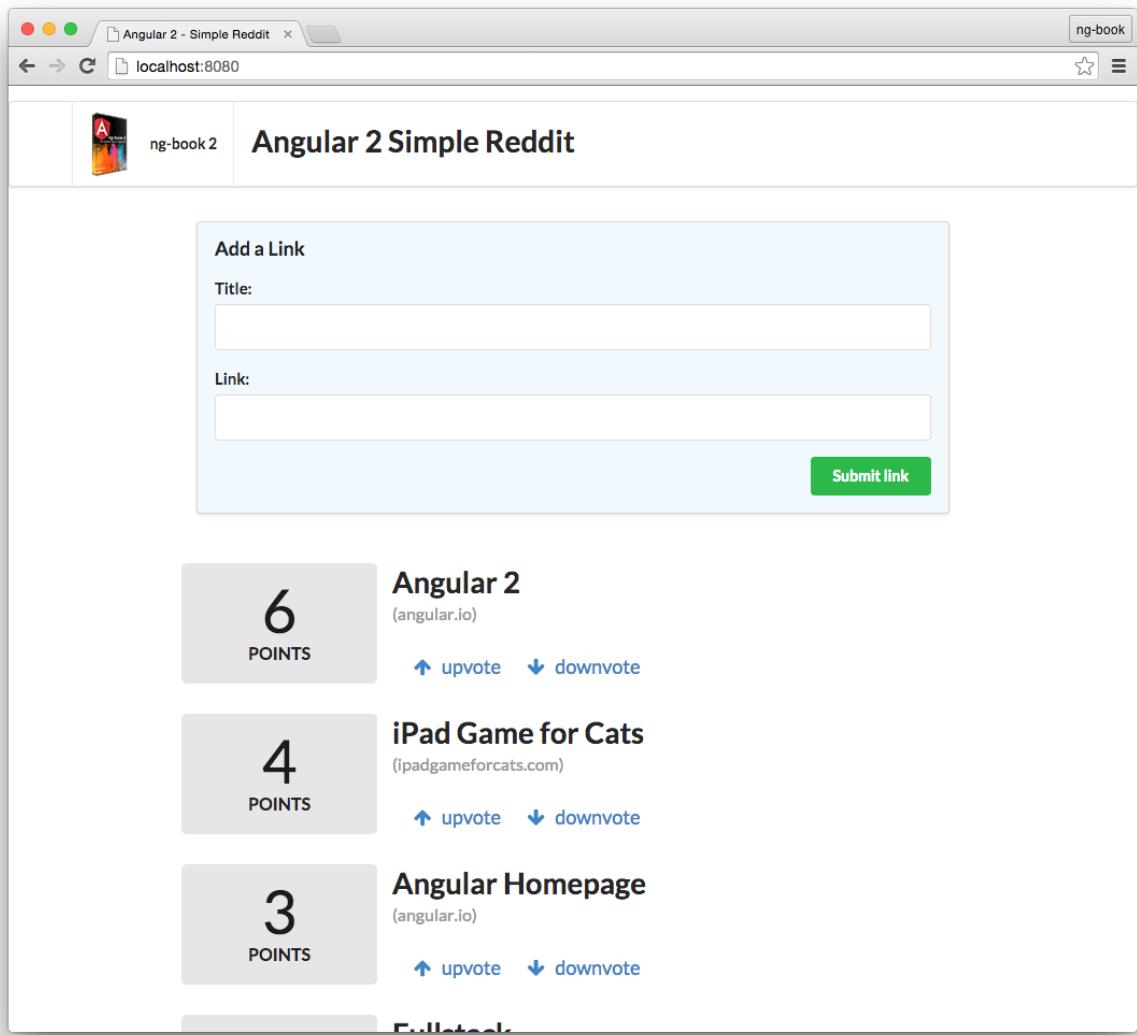
⁴<http://producthunt.com>

The screenshot shows a web browser window titled "Angular 2 - Simple Reddit" at "localhost:8080". The page has a header with a logo and the text "ng-book 2" and "Angular 2 Simple Reddit". Below the header is a form titled "Add a Link" with fields for "Title" (containing "iPad Game for Cats") and "Link" (containing "http://ipadgameforcats.com"). A green "Submit link" button is visible. Below the form, there are three listed links:

- Angular 2**
(angular.io)
3 POINTS
↑ upvote ↓ downvote
- Fullstack**
(fullstack.io)
2 POINTS
↑ upvote ↓ downvote
- Angular Homepage**
(angular.io)
1 POINTS
↑ upvote ↓ downvote

Completed application

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



App with new article

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

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

Getting started

TypeScript

To get started with TypeScript, you'll need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to the Node.js website for detailed information: <https://nodejs.org/download/>⁵.



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

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

```
1 $ npm install -g 'typescript@1.7.3'
```



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



Windows Users: We'll be using Linux/Mac-style commands on the commandline throughout this book. We'd highly recommend you install [Cygwin](#)⁶ as it will let you run commands just as we have them written out in this book.

Example Project

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

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

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

⁵<https://nodejs.org/download/>

⁶<https://www.cygwin.com/>



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

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

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

```
1 $ npm install
```

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

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

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

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

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

Angular's Dependencies



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

Angular 2 depends on these four libraries:

- `es6-shim` - (for older browsers)
- `angular2-polyfills`
- `SystemJS`
- `RxJS`

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

```

1 <script src="node_modules/es6-shim/es6-shim.js"></script>
2 <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
3 <script src="node_modules/systemjs/dist/system.src.js"></script>
4 <script src="node_modules/rxjs/bundles/Rx.js"></script>
5 <script src="node_modules/angular2/bundles/angular2.dev.js"></script>

```



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

ES6 Shim

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



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

For instance, check out this [ES6 Compatibility Table](#)⁷. Not every browser is completely compatible with every feature. By using different *shims* we're able to get standardized behavior across different browsers (and environments).

See also: [What is the difference between a shim and a polyfill?](#)⁸

⁷<https://kangax.github.io/compat-table/es6/>

⁸<http://www.2ality.com/2011/12/shim-vs-polyfill.html>

For more information on `es6-shim` checkout the project page⁹.

Angular 2 Polyfills

Like ES6 Shim, `angular2-polyfills` provide some foundational standardization across browsers. Specifically `angular2-polyfills` contains code for dealing with zones, promises, and reflection. If you don't know what those things are, you don't need to worry about it at this point.

SystemJS

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

RxJS

RxJS is a library for doing reactive programming in Javascript. Loosely speaking, RxJS gives us tools for working with *Observables*, which emit streams of data. Angular uses *Observables* in many places when dealing with asynchronous code (e.g. HTTP requests).

We talk a lot more about RxJS in the [RxJS Chapter of this book](#). While it isn't strictly needed for this chapter, it's worth mentioning here because you'll often use it in your projects.

You can read more about the [RxJS library](#) here at the official project page¹⁰.

All Dependencies Loaded

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

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

⁹<https://github.com/paulmillr/es6-shim>

¹⁰<https://github.com/ReactiveX/RxJS>

Adding CSS

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

```

1  <!doctype html>
2  <html>
3      <head>
4          <title>Angular 2 - Simple Reddit</title>
5          <!-- Libraries -->
6          <script src="node_modules/es6-shim/es6-shim.js"></script>
7          <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
8          <script src="node_modules/systemjs/dist/system.src.js"></script>
9          <script src="node_modules/rxjs/bundles/Rx.js"></script>
10         <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
11
12         <!-- Stylesheet -->
13         <link rel="stylesheet" type="text/css"
14             href="resources/vendor/semantic.min.css">
15         <link rel="stylesheet" type="text/css" href="styles.css">
16     </head>
17     <body>
18     </body>
19 </html>
```



For this project we're going to be using [Semantic-UI¹¹](#) to help with the styling. Semantic-UI is a CSS framework, similar to Foundation or Twitter Bootstrap. We've included it in the sample code download so all you need to do is add the `link` tag.

Our First TypeScript

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



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

¹¹<http://semantic-ui.com/>

code/first_app/hello-world/app.ts

```
1 import { bootstrap } from "angular2/platform/browser";
2 import { Component } from "angular2/core";
3
4 @Component({
5   selector: 'hello-world',
6   template: `
7     <div>
8       Hello world
9     </div>
10    `
11  })
12 class HelloWorld {
13 }
14
15 bootstrap(HelloWorld);
```

This snippet may seem scary at first, but don't worry. We're going to walk through it step by step. The `import` statement defines the modules we want to use to write our code. Here we're importing two things: `Component`, and `bootstrap`.

We `import Component` from the module `"angular2/core"`. The `"angular2/core"` portion tells our program **where to find the dependencies** that we're looking for.

Similarly we `import bootstrap` from the module `"angular2/platform/browser"`.

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

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

Making a Component

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

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

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



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

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

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

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

1. A Component annotation
2. A component definition class

Let's take these one at a time.

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

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

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

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

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

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

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

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

Adding a template

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

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

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



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

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

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

Booting Our Application

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

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

Loading our Application

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

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

Add the following to the body section:

```

1  <!doctype html>
2  <html>
3      <head>
4          <title>First App - Hello world</title>
5          <!-- Libraries -->
6          <script src="node_modules/es6-shim/es6-shim.js"></script>
7          <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
8          <script src="node_modules/systemjs/dist/system.src.js"></script>
9          <script src="node_modules/rxjs/bundles/Rx.js"></script>
10         <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
11
12         <!-- Stylesheet -->
13         <link rel="stylesheet" type="text/css" href="resources/vendor/semantic.min.c\
14 ss">
15         <link rel="stylesheet" type="text/css" href="styles.css">
16     </head>
17     <body>
18         <script>
19             System.config({
20                 packages: {
21                     app: {
22                         format: 'register',
23                         defaultExtension: 'js'
24                     }
25                 }
26             });
27             System.import('app.js')
28                 .then(null, console.error.bind(console));
29         </script>
30
31         <hello-world></hello-world>
32
33     </body>
34 </html>

```

Here in our script tag we're configuring our module loader, System.js. The important line to understand here is:

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

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

Running The App

Compiling the TypeScript Code to .js

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

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

```
1 tsc
```

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

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



Troubleshooting:

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

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



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

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

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

Using npm

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

Try running:

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

Serving The App

We have one more step to test our application: we need to run a local webserver to serve our app. If you did an `npm install` earlier, you've already got a local webserver installed. To run it, just run the following command:

```
1 npm run serve
```

Then open up your browser to <http://localhost:8080>¹².



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

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

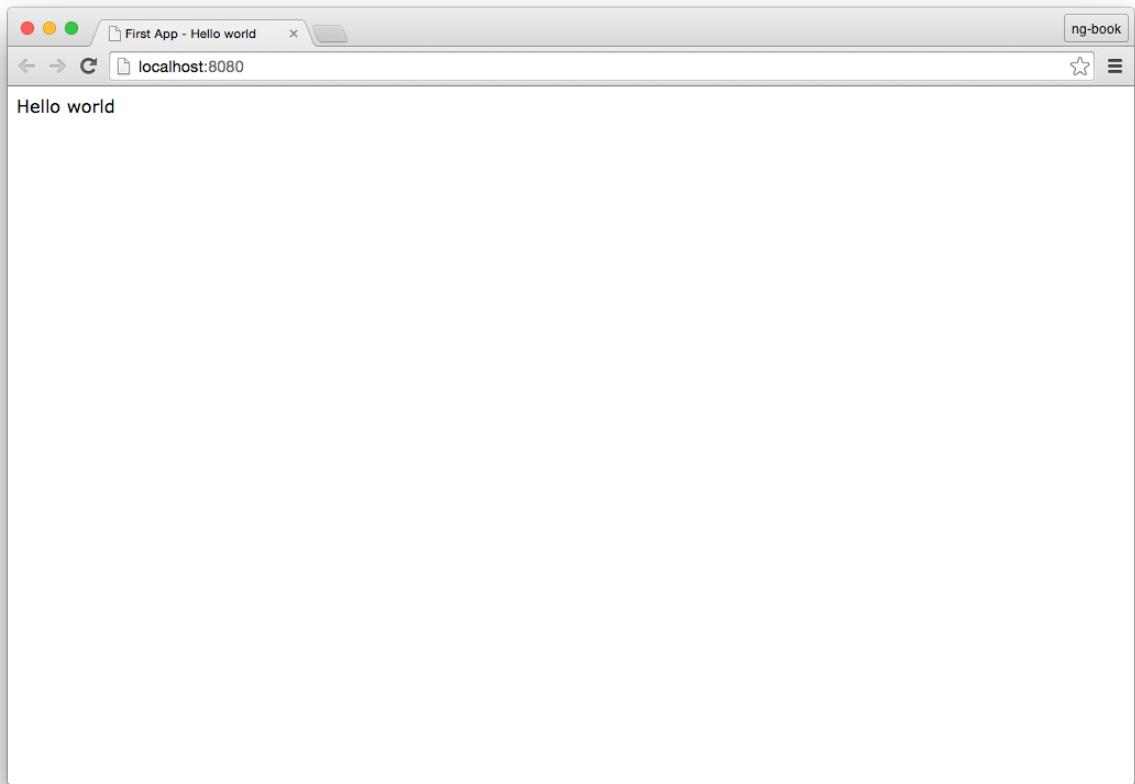
So instead we run a local webserver that simply serve whatever is on the filesystem. This is really convenient for testing, but not how you would deploy your production application.



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

If everything worked correctly, you should see the following:

¹²<http://localhost:8080>



Completed application



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

1. Make sure that your `app.js` file was created from the Typescript compiler `tsc`
2. Make sure that your webserver was started in the same directory as your `app.js` file
3. Make sure that your `index.html` file matches our code example above
4. Try opening the page in Chrome, right click, and pick "Inspect Element". Then click the "Console" tab and check for any errors.
5. If all else fails, [join us here to chat on Gitter!](#)¹³

¹³<https://gitter.im/ng-book/ng-book>

Compiling on every change

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

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

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

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

```
1 npm run go
```

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

Adding Data to the Component

Right now our component renders a static string, which means our component isn't very interesting. Let's introduce `name` as a new property of our component. This way we can reuse the same component for different inputs.

Make the following changes:

```
1 @Component({
2   selector: 'hello-world',
3   template: `<div>Hello {{ name }}</div>`
4 })
5 class HelloWorld {
6   name: string;
7
8   constructor() {
9     this.name = 'Felipe';
10  }
11 }
```

Here we've changed three things:

1. name Property

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

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

2. A Constructor

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

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

When we write:

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

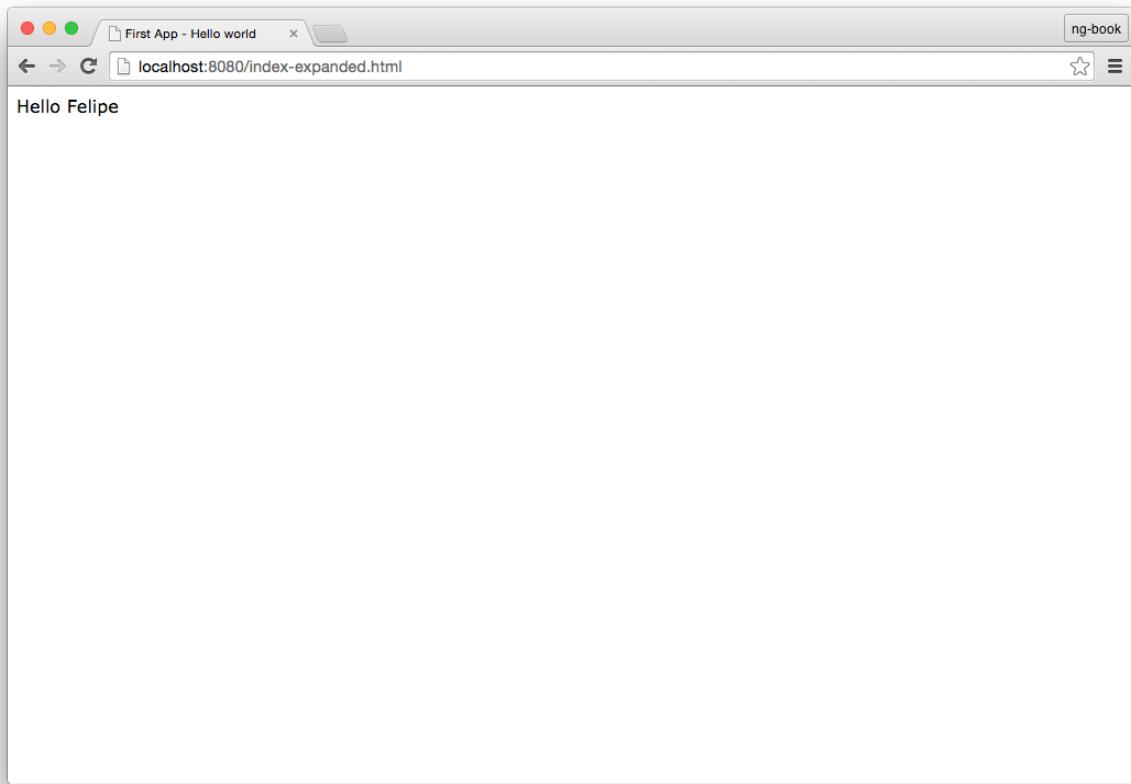
We're saying that whenever a new `HelloWorld` is created, set the name to 'Felipe'.

3. Template Variable

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

Try it out

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



Application with Data

Working with arrays

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

If you’ve worked with Angular 1 before, you probably used `ng-repeat` directive. In Angular 2, the analogous directive is called `NgFor`. Its syntax is slightly different but they have the same purpose: **repeat the same markup for a collection of objects**.

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

```

1 import { bootstrap } from "angular2/platform/browser";
2 import { Component } from "angular2/core";
3 import { NgFor } from "angular2/common";
4
5 @Component({
6   selector: 'hello-world',
7   template: `
8     <ul>
9       <li *ngFor="#name of names">Hello {{ name }}</li>
10    </ul>
11  `
12 })
13 class HelloWorld {
14   names: string[];
15
16   constructor() {
17     this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
18   }
19 }
20
21 bootstrap(HelloWorld);

```

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

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

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

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

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

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



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

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

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

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

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



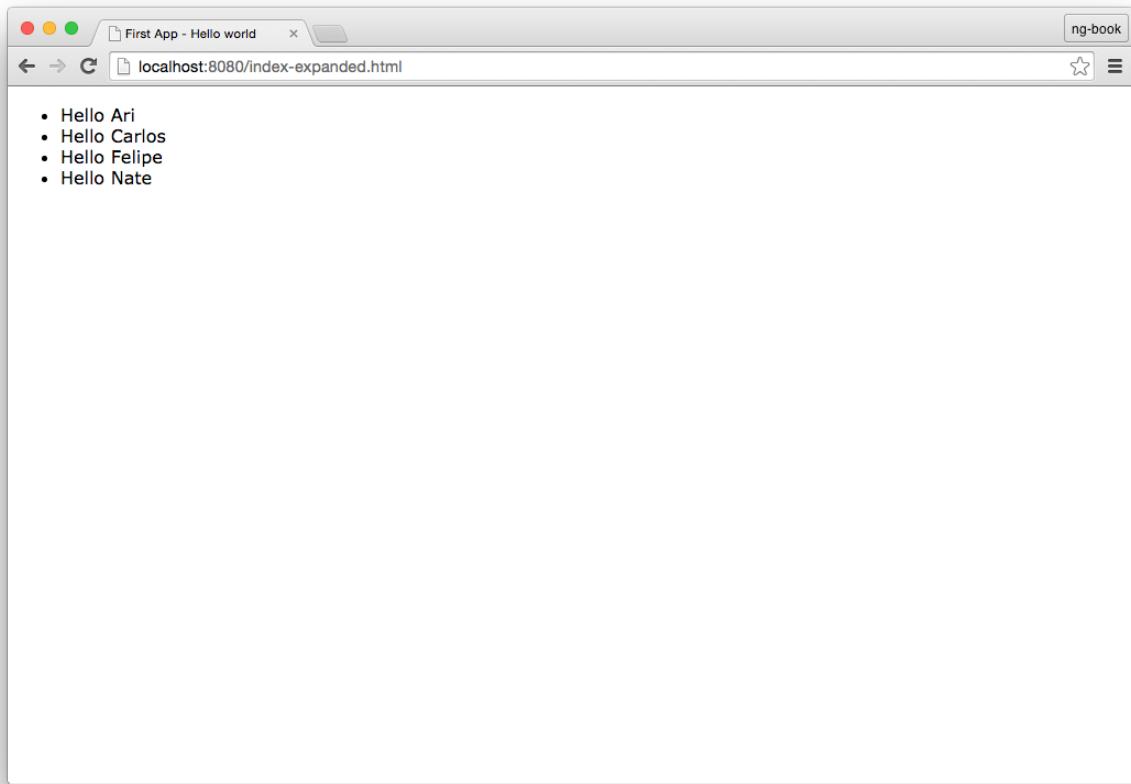
`NgFor` repeats the element that the `ngFor` is attached to. That is, we put it on the `li` tag and **not** the `ul` tag because we want to repeat the list element (`li`) and not the list itself (`ul`).



If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the `NgFor` directive [here¹⁴](#)

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

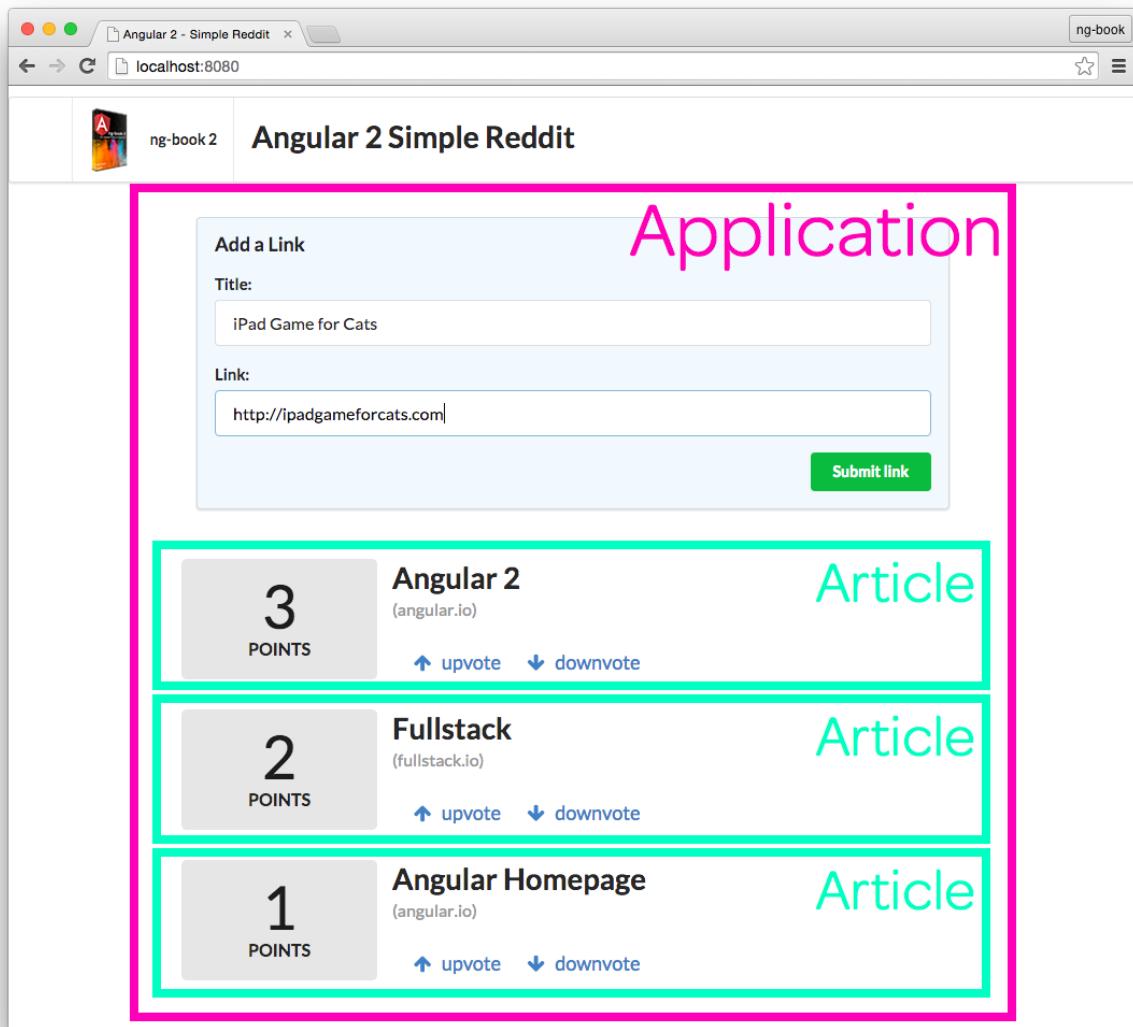
¹⁴https://github.com/angular/angular/blob/master/modules/angular2/src/common/directives/ng_for.ts



Application with Data

Expanding our Application

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



Application with Data

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

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



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

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

The Application Component

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

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

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



For this example we're using the [Semantic UI¹⁵](#) CSS Framework. In our template below when you see classes on the attributes - like `class="ui large form segment"` - these are styles coming from Semantic. It's a great way to have our app look nice without too much extra markup.

```
1 import { bootstrap } from 'angular2/platform/browser';
2 import { Component } from 'angular2/core';
3
4 @Component({
5   selector: 'reddit',
6   template: `
7     <form class="ui large form segment">
8       <h3 class="ui header">Add a Link</h3>
9
10      <div class="field">
11        <label for="title">Title:</label>
12        <input name="title">
13      </div>
14      <div class="field">
15        <label for="link">Link:</label>
16        <input name="link">
17      </div>
18    </form>
19
20  `})
21 class RedditApp {
22   constructor() {
23   }
24 }
25
26 bootstrap(RedditApp);
```

¹⁵<http://semantic-ui.com>

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

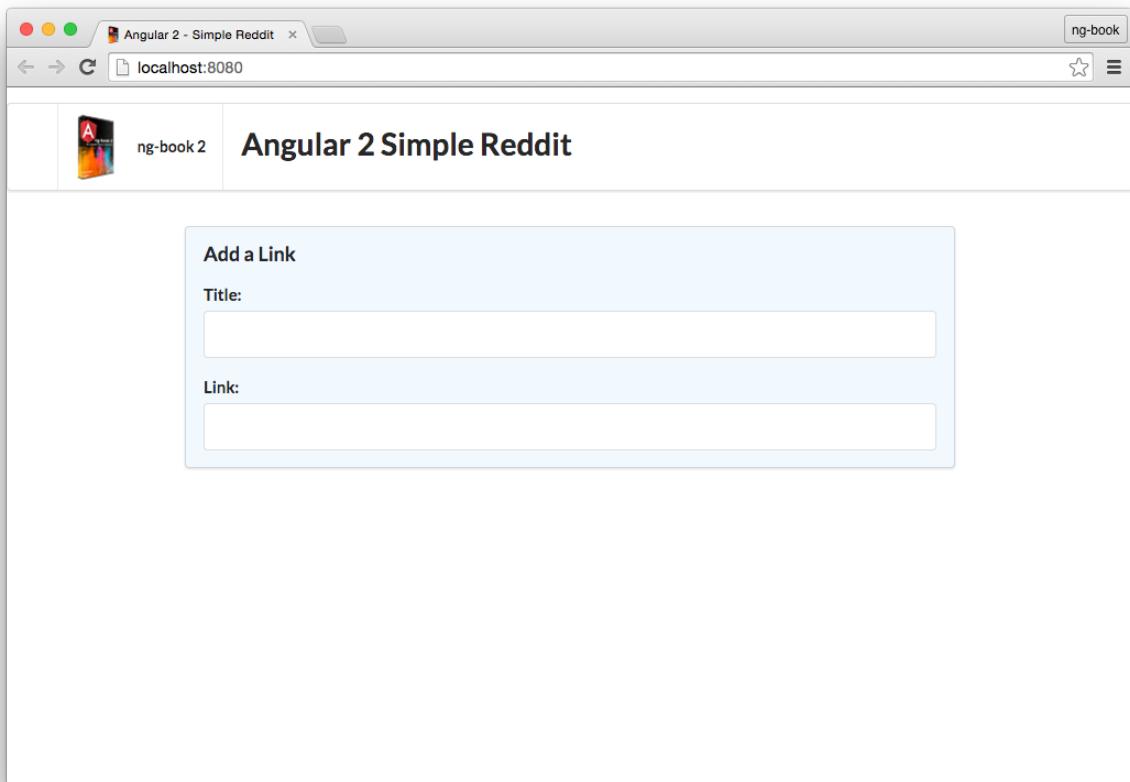
We're creating a template that defines two input tags: one for the title of the article and the other for the link URL.

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



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

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



Form



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

Adding Interaction

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

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

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

1. Created a button tag in our markup that shows the user where to click
2. We created a function named `addArticle` that defines what we want to do when the button is clicked

3. We added a `(click)` attribute on the button that says “call the function `addArticle` when this button is pressed”.
4. We added the attribute `#newtitle` and `#newlink` to the `<input>` tags

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

Binding inputs to values

Notice in our first input tag we have the following:

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

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

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

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

Binding actions to events

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

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

All together:

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



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

Defining the Action Logic

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

```
1   addArticle(title: HTMLInputElement, link: HTMLInputElement): void {  
2     console.log(`Adding article title: ${title.value} and link: ${link.value}`)\n3   );  
4 }
```

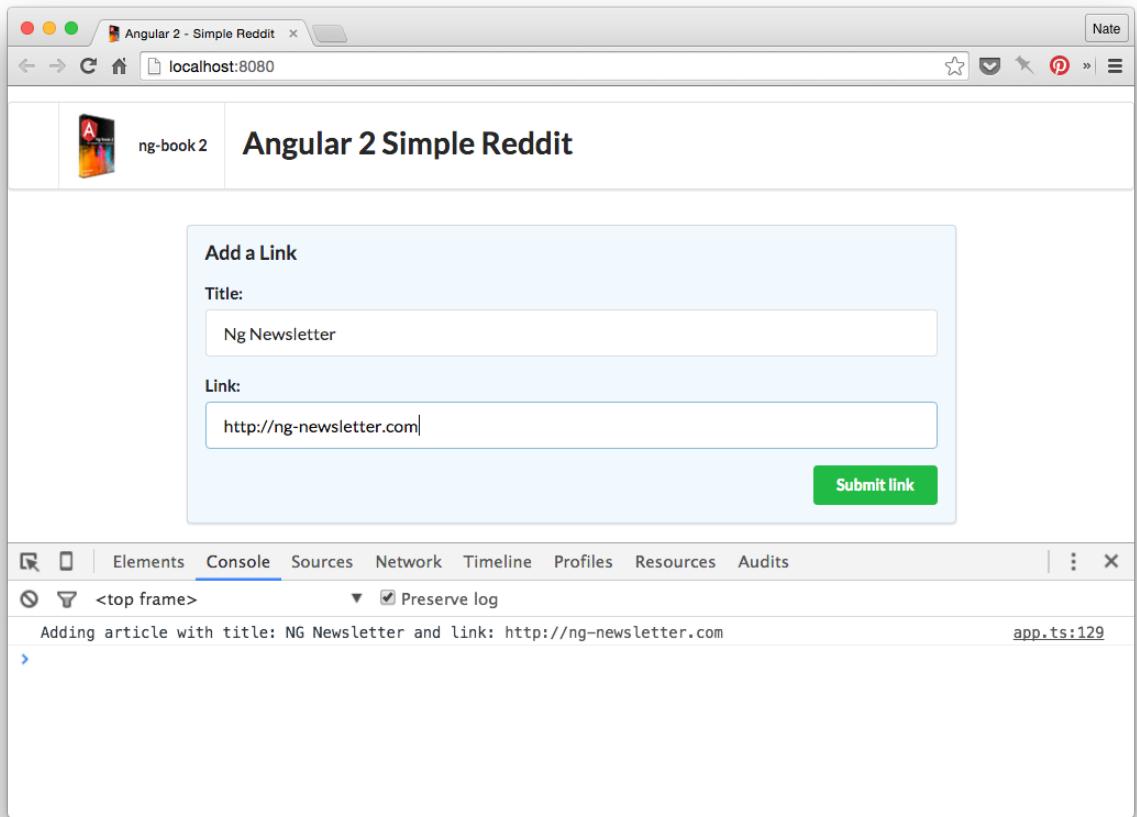


Notice that we're using backtick strings again. This is a really handy feature of ES6: backtick strings will expand template variables!

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

Try it out!

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



Clicking the Button

Adding the Article Component

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

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

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

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

```
1 @Component({
2   selector: 'reddit-article',
3   host: {
4     class: 'row'
5   },
6   template: `
7     <div class="four wide column center aligned votes">
8       <div class="ui statistic">
9         <div class="value">
10           {{ votes }}
11         </div>
12         <div class="label">
13           Points
14         </div>
15       </div>
16     </div>
17     <div class="twelve wide column">
18       <a class="ui large header" href="{{ link }}"
19         {{ title }}
20       </a>
21       <ul class="ui big horizontal list voters">
22         <li class="item">
23           <a href (click)="voteUp()">
24             <i class="arrow up icon"></i>
25             upvote
26           </a>
27         </li>
28         <li class="item">
29           <a href (click)="voteDown()">
30             <i class="arrow down icon"></i>
31             downvote
32           </a>
33         </li>
34       </ul>
35     </div>
36   `
37 })
38 class ArticleComponent {
39   votes: number;
40   title: string;
41   link: string;
```

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

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

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

Let's talk through each part in detail:

Creating the `reddit-article` Component

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

```
1 @Component({
2     selector: 'reddit-article',
3     host: {
4         class: 'row'
5     },
6 }
```

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

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

```

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

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

We want each `reddit-article` to be on its own row. We're using Semantic UI, and Semantic provides a [CSS class for rows¹⁶](#) called `row`.

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



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

Creating the `reddit-article` template

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

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

```

1  template: `
```

- 2 <div class="four wide column center aligned votes">
- 3 <div class="ui statistic">
- 4 <div class="value">
- 5 {{ votes }}
- 6 </div>
- 7 <div class="label">
- 8 Points
- 9 </div>
- 10 </div>
- 11 </div>
- 12 <div class="twelve wide column">
- 13
- 14 {{ title }}
- 15
- 16 <ul class="ui big horizontal list voters">
- 17 <li class="item">
- 18 <a href (click)="voteUp()">

¹⁶<http://semantic-ui.com/collections/grid.html>

```

19      <i class="arrow up icon"></i>
20      upvote
21    </a>
22  </li>
23  <li class="item">
24    <a href (click)="voteDown()">
25      <i class="arrow down icon"></i>
26      downvote
27    </a>
28  </li>
29  </ul>
30 </div>
31

```

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.

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

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

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

Creating the reddit-article ArticleComponent Definition Class

Finally, we create the `ArticleComponent` definition class:

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

```
1 class ArticleComponent {
2   votes: number;
3   title: string;
4   link: string;
5
6   constructor() {
7     this.title = 'Angular 2';
8     this.link = 'http://angular.io';
9     this.votes = 10;
10  }
11
12  voteUp() {
13    this.votes += 1;
14  }
15
16  voteDown() {
17    this.votes -= 1;
18  }
19 }
```

Here we create three properties on ArticleComponent:

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

In the constructor() we set some default attributes:

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

```
1 constructor() {
2   this.title = 'Angular 2';
3   this.link = 'http://angular.io';
4   this.votes = 10;
5 }
```

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

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

```
1  voteUp() {
2      this.votes += 1;
3  }
4
5  voteDown() {
6      this.votes -= 1;
7 }
```

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

Using the reddit-article Component

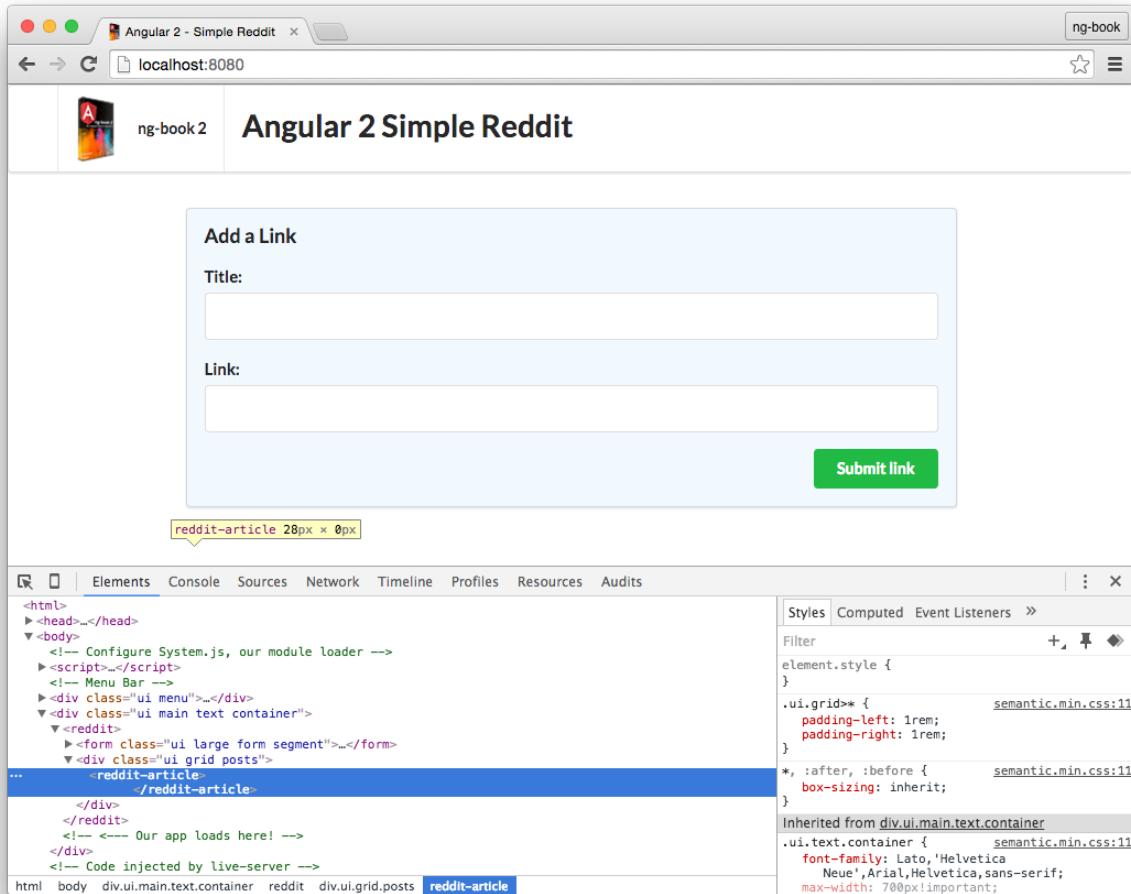
In order to use this component and make the data visible, we have to add a `<reddit-article></reddit-article>` tag somewhere in our markup.

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

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

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

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



Unexpanded tag when inspecting the DOM

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



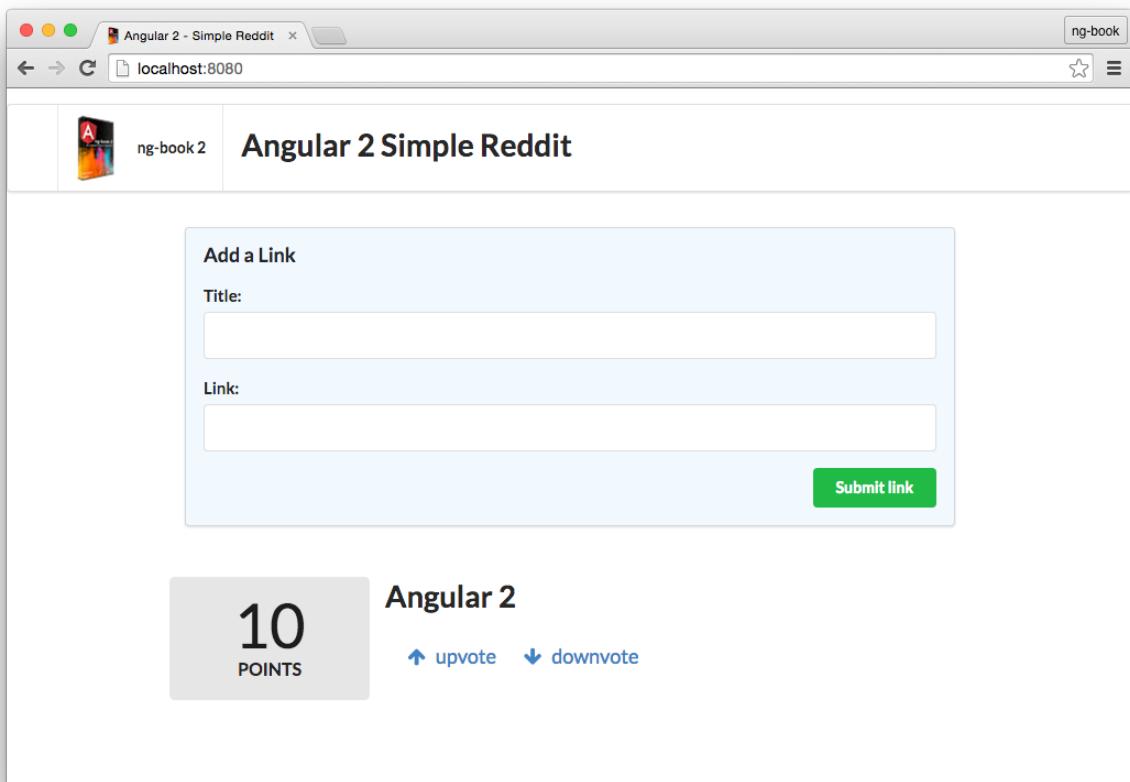
Angular 1 Note: If you've used Angular 1 it might be surprising that our app doesn't know about our new `reddit-article` component. This is because in Angular 1, directives match globally. However, in Angular 2 you need explicitly specify which components (and therefore, which selectors) you want to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

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

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

```
1 // for RedditApp
2 @Component({
3   selector: 'reddit',
4   directives: [ArticleComponent],
5   template: `
6     // ...`
```

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



Rendered ArticleComponent component

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

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

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

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

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

Rendering Multiple Rows

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

Creating an Article class

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

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

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

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

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

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

```
1  class ArticleComponent {
2      article: Article;
3
4      constructor() {
5          this.article = new Article('Angular 2', 'http://angular.io', 10);
6      }
7
8      voteUp(): boolean {
9          this.article.votes += 1;
10         return false;
11     }
12
13     voteDown(): boolean {
14         this.article.votes -= 1;
15         return false;
16     }
17 }
```

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

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

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

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

```
1 template: `
```

```
2   <div class="four wide column center aligned votes">
```

```
3     <div class="ui statistic">
```

```
4       <div class="value">
```

```
5         {{ article.votes }}
```

```
6       </div>
```

```
7       <div class="label">
```

```
8         Points
```

```
9       </div>
```

```
10      </div>
```

```
11    </div>
```

```
12    <div class="twelve wide column">
```

```
13      <a class="ui large header" href="{{ article.link }}">
```

```
14        {{ article.title }}
```

```
15      </a>
```

```
16      <ul class="ui big horizontal list voters">
```

```
17        <li class="item">
```

```
18          <a href (click)="voteUp()">
```

```
19            <i class="arrow up icon"></i>
```

```
20            upvote
```

```
21          </a>
```

```
22        </li>
```

```
23        <li class="item">
```

```
24          <a href (click)="voteDown()">
```

```
25            <i class="arrow down icon"></i>
```

```
26            downvote
```

```
27          </a>
```

```
28        </li>
```

```
29      </ul>
```

```
30    </div>
```

```
31  `
```

Reload the browser and everything should still work.

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



`voteUp` and `voteDown` current break the [Law of Demeter¹⁷](#) which says that a given object should assume as little as possible about the structure or properties any other objects. One way to detect this is to be suspicious when you see long method/property chains like `foo.bar.baz.bam`. This pattern of long-method chaining is also affectionately referred to as a “train-wreck”.

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

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

```

1  class Article {
2      title: string;
3      link: string;
4      votes: number;
5
6      constructor(title: string, link: string, votes?: number) {
7          this.title = title;
8          this.link = link;
9          this.votes = votes || 0;
10     }
11
12     voteUp(): void {
13         this.votes += 1;
14     }
15
16     voteDown(): void {
17         this.votes -= 1;
18     }

```

Then we’ll change `ArticleComponent` to call these methods:

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

```

1  class ArticleComponent {
2      article: Article;
3
4      voteUp(): boolean {
5          this.article.voteUp();
6          return false;
7      }
8

```

¹⁷http://en.wikipedia.org/wiki/Law_of_Demeter

```

9  voteDown(): boolean {
10    this.article.voteDown();
11    return false;
12  }
13 }
```



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](#)¹⁸. The idea is that we want to move most of our domain logic to our models so that our components do the minimum work possible.

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

Storing multiple Articles

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

Start by changing RedditApp to have a collection of articles:

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

```

1 class RedditApp {
2   articles: Article[];
3
4   constructor() {
5     this.articles = [
6       new Article('Angular 2', 'http://angular.io', 3),
7       new Article('Fullstack', 'http://fullstack.io', 2),
8       new Article('Angular Homepage', 'http://angular.io', 1),
9     ];
10  }
11
12  addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
13    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
```

Notice that our RedditApp has the line:

¹⁸<http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model>

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

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

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

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

```
1 constructor() {
2     this.articles = [
3         new Article('Angular 2', 'http://angular.io', 3),
4         new Article('Fullstack', 'http://fullstack.io', 2),
5         new Article('Angular Homepage', 'http://angular.io', 1),
6     ];
7 }
```

Configuring the ArticleComponent with inputs

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

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

Previously we had our ArticleComponent class defined like this:

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

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

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

```

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

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

```

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

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

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

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

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

Notice that `inputs` is an `Array`. This is because you can specify that a component has many inputs. Here's what our full `reddit-article` component now looks like using `inputs`:

[code/first_app/angular2-reddit-completed/intermediates/app-04.ts](#)

```

1 @Component({
2   selector: 'reddit-article',
3   inputs: ['article'],
4   host: {
5     class: 'row'
6   },
7   template: `
8     <div class="four wide column center aligned votes">
9       <div class="ui statistic">
10         <div class="value">
11           {{ article.votes }}
12         </div>
```

```
13      <div class="label">
14          Points
15      </div>
16      </div>
17  </div>
18  <div class="twelve wide column">
19      <a class="ui large header" href="{{ article.link }}">
20          {{ article.title }}
21      </a>
22
23      <ul class="ui big horizontal list voters">
24          <li class="item">
25              <a href (click)="voteUp()">
26                  <i class="arrow up icon"></i>
27                  upvote
28              </a>
29          </li>
30          <li class="item">
31              <a href (click)="voteDown()">
32                  <i class="arrow down icon"></i>
33                  downvote
34              </a>
35          </li>
36      </ul>
37  </div>
38
39 })
40 class ArticleComponent {
41     article: Article;
42
43     voteUp(): boolean {
44         this.article.voteUp();
45         return false;
46     }
47
48     voteDown(): boolean {
49         this.article.voteDown();
50         return false;
51     }
52 }
```

Rendering a List of Articles

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

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

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

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

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

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



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

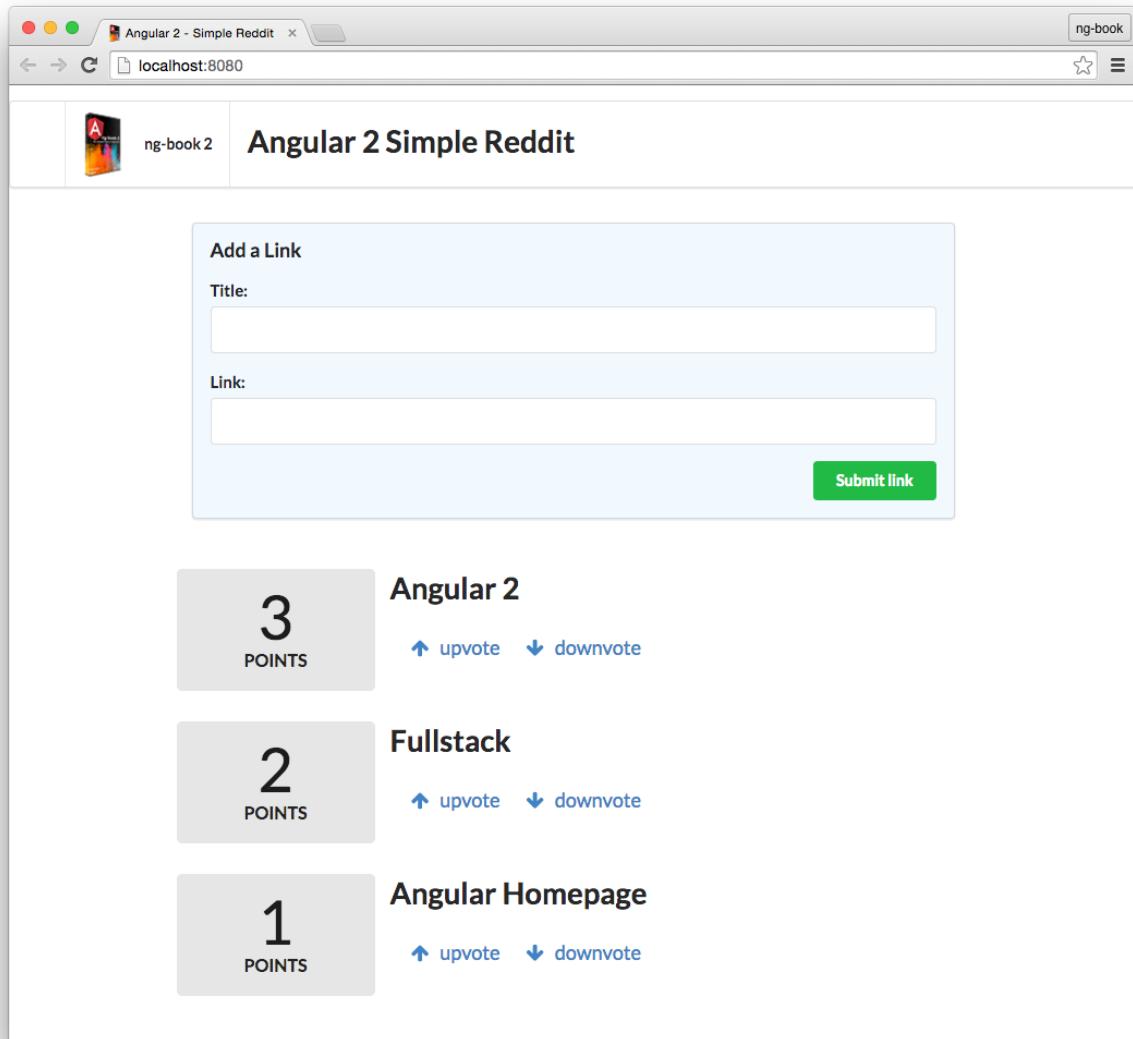
```
1  <reddit-article
2    *ngFor="#foobar of articles"
3    [article]="foobar">
4  </reddit-article>
```

So here we have three variables:

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

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

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



Multiple articles being rendered

Adding New Articles

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

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

```

1 addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
2   console.log(`Adding article title: ${title.value} and link: ${link.value}`);
3   this.articles.push(new Article(title.value, link.value, 0));
4   title.value = '';
5   link.value = '';
6 }
```

This will:

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



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

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

Finishing Touches

Displaying the Article Domain

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

Add this `domain` method to the `Article` class:

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

```

1 domain(): string {
2   try {
3     const link: string = this.link.split('//')[1];
4     return link.split('/')[0];
5   } catch (err) {
6     return null;
7   }
8 }
```

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

```

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

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

Re-sorting Based on Score

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

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

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

```

1 sortedArticles(): Article[] {
2   return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
3 }
```

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

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

```

1 <div class="ui grid posts">
2   <reddit-article
3     *ngFor="#article of sortedArticles()"
4       [article]="article"
5   </reddit-article>
6 </div>
```

Full Code Listing

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

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

```
1 import { bootstrap } from 'angular2/platform/browser';
2 import { Component } from 'angular2/core';
3
4 class Article {
5   title: string;
6   link: string;
7   votes: number;
8
9   constructor(title: string, link: string, votes?: number) {
10     this.title = title;
11     this.link = link;
12     this.votes = votes || 0;
13   }
14
15   domain(): string {
16     try {
17       const link: string = this.link.split('//')[1];
18       return link.split('/')[0];
19     } catch (err) {
20       return null;
21     }
22   }
23
24   voteUp(): void {
25     this.votes += 1;
26   }
27
28   voteDown(): void {
29     this.votes -= 1;
30   }
31 }
32
33 @Component({
34   selector: 'reddit-article',
35   inputs: ['article'],
36   host: {
37     class: 'row'
38   },
39   template: `
40     <div class="four wide column center aligned votes">
41       <div class="ui statistic">
```

```
42      <div class="value">
43          {{ article.votes }}
44      </div>
45      <div class="label">
46          Points
47      </div>
48  </div>
49  <div class="twelve wide column">
50      <a class="ui large header" href="{{ article.link }}">
51          {{ article.title }}
52      </a>
53      <div class="meta">({{ article.domain() }})</div>
54      <ul class="ui big horizontal list voters">
55          <li class="item">
56              <a href (click)="voteUp()">
57                  <i class="arrow up icon"></i>
58                  upvote
59              </a>
60          </li>
61          <li class="item">
62              <a href (click)="voteDown()">
63                  <i class="arrow down icon"></i>
64                  downvote
65              </a>
66          </li>
67      </ul>
68  </div>
69  <br>
70  </div>
71  })
72  class ArticleComponent {
73      article: Article;
74
75      voteUp(): boolean {
76          this.article.voteUp();
77          return false;
78      }
79
80      voteDown(): boolean {
81          this.article.voteDown();
82          return false;
83      }
```

```
84  }
85
86 @Component({
87   selector: 'reddit',
88   directives: [ArticleComponent],
89   template: `
90     <form class="ui large form segment">
91       <h3 class="ui header">Add a Link</h3>
92
93       <div class="field">
94         <label for="title">Title:</label>
95         <input name="title" #newtitle>
96       </div>
97       <div class="field">
98         <label for="link">Link:</label>
99         <input name="link" #newlink>
100      </div>
101
102      <button (click)="addArticle(newtitle, newlink)"
103          class="ui positive right floated button">
104        Submit link
105      </button>
106    </form>
107
108    <div class="ui grid posts">
109      <reddit-article
110        *ngFor="#article of sortedArticles()"
111        [article]="article">
112      </reddit-article>
113    </div>
114  `
115 })
116 class RedditApp {
117   articles: Article[];
118
119   constructor() {
120     this.articles = [
121       new Article('Angular 2', 'http://angular.io', 3),
122       new Article('Fullstack', 'http://fullstack.io', 2),
123       new Article('Angular Homepage', 'http://angular.io', 1),
124     ];
125   }
```

```
126
127     addArticle(title: HTMLInputElement, link: HTMLInputElement): void {
128         console.log(`Adding article title: ${title.value} and link: ${link.value}`);
129         this.articles.push(new Article(title.value, link.value, 0));
130         title.value = '';
131         link.value = '';
132     }
133
134     sortedArticles(): Article[] {
135         return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
136     }
137
138 }
139
140 bootstrap(RedditApp);
```

Wrapping Up

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

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

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

Getting Help

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

- Come join our (free!) community and chat with us on Gitter¹⁹
- Email us directly at us@fullstack.io²⁰

Onward!

¹⁹<https://gitter.im/ng-book/ng-book>

²⁰<mailto:us@fullstack.io>

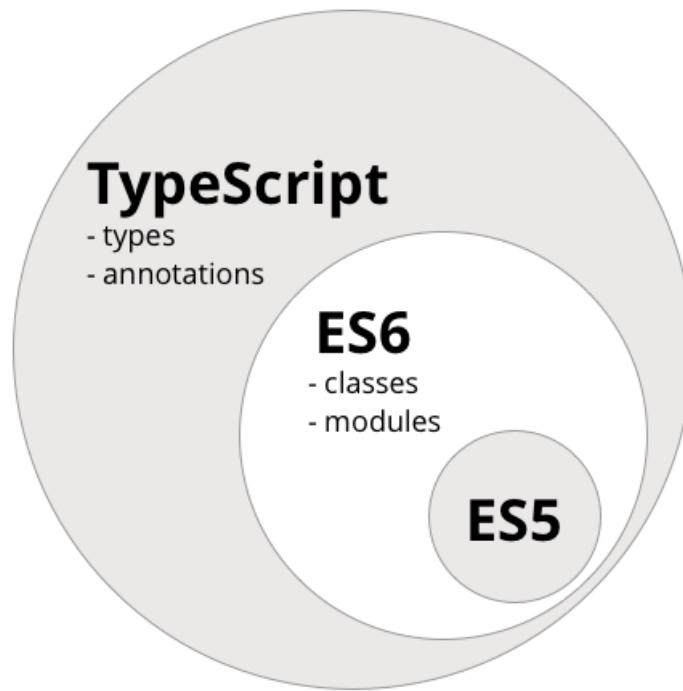
TypeScript

Angular 2 is built in TypeScript

Angular 2 is built in a Javascript-like language called [TypeScript²¹](#).

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.

²¹<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**²² by Google and **babel**²³ created by the JavaScript community. We're not going to be using either directly for this book, but they're both great projects that are worth knowing about.

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

```
npm install -g typescript
```

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

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

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

What do we get with TypeScript?

There are five big improvements that TypeScript bring over ES5:

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

Let's deal with these one at a time.

²²<https://github.com/google/traceur-compiler>

²³<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 name;`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Trying it out with a REPL

To play with the examples on this chapter, let's install a nice little utility called **TSUN**²⁴ (TypeScript Upgraded Node):

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

Now start `tsun`:

²⁴<https://github.com/HerringtonDarkholme/typescript-repl>

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

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

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

Built-in types

String

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

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

Number

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

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

Boolean

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

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

Array

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

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

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

Or similarly with a number:

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

Enums

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

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

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

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

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

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

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

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

Any

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

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

Void

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

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

Classes

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

A number of good practices have been adopted by the JavaScript community to compensate the lack of classes. A good summary of those good practices can be found in Mozilla Developer Network's [JavaScript Guide²⁵](#), and you can find a good overview on the [Introduction to Object-Oriented Javascript²⁶](#) page.

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

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

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

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

Properties

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

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

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

²⁵<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>

²⁶https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

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

Methods

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



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

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

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

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

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



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

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

```
1 // declare a variable of type Person
2 var p: Person;
3
4 // instantiate a new Person instance
5 p = new Person();
6
7 // give it a first_name
8 p.first_name = 'Felipe';
9
10 // call the greet method
11 p.greet();
```



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

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

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

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

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

Constructors

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

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



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

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

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

Is the same as:

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



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

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

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

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

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

It makes our previous example a little easier to write:

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

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

Inheritance

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



If you want to have a deeper understanding of how inheritance used to work in ES5, take a look at the Mozilla Developer Network article about it: [Inheritance and the prototype chain²⁷](#).

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:

²⁷https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

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

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



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

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

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

Running this should show:

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

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

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

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

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

Utilities

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

- fat arrow function syntax
- template strings

Fat Arrow Functions

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

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

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

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

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

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

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

Or as a statement:

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

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

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

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

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

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

Template Strings

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

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

Variables in strings

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

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

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

Multiline strings

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

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

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

Wrapping up

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

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

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

Let's get back to Angular!

How Angular Works

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



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

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

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

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

Application

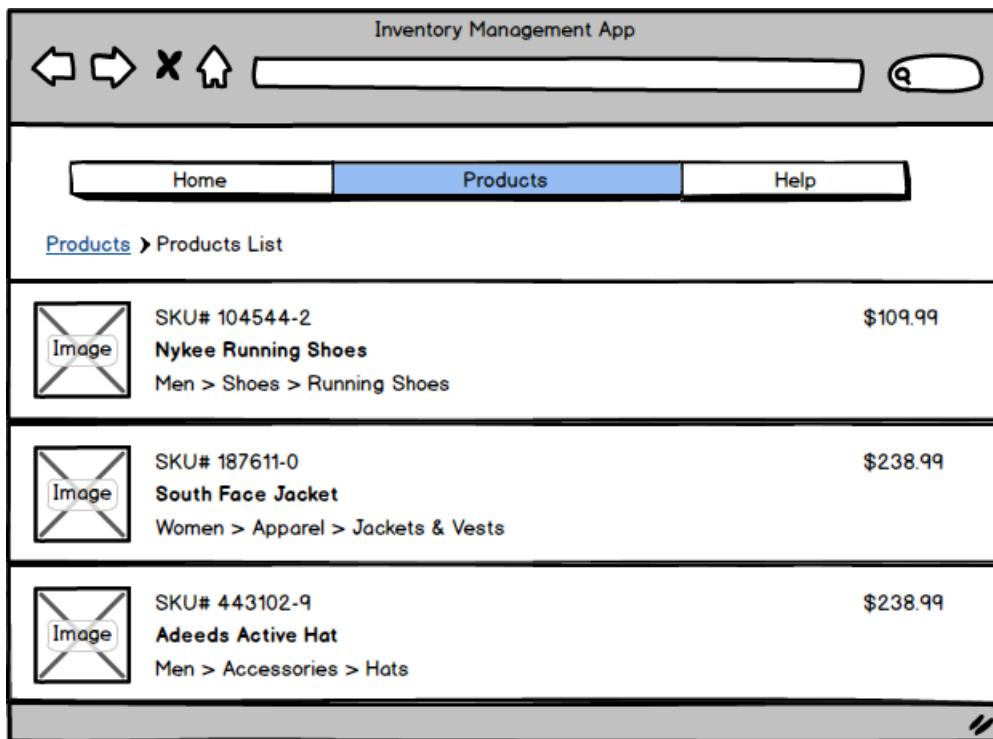
An Angular 2 Application is nothing more than a tree of Components.

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

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

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

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



Inventory Management App

Given this mockup, to write this application the first thing we want to do is split it into components. In this example, we could group the page into three high level components

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

The Navigation Component

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



Navigation Component

The Breadcrumbs Component

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

[Products](#) › Products List

Breadcrumbs Component

The Product List Component

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

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

Product List Component

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

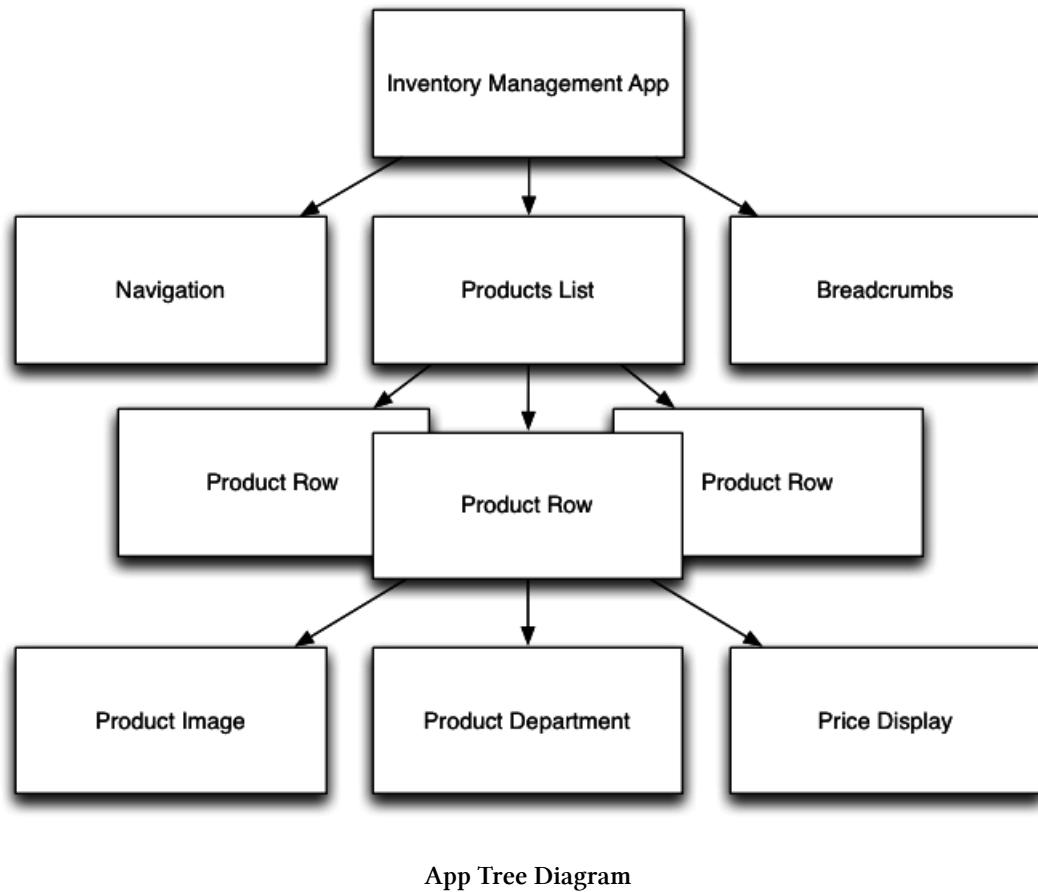
	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
---	--	----------

Product Row Component

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

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

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



App Tree Diagram

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

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

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

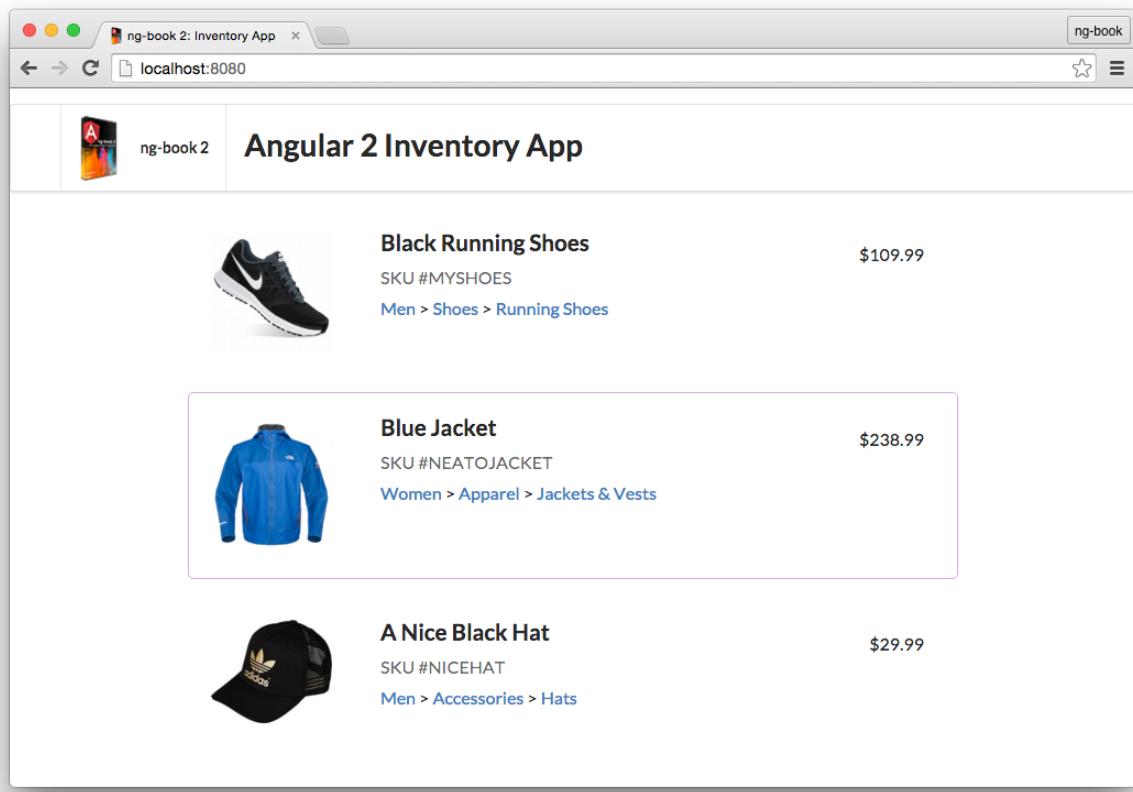
And the Product Row uses three components itself: one for the image, the department, and the price.

Let's work together to build this application.



You can find the full code listing for this chapter in the downloads under `how.angular_-works/inventory_app`.

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



Completed Inventory App

Product Model

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

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

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

code/how_angular_works/inventory_app/app.ts

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

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

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

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



If you're already familiar with JavaScript, you can quickly catch up on some of the differences, including the `public` constructor shorthand, [here at learnxinyminutes²⁸](#)

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

Components

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



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

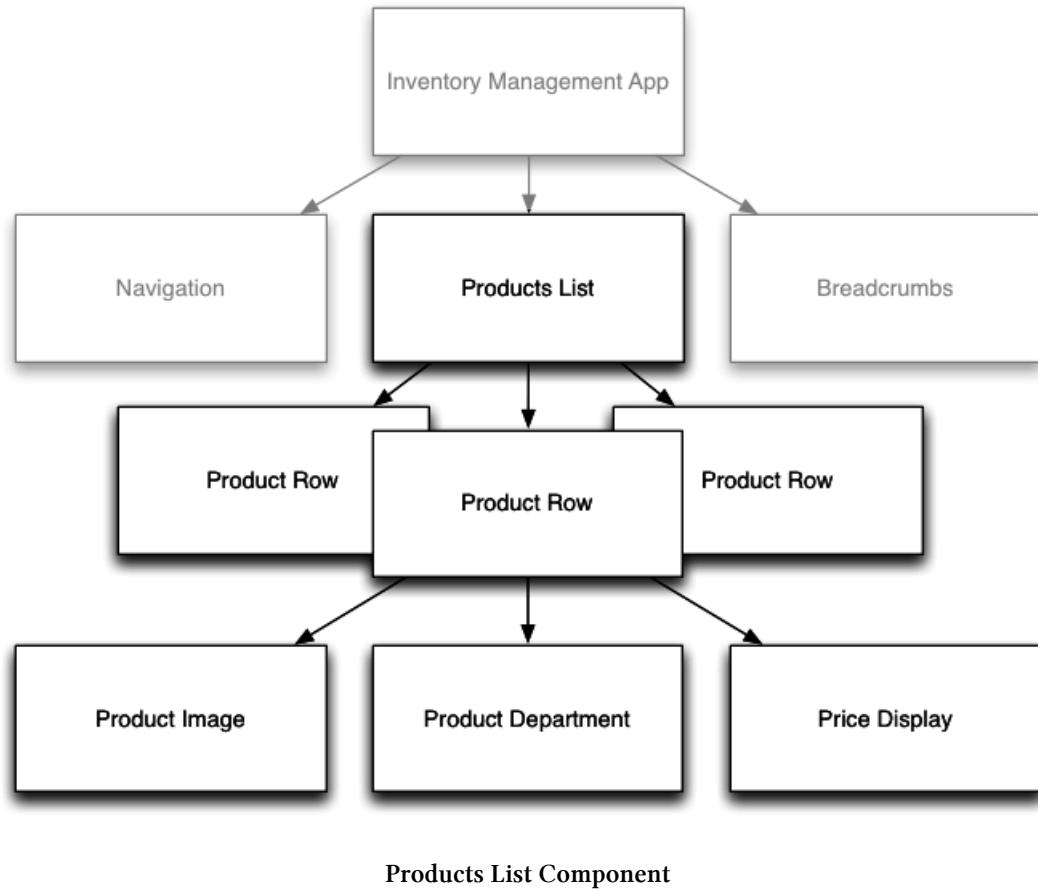
We'll be using Components a lot, so it's worth looking at them more closely.

Each components is composed of three parts:

²⁸<https://learnxinyminutes.com/docs/typescript/>

- 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** its child components:



Here's what a basic top-level InventoryApp:

```

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

```
10  }
11
12 bootstrap(InventoryApp);
```

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

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

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

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

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

Component Decorator

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

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

Component selector

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

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

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

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

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

Component template

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

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

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

Adding A Product

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

We can create a new Product like this:

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

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



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

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

But for now, a 5 argument constructor is fine.

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

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

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

or more concisely:

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

Notice that we did three things here:

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

Viewing the Product with Template Binding

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

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

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

So in this case, we have two bindings:

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

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

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

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

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

Adding More Products

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

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

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

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

code/how_angular_works/inventory_app/app.ts

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

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

Selecting a Product

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

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

code/how_angular_works/inventory_app/app.ts

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

Listing products using <products-list>

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

code/how_angular_works/inventory_app/app.ts

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

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

directives option

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

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

Inputs and Outputs

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

```

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

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

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

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

`[squareBrackets] pass inputs`

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

In our code where we show:

```

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

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

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

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

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

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



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

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

(parens) handle outputs

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

In our code where we show:

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

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

That is:

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

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

Full InventoryApp Listing

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

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

```
161 /**
162  * @InventoryApp: the top-level component for our application
163 */
164 @Component({
165   selector: 'inventory-app',
166   directives: [ProductsList],
167   template: `
168   <div class="inventory-app">
169     <products-list
170       [productList]="products"
171       (onProductSelected)="productWasSelected($event)">
172     </products-list>
173   </div>
174 `
```

```
175  })
176  class InventoryApp {
177    products: Product[];
178
179    constructor() {
180      this.products = [
181        new Product(
182          'MYSHOES', 'Black Running Shoes',
183          '/resources/images/products/black-shoes.jpg',
184          ['Men', 'Shoes', 'Running Shoes'],
185          109.99),
186        new Product(
187          'NEATOJACKET', 'Blue Jacket',
188          '/resources/images/products/blue-jacket.jpg',
189          ['Women', 'Apparel', 'Jackets & Vests'],
190          238.99),
191        new Product(
192          'NICEHAT', 'A Nice Black Hat',
193          '/resources/images/products/black-hat.jpg',
194          ['Men', 'Accessories', 'Hats'],
195          29.99)
196      ];
197    }
198
199    productWasSelected(product: Product): void {
200      console.log('Product clicked: ', product);
201    }
202  }
203
204 bootstrap(InventoryApp);
```

The ProductsList Component

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

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

Let's write the `ProductsList` Component in three steps:

- Configuring the `ProductsList` `@Component` options

- Writing the ProductsList controller class
- Writing the ProductsList view template

Configuring the ProductsList @Component Options

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

code/how_angular_works/inventory_app/app.ts

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

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

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

There are two new options though: `inputs` and `outputs`.

Component inputs

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

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

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

The `name` and `age` inputs map to the `name` and `age` properties on instances of the `MyComponent` class. If we want to use `MyComponent` from another template, we write something like: `<my-component name="myName" age="myAge"></my-component>`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Passing products through via the inputs

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

code/how_angular_works/inventory_app/app.ts

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

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

Component outputs

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

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

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

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

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

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

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

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

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

Emitting Custom Events

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

1. Specify outputs in the `@Component` configuration
2. Attach an `EventEmitter` to the `output` property
3. Emit an event from the `EventEmitter`, at the right time



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

An `EventEmitter` is simply an object that helps you implement the [Observer Pattern²⁹](#). That is, it's an object that can maintain a list of subscribers and publish events to them. That's it.

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

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

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

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

²⁹https://en.wikipedia.org/wiki/Observer_pattern

```

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

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

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

```

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

Again, notice that:

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

Writing the ProductsList Controller Class

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

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

Here's how we define those in code:

code/how_angular_works/inventory_app/app.ts

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

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

onProductSelected is our output.

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

Writing the ProductsList View Template

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

code/how_angular_works/inventory_app/app.ts

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

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

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

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



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

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

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

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

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

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

clicked`code/how_angular_works/inventory_app/app.ts`

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

This function does two things:

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

isSelected`code/how_angular_works/inventory_app/app.ts`

```
152  isSelected(product: Product): boolean {
153      if (!product || !this.currentProduct) {
154          return false;
155      }
156      return product.sku === this.currentProduct.sku;
157  }
```

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

The Full ProductsList Component

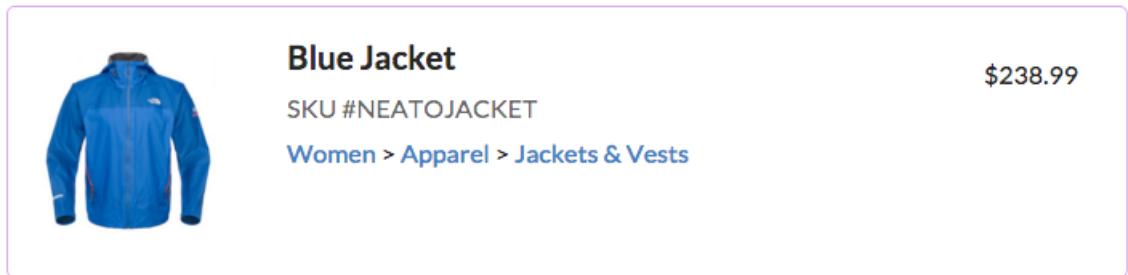
Here's the full code listing so we can see everything in context:

code/how_angular_works/inventory_app/app.ts

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

```
146
147     clicked(product: Product): void {
148         this.currentProduct = product;
149         this.onProductSelected.emit(product);
150     }
151
152     isSelected(product: Product): boolean {
153         if (!product || !this.currentProduct) {
154             return false;
155         }
156         return product.sku === this.currentProduct.sku;
157     }
158
159 }
```

The ProductRow Component

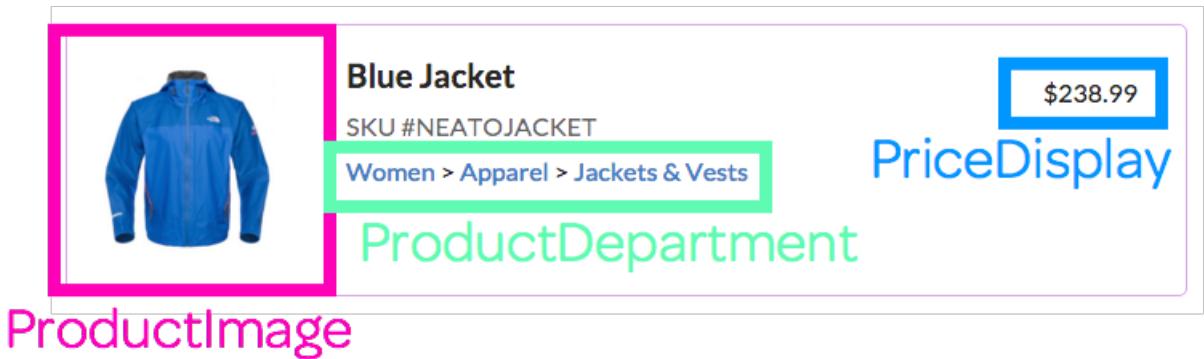


A Selected Product Row Component

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

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

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



ProductRow's Sub-components

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

ProductRow Component Configuration

`code/how-angular_works/inventory_app/app.ts`

```

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

```

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

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

The third option `host` is a new one. The `host` option lets us set attributes on the host element. In this case, we're using the [Semantic UI item class³⁰](#). Here when we say `host: {'class': 'item'}` we're saying that we want to attach the CSS class "item" to the host element.



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

³⁰<http://semantic-ui.com/views/item.html>

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

ProductRow Component Definition Class

The ProductRow Component definition class is straightforward:

code/how-angular-works/inventory_app/app.ts

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

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

ProductRow template

Now let's take a look at the template:

code/how-angular-works/inventory_app/app.ts

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

Our template doesn't have anything conceptually new.

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

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

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

ProductRow Full Listing

Here's the `ProductRow` component all together:

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

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

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

The ProductImage Component

code/how_angular_works/inventory_app/app.ts

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

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

We could have written this tag this way:

```
1 <!-- wrong, don't do it this way -->
2 \${{ price }}</div>
73   `
74 })
75 class PriceDisplay {
76   price: number;
77 }
```

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

The ProductDepartment Component

code/how_angular_works/inventory_app/app.ts

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

```
61     product: Product;  
62 }
```

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

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

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

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

```
1 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 `''`.

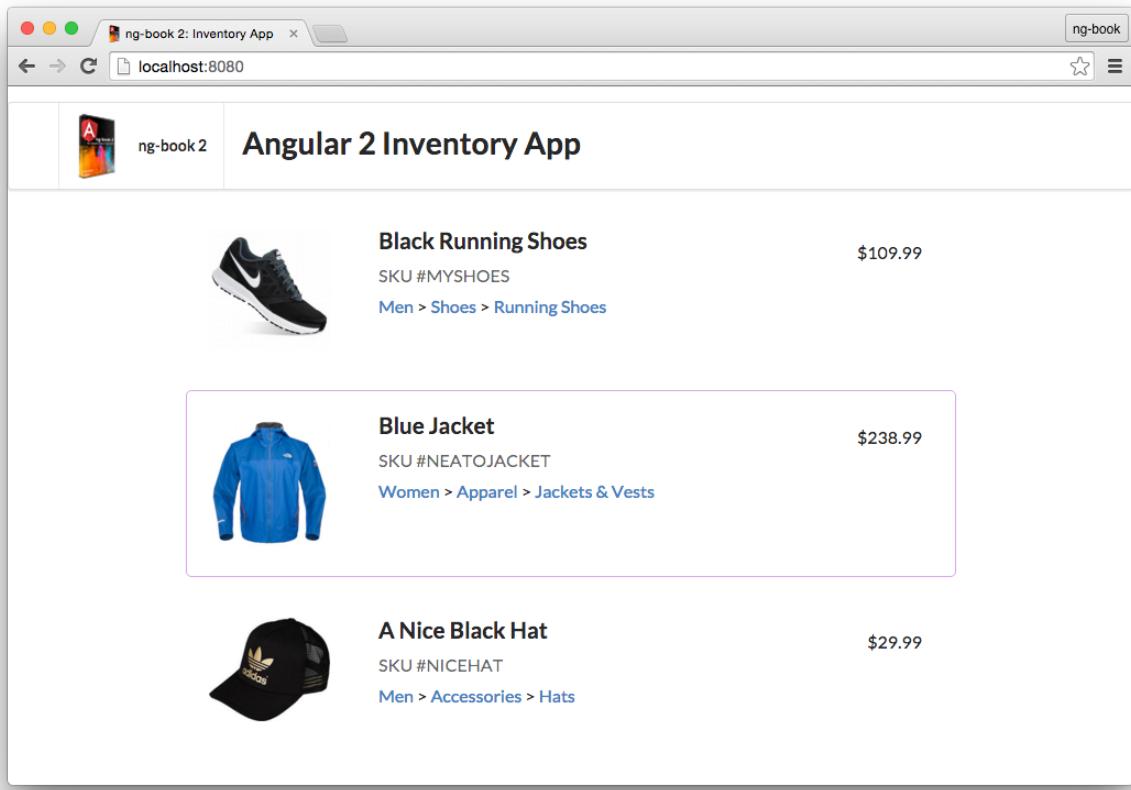


This format: `test ? valueIfTrue : valueIfFalse` is called the *ternary operator*.

The Completed Project

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

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



Completed Inventory App



You can run the code example in `how_angular_works/inventory_app`. See the README there.

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

A Word on Data Architecture

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Built-in Components

Introduction

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



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

NgIf

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

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

Some examples are:

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



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

NgSwitch

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

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

```

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

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

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

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

```

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

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

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

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

Once we have the result then we can:

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

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

```

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

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

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

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

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

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

code/built_in_components/ng_switch/app.ts

```
1 @Component({
2   selector: 'switch-sample-app',
3   template: `
4     <h4 class="ui horizontal divider header">
5       Current choice is {{ choice }}
6     </h4>
7
8     <div class="ui raised segment">
9       <ul [ngSwitch]="choice">
10      <li *ngSwitchWhen="1">First choice</li>
11      <li *ngSwitchWhen="2">Second choice</li>
12      <li *ngSwitchWhen="3">Third choice</li>
13      <li *ngSwitchWhen="4">Fourth choice</li>
14      <li *ngSwitchWhen="2">Second choice, again</li>
15      <li *ngSwitchDefault>Default choice</li>
16    </ul>
17  </div>
18
19  <div style="margin-top: 20px;">
20    <button class="ui primary button" (click)="nextChoice()">
21      Next choice
22    </button>
23  </div>
24  `,
25})
```

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

NgStyle

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

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

code/built_in_components/ng_style/app.ts

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

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

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

code/built_in_components/ng_style/app.ts

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



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

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

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

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

In our example, we are defining two input boxes:

code/built_in_components/ng_style/app.ts

```
1 <div class="ui input">
2   <input type="text" name="color" value="{{color}}" #colorinput>
3 </div>
4
5 <div class="ui input">
6   <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
7 </div>
```

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

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

code/built_in_components/ng_style/app.ts

```
1 <div>
2   <span [ngStyle]="{color: 'red'}" [style.fontSize.px]="fontSize">
3     red text
4   </span>
5 </div>
```

It's important to note that we have to specify units where appropriate. For instance, it isn't valid CSS to set a font-size of 12 - we have to specify a unit such as 12px or 1.2em. Angular provides a handy syntax for specifying units: here we used the notation [style.fontSize.px].

The .px suffix indicates that we're setting the font-size property value in pixels. You could easily replace that by [style.fontSize.em] to express the font size in ems or even in percentage using [style.fontSize.%].

The other two elements use the #colorinput to set the text and background colors:

code/built_in_components/ng_style/app.ts

```
1 <h4 class="ui horizontal divider header">
2   ngStyle with object property from variable
3 </h4>
4
5 <div>
6   <span [ngStyle]="{color: colorinput.value}">
7     {{ colorinput.value }} text
8   </span>
9 </div>
10
```

```
11  <h4 class="ui horizontal divider header">
12    style from variable
13  </h4>
14
15  <div [style.background-color]="colorinput.value"
16    style="color: white;">
17    {{ colorinput.value }} background
18  </div>
```

This way, when we click the **Apply settings** button, we call a method that sets the new values:

code/built_in_components/ng_style/app.ts

```
1 apply(color, fontSize) {
2   this.color = color;
3   this.fontSize = fontSize;
4 }
```

And with that, both the color and the font size will be applied to the elements using the `NgStyle` directive.

NgClass

The `NgClass` directive, represented by a `ngClass` attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.



If you're coming from Angular 1, the `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

The first way to use this directive is by passing in an object literal. The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

Let's assume we have a CSS class called `bordered` that adds a dashed black border to an element:

code/built_in_components/class/styles.css

```
1 .bordered {  
2   border: 1px dashed black;  
3   background-color: #eee;  
4 }
```

Let's add two `div` elements: one always having the `bordered` class (and therefore always having the border) and another one never having it:

code/built_in_components/ng_class/app.ts

```
1 <div [ngClass]="{bordered: false}">This is never bordered</div>  
2 <div [ngClass]="{bordered: true}">This is always bordered</div>
```

As expected, this is how those two `divs` would be rendered:

This is never bordered

This is always bordered

Simple class directive usage

Of course, it's a lot more useful to use the `NgClass` directive to make class assignments dynamic.

To make it dynamic we add a variable as the value for the object value, like this:

code/built_in_components/ng_class/app.ts

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

Alternatively, we can define the object in our component:

code/built_in_components/ng_class/app.ts

```
1 toggleBorder() {  
2     this.isBordered = !this.isBordered;  
3     this.classesObj = {  
4         bordered: this.isBordered  
5     };  
6 }
```

And use the object directly:

code/built_in_components/ng_class/app.ts

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



Again, be careful when you have class names that contains dashes, like bordered-box. JavaScript objects don't allow literal keys to have dashes. If you need to use them, you must make the key a string like this:

```
1 <div [ng-class]="{{ 'bordered-box': false }}">...</div>
```

We can also use a list of class names to specify which class names should be added to the element. For that, we can either pass in an array literal:

code/built_in_components/ng_class/app.ts

```
1 <div class="base" [ngClass]="['blue', 'round']">  
2     This will always have a blue background and  
3     round corners  
4 </div>
```

Or declare an array variable in our component:

```
1 this.classList = ['blue', 'round'];
```

And passing it in:

code/built_in_components/ng_class/app.ts

```

1 <div class="base" [ngClass]="classList">
2   This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
3   and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
4 </div>
```

In this last example, the `[class]` assignment works alongside existing values assigned by the HTML `class` attribute.

The resulting classes added to the element will always be the set of the classes provided by usual `class` HTML attribute and the result of the evaluation of the `[class]` directive.

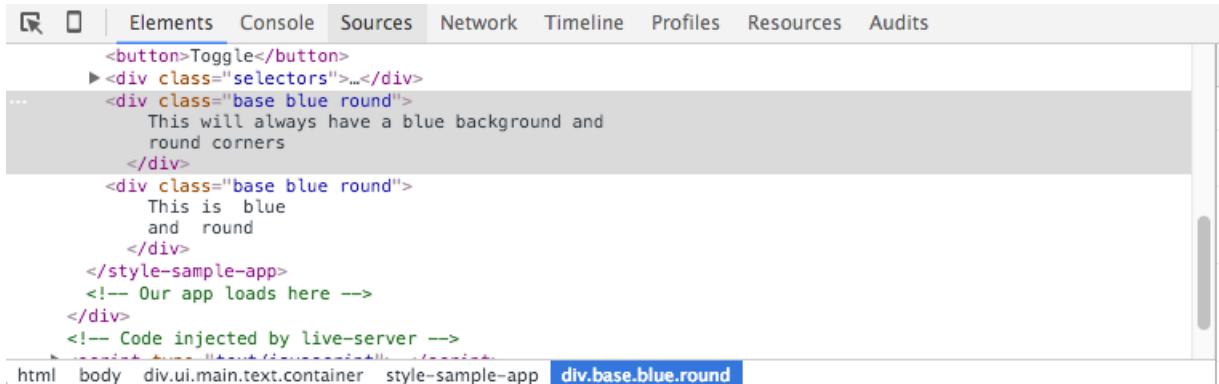
In this example:

code/built_in_components/ng_class/app.ts

```

1 <div class="base" [ngClass]="['blue', 'round']">
2   This will always have a blue background and
3   round corners
4 </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), each time passing it a different value from an array.



This directive is the successor of ng1's ng-repeat.

The syntax is *ngFor="#item of items".

- The #item is the template variable that's receiving each element of the items array;
- The items is the collection of items from your controller.

To illustrate, we can take a look at the code example. We declare an array of cities on our component controller:

```
1 this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

And then, in our template we can have the following HTML snippet:

code/built_in_components/ng_for/app.ts

```
1 <h4 class="ui horizontal divider header">
2   Simple list of strings
3 </h4>
4
5 <div class="ui list" *ngFor="#c of cities">
6   <div class="item">{{ c }}</div>
7 </div>
```

And it will render each city inside the div as you would expect:

Miami

Sao Paulo

New York

Simple list of strings

Result of the ng_for directive usage

We can also iterate through an array of objects like these:

code/built_in_components/ng_for/app.ts

```
1  this.people = [
2      { name: 'Anderson', age: 35, city: 'Sao Paulo' },
3      { name: 'John', age: 12, city: 'Miami' },
4      { name: 'Peter', age: 22, city: 'New York' }
5 ];
```

And then render a table based on each row of data:

code/built_in_components/ng_for/app.ts

```
1 <h4 class="ui horizontal divider header">
2     List of objects
3 </h4>
4
5 <table class="ui celled table">
6     <thead>
7         <tr>
8             <th>Name</th>
9             <th>Age</th>
10            <th>City</th>
11        </tr>
12    </thead>
13    <tr *ngFor="#p of people">
14        <td>{{ p.name }}</td>
15        <td>{{ p.age }}</td>
16        <td>{{ p.city }}</td>
17    </tr>
18 </table>
```

Getting the following result:

List of objects

Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

Rendering array of objects

We can also work with nested arrays. If we wanted to have the same table as above, broken down by city, we could easily declare a new array of objects:

code/built_in_components/ng_for/app.ts

```

1  this.peopleByCity = [
2      {
3          city: 'Miami',
4          people: [
5              { name: 'John', age: 12 },
6              { name: 'Angel', age: 22 }
7          ]
8      },
9      {
10         city: 'Sao Paulo',
11         people: [
12             { name: 'Anderson', age: 35 },
13             { name: 'Felipe', age: 36 }
14         ]
15     }
16 ];
17 }
```

And then we could use NgFor to render one h2 for each city:

code/built_in_components/ng_for/app.ts

```

1 <div *ngFor="#item of peopleByCity">
2     <h2 class="ui header">{{ item.city }}</h2>
```

And use a nested directive to iterate through the people for a given city:

code/built_in_components/ng_for/app.ts

```
1 <table class="ui celled table">
2   <thead>
3     <tr>
4       <th>Name</th>
5       <th>Age</th>
6     </tr>
7   </thead>
8   <tr *ngFor="#p of item.people">
9     <td>{{ p.name }}</td>
10    <td>{{ p.age }}</td>
11  </tr>
12 </table>
```

Resulting in the following template code:

code/built_in_components/ng_for/app.ts

```
1 <h4 class="ui horizontal divider header">
2   Nested data
3 </h4>
4
5 <div *ngFor="#item of peopleByCity">
6   <h2 class="ui header">{{ item.city }}</h2>
7
8   <table class="ui celled table">
9     <thead>
10       <tr>
11         <th>Name</th>
12         <th>Age</th>
13       </tr>
14     </thead>
15     <tr *ngFor="#p of item.people">
16       <td>{{ p.name }}</td>
17       <td>{{ p.age }}</td>
18     </tr>
19   </table>
20 </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 `#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 `#num = index` snippet like below:

code/built_in_components/ng_for/app.ts

```
1 <div class="ui list" *ngFor="#c of cities; #num = index">
2   <div class="item">{{ num+1 }} - {{ c }}</div>
3 </div>
```

It will add the position of the city before the name, like this:

List with index

- 1 - Miami
- 2 - Sao Paulo
- 3 - New York

Using an index

NgNonBindable

We use `ngNonBindable` when we want tell Angular **not** to compile or bind a particular section of our page.

Let's say we want to render the literal text `{{ content }}` in our template. Normally that text will be *bound* to the value of the `content` variable because we're using the `{{ }}` template syntax.

So how can we render the exact text `{{ content }}`? We use the `ngNonBindable` directive.

Let's say we want to have a `div` that renders the contents of that `content` variable and right after we want to point that out by outputting *<- this is what {{ content }} rendered* next to the actual value of the variable.

To do that, here's the template we'd have to use:

code/built_in_components/ng_non_bindable/app.ts

```
1 <div>
2   <span class="bordered">{{ content }}</span>
3   <span class="pre" ngNonBindable>
4     &larr; This is what {{ content }} rendered
5   </span>
6 </div>
```

And with that `ngNonBindable` attribute, ng2 will not compile within that second `span`'s context, leaving it intact:

Some text ← This is what {{ content }} rendered

Result of using `ngNonBindable`

Conclusion

Angular 2 has only a few core directives, but we can combine these simple pieces to create dynamic apps.

Forms in Angular 2

Forms are Crucial, Forms are Complex

Forms are probably the most crucial aspect of your web application. While we often get events from clicking on links or moving the mouse, it's through *forms* where we get the majority of our rich data input from users.

On the surface, forms seem straightforward: you make an `input` tag, the user fills it out, and hits submit. How hard could it be?

It turns out, forms can end up being really complex. Here's a few reasons why:

- Form inputs are meant to modify data, both on the page and the server
- Changes often need to be reflected elsewhere on the page
- Users have a lot of leeway in what they enter, so you need to validate values
- The UI needs to clearly state expectations and errors, if any
- Dependent fields can have complex logic
- We want to be able to test our forms, without relying on DOM selectors

Thankfully, Angular 2 has tools to help with all of these things.

- **Controls** 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.

Controls and Control Groups

The two fundamental objects in ng2 forms are `Control` and `ControlGroup`.

Control

A `Control` represents a single input field - it is the smallest unit of an Angular form.

`Controls` encapsulate the field's value, and states such as if it is valid, dirty (changed), or has errors.

For instance, here's how we might use a `Control` in TypeScript:

```

1 // create a new Control with the value "Nate"
2 let nameControl = new Control("Nate");
3
4 let name = nameControl.value; // -> Nate
5
6 // now we can query this control for certain values:
7 nameControl.errors // -> StringMap<string, any> of errors
8 nameControl.dirty // -> false
9 nameControl.valid // -> true
10 // etc.

```

To build up forms we create Controls (and groups of Controls) and then attach metadata and logic to them.

Like many things in Angular, we have a class (Control, in this case) that we attach to the DOM with an attribute (ngControl, in this case). For instance, we might have the following in our form:

```

1 <!-- part of some bigger form -->
2 <input type="text" ngControl="name" />

```

This will create a new Control object within the context of our form. We'll talk more about how that works below.

ControlGroup

Most forms have more than one field, so we need a way to manage multiple Controls. If we wanted to check the validity of our form, it's cumbersome to iterate over an array of Controls and check each Control for validity. ControlGroups solve this issue by providing a wrapper interface around a collection of Controls.

Here's how you create a ControlGroup:

```

1 let personInfo = new ControlGroup({
2   firstName: new Control("Nate"),
3   lastName: new Control("Murray"),
4   zip: new Control("90210")
5 })

```

ControlGroup and Control have a common ancestor ([AbstractControl³¹](#)). That means we can check the status or value of personInfo just as easily as a single Control:

³¹<https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/model.ts>

```
1 personInfo.value; // -> {  
2   //   firstName: "Nate",  
3   //   lastName: "Murray",  
4   //   zip: "90210"  
5 }  
6  
7 // now we can query this control group for certain values, which have sensible  
8 // values depending on the children Control's values:  
9 personInfo.errors // -> StringMap<string, any> of errors  
10 personInfo.dirty // -> false  
11 personInfo.valid // -> true  
12 // etc.
```

Notice that when we tried to get the value from the `ControlGroup` 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 `Control` individually.

Our First Form

There are lots of moving pieces to create a form, and several important ones we haven't touched on. Let's jump in to a full example and I'll explain each piece as we go along.



You can find the full code listing for this section in the code download under `forms/`

Here's a screenshot of the very first form we're going to build:

The screenshot shows a simple web form titled "Demo Form: Sku". At the top, it says "SKU". Below that is a text input field containing the placeholder "SKU". At the bottom is a "Submit" button.

Demo Form with Sku: Simple Version

In our imaginary application we're creating an e-commerce-type site where we're listing products for sale. In this app we need to store the product's SKU, so let's create a simple form that takes the SKU as the only input field.



SKU is an abbreviation for “stockkeeping unit”. It’s a term for a unique id for a product that is going to be tracked in inventory. When we talk about a SKU, we’re talking about a human-readable item ID.

Our form is super simple: we have a single input for `sku` (with a label) and a submit button.

Let’s turn this form into a Component. If you recall, there are three parts to defining a component:

- Configure the `@Component()` annotation
- Create the template
- Implement custom functionality in the component definition class

Let’s take these in turn:

Simple SKU Form: `@Component` Annotation

`code/forms/app/forms/demo_form_sku.ts`

```

1 import { Component } from 'angular2/core';
2 import { FORM_DIRECTIVES } from 'angular2/common';
3
4 @Component({
5   selector: 'demo-form-sku',
6   directives: [FORM_DIRECTIVES],

```

Here we define a `selector` of `demo-form-sku`. If you recall, `selector` tells Angular what elements this component will bind to. In this case we can use this component by having a `demo-form-sku` tag like so:

```

1 <demo-form-sku></demo-form-sku>

```

Next we specify `directives` that we’re going to use in this view. Notice that there is something funny about this injection: we’re injecting something called `FORM_DIRECTIVES`. `FORM_DIRECTIVES` is a constant that Angular provides for us as a shorthand to several directives that are all useful in a form. `FORM_DIRECTIVES` includes:

- `ngControl`
- `ngControlGroup`
- `ngForm`
- `ngModel`

... and several more³². We haven’t talked about how to use these directives or what they do, but we will shortly. For now, just know that by injecting `FORM_DIRECTIVES`, that means we can *use any of the directives in that list* in our view template.

³²<https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/directives.ts>

Simple SKU Form: template

Let's look at our template:

code/forms/app/ts/forms/demo_form_sku.ts

```

4  @Component({
5    selector: 'demo-form-sku',
6    directives: [FORM_DIRECTIVES],
7    template: `
8      <div class="ui raised segment">
9        <h2 class="ui header">Demo Form: Sku</h2>
10       <form #f="ngForm"
11         (ngSubmit)="onSubmit(f.value)"
12         class="ui form">
13
14         <div class="field">
15           <label for="skuInput">SKU</label>
16           <input type="text"
17             id="skuInput"
18             placeholder="SKU"
19             ngControl="sku">
20         </div>
21
22         <button type="submit" class="ui button">Submit</button>
23       </form>
24     </div>
25   `
26 })

```

form & NgForm

Now things get interesting: because we injected FORM_DIRECTIVES, that makes NgForm available to our view. Remember that whenever we make directives available to our view, they will get attached to any element that matches their selector.

NgForm does something handy but **non-obvious**: it includes the `form` tag in its selector (instead of requiring you to explicitly add `ngForm` as an attribute). What this means is that if you inject `FORM_DIRECTIVES`, `NgForm` will get *automatically* attached to any `<form>` tags you have in your view. This is really useful but potentially confusing because it happens behind the scenes.

There are two important pieces of functionality that `NgForm` gives us:

1. A ControlGroup named `ngForm`

2. A (`ngSubmit`) output

You can see that we use both of these in the `<form>` tag in our view:

`code/forms/app/ts/forms/demo_form_sku.ts`

```
10   <form #f="ngForm"
11     (ngSubmit)="onSubmit(f.value)"
```

First we have `#f="ngForm"`. If you recall the `#v=thing` syntax says that we want to create a local variable for this view (remember we've used this syntax before with `*ngFor="#product in products"`).

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 `ControlGroup`. That means we can use `f` as a `ControlGroup` 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 `ControlGroup` 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 `ControlGroup` that we specified above. And `.value` will return the key/value pairs of this `ControlGroup`

Put it all together and that line says "when I submit the form, call `onSubmit` on my component instance, passing the value of the form as the arguments".

input & NgControl

Our `input` tag has a few things we should touch on before we talk about `NgControl`:

code/forms/app/ts/forms/demo_form_sku.ts

```

10   <form #f="ngForm"
11     (ngSubmit)="onSubmit(f.value)"
12     class="ui form">
13
14   <div class="field">
15     <label for="skuInput">SKU</label>
16     <input type="text"
17       id="skuInput"
18       placeholder="SKU"
19       ngControl="sku">
20   </div>

```

- `class="ui form"` and `class="field"` - these two classes are totally optional. They come from the [CSS framework Semantic UI³³](#). 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 standard³⁴](#)
- 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 `NgControl` directive specifies a selector of `ngControl35`. This means we can attach it to our input tag by adding this sort of attribute: `ngControl="whatever"`. In this case, we say `ngControl="sku"`.



NgControl vs. ngControl: what's the difference? Generally, when we use PascalCase, like `NgControl`, we're specifying the `class` and referring to the object as it's defined in code. The lower case (CamelCase), as in `ngControl`, comes from the `selector` of the directive and it's only used in the DOM / template.

It's also worth pointing out that `NgControl` and `Control` are separate objects. `NgControl` is the *directive* that you use in your view, whereas `Control` is the object used for representing the data and validations.



It's worth pointing out that `ngControl` is concretely defined in the class `NgControlName` (not `NgControl`). You [can find the definition here³⁶](#)

³³<http://semantic-ui.com/>

³⁴<http://www.w3.org/TR/WCAG20-TECHS/H44.html>

³⁵https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/directives/ng_control_name.ts

³⁶https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/directives/ng_control_name.ts

NgControl creates a new Control that is automatically added to the parent ControlGroup (in this case, on the form) and then binds a DOM element to that new Control. That is, it sets up an association between the input tag in our view and the Control and the association is matched by a name, in this case "sku".



ngControl must be used as a child of NgForm (or NgFormModel, which we talk about below). Don't try to use ngControl without an NgForm parent or you'll have problems.

Simple SKU Form: Component Definition Class

Now let's look at our class definition:

code/forms/app/ts/forms/demo_form_sku.ts

```
27 export class DemoFormSku {  
28   onSubmit(value: string): void {  
29     console.log('you submitted value: ', value);  
30   }  
31 }
```

Here our class defines one function: onSubmit. This is the function that is called when the form is submitted. For now, we'll just console.log out the value that is passed in.

Try it out!

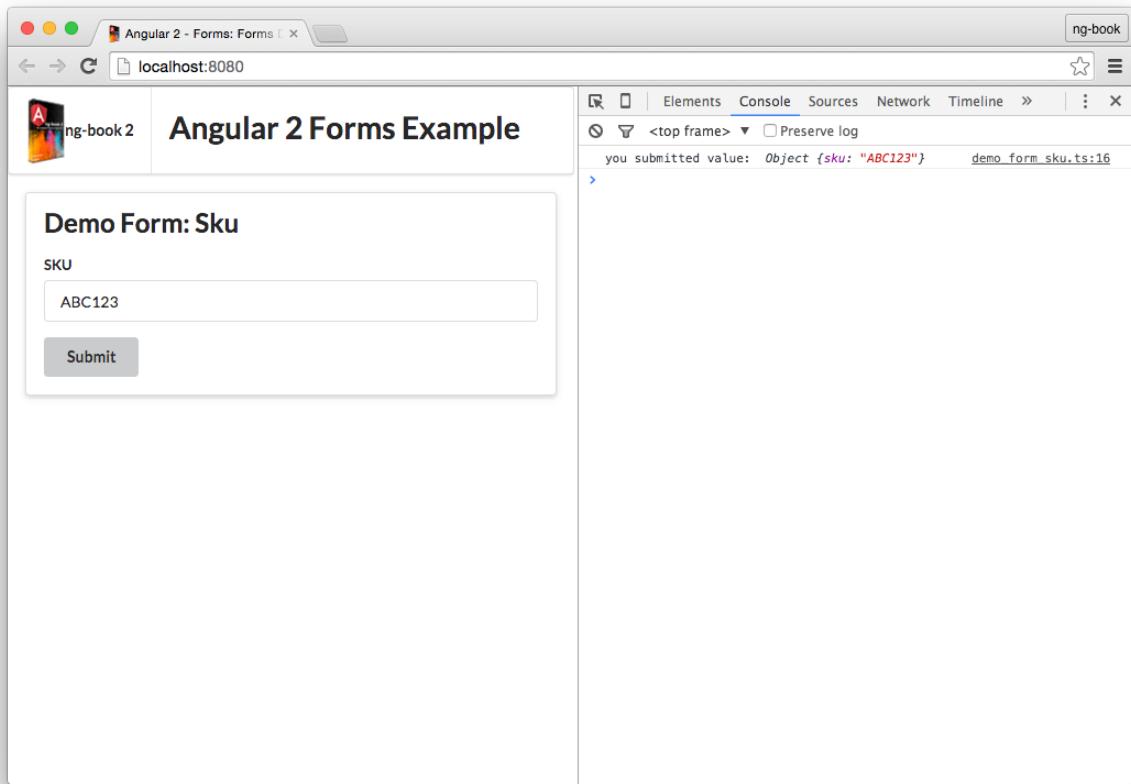
Putting it all together, here's what our code listing looks like:

code/forms/app/ts/forms/demo_form_sku.ts

```
1 import { Component } from 'angular2/core';  
2 import { FORM_DIRECTIVES } from 'angular2/common';  
3  
4 @Component({  
5   selector: 'demo-form-sku',  
6   directives: [FORM_DIRECTIVES],  
7   template: `  
8     <div class="ui raised segment">  
9       <h2 class="ui header">Demo Form: Sku</h2>  
10      <form #f="ngForm"  
11        (ngSubmit)="onSubmit(f.value)"  
12        class="ui form">
```

```
13      <div class="field">
14          <label for="skuInput">SKU</label>
15          <input type="text"
16              id="skuInput"
17              placeholder="SKU"
18              ngControl="sku">
19      </div>
20
21      <button type="submit" class="ui button">Submit</button>
22  </form>
23 </div>
24 <br>
25 })
26 })
27 export class DemoFormSku {
28     onSubmit(value: string): void {
29         console.log('you submitted value: ', value);
30     }
31 }
```

If we try this out in our browser, here's what it looks like:



Demo Form with Sku: Simple Version, Submitted

Using FormBuilder

Building our `Controls` and `ControlGroups` implicitly using `ngForm` and `ngControl` is convenient, but doesn't give us a lot of customization options. A more flexible and common way to configure forms is to use a `FormBuilder`.

`FormBuilder` is an aptly-named helper class that helps us build forms. As you recall, forms are made up of `Controls` and `ControlGroups` 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 `ControlGroup` on a `form` in the view

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](#) (forthcoming), so go there if you'd like to learn more about it in depth.

At a high level, Dependency Injection is a way to tell Angular what dependencies this component needs to function properly.

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```

31 export class DemoFormSkuBuilder {
32   myForm: ControlGroup;
33
34   constructor(fb: FormBuilder) {
35     this.myForm = fb.group({
36       'sku': ['ABC123']
37     });
38   }
39
40   onSubmit(value: string): void {
41     console.log('you submitted value: ', value);
42   }
43 }
```

During injection an instance of `FormBuilder` will be created and we assign it to the `fb` variable (in the constructor).

There are two main functions we'll use on `FormBuilder`:

- `control` - creates a new `Control`
- `group` - creates a new `ControlGroup`

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 `ControlGroup` and the `form` we had before.)

`myForm` is typed to be a `ControlGroup`. We create a `ControlGroup` by calling `fb.group()`. `.group` takes an object of key-value pairs that specify the `Controls` in this group.

In this case, we're setting up one control `sku`, and the value is `["ABC123"]` - this says that the default value of this control is `"ABC123"`. (You'll notice that is an array. That's because we'll be adding more configuration options there later.)

Now that we have `myForm` we need to use that in the view (i.e. we need to *bind* it to our `form` element).

Using myForm in the view

We want to change our `<form>` to use `myForm`. If you recall, in the last section we said that `ngForm` is applied for us automatically when we used `FORM_DIRECTIVES`. We also mentioned that `ngForm` creates its own `ControlGroup`. Well, in this case, we **don't** want to use an outside `ControlGroup`. 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 ControlGroup**: it's called `NgFormModel` and we use it like this:

`code/forms/app/ts/forms/demo_form_sku_with_builder.ts`

```
14 <form [ngFormModel]="myForm"
15   (ngSubmit)="onSubmit(myForm.value)"
```

Here we're telling Angular that we want to use `myForm` as the `ControlGroup` for this form.



Remember how earlier we said that when using `FORM_DIRECTIVES` that `NgForm` will be automatically applied to a `<form>` element? There is an exception: `NgForm` won't be applied to a `<form>` that has `ngFormModel`.

If you're curious, the selector for `NgForm` is:

```
1 form:not([ngNoForm]):not([ngFormModel]),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 `Control` to the `input` tag. Remember that `ngControl` creates a **new Control object**, and attaches it to the parent `ControlGroup`. But in this case, we used `FormBuilder` to create our own `Controls`.

When we want to bind an **existing Control** to an `input` we use `NgFormControl`:

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
20    <input type="text"
21        id="skuInput"
22        placeholder="SKU"
23        [ngFormControl]="myForm.controls['sku']">
```

Here we are instructing the `NgFormControl` directive to look at `myForm.controls` and use the existing `sku` Control for this input.

Try it out!

Here's what it looks like all together:

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
1 import { Component } from 'angular2/core';
2 import {
3   FORM_DIRECTIVES,
4   FormBuilder,
5   ControlGroup
6 } from 'angular2/common';
7
8 @Component({
9   selector: 'demo-form-sku-builder',
10  directives: [FORM_DIRECTIVES],
11  template: `
12    <div class="ui raised segment">
13      <h2 class="ui header">Demo Form: Sku with Builder</h2>
14      <form [ngFormModel]="myForm"
15        (ngSubmit)="onSubmit(myForm.value)"
16        class="ui form">
17
18        <div class="field">
19          <label for="skuInput">SKU</label>
20          <input type="text"
21              id="skuInput"
22              placeholder="SKU"
23              [ngFormControl]="myForm.controls['sku']">
24        </div>
25
26        <button type="submit" class="ui button">Submit</button>
```

```
27      </form>
28  </div>
29  `-
30  })
31 export class DemoFormSkuBuilder {
32   myForm: ControlGroup;
33
34   constructor(fb: FormBuilder) {
35     this.myForm = fb.group({
36       'sku': ['ABC123']
37     });
38   }
39
40   onSubmit(value: string): void {
41     console.log('you submitted value: ', value);
42   }
43 }
```

Remember:

To create a new ControlGroup and Controls implicitly use:

- ngForm and
- ngControl

But to bind to an existing ControlGroup and Controls use:

- ngFormModel and
- ngFormControl

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 Control will be considered invalid.

To use validators we need to do two things:

1. Assign a validator the Control object

2. Check the status of the validator in the view and take action accordingly

To assign a validator to a `Control` object we simply pass it as the second argument to our `Control` constructor:

```
1 let control = new Control('sku', Validators.required);
```

Or in our case, because we're using `FormBuilder` we will use the following syntax:

`code/forms/app/ts/forms/demo_form_with_validations_explicit.ts`

```
47 this.myForm = fb.group({  
48   'sku':  ['', Validators.required]  
49});
```

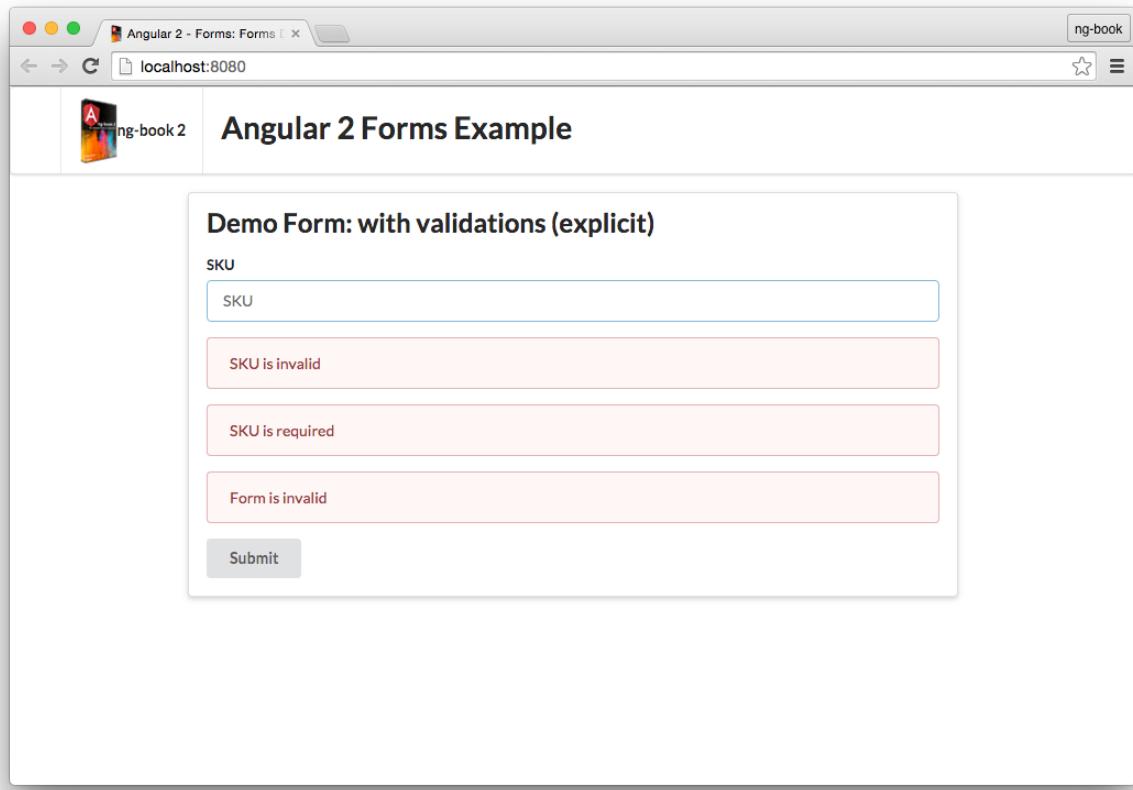
Now we need to use our validation in the view. There are two ways we can access the validation value in the view:

1. We can explicitly assign the `Control` `sku` to an instance variable of the class - which is more verbose, but gives us easy access to the `Control` in the view.
2. We can lookup the `Control` `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 Control 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 `Controls` in your view is to set each `Control` up as an instance variable in your component definition class. Here's how we could setup `sku` in our class:

`code/forms/app/ts/forms/demo_form_with_validations_explicit.ts`

```
42 export class DemoFormWithValidationsExplicit {
43   myForm: ControlGroup;
44   sku: AbstractControl;
45
46   constructor(fb: FormBuilder) {
47     this.myForm = fb.group({
48       'sku': ['', Validators.required]
49     });
50
51     this.sku = this.myForm.controls['sku'];
52   }
53
54   onSubmit(value: string): void {
```

```
55     console.log('you submitted value: ', value);
56 }
57 }
```

Notice that:

1. We setup `sku: AbstractControl` at the top of the class and
2. We assign `this.sku` after we've created `myForm` with the `FormBuilder`

This is great because it means we can reference `sku` anywhere in our component view. The downside is that by doing it this way, we'd have to setup an instance variable for **every field in our form**. For large forms, this can get pretty verbose.

Now that we have our `sku` being validated, I want to look at four different ways we can use it in our view:

1. Checking the validity of our whole form and displaying a message
2. Checking the validity of our individual field and displaying a message
3. Checking the validity of our individual field and coloring the field red if it's invalid
4. Checking the validity of our individual field on a particular requirement and displaying a message

Form message

We can check the validity of our whole form by looking at `myForm.valid`:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
34 <div *ngIf="!myForm.valid"
35   class="ui error message">Form is invalid</div>
```

Remember, `myForm` is a `ControlGroup` and a `ControlGroup` is valid if all of the children `Controls` are also valid.

Field message

We can also display a message for the specific field if that field's `Control` is invalid:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
28   <div *ngIf="!sku.valid"
29     class="ui error message">SKU is invalid</div>
```

Field coloring

I'm using the Semantic UI CSS Framework's CSS class `.error`, which means if I add the class `error` to the `<div class= "field">` it will show the input tag with a red border.

To do this, we can use the property syntax to set conditional classes:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
21   <div class="field"
22     [class.error]="!sku.valid && sku.touched">
```

Notice here that we have two conditions for setting the `.error` class: We're checking for `!sku.valid` and `sku.touched`. The idea here is that we only want to show the error state if the user has tried editing the form ("touched" it) and it's now invalid.

To try this out, enter some data into the `input` tag and then delete the contents of the field.

Specific validation

A form field can be invalid for many reasons. We often want to show a different message depending on the reason for a failed validation.

To look up a specific validation failure we use the `hasError` method:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
30   <div *ngIf="sku.hasError('required')"
31     class="ui error message">SKU is required</div>
```

Note that `hasError` is defined on both `Control` and `ControlGroup`. This means you can pass a second argument of path to lookup a specific field from `ControlGroup`. For example, we could have written the previous example as:

```
1   <div *ngIf="myForm.hasError('required', 'sku')"
2     class="error">SKU is required</div>
```

Putting it together

Here's the full code listing of our form with validations with the `Control` set as an instance variable:

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
1  /* tslint:disable:no-string-literal */
2  import { Component } from 'angular2/core';
3  import {
4    FORM_DIRECTIVES,
5    FormBuilder,
6    ControlGroup,
7    Validators,
8    AbstractControl
9  } from 'angular2/common';
10
11 @Component({
12   selector: 'demo-form-with-validations-explicit',
13   directives: [FORM_DIRECTIVES],
14   template: `
15     <div class="ui raised segment">
16       <h2 class="ui header">Demo Form: with validations (explicit)</h2>
17       <form [ngFormModel]="myForm"
18         (ngSubmit)="onSubmit(myForm.value)"
19         class="ui form">
20
21         <div class="field"
22           [class.error]="!sku.valid && sku.touched">
23           <label for="skuInput">SKU</label>
24           <input type="text"
25             id="skuInput"
26             placeholder="SKU"
27             [ngFormControl]="sku">
28             <div *ngIf="!sku.valid"
29               class="ui error message">SKU is invalid</div>
30             <div *ngIf="sku.hasError('required')"
31               class="ui error message">SKU is required</div>
32           </div>
33
34           <div *ngIf="!myForm.valid"
35             class="ui error message">Form is invalid</div>
36
37           <button type="submit" class="ui button">Submit</button>
38         </form>
39       </div>
40     `
```

41 })

```
42 export class DemoFormWithValidationsExplicit {
43   myForm: ControlGroup;
44   sku: AbstractControl;
45
46   constructor(fb: FormBuilder) {
47     this.myForm = fb.group({
48       'sku': ['', Validators.required]
49     });
50
51     this.sku = this.myForm.controls['sku'];
52   }
53
54   onSubmit(value: string): void {
55     console.log('you submitted value: ', value);
56   }
57 }
```

Explicitly setting the sku Control as an instance variable

As we mentioned in the last section, having to create an instance variable for every input tag in your form can get a bit verbose.

Can we get away without creating an instance variable for every Control? It turns out we can, though there are some trade-offs. It's useful to explore in any case, because we'll learn some new things about how to navigate forms.

First, let's take another look at the component definition class:

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```
41 export class DemoFormWithValidationsShorthand {
42   myForm: ControlGroup;
43
44   constructor(fb: FormBuilder) {
45     this.myForm = fb.group({
46       'sku': ['', Validators.required]
47     });
48   }
49
50   onSubmit(value: string): void {
51     console.log('you submitted value: ', value);
52   }
53 }
```

This example is similar but notice that we have removed the `sku: AbstractControl`.

Let's look at the three field-level validations that we did before and see what changes:

Declaring a local `sku` reference

Because we didn't expose the `sku Control` as an instance variable, we now need a way to get a reference to it. There are two ways we can get at it:

1. via `myForm.find`
2. via the `ngFormControl` directive

Field coloring via `myForm.find`

`ControlGroup` has a `.find` method which allows you to look up a child `Control` by path. Here's how we can find our `sku Control` and then check if it is valid / touched:

`code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts`

```
19   <div class="field"
20     [class.error]="!myForm.find('sku').valid && myForm.find('sku').touched">
```

This is a bit more verbose than before, but it's not too bad.

The `form` export from `NgFormControl`

There is another way we can get a reference to the `Control` and that is via the `ngForm` export of the `NgFormControl` directive. This is a new concept that we haven't covered so far:

Components can export a reference themselves so that you can use them in the view.

(We'll cover how to use `exportAs` in the Components chapter, but for now, just know that many of the built-in components do this already.)

In this case, `NgFormControl` exports itself as `ngForm`. You can use this export by using the `#reference` syntax. Here's what it looks like:

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```

22   <input type="text"
23     id="skuInput"
24     placeholder="SKU"
25     #sku="ngForm"
26     [ngFormControl]="myForm.controls['sku']">

```

What this does is make the `NgFormControl` directive itself available in the view as the variable `sku`. But note **this is the directive and not the Control**. To access the `sku` Control we must now call `sku.control`.



It's worth saying again - when we reference the `NgFormControl` directive with `#sku="ngForm"`, `sku` is now an instance of the directive and **not a Control**. To get a reference to the `Control` you need to call `sku.control`

Now that we have `sku` available to us, we can check the validity and errors like so:

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```

27   <div *ngIf="!sku.control.valid"
28     class="ui error message">SKU is invalid</div>
29   <div *ngIf="sku.control.hasError('required')"
30     class="ui error message">SKU is required</div>

```

Local reference to `sku` scope

When we create a local reference using the `#reference` syntax, it is only available to sibling and children elements, not parents.

For instance, we **can't** do this

```

1 // this won't work
2 <div class="field"
3   [class.error]="!sku.control.valid && sku.control.touched">

```

Why not? Because this `<div class="field"` is a parent of the `input` element that declares the reference.

Putting it together

Here's the full listing of our code showing the “shorthand” (non-instance-variable) version of our validated `Control` code:

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```
1 import { Component } from 'angular2/core';
2 import {
3   FORM_DIRECTIVES,
4   FormBuilder,
5   ControlGroup,
6   Validators
7 } from 'angular2/common';
8
9 @Component({
10   selector: 'demo-form-with-validations-shorthand',
11   directives: [FORM_DIRECTIVES],
12   template: `
13     <div class="ui raised segment">
14       <h2 class="ui header">Demo Form: with validations (shorthand)</h2>
15       <form [ngFormModel]="myForm"
16         (ngSubmit)="onSubmit(myForm.value)"
17         class="ui form">
18
19         <div class="field"
20           [class.error]="!myForm.find('sku').valid && myForm.find('sku').touched">
21           <label for="skuInput">SKU</label>
22           <input type="text"
23             id="skuInput"
24             placeholder="SKU"
25             #sku="ngForm"
26             [ngFormControl]="myForm.controls['sku']">
27           <div *ngIf="!sku.control.valid"
28             class="ui error message">SKU is invalid</div>
29           <div *ngIf="sku.control.hasError('required')"
30             class="ui error message">SKU is required</div>
31         </div>
32
33         <div *ngIf="!myForm.valid"
34           class="ui error message">Form is invalid</div>
35
36         <button type="submit" class="ui button">Submit</button>
37       </form>
38     </div>
39   `
40 })
41 export class DemoFormWithValidationsShorthand {
```

```

42     myForm: ControlGroup;
43
44     constructor(fb: FormBuilder) {
45         this.myForm = fb.group({
46             'sku': ['', Validators.required]
47         });
48     }
49
50     onSubmit(value: string): void {
51         console.log('you submitted value: ', value);
52     }
53 }
```

Custom Validations

We often are going to want to write our own custom validations. Let's take a look at how to do that. To see how validators are implemented, let's look at `Validators.required` from the Angular core source:

```

1 export class Validators {
2     static required(c: Control): StringMap<string, boolean> {
3         return isBlank(c.value) || c.value == "" ? {"required": true} : null;
4     }
```

A validator: - Takes a Control as it's input and - Returns a `StringMap<string, boolean>` where the key is "error code" and the value is `true` if it fails

Writing the Validator

Let's say we have specific requirements for our `sku`. For example, say our `sku` needs to begin with `123`. We could write a validator like so:

`code/forms/app/ts/forms/demo_form_with_custom_validations.ts`

```

20     function skuValidator(control: Control): { [s: string]: boolean } {
21         if (!control.value.match(/^123/)) {
22             return {invalidSku: true};
23         }
24     }
```

This validator will return an error code `invalidSku` if the input (the `control.value`) does not begin with `123`.

Assigning the Validator to the Control

Now we need to add the validator to our `Control`. However, there's one small problem: we already have a validator on `sku`. How can we add multiple validators to a single field?

For that, we use `Validators.compose`:

`code/forms/app/ts/forms/demo_form_with_custom_validations.ts`

```
63  this.myForm = fb.group({
64    'sku': ['', Validators.compose([
65      Validators.required, skuValidator])]
66  });
```

`Validators.compose` wraps our two validators and lets us assign them both to the `Control`. The `Control` is not `valid` unless both validations are valid.

Now we can use our new validator in the view:

`code/forms/app/ts/forms/demo_form_with_custom_validations.ts`

```
47  <div *ngIf="sku.hasError('invalidSku')"
48    class="ui error message">SKU must begin with <tt>123</tt></div>
```



Note that in this section, I'm using "explicit" notation of adding an instance variable for each `Control`. That means that in the view in this section, `sku` refers to a `Control`.

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 `ControlGroup` and `Control` 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 here³⁷](#)

To watch for changes on a control we:

³⁷<https://github.com/jhusain/observable-spec>

1. get access to the `EventEmitter` by calling `control.valueChanges`. Then we
2. add an *observer* using the `.observer` method

Here's an example:

`code/forms/app/ts/forms/demo_form_with_events.ts`

```

48  this.myForm = fb.group({
49    'sku': ['', Validators.required]
50  });
51
52  this.sku = this.myForm.controls['sku'];
53
54  this.sku.valueChanges.subscribe(
55    (value: string) => {
56      console.log('sku changed to: ', value);
57    }
58  );
59
60  this.myForm.valueChanges.subscribe(
61    (value: string) => {
62      console.log('form changed to: ', value);
63    }
64  );

```

Here we're observing two separate events: changes on the `sku` field and changes on the form as a whole.

The observable that we pass in is an object with a single key: `next` (there are other keys you can pass in, but we're not going to worry about those now). `next` is the function we want to call with the new value whenever the value changes.

If we type '`kj`' into the text box we will see in our console:

```

1  sku changed to: k
2  form changed to: Object {sku: "k"}
3  sku changed to: kj
4  form changed to: Object {sku: "kj"}

```

As you can see each keystroke causes the control to change, so our observable is triggered. When we observe the individual `Control` we receive a value (e.g. `kj`), but when we observe the whole form, we get an object of key-value pairs (e.g. `{sku: "kj"}`).

ngModel

NgModel is a special directive: it binds a model to a form. ngModel is special in that it **implements two-way data binding**. Two-way data binding is almost always more complicated and difficult to reason about vs. one-way data binding. Angular 2 is built to generally have data flow one-way: top-down. However, when it comes to forms, there are times where it is easier to opt-in to a two-way bind.



Just because you've used `ng-model` in Angular 1 in the past, don't rush to use `ngModel` right away. There are good reasons to avoid two-way data binding. Of course, `ngModel` can be really handy, but know that we don't necessarily rely on two-way data binding it as much as we did in Angular 1.

Let's change our form a little bit and say we want to input `productName`. We're going to use `ngModel` to keep the component instance variable in sync with the view.

First, here's our component definition class:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
41 export class DemoFormNgModel {
42   myForm: ControlGroup;
43   productName: string;
44
45   constructor(fb: FormBuilder) {
46     this.myForm = fb.group({
47       'productName': ['', Validators.required]
48     });
49   }
50
51   onSubmit(value: string): void {
52     console.log('you submitted value: ', value);
53   }
54 }
```

Notice that we're simply storing `productName: string` as an instance variable.

Next, let's use `ngModel` on our `input` tag:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
26      <input type="text"
27          id="productNameInput"
28          placeholder="Product Name"
29          [ngFormControl]="myForm.find('productName')"
30          [(ngModel)]="productName">
```

Now notice something - the syntax for `ngModel` is funny: we are using both brackets and parenthesis around the `ngModel` attribute! The idea this is intended to invoke is that we're using both the `input` [] brackets and the `output()` parenthesis. It's an indication of the two-way bind.

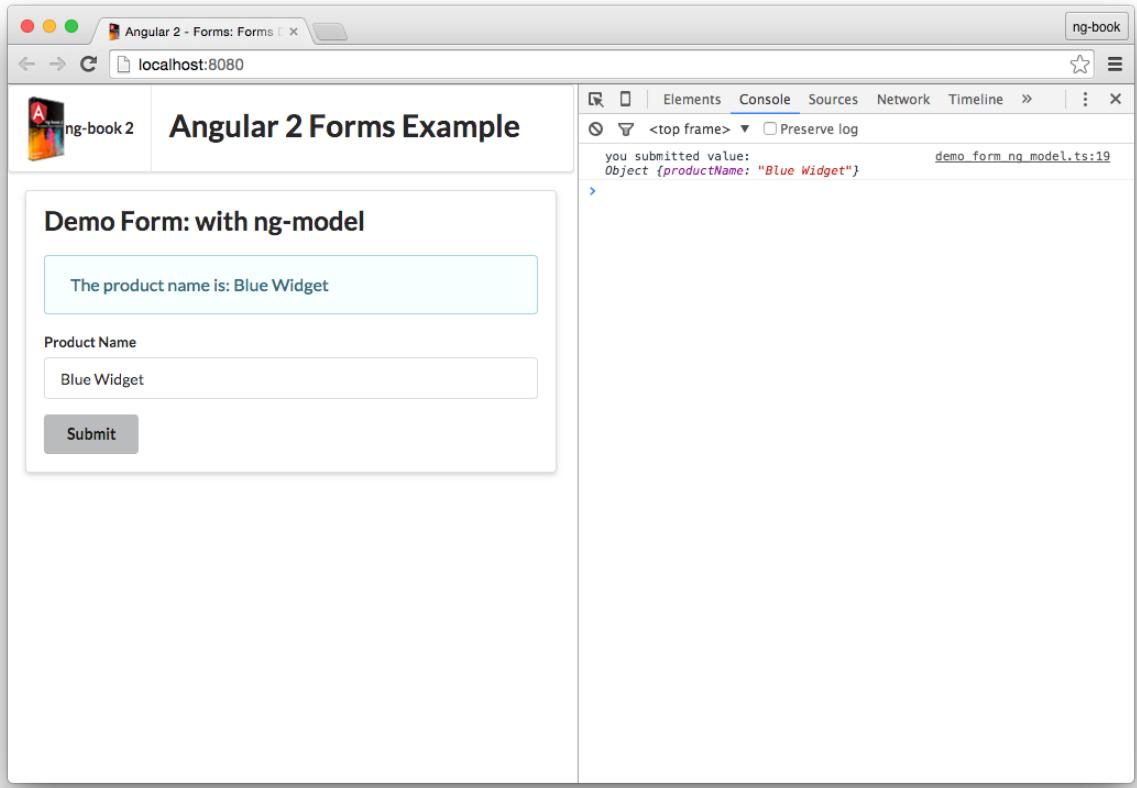
Notice something else here: we're still using `ngFormControl` to specify that this `input` should be bound to the `Control` on our form. We do this because `ngModel` is only binding the `input` to the instance variable - the `Control` is completely separate. But because we still want to validate this value and submit it as part of the form, we keep the `ngFormControl` directive.

Last, let's display our `productName` value in the view:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
16      <div class="ui info message">
17          The product name is: {{productName}}
18      </div>
```

Here's what it looks like:



Demo Form with ngModel

Wrapping Up

Forms have a lot of moving pieces, but Angular 2 makes it fairly straightforward. Once you get a handle on how to use ControlGroups, Controls, and Validations, it's pretty easy going from there!

Data Architecture in Angular 2

An Overview of Data Architecture

Managing data can be one of the trickiest aspects of writing a maintainable app. There are tons of ways to get data into your application:

- AJAX HTTP Requests
- Websockets
- 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:

- **MVV / Two-way data binding:** *Model-View-Whatever* is a term used³⁸ 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**³⁹: uses a unidirectional data flow. In Flux, Stores hold data, Views render what's in the Store, and Actions change the data in the Store. There is a bit more ceremony to setup Flux, but the idea is that because data only flows in one direction, it's easier to reason about.
- **Observables:** Observables give us streams of data. We subscribe to the streams and then perform operations to react to changes. RxJs⁴⁰ is the most popular reactive streams library for

³⁸See: Model View Whatever

³⁹<https://facebook.github.io/flux/>

⁴⁰<https://github.com/Reactive-Extensions/RxJS>

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.
- **Falcor⁴¹** is a powerful framework that helps bind your client-side models to the server-side data. Falcor often used with an Observables-type data architecture.

Data Architecture in Angular 2

Angular 2 is extremely flexible in what it allows for data architecture. A data strategy that works for one project doesn't necessarily work for another. So Angular doesn't prescribe a particular stack, but instead tries to make it easy to use whatever architecture we choose (while still retaining fast performance).

The benefit of this is that you have flexibility to fit Angular into almost any situation. The downside is that you have to make your own decisions about what's right for your project.

Don't worry, we're not going to leave you to make this decision on your own! In the chapters that follow, we're going to cover how to build applications using some of these patterns.

⁴¹<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.js⁴²](#) 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 here⁴³](#).

Learning Reactive Programming and RxJS

If you're just learning RxJS I recommend that you read this article first:

⁴²<http://underscorejs.org/>

⁴³<http://underscorejs.org/>

- [The introduction to Reactive Programming you've been missing⁴⁴](#) 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?](#)⁴⁵
- [Which instance operators to use on streams?](#)⁴⁶
- [RxMarbles](#)⁴⁷ - Interactive diagrams of the various operations on streams

Throughout this chapter I'll provide links to the API documentation of RxJS. The RxJS docs have tons of great example code that shed light on how the different streams and operators work.



Do I have to use RxJS to use Angular 2? - No, you definitely don't. Observables are just one pattern out of many that you can use with Angular 2. We talk more about [other data patterns you can use here](#).

I want to give you fair warning: learning RxJS can be a bit mind-bending at first. But trust me, you'll get the hang of it and it's worth it. Here's a few big ideas about streams that you might find helpful:

1. **Promises emit a single value whereas streams emit many values.** - Streams fulfill the same role in your application as promises. If you've made the jump from callbacks to promises, you know that promises are a big improvement in readability and data maintenance vs. callbacks. In the same way, streams improve upon the promise pattern in that we can continuously respond to data changes on a stream (vs. a one-time resolve from a promise)
2. **Imperative code “pulls” data whereas reactive streams “push” data** - In Reactive Programming our code subscribes to be notified of changes and the streams “push” data to these subscribers
3. **RxJS is *functional*** - If you're a fan of functional operators like `map`, `reduce`, and `filter` then you'll feel right at home with RxJS because streams are, in some sense, lists and so the powerful functional operators all apply
4. **Streams are composable** - Think of streams like a pipeline of operations over your data. You can subscribe to any part of your stream and even combine them to create new streams

⁴⁴<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>

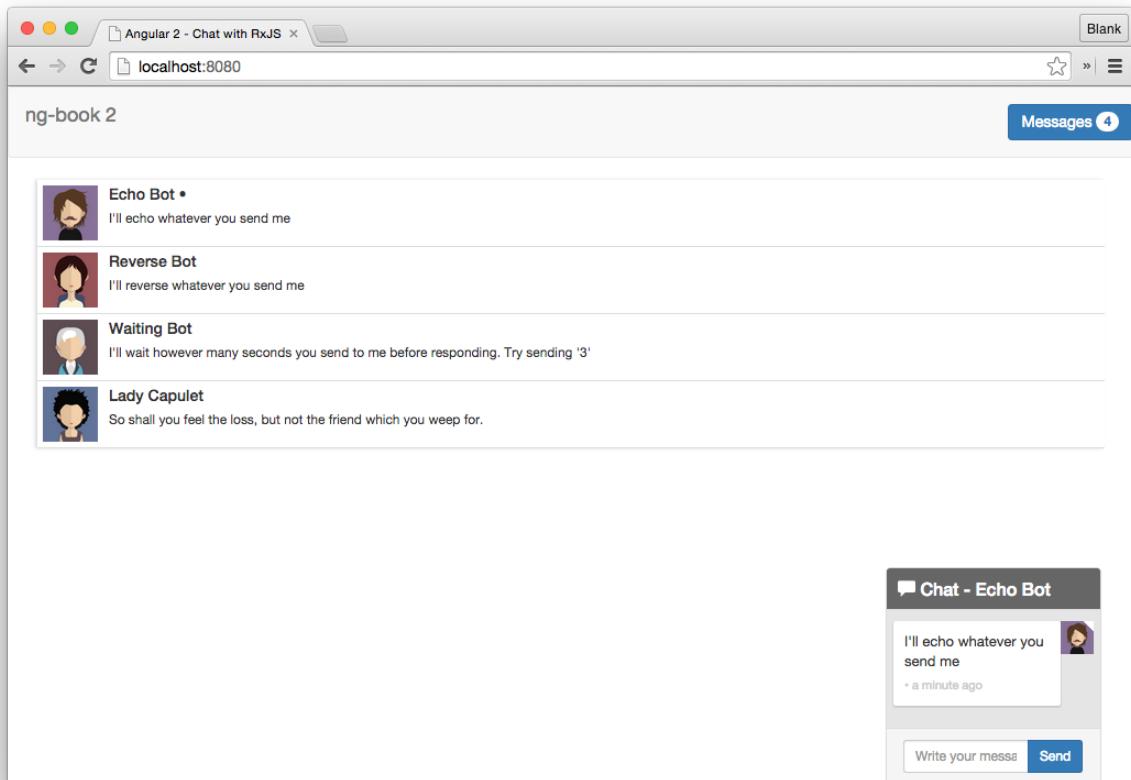
⁴⁵<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md>

⁴⁶<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md>

⁴⁷<http://staltz.com/rxmarbles>

Chat App Overview

In this chapter, we're going to use RxJS to build a chat app. Here's a screenshot:



Completed Chat Application



Usually we try to show every line of code here in the book text. However, this chat application has a lot of moving parts, so in this chapter we're not going to have every single line of code in the text. You can find the sample code for this chapter in the folder `code/rxjs/chat`. We'll call out each filter where you can view the context, where appropriate.

In this application we've provided a few bots you can chat with. Open up the code and try it out:

```
1 cd code/rxjs/chat  
2 make dev
```

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

Notice a few things:

- 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:

ng-book 2

ChatNavBar

Messages 4

Echo Bot •
I'll echo whatever you send me

Reverse Bot
I'll reverse whatever you send me

Waiting Bot
I'll wait however many seconds you send to me before responding. Try sending '3'

Lady Capulet
So shall you feel the loss, but not the friend which you weep for.

ChatThreads

ChatWindow

Chat - Echo Bot

I'll echo whatever you send me

a minute ago

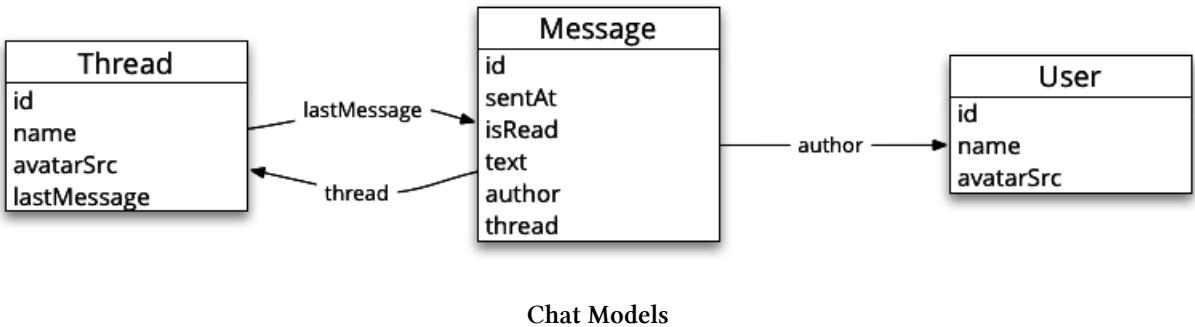
Write your message Send

Chat Top-Level Components

- ChatNavBar - contains the unread messages count
- ChatThreads - shows a clickable list of threads, along with the most recent message and the conversation avatar
- ChatWindow - shows the messages in the current thread with an input box to send new messages

Models

This application also has three models:



- User - stores information about a chat participant
- Message - stores an individual message
- Thread - stores a collection of Messages as well as some data about the conversation

Services

In this app, each of our models has a corresponding *service*. The services are singleton objects that play two roles:

1. **Provide streams** of data that our application can subscribe to
2. **Provide operations** to add or modify data

For instance, the UserService:

- publishes a stream that emits the current user and
- offers a setCurrentUser function which will set the current user (that is, emit the current user from the currentUser stream)

Summary

At a high level, the application data architecture is straightforward:

- The **services** maintain streams which emit models (e.g. `Messages`)
- The **components** subscribe to those streams and render according to the most recent values

For instance, the `ChatThreads` component listens for the most recent list of threads from the `ThreadService` and the `ChatWindow` subscribes for the most recent list of messages.

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular 2 and RxJS. We'll start by implementing our models, then look at how we create Services to manage our streams, and then finally implement the Components.

Implementing the Models

Let's start with the easy stuff and take a look at the models.

User

Our `User` class is straightforward. We have an `id`, `name`, and `avatarSrc`.

code/rxjs/chat/app/ts/models.ts

```
1 export class User {
2   id: string;
3
4   constructor(public name: string,
5             public avatarSrc: string) {
6     this.id = uuid();
7   }
8 }
```



Notice above that we're using a TypeScript shorthand in the constructor. When we say `public name: string` we're telling TypeScript that 1. we want `name` to be a public property on this class and 2. assign the argument value to that property when a new instance is created.

Thread

Similarly, `Thread` is also a straightforward TypeScript class:

code/rxjs/chat/app/ts/models.ts

```

1  export class Thread {
2    id: string;
3    lastMessage: Message;
4    name: string;
5    avatarSrc: string;
6
7    constructor(id?: string,
8              name?: string,
9              avatarSrc?: string) {
10      this.id = id || uuid();
11      this.name = name;
12      this.avatarSrc = avatarSrc;
13    }
14 }
```

Note that we store a reference to the `lastMessage` in our `Thread`. This let's us show a preview of the most recent message in the threads list.

Message

`Message` is also a simple TypeScript class, however in this case we use a slightly different form of constructor:

code/rxjs/chat/app/ts/models.ts

```

1  export class Message {
2    id: string;
3    sentAt: Date;
4    isRead: boolean;
5    author: User;
6    text: string;
7    thread: Thread;
8
9    constructor(obj?: any) {
10      this.id          = obj && obj.id          || uuid();
11      this.isRead     = obj && obj.isRead     || false;
12      this.sentAt    = obj && obj.sentAt    || new Date();
13      this.author    = obj && obj.author    || null;
14      this.text      = obj && obj.text      || null;
15      this.thread    = obj && obj.thread    || null;
```

```
16      }
17 }
```

The pattern you see here in the constructor allows us to simulate using keyword arguments in the constructor. Using this pattern, we can create a new `Message` using whatever data we have available and we don't have to worry about the order of the arguments. For instance we could do this:

```
1 let msg1 = new Message();
2
3 # or this
4
5 let msg2 = new Message({
6   text: "Hello Nate Murray!"
7 })
```

Now that we've looked at our models, let's take a look at our first service: the `UserService`.

Implementing UserService

The point of the `UserService` is to provide a place where our application can learn about the current user and also notify the rest of the application if the current user changes.

The first thing we need to do is create a TypeScript class and make it *injectable* by using the `@Injectable` annotation.

`code/rxjs/chat/app/ts/services/UserService.ts`

```
1 @Injectable()
2 export class UserService {
```



When we make something *injectable* that means we will be able to use it as a dependency to other components in our application. Briefly, two benefits of dependency-injection are:

1. we let Angular handle the lifecycle of the object and
2. it's easier to test injected components.

We talk more about `@Injectable` in the [chapter on dependency injection](#), but the result is that now we can inject it as a dependency to our components like so:

```
1  class MyComponent {
2    constructor(public userService: UserService) {
3      // do something with `userService` here
4    }
5 }
```

currentUser stream

Next we setup a stream which we will use to manage our current user:

`code/rxjs/chat/app/ts/services/UserService.ts`

```
1 currentUser: Rx.Subject<User> = new Rx.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`⁴⁸ inherits from both `Observable`⁴⁹ and `Observer`⁵⁰

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.

⁴⁸<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/subject.md>

⁴⁹<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md>

⁵⁰<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md>

BehaviourSubject⁵¹ has a special property in that it **stores the last value**. Meaning that any subscriber to the stream will receive the latest value. This is great for us because it means that any part of our application can subscribe to the `UserService.currentUser` stream and immediately know who the current user is.

Setting a new user

We need a way to publish a new user to the stream whenever the current user changes (e.g. logging in).

There's two ways we can expose an API for doing this:

1. Add new users to the stream directly:

The most straightforward way to update the current user is to have clients of the `UserService` simply publish a new `User` directly to the stream like this:

```

1 userService.subscribe((newUser) => {
2   console.log('New User is: ', newUser.name);
3 })
4
5 // => New User is: originalUserName
6
7 let u = new User('Nate', 'anImgSrc');
8 userService.currentUser.onNext(u);
9
10 // => New User is: Nate

```



Note here that we use the `onNext` method on a `Subject` to push a new value to the stream

The pro here is that we're able to reuse the existing API from the stream, so we're not introducing any new code or APIs

2. Create a `setCurrentUser(newUser: User)` method

The other way we could update the current user is to create a helper method on the `UserService` like this:

⁵¹<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md>

code/rxjs/chat/app/ts/services/UserService.ts

```
1  public setCurrentUser(newUser: User): void {
2      this.currentUser.onNext(newUser);
3  }
```

You'll notice that we're still using the `onNext` method on the `currentUser` stream, so why bother doing this?

Because there's value in decoupling the implementation of the `currentUser` from the implementation of the stream. By wrapping the `onNext` in the `setCurrentUser` call we give ourselves room to change the implementation of the `UserService` without breaking our clients.

In this case, I wouldn't recommend one method very strongly over the other, but it can make a big difference on the maintainability of larger projects.



A third option could be to have the updates expose streams of their own (that is, a stream where we place the action of changing the current user). We explore this pattern in the `MessagesService` below.

UserService.ts

Putting it together, our `UserService` looks like this:

code/rxjs/chat/app/ts/services/UserService.ts

```
1  export class UserService {
2      // `currentUser` contains the current user
3      currentUser: Rx.Subject<User> = new Rx.BehaviorSubject<User>(null);
4
5      public setCurrentUser(newUser: User): void {
6          this.currentUser.onNext(newUser);
7      }
8  }
```

The MessagesService

The `MessagesService` is the backbone of this application. In our app, all messages flow through the `MessagesService`.

Our `MessagesService` has much more sophisticated streams compared to our `UserService`. There are five streams that make up our `MessagesService`: 3 "data management" streams and 2 "action" streams.

The three data management streams are:

- newMessages - emits each new Message only once
- messages - emits an array of the current Messages
- updates - performs operations on messages

the newMessages stream

newMessages is a Subject that will publish each new Message only once.

code/rxjs/chat/app/ts/services/MessagesService.ts

```
1 export class MessagesService {  
2   // a stream that publishes new messages only once  
3   newMessages: Rx.Subject<Message> = new Rx.Subject<Message>();
```

If we want, we can define a helper method to add Messages to this stream:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
1 addMessage(message: Message): void {  
2   this.newMessages.onNext(message);  
3 }
```

It would also be helpful to have a stream that will get all of the messages from a thread that are not from a particular user. For instance, consider the Echo Bot:



Real mature, Echo Bot

When we are implementing the Echo Bot, we don't want to enter an infinite loop and repeat back the bot's messages to itself.

To implement this we can subscribe to the newMessages stream and filter out all messages that are 1. part of this thread and 2. not written by the bot. You can think of this as saying, for a given Thread I want a stream of the messages that are “for” this User.

code/rxjs/chat/app/ts/services/MessagesService.ts

```

1  messagesForThreadUser(thread: Thread, user: User): Rx.Observable<Message> {
2      return this.newMessages
3          .filter((message: Message) => {
4              // belongs to this thread
5              return (message.thread.id === thread.id) &&
6                  // and isn't authored by this user
7                  (message.author.id !== user.id);
8          });
9      }

```

messagesForThreadUser takes a Thread and a User and returns a new stream of Messages that are filtered on that Thread and not authored by the User. That is, it is a stream of “everyone else’s” messages in this Thread.

the messages stream

Whereas newMessages emits individual Messages, the messages stream emits an Array of the most recent Messages.

code/rxjs/chat/app/ts/services/MessagesService.ts

```

1  messages: Rx.Observable<Message[]>;

```



The type Message[] is the same as Array<Message>. Another way of writing the same thing would be: Rx.Observable<Array<Message>>. When we define the type of messages to be Rx.Observable<Message[]> we mean that this stream emits an Array (of Messages), not individual Messages.

So how does messages get populated? For that we need to talk about the updates stream and a new pattern: the Operation stream.

The Operation Stream Pattern

Here's the idea:

- We'll maintain state in `messages` which will hold an `Array` of the most current `Messages`
- We use an `updates` stream which is a **stream of functions** to apply to `messages`

You can think of it this way: any function that is put on the `updates` stream will change the list of the current messages. A function that is put on the `updates` stream should **accept a list of Messages** and then **return a list of Messages**. Let's formalize this idea by creating an interface in code:

`code/rxjs/chat/app/ts/services/MessagesService.ts`

```

1  interface IMessagesOperation extends Function {
2      (messages: Message[]): Message[];
3  }

```

Let's define our `updates` stream:

`code/rxjs/chat/app/ts/services/MessagesService.ts`

```

1  updates: Rx.Subject<any> =
2      new Rx.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/chat/app/ts/services/MessagesService.ts`

```

1  constructor() {
2      this.messages = this.updates
3          // watch the updates and accumulate operations on the messages
4          .scan(initialMessages, (messages: Message[],
5              operation: IMessagesOperation) => {
6                  return operation(messages);
7          })

```

This code introduces a new stream function: `scan`⁵². If you're familiar with functional programming, `scan` is a lot like `reduce`: 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:

⁵²<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md>

1. the messages we're accumulating and
2. the new operation to apply.

and then we return the new Message[].

Sharing the Stream

One thing to know about streams is that they aren't shareable by default. That is, if one subscriber reads a value from a stream, it can be gone forever. In the case of our messages, we want to 1. share the same stream among many subscribers and 2. replay the last value for any subscribers who come "late".

To do that, we use the `shareReplay`⁵³ method like so:

code/rxjs/chat/app/ts/services/MessagesService.ts

```

1  constructor() {
2      this.messages = this.updates
3          // watch the updates and accumulate operations on the messages
4          .scan(initialMessages, (messages: Message[],
5              operation: IMessagesOperation) => {
6                  return operation(messages);
7          })
8          // make sure we can share the most recent list of messages across anyone
9          // who's interested in subscribing and cache the last known list of
10         // messages
11         .shareReplay(1);

```

Adding Messages to the messages Stream

Now we could add a Message to the messages stream like so:

```

1 var myMessage = new Message(/* params here... */);
2
3 updates.onNext( (messages: Message[]): Message[] => {
4     return messages.concat(myMessage);
5 })

```

Above, we're adding an operation to the `updates` stream. `messages` is subscribe to that stream and so it will apply that operation which will concat our `newMessage` on to the accumulated list of messages.

⁵³<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/sharereplay.md>



It's okay if this takes a few minutes to mull over. It can feel a little foreign if you're not used to this style of programming.

One problem with the above approach is that it's a bit verbose to use. It would be nice to not have to write that inner function every time. We could do something like this:

```

1 addMessage(newMessage: Message) {
2   updates.onNext( (messages: Message[]): Message[] => {
3     return messages.concat(newMessage);
4   })
5 }
6
7 // somewhere else
8
9 var myMessage = new Message(/* params here... */);
10 MessagesService.addMessage(myMessage);

```

This is a little bit better, but it's not “the reactive way”. In part, because this action of creating a message isn't composable with other streams. (Also this method is circumventing our `newMessages` stream. More on that later.)

A reactive way of creating a new message would be to **have a stream that accepts `Messages` to add to the list**. Again, this can be a bit new if you're not used to thinking this way. Here's how you'd implement it:

First we make an “action stream” called `create`. (The term “action stream” is only meant to describe its role in our service. The stream itself is still a regular `Subject`):

`code/rxjs/chat/app/ts/services/MessagesService.ts`

```

1 // action streams
2 create: Rx.Subject<Message> = new Rx.Subject<Message>();

```

Next, in our constructor we configure the `create` stream:

code/rxjs/chat/app/ts/services/MessagesService.ts

```

1   this.create
2     .map( function(message: Message): IMessagesOperation {
3       return (messages: Message[]) => {
4         return messages.concat(message);
5       };
6     })

```

The `map`⁵⁴ 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`⁵⁵.

code/rxjs/chat/app/ts/services/MessagesService.ts

```

1   this.create
2     .map( function(message: Message): IMessagesOperation {
3       return (messages: Message[]) => {
4         return messages.concat(message);
5       };
6     })
7     .subscribe(this.updates);

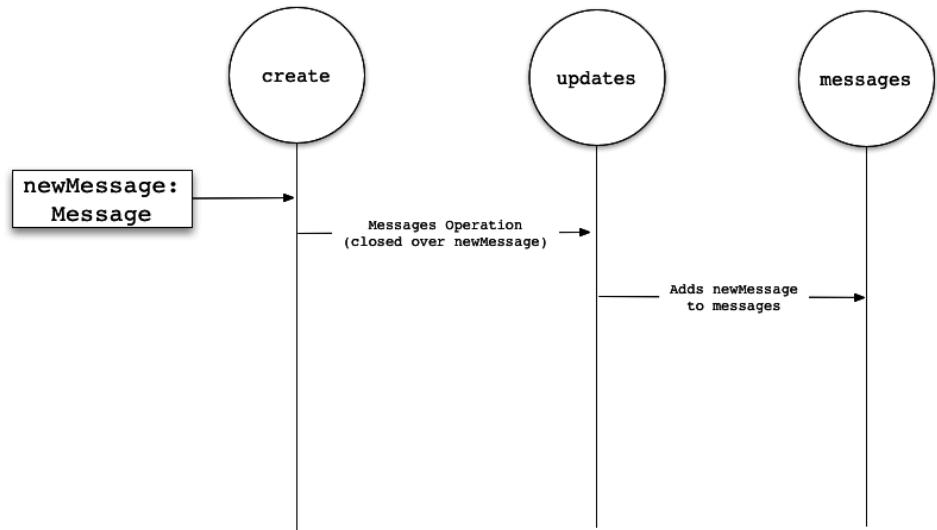
```

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:

⁵⁴<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/select.md>

⁵⁵<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.

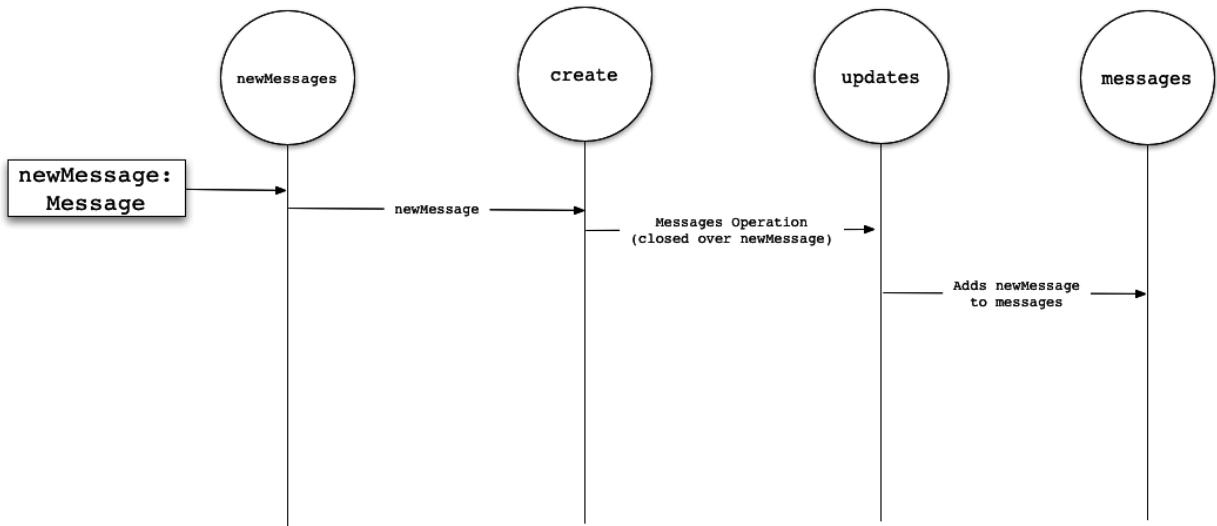
It would be great if we had a way to easily connect this stream with any `Message` that comes from `newMessages`. It turns out, it's really easy:

`code/rxjs/chat/app/ts/services/MessagesService.ts`

```

1   this.newMessages
2     .subscribe(this.create);
  
```

Now our diagram looks like this:



Creating a new message, starting with the `newMessages` stream

Now our flow is complete! It's the best of both worlds: we're able to subscribe to the stream of individual messages through `newMessages`, but if we just want the most up-to-date list, we can subscribe to `messages`.



It's worth pointing out some implications of this design: if you subscribe to `newMessages` directly, you have to be careful about changes that may happen downstream. Here are three things to consider:

First, you obviously won't get any downstream updates that are applied to the `Messages`.

Second, in this case, we have **mutable** `Message` objects. So if you subscribe to `newMessages` and store a reference to a `Message`, that `Message`'s attributes may change.

Third, in the case where you want to take advantage of the mutability of our `Messages` you may not be able to. Consider the case where we could put an operation on the `updates` queue that makes a copy of each `Message` and then mutates the copy. (This is probably a better design than what we're doing here.) In this case, you couldn't rely on any `Message` emitted directly from `newMessages` being in its "final" state.

That said, as long as you keep these considerations in mind, you shouldn't have too much trouble.

Our completed `MessagesService`

Here's what the completed `MessagesService` looks like:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
1 let initialMessages: Message[] = [];
2
3 interface IMessagesOperation extends Function {
4   (messages: Message[]): Message[];
5 }
6
7 @Injectable()
8 export class MessagesService {
9   // a stream that publishes new messages only once
10  newMessages: Rx.Subject<Message> = new Rx.Subject<Message>();
11
12  // `messages` is a stream that emits an array of the most up to date messages
13  messages: Rx.Observable<Message[]>;
14
15  // `updates` receives _operations_ to be applied to our `messages`
16  // it's a way we can perform changes on *all* messages (that are currently
17  // stored in `messages`)
18  updates: Rx.Subject<any> =
19    new Rx.Subject<any>();
20
21  // action streams
22  create: Rx.Subject<Message> = new Rx.Subject<Message>();
23  markThreadAsRead: Rx.Subject<any> = new Rx.Subject<any>();
24
25  constructor() {
26    this.messages = this.updates
27      // watch the updates and accumulate operations on the messages
28      .scan(initialMessages, (messages: Message[],
29        operation: IMessagesOperation) => {
30        return operation(messages);
31      })
32      // make sure we can share the most recent list of messages across anyone
33      // who's interested in subscribing and cache the last known list of
34      // messages
35      .shareReplay(1);
36
37  // `create` takes a Message and then puts an operation (the inner function)
38  // on the `updates` stream to add the Message to the list of messages.
39  //
40  // That is, for each item that gets added to `create` (by using `onNext`)
41  // this stream emits a concat operation function.
```

```
42  //  
43  // Next we subscribe `this.updates` to listen to this stream, which means  
44  // that it will receive each operation that is created  
45  //  
46  // Note that it would be perfectly acceptable to simply modify the  
47  // "addMessage" function below to simply add the inner operation function to  
48  // the update stream directly and get rid of this extra action stream  
49  // entirely. The pros are that it is potentially clearer. The cons are that  
50  // the stream is no longer composable.  
51  this.create  
52    .map( function(message: Message): IMessagesOperation {  
53      return (messages: Message $[]$ ) => {  
54        return messages.concat(message);  
55      };  
56    })  
57    .subscribe(this.updates);  
58  
59  this.newMessages  
60    .subscribe(this.create);  
61  
62  // similarly, `markThreadAsRead` takes a Thread and then puts an operation  
63  // on the `updates` stream to mark the Messages as read  
64  this.markThreadAsRead  
65    .map( (thread: Thread) => {  
66      return (messages: Message $[]$ ) => {  
67        return messages.map( (message: Message) => {  
68          // note that we're manipulating `message` directly here. Mutability  
69          // can be confusing and there are lots of reasons why you might want  
70          // to, say, copy the Message object or some other 'immutable' here  
71          if (message.thread.id === thread.id) {  
72            message.isRead = true;  
73          }  
74          return message;  
75        });  
76      };  
77    })  
78    .subscribe(this.updates);  
79  }  
80  
82  // an imperative function call to this action stream  
83  addMessage(message: Message): void {
```

```

84     this.newMessages.onNext(message);
85 }
86
87 messagesForThreadUser(thread: Thread, user: User): Rx.Observable<Message> {
88     return this.newMessages
89         .filter((message: Message) => {
90             // belongs to this thread
91             return (message.thread.id === thread.id) &&
92                 // and isn't authored by this user
93                 (message.author.id !== user.id);
94         });
95     }
96 }
```

Trying out MessagesService

If you haven't already, this would be a good time to open up the code and play around with the `MessagesService` to get a feel for how it works. We've got an example you can start with in `test/services/MessagesService.spec.ts`.



To run the tests in this project, open up your terminal then:

```

1 cd /path/to/code/rxjs/chat // <-- your path will vary
2 npm install
3 karma start
```

Let's start by creating a few instances of our models to use:

`code/rxjs/chat/test/services/MessagesService.spec.ts`

```

1 let user: User = new User("Nate", "");
2 let thread: Thread = new Thread("t1", "Nate", "");
3 let m1: Message = new Message({
4     author: user,
5     text: "Hi!",
6     thread: thread
7 });
8
9 let m2: Message = new Message({
10    author: user,
11    text: "Bye!",
```

```
12     thread: thread
13 );
```

Next let's subscribe to a couple of our streams:

code/rxjs/chat/test/services/MessagesService.spec.ts

```
1  let messagesService = new MessagesService();
2
3  // listen to each message individually as it comes in
4  messagesService.newMessages
5    .subscribe( (message: Message) => {
6      console.log("=> newMessages: " + message.text);
7    });
8
9  // listen to the stream of most current messages
10 messagesService.messages
11   .subscribe( (messages: Message[]) => {
12     console.log("=> messages: " + messages.length);
13   });
14
15 messagesService.addMessage(m1);
16 messagesService.addMessage(m2);
17
18 // => messages: 0
19 // => messages: 1
20 // => newMessages: Hi!
21 // => messages: 2
22 // => newMessages: Bye!
```

Two things to notice here:

1. Our messages subscription immediately receives a zero-length array. This is because we used `shareReplay` on the stream which, as you recall, means that subscribers are given the last known result.
2. 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 we're going to define four streams that emit respectively:

1. A map of the current set of Threads (in `threads`)
2. A chronological list of Threads, newest-first (in `orderedThreads`)
3. The currently selected Thread (in `currentThread`)
4. The list of Messages for the currently selected Thread (in `currentThreadMessages`)

Let's walk through how to build each of these streams, and we'll learn a little more about RxJS along the way.

A map of the current set of Threads (in `threads`)

Let's start by defining our ThreadsService class and the instance variable that will emit the Threads:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 @Injectable()
2 export class ThreadsService {
3
4   // `threads` is an observable that contains the most up to date list of threads
5   threads: Rx.Observable<{ [key: string]: Thread }>;
```

Notice that this stream will emit a map (an object) with the id of the Thread being the string key and the Thread itself will be the value.

To create a stream that maintains the current list of threads, we start by attaching to the `messagesService.messages` stream:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 this.threads = messagesService.messages
```

Recall that each time a new Message is added to the stream, `messages` will emit an array of the current Messages. We're going to look at each Message and we want to return a unique list of the Threads.

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1   this.threads = messagesService.messages
2   .map( (messages: Message[]) => {
3     let threads: {[key: string]: Thread} = {};
4     // Store the message's thread it in our accumulator `threads`
5     messages.map((message: Message) => {
6       threads[message.thread.id] = threads[message.thread.id] ||
7         message.thread;

```

Notice above that each time we will create a new list of threads. The reason for this is because we might delete some messages down the line (e.g. leave the conversation). Because we're recalculating the list of threads each time, we naturally will "delete" a thread if it has no messages.

In the threads list, we want to show a preview of the chat by using the text of the most recent Message in that Thread.



List of Threads with Chat Preview

In order to do that, we'll store the most recent Message for each Thread. We know which Message is newest by comparing the `sentAt` times:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1   // Cache the most recent message for each thread
2   let messagesThread: Thread = threads[message.thread.id];
3   if (!messagesThread.lastMessage || 
4       messagesThread.lastMessage.sentAt < message.sentAt) {
5       messagesThread.lastMessage = message;
6   }
7 });
8   return threads;
9 )

```

Lastly, we want to make sure that all subscribers get the same stream and that new subscribers get the most recent value. To do that, we'll use `shareReplay` again.

Putting it all together, `threads` looks like this:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1 this.threads = messagesService.messages
2   .map( (messages: Message[]) => {
3     let threads: {[key: string]: Thread} = {};
4     // Store the message's thread in our accumulator `threads`
5     messages.map((message: Message) => {
6       threads[message.thread.id] = threads[message.thread.id] ||
7         message.thread;
8
9       // Cache the most recent message for each thread
10      let messagesThread: Thread = threads[message.thread.id];
11      if (!messagesThread.lastMessage || 
12          messagesThread.lastMessage.sentAt < message.sentAt) {
13          messagesThread.lastMessage = message;
14      }
15    });
16    return threads;
17  })
18  // share this stream across multiple subscribers and makes sure everyone
19  // receives the current list of threads when they first subscribe
20  .shareReplay(1);

```

Trying out the ThreadsService

Let's try out our `ThreadsService`. First we'll create a few models to work with:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```

1  let nate: User = new User("Nate Murray", "");
2  let felipe: User = new User("Felipe Coury", "");
3
4  let t1: Thread = new Thread("t1", "Thread 1", "");
5  let t2: Thread = new Thread("t2", "Thread 2", "");
6
7  let m1: Message = new Message({
8    author: nate,
9    text: "Hi!",
10   thread: t1
11 });
12
13 let m2: Message = new Message({
14   author: felipe,
15   text: "Where did you get that hat?",
16   thread: t1
17 });
18
19 let m3: Message = new Message({
20   author: nate,
21   text: "Did you bring the briefcase?",
22   thread: t2
23 });

```

Now let's create an instance of our services:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```

1  let messagesService = new MessagesService();
2  let 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/chat/test/services/ThreadsService.spec.ts

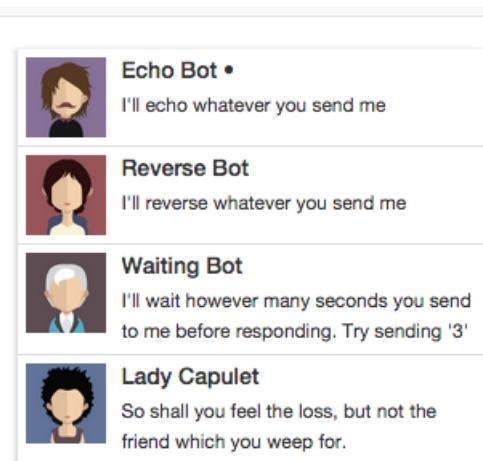
```

1   threadsService.threads
2     .subscribe( (threadIdx) => {
3       var threads = _.values(threadIdx);
4       var threadNames = _.map(threads, 'name').join(', ')
5       console.log(`=> threads (${threads.length}): ${threadNames}`);
6     })
7
8     messagesService.addMessage(m1);
9     messagesService.addMessage(m2);
10    messagesService.addMessage(m3);
11
12    // => threads (1): Thread 1
13    // => threads (1): Thread 1
14    // => threads (2): Thread 1, Thread 2
15
16  });
17 });

```

A chronological list of Threads, newest-first (in orderedThreads)

threads gives us a map which acts as an “index” of our list of threads. But we want the threads view to be ordered according the most recent message.



Time Ordered List of Threads

Let's create a new stream that returns an Array of Threads ordered by the most recent Message time: We'll start by defining orderedThreads as an instance property:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 // `orderedThreads` contains a newest-first chronological list of threads
2 orderedThreads: Rx.Observable<Thread[]>;
```

Next, in the constructor we'll define orderedThreads by subscribing to threads and ordered by the most recent message:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 this.orderedThreads = this.threads
2 .map((threadGroups: { [key: string]: Thread }) => {
3     let threads: Thread[] = _.values(threadGroups);
4     return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
5 })
6 .shareReplay(1);
```

The currently selected Thread (in currentThread)

Our application needs to know which Thread is the currently selected thread. This lets us know:

1. which thread should be shown in the messages window
2. which thread should be marked as the current thread in the list of threads



The current thread is marked by a ‘•’ symbol

Let's create a BehaviorSubject that will store the currentThread:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 // `currentThread` contains the currently selected thread
2 currentThread: Rx.Subject<Thread> =
3   new Rx.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 `onNext` directly or 2. add a helper method to do it. Let's define a helper method `setCurrentThread` that we can use to set the next thread:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 setCurrentThread(newThread: Thread): void {
2   this.currentThread.onNext(newThread);
3 }
```

Marking the Current Thread as Read

We want to keep track of the number of unread messages. If we switch to a new Thread then we want to mark all of the Messages in that Thread as read. We have the parts we need to do this:

1. The `messagesService.makeThreadAsRead` accepts a Thread and then will mark all Messages in that Threaad as read
2. Our `currentThread` emits a single Thread that represents the current Thread

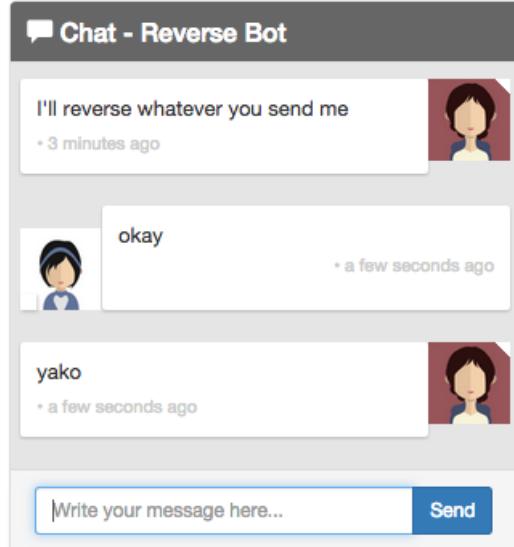
So all we need to do is hook them together:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1 this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

The list of Messages for the currently selected Thread (in currentThreadMessages)

Now that we have the currently selected thread, we need to make sure we can show the list of Messages in that Thread.



The current list of messages is for the Reverse Bot

Implementing this is a little bit more complicated than it may seem at the surface. Say we implemented it like this:

```

1 var the.currentThread: Thread;
2
3 this.currentThread.subscribe((thread: Thread) => {
4   the.currentThread = thread;
5 })
6
7 this.currentThreadMessages.map(
8   (messages: Message[]) => {
9     return _.filter(messages,
10       (message: Message) => {
11         return message.thread.id == the.currentThread.id;
12       })
13   })

```

What's wrong with this approach? Well, if the `currentThread` changes, `currentThreadMessages` won't know about it and so we'll have an outdated list of `currentThreadMessages`!

What if we reversed it, and stored the current list of messages in a variable and subscribed to the changing of `currentThread`? We'd have the same problem only this time we would know when the thread changes but not when a new message came in.

How can we solve this problem?

It turns out, RxJS has a set of operators that we can use to **combine multiple streams**. In this case we want to say “if *either* `currentThread` **or** `messagesService.messages` changes, then we want to emit something.” For this we use the `combineLatest`⁵⁶ operator.

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1   this.currentThreadMessages = this.currentThread
2     .combineLatest(messagesService.messages,
3                   (currentThread: Thread, messages: Message[]) => {
```

When we're combining two streams one or the other will arrive first and there's no guarantee that we'll have a value on both streams, so we need to check to make sure we have what we need otherwise we'll just return an empty list.

Now that we have both the current thread and messages, we can filter out just the messages we're interested in:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1   .combineLatest(messagesService.messages,
2                   (currentThread: Thread, messages: Message[]) => {
3                     if (currentThread && messages.length > 0) {
4                       return _.chain(messages)
5                         .filter((message: Message) =>
6                           (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.

⁵⁶<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md>

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1   .filter((message: Message) =>
2       (message.thread.id === currentThread.id))
3   .map((message: Message) => {
4       message.isRead = true;
5       return message; })
6   .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 it's messages marked as read. That said, the "read with side-effects" is not a pattern I recommend in general.

Putting it together, here's what `currentThreadMessages` looks like:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```

1  this.currentThreadMessages = this.currentThread
2  .combineLatest(messagesService.messages,
3      (currentThread: Thread, messages: Message[]) => {
4          if (currentThread && messages.length > 0) {
5              return _.chain(messages)
6                  .filter((message: Message) =>
7                      (message.thread.id === currentThread.id))
8                  .map((message: Message) => {
9                      message.isRead = true;
10                     return message; })
11                 .value();
12         } else {
13             return [];
14         }
15     })
16     .shareReplay(1);

```

Our Completed ThreadsService

Here's what our `ThreadService` looks like:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
1  @Injectable()
2  export class ThreadsService {
3
4      // `threads` is a observable that contains the most up to date list of threads
5      threads: Rx.Observable<{ [key: string]: Thread }>;
6
7      // `orderedThreads` contains a newest-first chronological list of threads
8      orderedThreads: Rx.Observable<Thread[]>;
9
10     // `currentThread` contains the currently selected thread
11     currentThread: Rx.Subject<Thread> =
12         new Rx.BehaviorSubject<Thread>(new Thread());
13
14     // `currentThreadMessages` contains the set of messages for the currently
15     // selected thread
16     currentThreadMessages: Rx.Observable<Message[]>;
17
18     constructor(public messagesService: MessagesService) {
19
20         this.threads = messagesService.messages
21             .map( (messages: Message[]) => {
22                 let threads: {[key: string]: Thread} = {};
23                 // Store the message's thread it in our accumulator `threads`
24                 messages.map((message: Message) => {
25                     threads[message.thread.id] = threads[message.thread.id] ||
26                         message.thread;
27
28                     // Cache the most recent message for each thread
29                     let messagesThread: Thread = threads[message.thread.id];
30                     if (!messagesThread.lastMessage ||
31                         messagesThread.lastMessage.sentAt < message.sentAt) {
32                         messagesThread.lastMessage = message;
33                     }
34                 });
35                 return threads;
36             })
37             // share this stream across multiple subscribers and makes sure everyone
38             // receives the current list of threads when they first subscribe
39             .shareReplay(1);
40
41         this.orderedThreads = this.threads
```

```

42     .map((threadGroups: { [key: string]: Thread }) => {
43       let threads: Thread[] = _.values(threadGroups);
44       return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
45     })
46     .shareReplay(1);
47
48   this.currentThreadMessages = this.currentThread
49     .combineLatest(messagesService.messages,
50                   (currentThread: Thread, messages: Message[]) => {
51       if (currentThread && messages.length > 0) {
52         return _.chain(messages)
53           .filter((message: Message) =>
54             (message.thread.id === currentThread.id))
55           .map((message: Message) => {
56             message.isRead = true;
57             return message; })
58           .value();
59       } else {
60         return [];
61       }
62     })
63     .shareReplay(1);
64
65   this.currentThread.subscribe(this.messagesService.markThreadAsRead);
66
67 }
68
69 setCurrentThread(newThread: Thread): void {
70   this.currentThread.onNext(newThread);
71 }
72
73 }
```

Data Model Summary

Our data model and services are complete! Now we have everything we need now to start hooking it up to our view components! In the next chapter we'll build out our 3 major components to render and interact with these streams.

Data Architecture with Observables - Part 2: View Components

Building Our Views: The ChatApp Top-Level Component

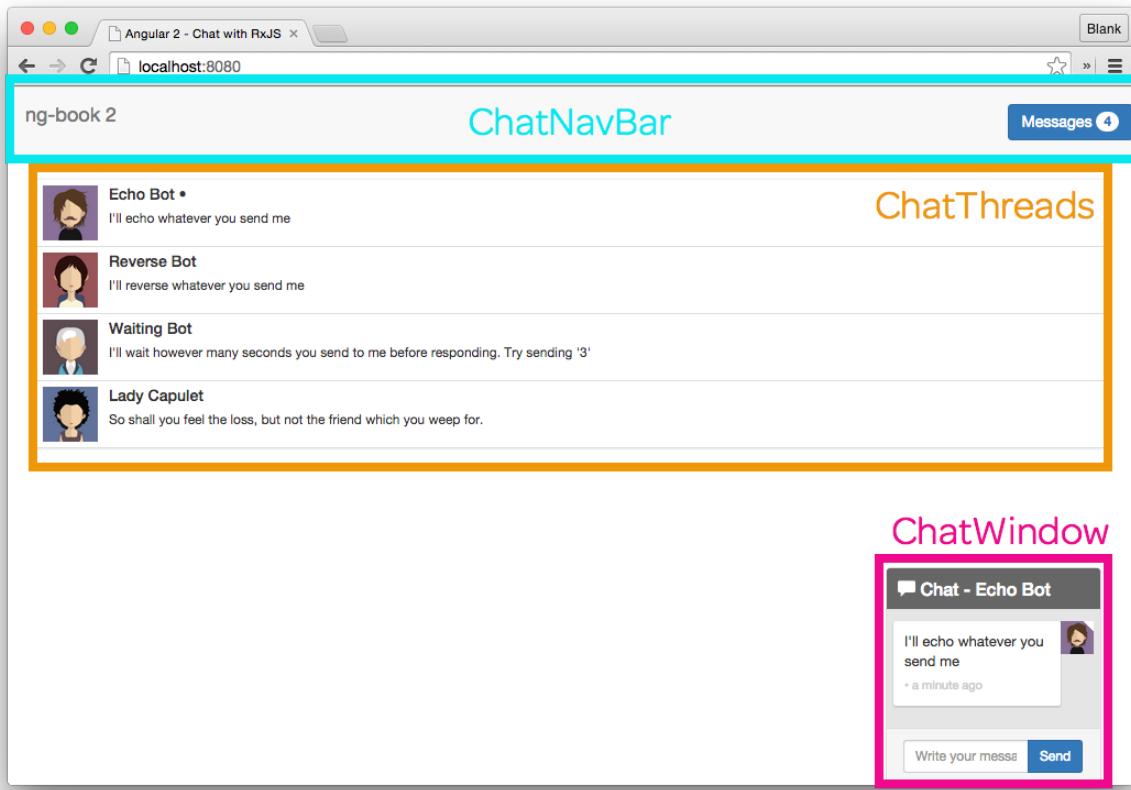
Let's turn our attention to our app and implement our view components.



For the sake of clarity and space, in the following sections I'll be leaving out 1. `import` statements, CSS, and a few other housekeeping things along those lines. 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

- ChatNavBar - contains the unread messages count
- ChatThreads - shows a clickable list of threads, along with the most recent message and the conversation avatar
- ChatWindow - shows the messages in the current thread with an input box to send new messages

Here's what our component looks like in code:

code/rxjs/chat/app/ts/app.ts

```
1  @Component({
2    selector: "chat-app"
3  })
4  @View({
5    directives: [ChatNavBar,
6                  ChatThreads,
7                  ChatWindow],
8    template: `
9      <div>
10        <nav-bar></nav-bar>
11        <div class="container">
12          <chat-threads></chat-threads>
13          <chat-window></chat-window>
14        </div>
15      </div>
16    `
17  })
18  class ChatApp {
19    constructor(public messagesService: MessagesService,
20                public threadsService: ThreadsService,
21                public userService: UserService) {
22      ChatExampleData.init(messagesService, threadsService, userService);
23    }
24  }
25 }
```

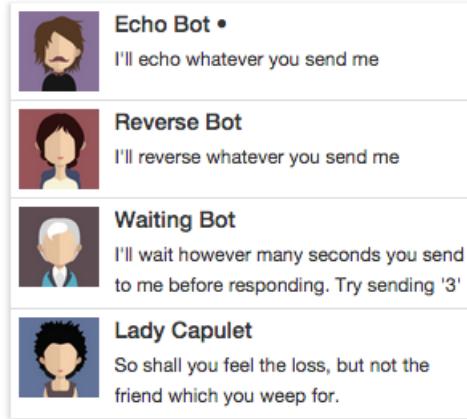
Take a look at the constructor. Here we're injecting our three services: the `MessagesService`, `ThreadsService`, and `UserService`. We're using those services to initialize our example data.



If you're interested in the example data you can find it in `code/rxjs/chat/app/ts/ChatExampleData.ts`.

The ChatThreads Component

Next let's build our thread list in the `ChatThreads` component.



Time Ordered List of Threads

Our @Component is straightforward, we want to match the selector `chat-threads`.

code/rxjs/chat/app/ts/components/ChatThreads.ts

```

1  @Component({
2    selector: "chat-threads"
3  })

```

ChatThreads Controller

Take a look at our component controller `ChatThreads`:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```

1  export class ChatThreads {
2    threads: Rx.Observable<any>;
3
4    constructor(public threadsService: ThreadsService) {
5      this.threads = threadsService.orderedThreads;
6    }
7  }

```

Here we're injecting `ThreadsService` and then we're keeping a reference to the `orderedThreads`.

ChatThreads @View

Lastly, let's look at the @View:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```

1  @View({
2    directives: [NgFor, ChatThread],
3    pipes: [RxPipe],
4    template: `
5      <!-- conversations -->
6      <div class="row">
7        <div class="conversation-wrap">
8
9          <chat-thread
10            *ng-for="#thread of threads | rx"
11            [thread]="thread">
12          </chat-thread>
13
14        </div>
15      </div>
16
17  })

```

There's three things to look at here: NgFor, ChatThread and RxPipe.

The ChatThread directive component (which matches `chat-thread` in the markup) will show the view for the Threads. We'll define that in a moment.

The NgFor iterates over our threads, and passes the property `[thread]` to our ChatThread directive. But you probably notice something new in our `*ng-for`: the pipe to `rx`.

RxPipe is the class that defines this behavior. RxPipe is a custom, pipe that lets us use an RxJS Observable here in our view. We're not going to spend too much time talking about the implementation details of RxPipe, but you can find the definition in `code/rxjs/app/ts/util/RxPipe.ts`.



RxPipe is going to become unnecessary as Angular 2 will support RxJS Observables using the built-in `AsyncPipe`. However as of today (`v2.0.0-alpha.35`) we have to use our custom pipe.

What's great about this is that Angular 2 is letting us use our async observable as if it was a sync collection. This is super convenient and really cool.

Here's what our total ChatThreads component looks like:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
1  @Component({
2    selector: "chat-threads"
3  })
4  @View({
5    directives: [NgFor, ChatThread],
6    pipes: [RxPipe],
7    template: `
8      <!-- conversations -->
9      <div class="row">
10        <div class="conversation-wrap">
11
12          <chat-thread
13            *ng-for="#thread of threads | rx"
14            [thread]="thread">
15          </chat-thread>
16
17        </div>
18      </div>
19
20    })
21  export class ChatThreads {
22    threads: Rx.Observable<any>;
23
24    constructor(public threadsService: ThreadsService) {
25      this.threads = threadsService.orderedThreads;
26    }
27 }
```



ON_PUSH change detection?

If you've been keeping up with Angular 2, you'll know it's got 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.

As of today (Aug 2015, v2.0.0-alpha.35), the Angular team is working hard to convert Angular2 to RxJS. When that happens we'll be able to use our RxJS observables with ON_PUSH change detection in a clean and easy way.

For this revision of the book, I've left ON_PUSH out, because the details are still being worked out.

Everything in this chapter is still worth learning: the data architecture will stay the same and the components will act the same and more-or-less be implemented the same way. The only difference will be that with ON_PUSH change detection our app will run much more efficiently.

We'll update this section once RxJS is integrated into Angular 2.

The Single ChatThread Component

Let's look at our ChatThread component. This is the component that will be used to display a **single thread**. Starting with the @Component:

`code/rxjs/chat/app/ts/components/ChatThreads.ts`

```

1  @Component({
2    lifecycle: [ LifecycleEvent.onInit ],
3    properties: ["thread"],
4    selector: "chat-thread"
5  })

```

We've got a new key here: `lifecycle`. Angular components can declare that they want to listen for certain lifecycle events. We talk more about lifecycle events [here in the Components chapter](#). The `lifecycle` key accepts an array of `LifecycleEvents` that we want to listen for.

ChatThread Controller and onInit

In this case, `onInit` will be called on our component after the component has been checked for changes the first time.

A key reason we will use `onInit` is because **our `thread` property won't be available in the constructor. Let's take a look at the component controller:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
1 class ChatThread {
2   thread: Thread;
3   selected: boolean = false;
4
5   constructor(public threadsService: ThreadsService) {
6   }
7
8   onInit(): void {
9     this.threadsService.currentThread
10    .subscribe( (currentThread: Thread) => {
11      this.selected = currentThread &&
12        this.thread &&
13        (currentThread.id === this.thread.id);
14    });
15  }
16
17  clicked(event: any): void {
18    this.threadsService.setCurrentThread(this.thread);
19    event.preventDefault();
20  }
21 }
```

Above you can see that in `onInit` 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 `@View` below, we will bind `clicked()` to clicking on the thread view. If we receive `clicked()` then we tell the `threadsService` we want to set the current thread to the `Thread` of this component.

ChatThread @View

Here's the code for our `@View`:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
1 @View({
2   directives: [NgIf],
3   template: `
4     <div class="media conversation">
5       <div class="pull-left">
6         
10      <h5 class="media-heading contact-name">{{thread.name}}</h5>
11      <span *ng-if="selected">&bull;</span>
12    </div>
13      <small class="message-preview">{{thread.lastMessage.text}}</small>
14    </div>
15      <a (click)="clicked($event)" class="div-link">Select</a>
16    </div>
17  `>
18 })
```

Notice we've got some straight-forward bindings like {{thread.avatarSrc}}, {{thread.name}}, and {{thread.lastMessage.text}}.

We've got an *ng-if which will show the • symbol only if this is the selected thread.

Lastly, we're binding to the (click) event to call our clicked() handler. Notice that when we call clicked we're passing the argument \$event. This is a special variable provided by Angular that describes the event. We use that in our clicked handler by calling event.preventDefault();. This makes sure that we don't navigate to a different page.

ChatThread Complete Code

Here's the whole of the ChatThread component:

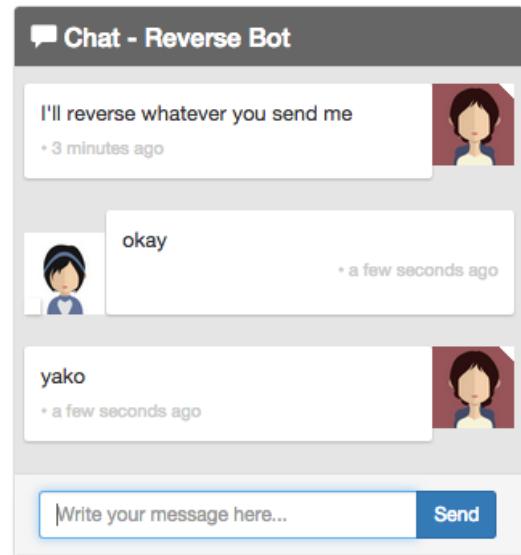
code/rxjs/chat/app/ts/components/ChatThreads.ts

```
1 @Component({
2   lifecycle: [ LifecycleEvent.onInit ],
3   properties: ["thread"],
4   selector: "chat-thread"
5 })
6 @View({
7   directives: [NgIf],
8   template: `
9     <div class="media conversation">
10       <div class="pull-left">
11         
15         <h5 class="media-heading contact-name">{{thread.name}}</h5>
16         <span *ng-if="selected">&bull;</span>
17       </div>
18       <small class="message-preview">{{thread.lastMessage.text}}</small>
19     </div>
20     <a (click)="clicked($event)" class="div-link">Select</a>
21   </div>
22   `
23 })
24 class ChatThread {
25   thread: Thread;
26   selected: boolean = false;
27
28   constructor(public threadsService: ThreadsService) {
29   }
30
31   OnInit(): void {
32     this.threadsService.currentThread
33       .subscribe( (currentThread: Thread) => {
34       this.selected = currentThread &&
35         this.thread &&
36         (currentThread.id === this.thread.id);
37     });
38   }
39
40   clicked(event: any): void {
41     this.threadsService.setCurrentThread(this.thread);
```

```
42     event.preventDefault();
43 }
44 }
```

The ChatWindow Component

The ChatWindow is the most complicated component in our app. Let's take it one section at a time:



The Chat Window

We start by defining our @Component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 @Component({
2   lifecycle: [ LifecycleEvent.onInit ],
3   selector: "chat-window"
4 })
```

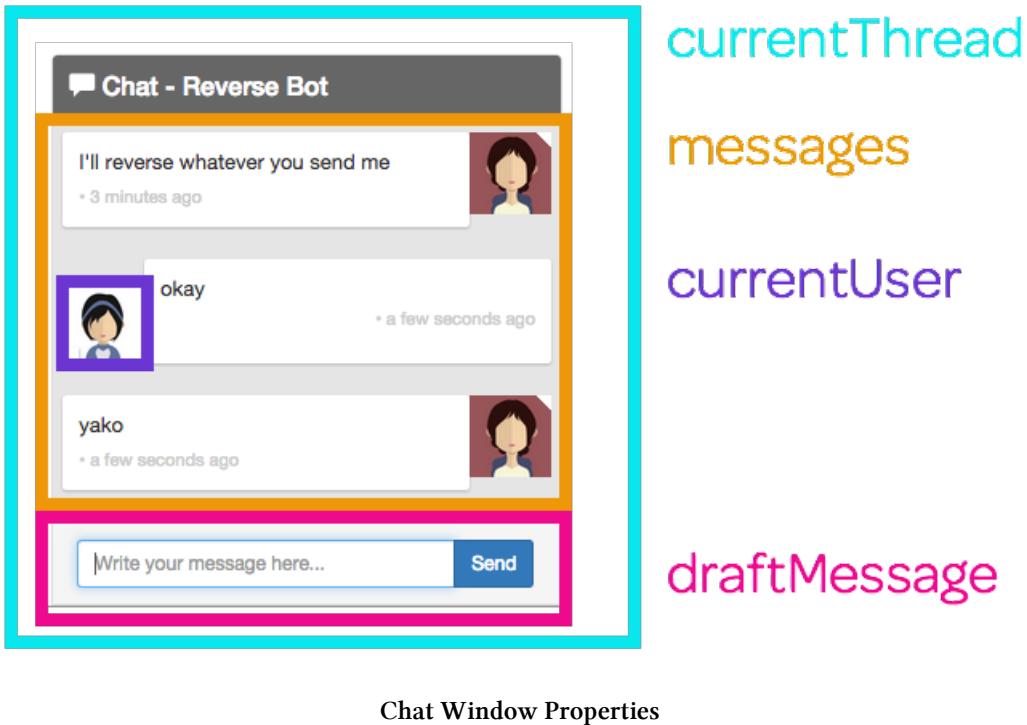
ChatWindow Properties

Our Chat window has four properties:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 export class ChatWindow {
2   messages: Rx.Observable<any>;
3   currentThread: Thread;
4   draftMessage: Message;
5   currentUser: User;
```

Here's a diagram of where each one is used:



In our constructor we're going to inject four things:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 constructor(public messagesService: MessagesService,
2             public threadsService: ThreadsService,
3             public userService: UserService,
4             public el: ElementRef) {
5 }
```

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`

ChatWindow `onInit`

We're going to put the initialization of this component in `onInit`. The main thing we're going to be doing here is setting up the subscriptions on our observables which will then change our component properties.

`code/rxjs/chat/app/ts/components/ChatWindow.ts`

```

1  onInit(): void {
2      this.messages = this.threadsService.currentThreadMessages;
3
4      this.draftMessage = new Message();

```

First, we'll save the `currentThreadMessages` into `messages`. Next we create an empty `Message` for the default `draftMessage`.

When we send a new message we need to make sure that `Message` stores a reference to the sending `Thread`. The sending thread is always going to be the current thread, so let's store a reference to the currently selected thread:

`code/rxjs/chat/app/ts/components/ChatWindow.ts`

```

1  this.threadsService.currentThread.subscribe(
2      (thread: Thread) => {
3          this.currentThread = thread;
4      });

```

We also want new messages to be sent from the current user, so let's do the same with `currentUser`:

`code/rxjs/chat/app/ts/components/ChatWindow.ts`

```

1  this.userService.currentUser
2      .subscribe(
3          (user: User) => {
4              this.currentUser = user;
5          });

```

ChatWindow `sendMessage`

Since we're talking about it, let's implement a `sendMessage` function that will send a new message:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 sendMessage(): void {
2     let m: Message = this.draftMessage;
3     m.author = this.currentUser;
4     m.thread = this.currentThread;
5     m.isRead = true;
6     this.messagesService.addMessage(m);
7     this.draftMessage = new Message();
8 }
```

The `sendMessage` function above takes the `draftMessage`, sets the `author` and `thread` using our component properties. Every message we send has “been read” already (we wrote it) so we mark it as read.

Notice here that we’re not updating the `draftMessage` text. That’s because we’re going to bind the value of the `messages` text in the view in a few minutes.

After we’ve updated the `draftMessage` properties we send it off to the `messagesService` and then **create a new `Message`** and set that new `Message` to `this.draftMessage`. We do this to make sure we don’t mutate an already sent message.

ChatWindow `onEnter`

In our view, we want to send the message in two scenarios 1. the user hits the “Send” button or 2. the user hits the Enter (or Return) key.

Let’s define a function that will handle that event:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 onEnter(event: any): void {
2     this.sendMessage();
3     event.preventDefault();
4 }
```

ChatWindow `scrollToBottom`

When we send a message, or when a new message comes in, we want to scroll to the bottom of the chat window. To do that, we’re going to set the `scrollTop` property of our host element:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 scrollToBottom(): void {
2     let scrollPane: any = this.el
3     .nativeElement.querySelector(".msg-container-base");
4     scrollPane.scrollTop = scrollPane.scrollHeight;
5 }
```

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 `onInit` let's subscribe to the list of `currentThreadMessages` and scroll to the bottom any time we get a new message:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 this.messages
2     .subscribe(
3         (messages: Array<Message>) => {
4             setTimeout(() => {
5                 this.scrollToBottom();
6             });
7         });

```



Why do we have the `setTimeout`?

If we call `scrollToBottom` immediately when we get a new message then what happens is we scroll to the bottom before the new message is rendered. By using a `setTimeout` we're telling Javascript that we want to run this function when it is finished with the current execution queue. This happens **after** the component is rendered, so it does what we want.

ChatWindow @View

The opening of our `@View` should look familiar, we're just describing the directives and pipes we're going to use:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 @View({
2   directives: [NgFor,
3               ChatMessage,
4               FORM_DIRECTIVES],
5   pipes: [RxPipe],
```

We start by defining some markup and the panel header:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 template: `
2   <div class="chat-window-container">
3     <div class="chat-window">
4       <div class="panel-container">
5         <div class="panel panel-default">
6
7           <div class="panel-heading top-bar">
8             <div class="panel-title-container">
9               <h3 class="panel-title">
10                 <span class="glyphicon glyphicon-comment"></span>
11                 Chat - {{currentThread.name}}
12               </h3>
13             </div>
14             <div class="panel-buttons-container">
15               <!-- you could put minimize or close buttons here -->
16             </div>
17           </div>
```

Next we show the list of messages. Here we use `ng-for` along with the `rx` pipe to iterate over our list of messages. We'll describe the individual `chat-message` component in a minute.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1   <div class="panel-body msg-container-base">
2     <chat-message
3       *ng-for="#message of messages | rx"
4         [message]="message">
5           </chat-message>
6     </div>

```

Lastly we have the message input box and closing tags:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1   <div class="panel-footer">
2     <div class="input-group">
3       <input type="text"
4         class="chat-input"
5         placeholder="Write your message here...""
6         (keydown.enter)="onEnter($event)"
7         [(ng-model)]="draftMessage.text" />
8       <span class="input-group-btn">
9         <button class="btn-chat"
10          (click)="onEnter($event)"
11          >Send</button>
12       </span>
13     </div>
14   </div>
15
16   </div>
17 </div>
18 </div>
19 </div>
20

```

The message input box is the most interesting part of this view, so let's talk about a couple of things. Our `input` tag has two interesting properties: 1. `(keydown.enter)` and 2. `[(ng-model)]`.

Handling keystrokes

Angular 2 provides a straightforward way to handle keyboard actions: we bind to the event on an element. In this case, we're binding to `keydown.enter` which says if "Enter" is pressed, call the function in the expression, which in this case is `onEnter($event)`.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1      <input type="text"
2          class="chat-input"
3          placeholder="Write your message here...""
4          (keydown.enter)="onEnter($event)"
5          [(ng-model)]="draftMessage.text" />

```

Using ng-model

As we've talked about before, Angular doesn't have a general model for two-way binding. However it can be very useful to have a two-way binding between a component and its view. As long as the side-effects are kept local to the component, it can be a very convenient way to keep a component property in sync with the view.

In this case, we're establishing a two-way bind **between the value of the input tag and draftMessage.text**. That is, if we type into the input tag, draftMessage.text will automatically be set to the value of that input. Likewise, if we were to update draftMessage.text in our code, the value in the input tag would change in the view.

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1      <input type="text"
2          class="chat-input"
3          placeholder="Write your message here...""
4          (keydown.enter)="onEnter($event)"
5          [(ng-model)]="draftMessage.text" />

```

Clicking "Send"

On our "Send" button we bind the `(click)` property to the `onEnter` function of our component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1      <span class="input-group-btn">
2          <button class="btn-chat"
3              (click)="onEnter($event)"
4              >Send</button>
5      </span>

```

The Entire ChatWindow Component

Here's the code listing for the entire ChatWindow Component:

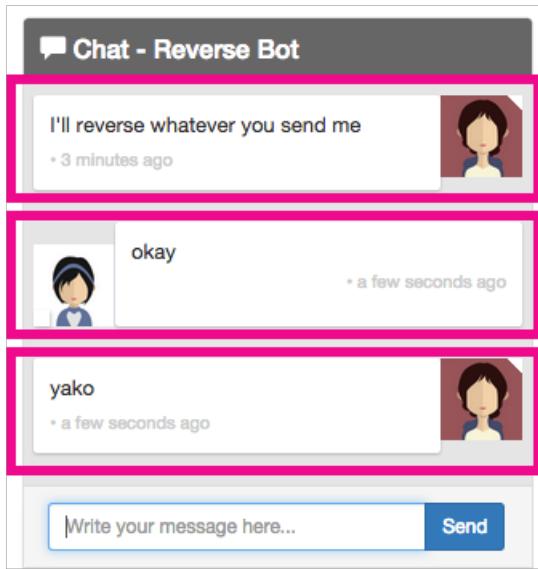
code/rxjs/chat/app/ts/components/ChatWindow.ts

```
42         <span class="input-group-btn">
43             <button class="btn-chat"
44                 (click)="onEnter($event)"
45                 >Send</button>
46         </span>
47     </div>
48 </div>
49
50     </div>
51 </div>
52 </div>
53 </div>
54 `
55 })
56 export class ChatWindow {
57     messages: Rx.Observable<any>;
58     currentThread: Thread;
59     draftMessage: Message;
60     currentUser: User;
61
62     constructor(public messagesService: MessagesService,
63                 public threadsService: ThreadsService,
64                 public userService: UserService,
65                 public el: ElementRef) {
66 }
67
68     ngOnInit(): void {
69         this.messages = this.threadsService.currentThreadMessages;
70
71         this.draftMessage = new Message();
72
73         this.threadsService.currentThread.subscribe(
74             (thread: Thread) => {
75                 this.currentThread = thread;
76             });
77
78         this.userService.currentUser
79             .subscribe(
80                 (user: User) => {
81                     this.currentUser = user;
82                 });
83 }
```

```
84     this.messages
85       .subscribe(
86         (messages: Array<Message>) => {
87           setTimeout(() => {
88             this.scrollToBottom();
89           });
90         });
91     }
92
93   onEnter(event: any): void {
94     this.sendMessage();
95     event.preventDefault();
96   }
97
98   sendMessage(): void {
99     let m: Message = this.draftMessage;
100    m.author = this.currentUser;
101    m.thread = this.currentThread;
102    m.isRead = true;
103    this.messagesService.addMessage(m);
104    this.draftMessage = new Message();
105  }
106
107  scrollToBottom(): void {
108    let scrollPane: any = this.el
109      .nativeElement.querySelector(".msg-container-base");
110    scrollPane.scrollTop = scrollPane.scrollHeight;
111  }
112
113 }
```

The ChatMessage Component

Each Message is rendered by the ChatMessage component.



ChatMessage

ChatMessage

ChatMessage

The ChatMessage Component

This component is relatively straightforward. The main logic here is rendering a slightly different view depending on if the message was authored by the current user. If the Message was **not** written by the current user, then we consider the message `incoming`.

We start by defining the `@Component`:

`code/rxjs/chat/app/ts/components/ChatWindow.ts`

```

1  @Component({
2    lifecycle: [ LifecycleEvent.onInit ],
3    properties: ["message"],
4    selector: "chat-message"
5  })

```

Setting incoming

Remember that each `ChatMessage` belongs to one `Message`. So in `onInit` we will subscribe to the `currentUser` stream and set `incoming` depending on if this `Message` was written by the current user:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1  export class ChatMessage {
2      message: Message;
3      currentUser: User;
4      incoming: boolean;
5
6      constructor(public userService: UserService) {
7      }
8
9      ngOnInit(): void {
10         this.userService.currentUser
11             .subscribe(
12                 (user: User) => {
13                     this.currentUser = user;
14                     if (this.message.author && user) {
15                         this.incoming = this.message.author.id !== user.id;
16                     }
17                 });
18     }
19
20 }
```

The ChatMessage @View

In our @View we have two interesting ideas:

1. the FromNowPipe
2. [ng-class]

First, here's the code:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```

1  @View({
2      directives: [NgIf,
3                  NgClass],
4      pipes: [FromNowPipe],
5      template: `
6          <div class="msg-container"
7              [ng-class]="{{ 'base-sent': !incoming, 'base-receive': incoming }}"
8      
```

```

9   <div class="avatar"
10    *ng-if="!incoming">
11      
12  </div>
13
14  <div class="messages"
15  [ng-class]="{'msg-sent': !incoming, 'msg-receive': incoming}">
16    <p>{{message.text}}</p>
17    <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
18  </div>
19
20  <div class="avatar"
21    *ng-if="incoming">
22      
23  </div>
24 </div>
25
26 })

```

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`⁵⁷ library. If you’d like to learn about creating your own custom pipes, checkout the [Pipes](#) chapter. You can also read the source of the `FromNowPipe` in `code/rxjs/chat/app/ts/util/FromNowPipe.ts`

We also make extensive use of `ng-class` in this view. The idea is, when we say:

```
1 [ng-class]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

We’re asking Angular to apply the `msg-receive` class if `incoming` is truthy (and apply `msg-sent` if `incoming` is falsey).

By using the `incoming` property, we’re able to display incoming and outgoing messages differently.

The Complete ChatMessage Code Listing

Here’s our completed `ChatMessage` component:

⁵⁷<http://momentjs.com/>

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
1 @Component({
2   lifecycle: [ LifecycleEvent.onInit ],
3   properties: ["message"],
4   selector: "chat-message"
5 })
6 @View({
7   directives: [NgIf,
8     NgClass],
9   pipes: [FromNowPipe],
10  template: `
11    <div class="msg-container"
12      [ng-class]="{{ 'base-sent': !incoming, 'base-receive': incoming }}"
13
14      <div class="avatar"
15        *ng-if="!incoming">
16          
17      </div>
18
19      <div class="messages"
20        [ng-class]="{{ 'msg-sent': !incoming, 'msg-receive': incoming }}"
21        <p>{{ message.text }}</p>
22        <time>{{ message.sender }} • {{ message.sentAt | fromNow }}</time>
23      </div>
24
25      <div class="avatar"
26        *ng-if="incoming">
27          
28      </div>
29    </div>
30  `
31 })
32 export class ChatMessage {
33   message: Message;
34   currentUser: User;
35   incoming: boolean;
36
37   constructor(public userService: UserService) {
38   }
39
40   OnInit(): void {
41     this.userService.currentUser
```

```

42     .subscribe(
43       (user: User) => {
44         this.currentUser = user;
45         if (this.message.author && user) {
46           this.incoming = this.message.author.id !== user.id;
47         }
48       });
49   }
50
51 }
```

The ChatNavBar Component

The last component we have to talk about is the `ChatNavBar`. In the nav-bar we'll show an unread messages count to the user.



The best way to try out the unread messages count is to use the “Waiting Bot”. If you haven't already, try sending the message ‘3’ to the Waiting Bot and then switch to another window. The Waiting Bot will then wait 3 seconds before sending you a message and you will see the unread messages counter increment.

The ChatNavBar @Component

First we define a pretty plain `@Component`:

`code/rxjs/chat/app/ts/components/ChatNavBar.ts`

```

1  @Component({
2    lifecycle: [ LifecycleEvent.onInit ],
3    selector: "nav-bar"
```

The ChatNavBar Controller

The only thing the `ChatNavBar` controller needs to keep track of is the `unreadMessagesCount`. This is slightly more complicated than it seems on the surface.

The most straightforward way would be to simply listen to `messagesService.messages` and sum the number of `Messages` where `isRead` is false. This works fine for all messages outside of the current thread. However new messages in the current thread aren't guaranteed to be marked as read by the time `messages` emits new values.

The safest way to handle this is to combine the `messages` and `currentThread` streams and make sure we don't count any messages that are part of the current thread.

We do this using the `combineLatest` operator, which we've already used earlier in the chapter:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
1  export class ChatNavBar {
2      unreadMessagesCount: number;
3
4      constructor(public messagesService: MessagesService,
5                  public threadsService: ThreadsService) {
6      }
7
8      ngOnInit(): void {
9          this.messagesService.messages
10         .combineLatest(
11             this.threadsService.currentThread,
12             (messages: Message[], currentThread: Thread) =>
13             [currentThread, messages] )
14
15         .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
16             this.unreadMessagesCount =
17                 _.reduce(
18                     messages,
19                     (sum: number, m: Message) => {
20                         let messageIsInCurrentThread: boolean = m.thread &&
21                             currentThread &&
22                             (currentThread.id === m.thread.id);
23                         if (m && !m.isRead && !messageIsInCurrentThread) {
24                             sum = sum + 1;
25                         }
26                         return sum;
27                     },
28                     0);
29         });
30     }
31 }
```

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 it's two elements.

Then we subscribe to that stream and we're *destructuring* those objects in the function call. Next we reduce over the `messages` and count the number of messages that are unread and not in the current thread.

The ChatNavBar @View

In our view, the only thing we have left to do is display our `unreadMessagesCount`:

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
1  @View({
2    directives: [],
3    template: `
4      <nav class="navbar navbar-default">
5        <div class="container-fluid">
6          <div class="navbar-header">
7            <a class="navbar-brand" href="https://ng-book.com/2">
8              </img> ng\
9 -book 2
10           </a>
11         </div>
12         <p class="navbar-text navbar-right">
13           <button class="btn btn-primary" type="button">
14             Messages <span class="badge">{{unreadMessagesCount}}</span>
15           </button>
16         </p>
17       </div>
18     </nav>
19   `
20   `
21 })
```

The Completed ChatNavBar

Here's the full code listing for ChatNavBar:

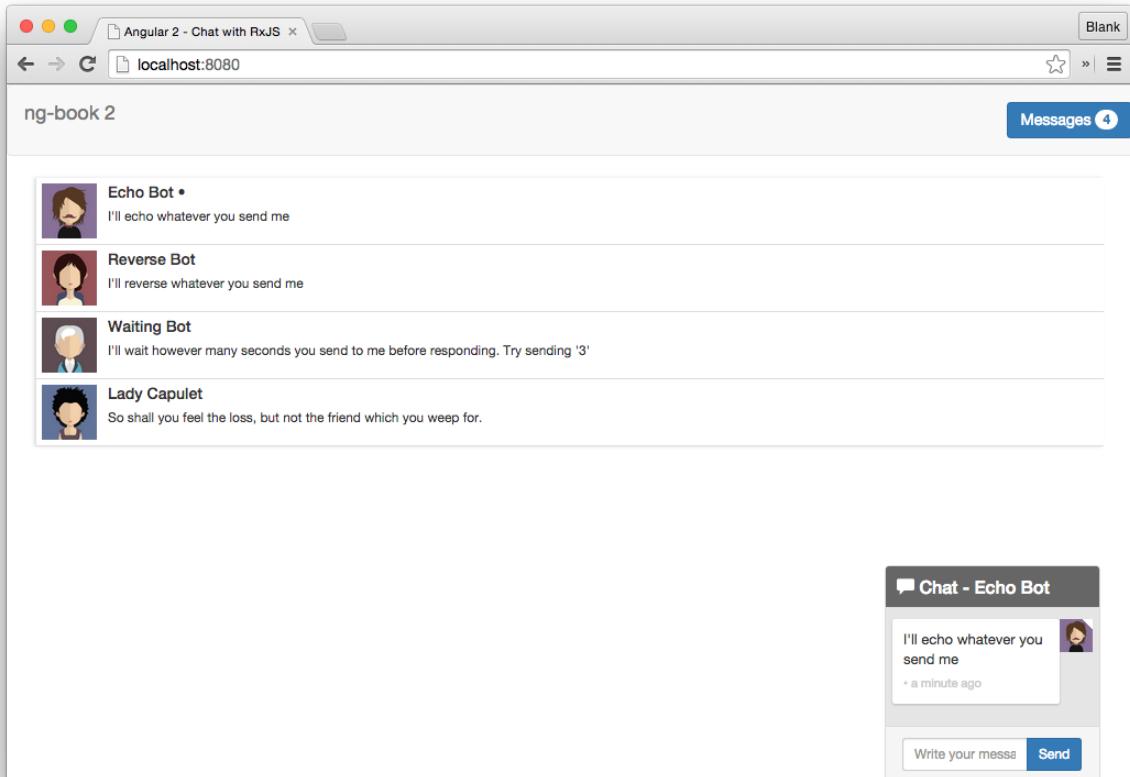
code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
1 @Component({
2   lifecycle: [ LifecycleEvent.onInit ],
3   selector: "nav-bar"
4 })
5 @View({
6   directives: [],
7   template: `
8     <nav class="navbar navbar-default">
9       <div class="container-fluid">
10         <div class="navbar-header">
11           <a class="navbar-brand" href="https://ng-book.com/2">
12             </img> ng\
13             -book 2
14           </a>
15         </div>
16         <p class="navbar-text navbar-right">
17           <button class="btn btn-primary" type="button">
18             Messages <span class="badge">{{unreadMessagesCount}}</span>
19           </button>
20         </p>
21       </div>
22     </nav>
23   `
24 }
25 })
26 export class ChatNavBar {
27   unreadMessagesCount: number;
28
29   constructor(public messagesService: MessagesService,
30               public threadsService: ThreadsService) {
31   }
32
33   OnInit(): void {
34     this.messagesService.messages
35       .combineLatest(
36         this.threadsService.currentThread,
37         (messages: Message[], currentThread: Thread) =>
38         [currentThread, messages] )
39
40     .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
41       this.unreadMessagesCount =
```

```
42     _.reduce(  
43       messages,  
44       (sum: number, m: Message) => {  
45         let messageIsInCurrentThread: boolean = m.thread &&  
46           currentThread &&  
47           (currentThread.id === m.thread.id);  
48         if (m && !m.isRead && !messageIsInCurrentThread) {  
49           sum = sum + 1;  
50         }  
51         return sum;  
52       },  
53       0);  
54     });  
55   }  
56 }
```

Summary

There we go, if we put them all together we've got a fully functional chat app!



Completed Chat Application

If you checkout code/rxjs/chat/app/ts/ChatExampleData.ts you'll see we've written a handful of bots for you that you can chat with. Here's a code excerpt from the Reverse Bot:

```
1 let rev: User      = new User("Reverse Bot", require("images/avatars/female-avata\
2 r-4.png"));
3 let tRev: Thread   = new Thread("tRev", rev.name, rev.avatarSrc);
```

[code/rxjs/chat/app/ts/ChatExampleData.ts](#)

```
1 // reverse bot
2 messagesService.messagesForThreadUser(tRev, rev)
3 .forEach( (message: Message) => {
4     messagesService.addMessage(
5         new Message({
6             author: rev,
7             text: message.text.split("").reverse().join(""),
8             thread: tRev
9         })
10    );
11 }
12 )
13 
```

```
9      })
10     );
11   });
```

Above you can see that we've subscribed to the messages for the "Reverse Bot" by using `messages-ForThreadUser`. Try writing a few bots of your own.

Next Steps

Some ways to improve this chat app would be to become stronger at RxJS and then hook it up to an actual API. We'll talk about how to make API requests in the [HTTP Chapter](#). For now, enjoy your fancy chat application!

HTTP

Introduction

Angular comes with its own HTTP library which we can use to call out to external APIs.

When we make calls to an external server, we want our user to still 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* is, historically, more tricky than dealing with synchronous code. In Javascript, there are generally three approaches to dealing with async code:

1. Callbacks
2. Promises
3. Observables

In Angular 2, the preferred method of dealing with async code is by using Observables, and so that's what we'll cover in this chapter.



There's a whole chapter on RxJS and Observables: In this chapter we're going to be using Observables and not explaining them much. If you're just starting to read this book at this chapter, you should know that there's a [whole chapter on Observables](#) that goes into RxJS in more detail.

In this chapter we're going to:

1. show a basic example of `Http`
2. create a YouTube search-as-you-type component
3. discuss API details about the `Http` library



Sample Code The complete code for the examples in this chapter can be found in the `http` folder of the sample code. That folder contains a `README.md` which gives instructions for building and running the project.

Try running the code while reading the chapter and feel free play around to get a deeper insight about how it all works.

Using angular2/http

HTTP has been split into a separate module in Angular 2. This means that to use it you need to:

1. Add the `http.js` in your `index.html`
2. Add the typings `http.d.ts`, if you're using typescript
3. import constants from `angular2/http`

Adding `http.js`

In the sample code for this chapter we've vendored the `http.js` library for you. So in our `index.html` we simply put it in a `script` tag like this:

code/http/app/index.html

```
1 <script src="/_/vendor/angular2.dev.2.0.0-alpha.37.js"></script>
2 <script src="/_/vendor/angular2-http.dev.2.0.0-alpha.37.js"></script>
```

Adding the typings

If you're using TypeScript, like we are, then you'll want to add a reference to the `http.d.ts` typings. These are included with the `angular2` bundles. For instance, in our project it looks like this:



Each reference is a single line. Be careful about copying and pasting newlines if you're reading a digital version.

code/http/app/typings/app.d.ts

```
1 /*
2  * Include types from npm
3  */
4
5 /// <reference path="../../node_modules/angular2/bundles/typings/angular2/angular\
6 r2.d.ts" />
7 /// <reference path="../../node_modules/angular2/bundles/typings/angular2/http.d\
8 .ts" />
```

import from angular2/http

In our `app.ts` we're going to import `HTTP_BINDINGS` which is a convenience collection of modules.

code/http/app/ts/app.ts

```
1 import {Component, bootstrap, View} from "angular2/angular2";
2 import {HTTP_BINDINGS} from "angular2/http";
```

When we bootstrap our app we will add `HTTP_BINDINGS` as a dependency. The effect is that we will be able to inject `Http` (and a few other modules) into our components.

```
1 bootstrap(HttpApp, [HTTP_BINDINGS]);
```

Now we can inject the `Http` service into our components (or anywhere we use DI, actually).

```
1 class MyFooComponent {
2   constructor(public http: Http) {
3   }
4
5   makeRequest(): void {
6     // do something with this.http ...
7   }
8 }
```

A Basic Request

The first thing we're going to do is make a simple GET request to the [jsonplaceholder API⁵⁸](#).

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:

⁵⁸<http://jsonplaceholder.typicode.com>

Basic Request

Make Request

```
{  
  "userId": 1,  
  "id": 1,  
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",  
  "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehende  
rit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto"  
}
```

Basic Request

Building the SimpleHTTPComponent @Component

The first thing we're going to do is import a few modules and then specify a selector for our @Component:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 import {Component, View, NgIf} from "angular2/angular2";  
2 import {Http, Response} from "angular2/http";  
3  
4 @Component({  
5   selector: "simple-http"  
6 })
```

Building the SimpleHTTPComponent @View

Next we build our view:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 @View({  
2   directives: [NgIf],  
3   template: `  
4     <h2>Basic Request</h2>  
5     <button type="button" (click)="makeRequest()">Make Request</button>  
6     <div *ng-if="loading">loading...</div>  
7     <pre>{{data | json}}</pre>  
8   `  
9 })
```

First we specify that we're going to use the `NgIf` directive.

Our template has three interesting parts:

1. The button
2. The loading indicator
3. The data

On the button we bind to `(click)` to call the `makeRequest` function in our controller, which we'll define in a minute.

We want to indicate to the user that our request is loading, so to do that we will show `loading...` if the instance variable `loading` is true, using `ng-if`.

The data is an `Object`. A great way to debug objects is to use the `json` pipe as we do here. We've put this in a `pre` tag to give us nice, easy to read formatting.

Building the SimpleHTTPComponent Controller

We start by defining a new class for our `SimpleHTTPComponent`:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 export class SimpleHTTPComponent {
2   data: Object;
3   loading: boolean;
```

We have two instance variables: `data` and `loading`. This will be used for our API return value and loading indicator respectively.

Next we define our constructor:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 constructor(public http: Http) {
2 }
```

The constructor body is empty, but we inject one key module: `Http`.



Remember that when we use the `public` keyword in `public http: Http` TypeScript will assign `http` to `this.http`. It's a shorthand for:

```
1  // other instance variables here
2  http: Http;
3
4  constructor(http: Http) {
5      this.http = http;
6  }
```

Now let's make our first HTTP request by implementing the `makeRequest` function:

`code/http/app/ts/components/SimpleHTTPComponent.ts`

```
1  makeRequest(): void {
2      this.loading = true;
3      this.http.request("http://jsonplaceholder.typicode.com/posts/1")
4          .toRx()
5          .subscribe((res: Response) => {
6              this.data = res.json();
7              this.loading = false;
8          });
9  }
```

When we call `makeRequest`, the first thing we do is set `this.loading = true`. This will turn on the loading indicator in our view.

To make an HTTP request is straightforward: we call `this.http.request` and pass the URL to which we want to make a GET request.

`http.request` returns an `ng2.EventEmitter`. The easiest way, today, to use this object is to convert it to an `Rx.Observable` using `toRx()`.



Alpha note: The API for `angular2/http` in this area is still under development. The `toRx()` may not be necessary in the future. For now, it's the most straightforward way to work with the response from `Http`.

If you're interested in more background about this issue, you can follow [angular issue #2794⁵⁹](#)

When we call `toRx()` we get an `Rx.Observable`. We can subscribe to changes (akin to using `then` from a `Promise`) using `subscribe`.

⁵⁹<https://github.com/angular/angular/issues/2794>

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1   .subscribe((res: Response) => {
2     this.data = res.json();
3     this.loading = false;
4   });

```

When our `http.request` returns the stream will emit a `Response` object. We extract the body of the response as an `Object` by using `json` and then we set `this.data` to that `Object`.

Since we have a response, we're not loading anymore so we set `this.loading = false`



`.subscribe` can also handle failures and stream completion by passing a function to the second and third arguments respectively. In a production app it would be a good idea to handle those cases, too. That is, `this.loading` should also be set to `false` if the request fails (i.e. the stream emits an error).

Full SimpleHTTPComponent

Here's what our `SimpleHTTPComponent` looks like altogether:

code/http/app/ts/components/SimpleHTTPComponent.ts

```
1 import {Component, View, NgIf} from "angular2/angular2";
2 import {Http, Response} from "angular2/http";
3
4 @Component({
5   selector: "simple-http"
6 })
7 @View({
8   directives: [NgIf],
9   template: `
10   <h2>Basic Request</h2>
11   <button type="button" (click)="makeRequest()">Make Request</button>
12   <div *ng-if="loading">loading...</div>
13   <pre>{{data | json}}</pre>
14   `
15 })
16 export class SimpleHTTPComponent {
17   data: Object;
18   loading: boolean;
19 }
```

```
20  constructor(public http: Http) {  
21  }  
22  
23  makeRequest(): void {  
24    this.loading = true;  
25    this.http.request("http://jsonplaceholder.typicode.com/posts/1")  
26      .toRx()  
27      .subscribe((res: Response) => {  
28        this.data = res.json();  
29        this.loading = false;  
30      });  
31  }  
32 }
```

Writing a YouTubeSearchComponent

The last example was a minimal way to get the data from an API server into your code. Now let's try to build a more involved example.

In this section, we're going to build a way to search YouTube as you type. When the search returns we'll show a list of video thumbnail results, along with a description and link to each video.

Here's a screenshot of what happens when I search for "cats playing ipads":

YouTube Search

cats playing ipads|



Funny Cats Playing On iPads Compilation - Funny Videos 2015

You may or may not be surprised, but there are many animals playing on tablet computer. New video funny 2015 Thanks for watching, rating the video and ...

[Watch](#)



Animals Playing On iPads Compilation

You may or may not be surprised, but there are many animals playing on tablet computer. Join Us On Facebook <http://www.facebook.com/CompilarizNo> ...

[Watch](#)



Cute cats try to catch a mouse from an iPad

Cute cats try to catch a mouse from an iPad.

[Watch](#)



Charlie The Cat - Kitten Playing iPad 2 !!! Game For Cats Cute Funny Clever Pets Bloopers

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos - <http://youtu.be/xZHwYNrfWd0> Check My Other Videos Kitten HArlem Shake ...

[Watch](#)



Cats playing "Game for Cats" with Apple iPad

Two Siberian cats like to play "Game for Cats" with Apple iPad :) Note that the iPad has Invisible Shield screen protector. Siperlankissat leikkivät



White Tiger Plays iPad - Game for Cats Gone Wild! Lions, servals, and more!

<http://www.ipadgameforcats.com> and <http://www.conservatorscenter.org/>



Cat Plays with iPad - Friskies Games for Cats

Mr. Kitty playing Cat Fishing on my girlfriends 1st gen iPad, via Friskies Games for Cats <http://www.gamesforcats.com>.



Cute Cat plays on iPad

Cute Cat plays on iPad.

[Watch](#)

Can I get my cat to write Angular 2?

For this example we're going to write several things:

1. A SearchResult object that will hold the data we want from each result
2. A YouTubeService which will manage the API request to YouTube and convert the results to a stream of SearchResult[]
3. A SearchBox component which will call out to the YouTube service as the user types
4. A SearchResultComponent which will render a specific SearchResult
5. A YouTubeSearchComponent which will encapsulate our whole YouTube searching app and render the list of results

Let's handle each part one at a time.



Patrick Stapleton has an excellent repository named [angular2-webpack-starter⁶⁰](https://github.com/angular-class/angular2-webpack-starter). This repo has an RxJS example which autocompletes Github repositories. Some of the ideas in this section are inspired from that example. It's a fantastic project with lots of examples and you should check it out.

Writing a SearchResult

First let's start with writing a basic SearchResult class. This class is just a convenient way to store the specific fields we're interested in from our search results.

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```

1  class SearchResult {
2    id: string;
3    title: string;
4    description: string;
5    thumbnailUrl: string;
6    videoUrl: string;
7
8    constructor(obj?: any) {
9      this.id          = obj && obj.id          || null;
10     this.title       = obj && obj.title       || null;
11     this.description = obj && obj.description || null;
12     this.thumbnailUrl = obj && obj.thumbnailUrl || null;
13     this.videoUrl   = obj && obj.videoUrl   ||
14                               `https://www.youtube.com/watch?v=${this.id}`;
15   }
16 }
```

This pattern of taking an `obj?: any` lets us simulate keyword arguments. The idea is that we can create a new SearchResult and just pass in an object containing the keys we want to specify.

The only thing to point out here is that we're constructing the `videoUrl` using a hard-coded URL format. You're welcome to change this to a function which takes more arguments, or use the `video id` directly in your view to build this URL if you need to.

Writing the YouTubeService

The API

For this example we're going to be using [the YouTube v3 search API⁶¹](https://developers.google.com/youtube/v3/docs/search/list).

⁶⁰<https://github.com/angular-class/angular2-webpack-starter>

⁶¹<https://developers.google.com/youtube/v3/docs/search/list>



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](#)⁶². For simplicities sake, I've registered a server key, but you should probably use a browser key if you're going to put your javascript code online.

We're going to setup two constants for our YouTubeService mapping to our API key and the API URL:

```
1 let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
2 let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

Eventually we're going to want to test our app. One of the things we find when testing is that we don't always want to test against production - we often want to test against staging or a development API.

To help with this environment configuration, one of the things we can do is **make these constants injectable**.

Why should we inject these constants instead of just using them in the normal way? Because if we make them injectable we can

1. have code that injects the right constants for a given environment at deploy time and
2. replace the injected value easily at test-time

By injecting these values, we have a lot more flexibility about their values down the line.

In order to make these values injectable, we use the `bind(...).toValue(...)` syntax like this:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 export var youTubeServiceInjectables: Array<any> = [
2   bind(YouTubeService).toClass(YouTubeService),
3   bind(YOUTUBE_API_KEY).toValue(YOUTUBE_API_KEY),
4   bind(YOUTUBE_API_URL).toValue(YOUTUBE_API_URL)
5 ];
```

Here we're specifying that we want to bind `YOUTUBE_API_KEY` "injectably" to the value of `YOUTUBE_API_KEY`. (Same for `YOUTUBE_API_URL`, and we'll define `YouTubeService` in a minute.)

If you recall, to make something available to be injected throughout our application, we need to make it a dependency at bootstrap. Since we're exporting `youTubeServiceInjectables` here we can import it in our `app.ts`

⁶²https://developers.google.com/youtube/registering_an_application#create_API_Keys

```
1 // http/app.ts
2 import {youTubeServiceInjectables} from "components/YouTubeSearchComponent";
3
4 // ....
5 // further down
6
7 bootstrap(HttpApp, [HTTP_BINDINGS, youTubeServiceInjectables]);
```

Now we can inject YOUTUBE_API_KEY instead of using the variable directly.

YouTubeService constructor

We create our YouTubeService by making a class and annotating it as @Injectable:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 @Injectable()
2 export class YouTubeService {
3   constructor(public http: Http,
4             @Inject(YOUTUBE_API_KEY) private apiKey: string,
5             @Inject(YOUTUBE_API_URL) private apiUrl: string) {
6 }
```

In the constructor we inject three things:

1. Http
2. YOUTUBE_API_KEY
3. YOUTUBE_API_URL

Notice that we make instance variables from all three arguments, meaning we can access them as `this.http`, `this.apiKey`, and `this.apiUrl` respectively.

Notice that we explicitly inject using the `@Inject(YOUTUBE_API_KEY)` notation.

YouTubeService search

Next let's implement the `search` function. `search` takes a query string and returns an `Rx.Observable` which will emit a stream of `SearchResult[]`. That is, each item emitted is an *array* of `SearchResults`.

code/http/app/ts/components/YouTubeSearchComponent.ts

```

1  search(query: string): Rx.Observable<SearchResult[]> {
2      let params: string = [
3          `q=${query}`,
4          `key=${this.apiKey}`,
5          `part=snippet`,
6          `type=video`,
7          `maxResults=10`
8      ].join("&");
9      let queryUrl: string = `${this.apiUrl}?${params}`;

```

We're building the `queryUrl` in a manual way here. We start by simply putting the query params in the `params` variable. (You can find the meaning of each of those values by [reading the search API docs⁶³](#).)

Then we build the `queryUrl` by concatenating the `apiUrl` and the `params`.

Now that we have a `queryUrl` we can make our request:

code/http/app/ts/components/YouTubeSearchComponent.ts

```

1  return this.http.get(queryUrl)
2      .toRx() // convert to Rx stream
3      .map((response: Response) => {
4          return (<any>response.json()).items.map(item => {
5              // console.log("raw item", item); // uncomment if you want to debug
6              return new SearchResult({
7                  id: item.id.videoId,
8                  title: item.snippet.title,
9                  description: item.snippet.description,
10                 thumbnailUrl: item.snippetthumbnails.high.url
11             });
12         });
13     });

```

Here we take the return value of `http.get` and immediately convert it to an `Rx.Observable` using `toRx()`.

We 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`.

⁶³<https://developers.google.com/youtube/v3/docs/search/list>



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? Here we're telling TypeScript that we're not interested in doing strict type checking here.

When working with a JSON API, we don't generally have typing definitions for the API responses, and so TypeScript won't know that the object returned even has an `items` key, so the compiler will complain.

We could call `response.json()["items"]` and then cast that to an `Array` etc., but in this case (and in creating the `SearchResult`, it's just cleaner to use an `any` type, at the expense of strict type checking

YouTubeService Full Listing

Here's the full listing of our `YouTubeService`:

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```
1 @Injectable()
2 export class YouTubeService {
3   constructor(public http: Http,
4             @Inject(YOUTUBE_API_KEY) private apiKey: string,
5             @Inject(YOUTUBE_API_URL) private apiUrl: string) {
6 }
7
8   search(query: string): Rx.Observable<SearchResult[]> {
9     let params: string = [
10       `q=${query}`,
11       `key=${this.apiKey}`,
12       `part=snippet`,
13       `type=video`,
14       `maxResults=10`
15     ].join("&");
16     let queryUrl: string = `${this.apiUrl}?${params}`;
17     return this.http.get(queryUrl)
18       .toRx() // convert to Rx stream
19       .map((response: Response) => {
20         return (<any>response.json()).items.map(item => {
21           // console.log("raw item", item); // uncomment if you want to debug
22           return new SearchResult({
```

```
23      id: item.id.videoId,
24      title: item.snippet.title,
25      description: item.snippet.description,
26      thumbnailUrl: item.snippet.thumbnails.high.url
27    });
28  });
29 });
30 }
31 }
```

Writing the SearchBox

The `SearchBox` component plays a key role in our app: it is the mediator between our UI and the `YouTubeService`.

The `SearchBox` will:

1. Watch for `keyup` on an `input` and submit a search to the `YouTubeService`
2. Emit a `loading` event when we're loading (or not)
3. Emit a `results` event when we have new results

`SearchBox` @Component Definition

Let's define our `SearchBox` @Component:

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```
1  @Component({
2    events: ["loading", "results"],
3    selector: "search-box"
4  })
```

The `selector` we've seen many times before: this allows us to craete a `<search-box>` tag.

The `events` key specifies events that will be emitted from this component. That is, we can use the `(event)="callback()"` syntax in our view to listen to events on this component. For example, here's how we will use the `search-box` tag in our view later on:

```
1 <search-box
2   (results)="updateResults($event)"
3   (loading)="loading = $event"
4 ></search-box>
```

In this example, when the `SearchBox` component emits a `loading` event, we will set the variable `loading` in the parent context. Likewise, when the `SearchBox` emits a `results` event, we will call the `updateResults()` function, with the value, in the parent's context.

In the `@Component` configuration we're simply specifying the names of the events with the strings "loading" and "results". In this example, each event will have a corresponding `EventEmitter` as an *instance variable of the controller class*. We'll implement that in a few minutes.

For now, remember that `@Component` is like the public API for our component, so here we're just specifying the name of the events, and we'll worry about implementing the `EventEmitters` later.

SearchBox @View Definition

Our `@View` is straightforward. We have one `input` tag:

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```
1 @View({
2   template: `
3     <input type="text" class="form-control" placeholder="Search" autofocus>
4   `
5 })
```

SearchBox Controller Definition

Our `SearchBox` controller is a new class:

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```
1 class SearchBox implements OnInit {
2   loading: EventEmitter = new EventEmitter();
3   results: EventEmitter = new EventEmitter();
```

We say that this class `implements OnInit` because we want to use the `onInit` lifecycle callback. If a class `implements OnInit` then the `onInit` function will be called after the first change detection check.

`onInit` is a good place to do initialization (vs. the constructor) because properties set on a component are not available in the constructor.

SearchBox Controller Definition constructor

Let's talk about the `SearchBox` constructor:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1  constructor(public youtube: YouTubeService,
2               private el: ElementRef) {
3 }
```

In our constructor we inject:

1. Our YouTubeService and
2. The element el that this component is attached to. el is an object of type ElementRef, which is an Angular wrapper around a native element.

We set both injections as instance variables.

SearchBox Controller Definition onInit

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's three things we can do to improve the user experience:

1. Filter out any empty or short queries
2. "debounce" the input, that is, don't search on every character but only after the user has stopped typing after a short amount of time
3. discard any old searches, if the user has made a new search

We could manually bind to keyup and call a function on each keyup event and then implement filtering and debouncing from there. However, there is a better way: turn the keyup events into an observable stream.

RxJS provides a way to listen to events on an element using Rx.Observable.fromEvent. We can use it like so:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1  onInit(): void {
2      // convert the `keyup` event into an observable stream
3      (<any>Rx).Observable.fromEvent(this.el.nativeElement, "keyup")
```

Notice that in fromEvent:

- the first argument is this.el.nativeElement (the native DOM element this component is attached to)

- the second argument is the string "keyup", which is the name of the event we want to turn into a stream

We can now perform some RxJS magic over this stream to turn it into `SearchResults`. Let's walk through step by step.

Given the stream of keyup events we can chain on more methods. In the next few paragraphs we're going to chain several functions on to our stream which will transform the stream. Then at the end we'll show the whole example together.

First, let's extract the value of the input tag:

```
1 .map((e: any) => e.target.value) // extract the value of the input
```

Above says, map over each keyup event, then find the event target (`e.target`, that is, our `input` element) and extract the `value` of that element. This means our stream is now a stream of strings.

Next:

```
1 .filter((text: string) => text.length > 1)
```

This `filter` means the stream will not emit any search strings for which the length is less than one. You could set this to a higher number if you want to ignore short searches.

```
1 .debounce(250)
```

`debounce` means we will throttle requests that come in faster than 250ms. That is, we won't search on every keystroke, but rather after the user has paused a small amount.

```
1 .do(() => this.loading.next(true)) // enable loading
```

Using `do` on a stream is way to perform a function mid-stream for each event, but it does not change anything in the stream. The idea here is that we've got our search, it has enough characters, and we've debounced, so now we're about to search, so we turn on `loading`.

`this.loading` is an `EventEmitter`. We "turn on" `loading` by emitting `true` as the next event. We emit something on an `EventEmitter` by calling `next`. Writing `this.loading.next(true)` means, emit a `true` event on the `loading` `EventEmitter`. When we listen to the `loading` event on this component, the `$event` value will now be `true` (we'll look more closely at using `$event` below).

```
1 .flatMapLatest((query: string) => this.youtube.search(query))
```

Essentially, by using `flatMapLatest` we're saying "ignore all search events but the most recent"⁶⁴. That is, if a new search comes in, we want to use the most recent and discard the rest.

For each query that comes in, we're going to perform a search on our `YouTubeService`.

Putting the chain together we have this:

⁶⁴Reactive experts will note that I'm handwaving here. 'flatMapLatest' has a more specific technical definition which you can [read about in the RxJS docs here](<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/flatmaplatest.md>).

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```

1  ngOnInit(): void {
2      // convert the `keyup` event into an observable stream
3      (<any>Rx).Observable.fromEvent(this.el.nativeElement, "keyup")
4          .map((e: any) => e.target.value) // extract the value of the input
5          .filter((text: string) => text.length > 1) // filter out if empty
6          .debounce(250) // only once every 250ms
7          .do(() => this.loading.next(true)) // enable loading
8          // search, discarding old events if new input comes in
9          .flatMapLatest((query: string) => this.youtube.search(query))

```

The API of RxJS can be a little intimidating because the API surface area is large. That said, we've implemented a sophisticated event-handling stream in very few lines of code!

Because we are calling out to our YouTubeService our stream is now a stream of SearchResult[]. We can subscribe to this stream and perform actions accordingly.

subscribe takes three arguments: onSuccess, onError, onCompletion.

[code/http/app/ts/components/YouTubeSearchComponent.ts](#)

```

1  .subscribe(
2      (results: SearchResult[]) => { // on sucesss
3          this.loading.next(false);
4          this.results.next(results);
5      },
6      (err: any) => { // on error
7          console.log(err);
8          this.loading.next(false);
9      },
10     () => { // on completion
11         this.loading.next(false);
12     }
13 );

```

The first argument specifies what we want to do when the stream emits a regular event. Here we emit an event on both of our EventEmitters:

1. We call `this.loading.next(false)`, indicating we've stopped loading
2. We call `this.results.next(results)`, which will emit an event containing the list of results

The second argument specifies what should happen when the stream has an event. Here we set `this.loading.next(false)` and log out the error.

The third argument specifies what should happen when the stream completes. Here we also emit that we're done loading.

SearchBox Component: Full Listing

All together, here's the full listing of our SearchBox Component:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1  @Component({
2    events: ["loading", "results"],
3    selector: "search-box"
4  })
5  @View({
6    template: `
7      <input type="text" class="form-control" placeholder="Search" autofocus>
8
9  `})
10 class SearchBox implements OnInit {
11   loading: EventEmitter = new EventEmitter();
12   results: EventEmitter = new EventEmitter();
13
14   constructor(public youtube: YouTubeService,
15             private el: ElementRef) {
16   }
17
18   //
19   // Note:
20   // The (<any>Rx) syntax below is a temporary hack to disable type checking
21   // because the Typescript definition files bundled with angular for rx don't
22   // include `fromEvent`. See:
23   // http://stackoverflow.com/questions/23217334/how-do-i-extend-a-typescript-cl\
24   ass-definition-in-a-separate-definition-file
25   //
26   ngOnInit(): void {
27     // convert the `keyup` event into an observable stream
28     (<any>Rx).Observable.fromEvent(this.el.nativeElement, "keyup")
29       .map((e: any) => e.target.value) // extract the value of the input
30       .filter((text: string) => text.length > 1) // filter out if empty
31       .debounce(250) // only once every 250ms
32       .do(() => this.loading.next(true)) // enable loading
```

```

33     // search, discarding old events if new input comes in
34     .flatMapLatest((query: string) => this.youtube.search(query))
35     // act on the return of the search
36     .subscribe(
37       (results: SearchResult[]) => { // on success
38         this.loading.next(false);
39         this.results.next(results);
40       },
41       (err: any) => { // on error
42         console.log(err);
43         this.loading.next(false);
44       },
45       () => { // on completion
46         this.loading.next(false);
47       }
48     );
49
50   }
51 }
```

Writing SearchResultComponent

The SearchBox was pretty complicated. Let's handle a **much** easier component now: the SearchResultComponent. The SearchResultComponent's job is to render a single SearchResult.

There's not really any new ideas here, so let's take it all at once:

code/http/app/ts/components/YouTubeSearchComponent.ts

```

1  @Component({
2    properties: ["result"],
3    selector: "search-result"
4  })
5  @View({
6    template: `
7      <div class="col-sm-6 col-md-3">
8        <div class="thumbnail">
9          
10         <div class="caption">
11           <h3>{{result.title}}</h3>
12           <p>{{result.description}}</p>
13           <p><a href="{{result.videoUrl}}" class="btn btn-default" role="button">Watch</a></p>
```

```

15      </div>
16      </div>
17      </div>
18  )
19 })
20 export class SearchResultComponent {
21   result: SearchResult;
22 }

```

A few things:

The `@Component` takes a single property `result`, on which we will put the `SearchResult` assigned to this component.

The `@View` shows the title, description, and thumbnail of the video and then links to the video via a button.

The `SearchResultComponent` simply stores the `SearchResult` in the instance variable `result`.



**Charlie The Cat -
Kitten Playing iPad 2
!!! Game For Cats
Cute Funny Clever
Pets Bloopers**

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos
- <http://youtu.be/xZHwYNrfWd0>
Check My Other Videos Kitten HAriem Shake ...

[Watch](#)

Writing YouTubeSearchComponent

The last component we have to implement is the `YouTubeSearchComponent`. This is the component that ties everything together.

YouTubeSearchComponent `@Component`

<code/http/app/ts/components/YouTubeSearchComponent.ts>

```

1 @Component({
2   selector: "youtube-search"
3 })

```

Single Search Result Component

Our `@Component` annotation is straightforward: use the `selector` `youtube-search`.

YouTubeSearchComponent Controller

Before we look at the `@View`, let's take a look at the `YouTubeSearchComponent` controller:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 export class YouTubeSearchComponent {
2   results: SearchResult[];
3
4   updateResults(results: SearchResult[]): void {
5     this.results = results;
6     // console.log("results:", this.results); // uncomment to take a look
7   }
8 }
```

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 `@View`.

YouTubeSearchComponent `@View`

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

The first thing we need to do is tell Angular what directives we'll be using:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 @View({
2   directives: [CORE_DIRECTIVES, SearchBox, SearchResultComponent],
```

Here we use the `CORE_DIRECTIVES` (which gives us `ng-if` and `ng-for`) along with our two custom components: `SearchBox` and `SearchResultComponent`.

Next lets look at our template. Let's build some basic structure and show the loading gif next to the header:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 template: `
2 <div class='container'>
3   <div class="page-header">
4     <h1>YouTube Search
5     <img
6       style="float: right;"
7       *ng-if="loading"
8       src=${loadingGif} />
9   </h1>
10  </div>
```



Notice that our `img` has a `src` of `${loadingGif}` - that `loadingGif` variable came from a `require` statement earlier in the program. Here we're taking advantage of webpack's image loading feature. If you want to learn more about how this works, take a look at the webpack config in the sample code for this chapter or checkout `image-webpack-loader`⁶⁵.

We only want to show this loading image if `loading` is true, so we use `ng-if` to implement that functionality.

Next lets look at the markup for the search-box:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 <div class="row">
2   <div class="input-group input-group-lg col-md-12">
3     <search-box
4       (loading)="loading = $event"
5       (results)="updateResults($event)"
6       ></search-box>
7     </div>
8   </div>
```

The interesting part here is how we bind to the `loading` and `results` events. Notice, that we use the `(event)="action()"` syntax here.

For the `loading` event, we run the expression `loading = $event`. `$event` will be substituted with the value of the event that is emitted from the `EventEmitter`. That is, in our `SearchBox` component, when we call `this.loading.next(true)` then `$event` will be true.

⁶⁵<https://github.com/tcoopman/image-webpack-loader>

Similarly, for the results event, 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/app/ts/components/YouTubeSearchComponent.ts

```
1 <div class="row">
2   <search-result
3     *ng-for="#result of results"
4     [result]="result">
5   </search-result>
6 </div>
7 </div>
8   ` 
9 })
```

YouTubeSearchComponent Full Listing

Here's the full listing for the YouTubeSearchComponent:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
1 @Component({
2   selector: "youtube-search"
3 })
4 @View({
5   directives: [CORE_DIRECTIVES, SearchBox, SearchResultComponent],
6   template: `
7     <div class='container'>
8       <div class="page-header">
9         <h1>YouTube Search
10        <img
11          style="float: right;" 
12          *ng-if="loading"
13          src=${loadingGif} />
14      </h1>
15    </div>
16
17    <div class="row">
18      <div class="input-group input-group-lg col-md-12">
19        <search-box
20          (loading)="loading = $event"
```

```
21         (results)="updateResults($event)"
22     > </search-box>
23   </div>
24 </div>
25
26   <div class="row">
27     <search-result
28       *ng-for="#result of results"
29       [result]="result">
30     </search-result>
31   </div>
32 </div>
33   `
34 })
35 export class YouTubeSearchComponent {
36   results: SearchResult[];
37
38   updateResults(results: SearchResult[]): void {
39     this.results = results;
40     // console.log("results:", this.results); // uncomment to take a look
41   }
42 }
```

There we have it! A functional search-as-you-type implemented for YouTube videos! Try running it from the code examples if you haven't already.

angular/http API

Of course, all of the HTTP requests we've made so far have simply been GET requests. It's important that we know how we can make other requests too.

Making a POST request

Making POST request with angular/http is very much like making a GET request except that we have one additional parameter: a body.

jsonplaceholder API⁶⁶ also provides a convenient URL for testing our POST requests, so let's use it for a POST:

⁶⁶<http://jsonplaceholder.typicode.com>

code/http/app/ts/components/MoreHTTPRequests.ts

```
1 makePost(): void {
2     this.loading = true;
3     this.http.post(
4         "http://jsonplaceholder.typicode.com/posts",
5         JSON.stringify({
6             body: "bar",
7             title: "foo",
8             userId: 1
9         }))
10    .toRx()
11    .subscribe((res: Response) => {
12        this.data = res.json();
13        this.loading = false;
14    });
15 }
```

Notice in the second argument we're taking an Object and converting it to a JSON string using `JSON.stringify`.

PUT / PATCH / DELETE / HEAD

There are a few other fairly common HTTP requests and we call them in much the same way.

- `http.put` and `http.patch` map to PUT and PATCH respectively and both take a URL and a body
- `http.delete` and `http.head` map to DELETE and HEAD respectively and both take a URL (no body)

Here's how we might make a DELETE request:

code/http/app/ts/components/MoreHTTPRequests.ts

```
1 makeDelete(): void {
2     this.loading = true;
3     this.http.delete("http://jsonplaceholder.typicode.com/posts/1")
4         .toRx()
5         .subscribe((res: Response) => {
6             this.data = res.json();
7             this.loading = false;
8         });
9 }
```

RequestOptions



Alpha warning angular/http is still under heavy development and so the API here may change in future versions.

As of writing, the team is hard at work on the documentation for angular/http. Until then, you can find some documentation on [RequestOptions by reading the source directly](#)⁶⁷

All of the http methods we've covered so far also take an optional last argument: RequestOptions. The RequestOptions object encapsulates:

- method
- headers
- body
- mode
- credentials
- cache
- url
- search

Let's say we want to craft a GET request that uses a special X-API-TOKEN header. We can create a request with this header like so:

code/http/app/ts/components/MoreHTTPRequests.ts

```
1 makeHeaders(): void {
2   let headers: Headers = new Headers();
3   headers.append("X-API-TOKEN", "ng-book");
4
5   let opts: RequestOptions = new RequestOptions();
6   opts.headers = headers;
7
8   this.http.get("http://jsonplaceholder.typicode.com/posts/1", opts)
9     .toRx()
10    .subscribe((res: Response) => {
11      this.data = res.json();
12    });
13 }
```

⁶⁷https://github.com/angular/angular/blob/master/modules/angular2/src/http/base_request_options.ts

Summary

`angular/http` is still young but it's already full-featured enough for a wide variety of APIs.

One of the great things about `angular/http` is that it has support for mocking the backend which is very useful in testing. To learn about testing HTTP, flip on over to [the testing chapter](#).

Routing

In web development, *routing* means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

For instance, if we visit the / path of a website, we may be visiting the **home route** of that website. Or if we visit /about we want to render the “about page”, and so on.

Why 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.js⁶⁸](#) you might write something like this:

```
1 var express = require('express');
2 var router = express.Router();
3
4 // define the about route
5 router.get('/about', function(req, res) {
6   res.send('About us');
7 });


```

Or with [Ruby on Rails⁶⁹](#) you might have:

```
1 # routes.rb
2 get '/about', to: 'pages#about'
3
4 # PagesController.rb
5 class PagesController < ApplicationController::Base
6   def about
7     render
8   end
9 end


```

The pattern varies per framework, but in both of these cases you have a **server** that accepts a request and *routes* to a **controller** and the controller runs a specific **action**, depending on the path and parameters.

Client-side routing is very similar in concept but different in implementation. With client-side routing **we're not necessarily making a request to the server** on every URL change. With our Angular apps, we refer to them as "Single Page Apps" (SPA) because our server only gives us a single page and it's our JavaScript that renders the different pages.

So how can we have different routes in our JavaScript code?

⁶⁸<http://expressjs.com/guide/routing.html>

⁶⁹<http://rubyonrails.org/>

The beginning: using anchor tags

Client-side routing started out with a clever hack: Instead of using the page page, instead use the *anchor tag* as the client-side URL.

As you may already know, anchor tags were traditionally used to link directly to a place *within* the webpage and make the browser scroll all the way to where that anchor was defined. For instance, if we define an anchor tag in an HTML page:

```
1 <!-- ... lots of page content here ... -->
2 <a name="about"><h1>About</h1></a>
```

And we visited the URL `http://something/#about`, the browser would jump straight to that H1 tag that identified by the `about` anchor.

The clever move for client-side frameworks used for SPAs was to take the anchor tags and use them represent the routes within the app by formatting them as paths.

For example, the `about` route for an SPA would be something like `http://something/#/about`. This is what is known as **hash-based routing**.

What's neat about this trick is that it looks like a "normal" URL because we're starting our anchor with a slash (`/about`).

The evolution: HTML5 client-side routing

With the introduction of HTML5, browsers acquired the ability to programmatically create new browser history entries that change the displayed URL *without the need for a new request*.

This is achieved using the `history.pushState` method that exposes the browser's navigational history to JavaScript.

So now, instead of relying on the anchor hack to navigate routes, modern frameworks can rely on `pushState` to perform history manipulation without reloads.



Angular 1 Note: This way of routing already works in Angular 1, but it needs to be explicitly enabled using `$locationProvider.html5Mode(true)`.

In Angular 2, however, the HTML5 is the default mode. Later in this chapter we show how to change from HTML5 mode to the old anchor tag mode.



There's two things you need to be aware of when using HTML5 mode routing, though

1. Not all browsers support HTML5 mode routing, so if you need to support older browsers you might be stuck with hash-based routing for a while.
2. **The server has to support HTML5 based routing.**

It may not be immediately clear why the server has to support HTML5 based-routing, we'll talk more about why later in this chapter.

Writing our first routes

In Angular we configure routes by mapping *paths* to the component that will handle them.

Let's create a small app that has multiple routes. On this sample application we will have 3 routes:

- A main page route, using the `/#/home` path;
- An about page, using the `/#/about` path;
- A contact us page, using the `/#/contact` path;

And when the user visits the root path (`/#/`), it will redirect to the home path.

Components of Angular 2 routing

There are three main components that we use to configure routing in Angular:

- `RouteConfig annotation` describes the routes our application supports
- `RouterOutlet` is a “placeholder” component that gets expanded to each route’s content
- `RouterLink` is used to link to routes

Let's look at each one more closely.

RouteConfig

To define routes for our application, we annotate our main component with the `RouteConfig` annotation:

code/routes/basic/app/ts/app.ts

```
45 @RouteConfig([
46   { path: '/', name: 'root', redirectTo: ['/Home'] },
47   { path: '/home', name: 'Home', component: HomeComponent },
48   { path: '/about', name: 'About', component: AboutComponent },
49   { path: '/contact', name: 'Contact', component: ContactComponent },
50   { path: '/contactus', name: 'ContactUs', redirectTo: ['/Contact'] },
51 ])
```

Notice a few things about the routes:

- path specifies the URL this route will handle
- name uniquely identifies the route and is used to navigate to it either by using a RouterLink, a redirectTo or from the Router API
- component is what ties a given route path to a component that will handle the route
- the optional redirectTo is used to redirect a given path to an existing route

As a summary, the goal of routes is to specify which component will handle a given path.

Redirections

When we use redirectTo on a route definition, it will tell the router that when we visit the path of the route, we want the browser to be redirected to another route.

In our sample code above, if we visit the root path at <http://localhost:8080/#/>⁷⁰, we'll be redirected to the default route Home.

Another example is the ContactUs route:

```
1 { path: '/contactus', name: 'ContactUs', redirectTo: ['/Contact'] },
```

In this case, if we visit the URL <http://localhost:8080/#/contactus>⁷¹, we'll see that the browser redirects to /contact.



Sample Code The complete code for the examples in this section can be found in the routes/basic folder of the sample code. That folder contains a README.md, which gives instructions for building and running the project.

Try running the code while reading this section and feel free play around to get a deeper insight about how it all works.

⁷⁰<http://localhost:8080/#/>

⁷¹<http://localhost:8080/#/contactus>

RouterOutlet

Our component @View has a template which specifies some div structure, a section for Navigation, and a directive called router-outlet.

When we change routes, we want to keep our outer template but only substitute out the “inner section” of the page.

In order to describe to Angular where in our page we want to render the contents for each route, we use the RouterOutlet directive.

The router-outlet element indicates where the contents of each route component will be rendered.

In order to use it, you need to declare the ROUTER_DIRECTIVES as one of the directives your component needs and add a <router-outlet> tag to your HTML:



In Angular2 router, ROUTER_DIRECTIVES is an array that holds all router directives you need to import, including RouterOutlet.

code/routes/basic/app/ts/app.ts

```
1  @Component({
2    selector: 'router-app',
3    directives: [ROUTER_DIRECTIVES],
4    template: `
5      <div>
6        <nav>
7          <a>Navigation:</a>
8          <ul>
9            <li><a [routerLink]="/Home">Home</a></li>
10           <li><a [routerLink]="/About">About</a></li>
11           <li><a [routerLink]="/Contact">Contact us</a></li>
12         </ul>
13       </nav>
14
15       <router-outlet></router-outlet>
16     </div>
17   `
```

If we look at the template contents above, you will note the router-outlet element right below the navigation menu. When we visit /home, that's where HomeComponent template will be rendered. The same happens for the other components.

RouterLink

Now that we know where route templates will be rendered, how do we tell Angular 2 to navigate to a given route?

We might try linking to the routes directly using pure HTML:

```
1 <a href="/#/home">Home</a>
```

But if we do this, we'll notice that clicking the link triggers a page reload and that's definitely not what we want when programming single page apps.

To solve this problem, Angular 2 provides a solution that can be used to link to routes **with no page reload**: the `RouterLink` directive.

This directive allows you to write links using a special syntax:

code/routes/basic/app/ts/app.ts

```
1 <li><a [routerLink]="['/Home']">Home</a></li>
2 <li><a [routerLink]="['/About']">About</a></li>
3 <li><a [routerLink]="['/Contact']">Contact us</a></li>
```

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 name as the first element, like `"['/Home']"` or `"['/About']"` that will indicate which route to navigate to when we click the element.

It might seem a little odd that the value of `routerLink` is a string with an array containing a string (`"['/Home']"`, for example). This is because there are more things you can provide when linking to routes, but we'll look at this into more detail when we talk about child routes and route parameters.

For now, we're only using routes names from the root app component.

Putting it all together

So now that we have all the basic pieces, let's make them work together to transition from one route to the other.

The first thing we need to write for our application is the `index.html` file.

Here's the full code for that:

code/routes/basic/app/index.html

```
1 <!doctype html>
2 <html>
3   <head>
4     <base href="/">
5     <title>ng-book 2: Angular 2 Router</title>
6
7     {% for (var css in o.htmlWebpackPlugin.files.css) { %}
8       <link href="{%o.htmlWebpackPlugin.files.css[css] %}" rel="stylesheet">
9     {% } %}
10    </head>
11    <body>
12      <router-app></router-app>
13      <script src="/core.js"></script>
14      <script src="/vendor.js"></script>
15      <script src="/bundle.js"></script>
16    </body>
17 </html>
```



The section describing `htmlWebpackPlugin` comes from the [webpack module bundler](#)⁷². We're using webpack in this chapter because it's a tool for bundling your assets

The code should be familiar by now, with the exception of this line:

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

This line declares the `base` HTML tag. This tag is traditionally used to tell the browser where to look for images and other resources declared using relative paths.

It turns out Angular Router also relies on this tag to determine how to construct its routing information.

For instance, if we have a route with a path of `/hello` and our `base` element declares `href="/app"`, the application will use `/app/#` as the concrete path.

Sometimes though, coders of an Angular application don't have access to the `head` section of the application HTML. This is true for instance, when reusing headers and footers of a larger, pre-existing application.

Fortunately there is a workaround for this cases. You can declare the application base path programmatically, when bootstrapping the Angular application:

⁷²<https://webpack.github.io/>

```
1 bootstrap(RoutesDemoApp, [
2   ROUTER_PROVIDERS,
3   provide(APP_BASE_HREF, {useValue: '/'})
4 ]);
```

Injecting the result from `provide(APP_BASE_HREF, {useValue: '/'})` is the equivalent of using `<base href="/">` on our application HTML header.

Creating the components

Before we get to the main app component, let's create 3 simple components, one for each of the routes.

HomeComponent

The HomeComponent will just have an `h1` tag that says "Welcome!". Here's the full code for our HomeComponent:

`code/routes/basic/app/ts/components/HomeComponent.ts`

```
1 /**
2  * Angular
3  */
4 import {Component} from 'angular2/core';
5
6 @Component({
7   selector: 'home',
8   template: `<h1>Welcome!</h1>`
9 })
10 export class HomeComponent {
```

AboutComponent

Similarly, the AboutComponent will just have a basic `h1`:

code/routes/basic/app/ts/components/AboutComponent.ts

```
1  /*
2   * Angular
3   */
4  import {Component} from 'angular2/core';
5
6  @Component({
7    selector: 'about',
8    template: `<h1>About</h1>`
9  })
10 export class AboutComponent {
11 }
```

ContactComponent

And, likewise with AboutComponent:

code/routes/basic/app/ts/components/ContactComponent.ts

```
1  /*
2   * Angular
3   */
4  import {Component} from 'angular2/core';
5
6  @Component({
7    selector: 'contact',
8    template: `<h1>Contact Us</h1>`
9  })
10 export class ContactComponent {
11 }
```

Nothing really very interesting about those components, so let's move on to the main `app.ts` file.

Application component

Now we need to create the root-level “application” component that will tie everything together.

We start with the imports we'll need, both from the core and router bundles:

code/routes/basic/app/ts/app.ts

```
5 import {provide, Component} from 'angular2/core';
6 import {bootstrap} from 'angular2/platform/browser';
7 import {
8   ROUTER_DIRECTIVES,
9   ROUTER_PROVIDERS,
10  HashLocationStrategy,
11  LocationStrategy,
12  RouteConfig,
13 } from 'angular2/router';
```

Next step is to import the three components we created above:

code/routes/basic/app/ts/app.ts

```
18 import {HomeComponent} from 'components/HomeComponent';
19 import {AboutComponent} from 'components/AboutComponent';
20 import {ContactComponent} from 'components/ContactComponent';
```

Now let's get to the real component code. We start with the declaration of the component selector, directives and template:

code/routes/basic/app/ts/app.ts

```
27 @Component({
28   selector: 'router-app',
29   directives: [ROUTER_DIRECTIVES],
30   template: `
31     <div>
32       <nav>
33         <a>Navigation:</a>
34         <ul>
35           <li><a [routerLink]="/Home">Home</a></li>
36           <li><a [routerLink]="/About">About</a></li>
37           <li><a [routerLink]="/Contact">Contact us</a></li>
38         </ul>
39       </nav>
40
41       <router-outlet></router-outlet>
42     </div>
43   `)
```

For this component, we're going to use two router directives: RouterOutlet and the RouterLink. Those directives, along with all other *common* router directives are declared on a special array called ROUTER_DIRECTIVES. When we import that constant on our directives area, it's basically importing all of those directives for us, without the need to do it one by one.

As a recap, the RouterOutlet directive is then used to indicate where in our template the route contents should be rendered. That's represented by the <router-outlet></router-outlet> snippet in our template code.

The RouterLink directive is used to create navigation links to our routes:

code/routes/basic/app/ts/app.ts

```
35   <li><a [routerLink]="/Home">Home</a></li>
36   <li><a [routerLink]="/About">About</a></li>
37   <li><a [routerLink]="/Contact">Contact us</a></li>
```

This will make Angular take ownership of the clicking event and redirect to the right place based on the route definition.

Next, we declare all the routes by using the RouteConfig annotation:

code/routes/basic/app/ts/app.ts

```
45 @RouteConfig([
46   { path: '/', name: 'root', redirectTo: ['/Home'] },
47   { path: '/home', name: 'Home', component: HomeComponent },
48   { path: '/about', name: 'About', component: AboutComponent },
49   { path: '/contact', name: 'Contact', component: ContactComponent },
50   { path: '/contactus', name: 'ContactUs', redirectTo: ['/Contact'] },
51 ])
```

And finally, we declare our RoutesDemoApp class:

code/routes/basic/app/ts/app.ts

```
1 class RoutesDemoApp {
2 }
```

In the last section of the app.ts file, we bootstrap the application:

code/routes/basic/app/ts/app.ts

```
55 bootstrap(RoutesDemoApp, [
56   ROUTER_PROVIDERS,
57   provide(LocationStrategy, {useClass: HashLocationStrategy})
58 ]);
```

Just like we have been doing so far, we are now bootstrapping the app and telling that RoutesDemoApp is the root component.

The difference now is that we're providing a second parameter to the `bootstrap` method. This second parameter is an array of injectables you're injecting into our application.

The first thing we're injecting is the `ROUTER_PROVIDERS` constant. This constant is an array of all the router classes, like `RouteRegistry` and `Location`, that are necessary to make routing work.

The second seems a little more complicated:

```
1 provide(LocationStrategy, {useClass: HashLocationStrategy})
```

Let's take an in depth look of what we want to achieve with this line.

Routing strategies

The way the Angular application parses and creates paths from and to route definitions is now *location strategy*.



In Angular 1 this is called *routing modes* instead

The default strategy is `PathLocationStrategy`, which is what we call HTML5 routing. While using this strategy, routes are represented by regular paths, like `/home` or `/contact`.

We can change the location strategy used for our application by binding the `LocationStrategy` class to a new, concrete strategy class.

Instead of using the default `PathLocationStrategy` we can also use the `HashLocationStrategy`.

The reason we're using the hash strategy as a default is because if we were using HTML5 routing, our URLs would end up being regular paths (that is, not using hash/anchor tags).

This way, the routes would work when you click a link and navigate on the client side, let's say from `/about` to `/contact`.

If we were to refresh the page, instead of asking the server for the root URL, which is what is being served, instead we'd be asking for /about or /contact. Because there's no known page at /about the server would return a 404.

This default strategy works with hash based paths, like /#/home or /#/contact that the server understands as being the / path. (This is also the default mode in Angular 1.)



Let's say you want to use HTML5 mode in production, what can you do?

In order to use HTML5 mode routing, you have to configure your server to redirect every "missing" route to the root URL.

In the routes/basic project we've included a script you can use to develop with webpack-dev-server and use HTML5 paths at the same time.

To use it cd routes/basic and run node html5-dev-server.js.

Finally, in order to make our example application work with this new strategy, first we have to import LocationStrategy and HashLocationStrategy:

code/routes/basic/app/ts/app.ts

```
5 import {provide, Component} from 'angular2/core';
6 import {bootstrap} from 'angular2/platform/browser';
7 import {
8   ROUTER_DIRECTIVES,
9   ROUTER_PROVIDERS,
10  HashLocationStrategy,
11  LocationStrategy,
12  RouteConfig,
13 } from 'angular2/router';
```

and then just add that link to the bootstrap section:

code/routes/basic/app/ts/app.ts

```
55 bootstrap(RoutesDemoApp, [
56   ROUTER_PROVIDERS,
57   provide(LocationStrategy, {useClass: HashLocationStrategy})
58 ]);
```



You could write your own strategy if you wanted to. All you need to do is extend the LocationStrategy class and implement the methods. A good way to start is reading the Angular 2 source for the HashLocationStrategy or PathLocationStrategy classes.

Path location strategy

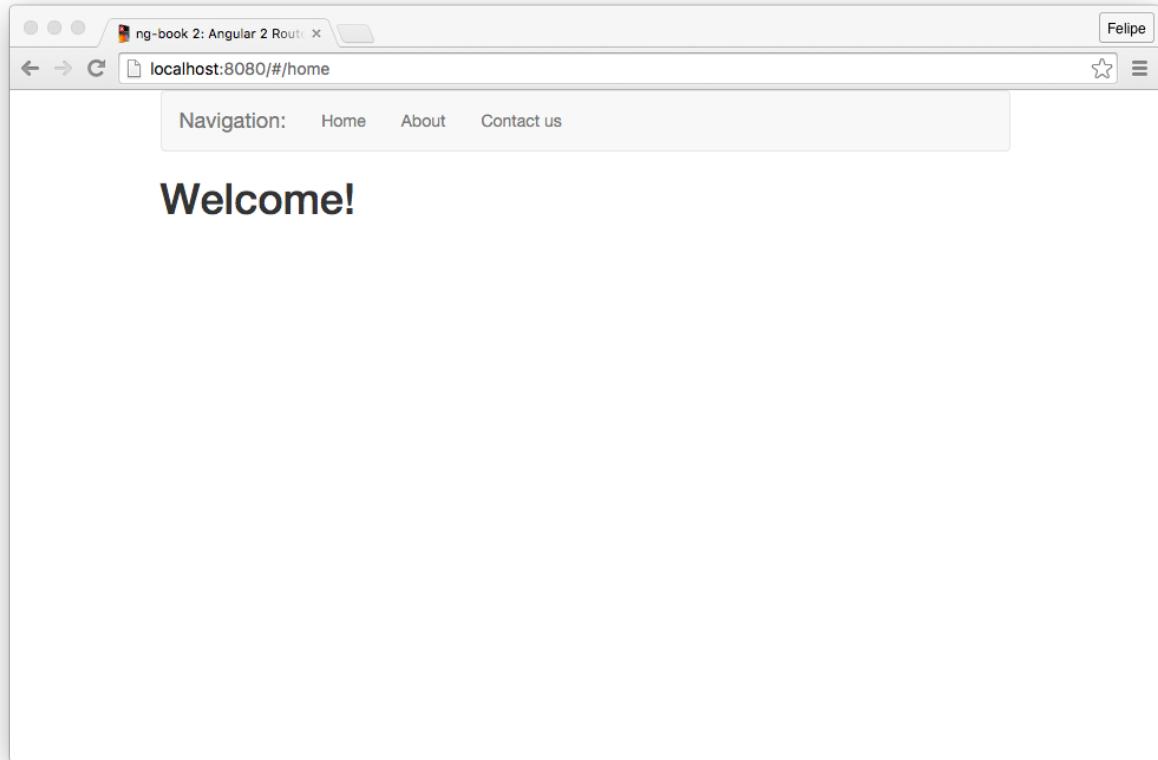
In our sample application folder, you'll find a file called `app/ts/app.html5.ts`.

If we want to play with the default `PathLocationStrategy`, we just need to copy the contents of that file to `app/ts/app.ts`, then reload the application.

Running the application

You can now go into the application root folder (`code/routes`) and run `npm run server` to boot the application.

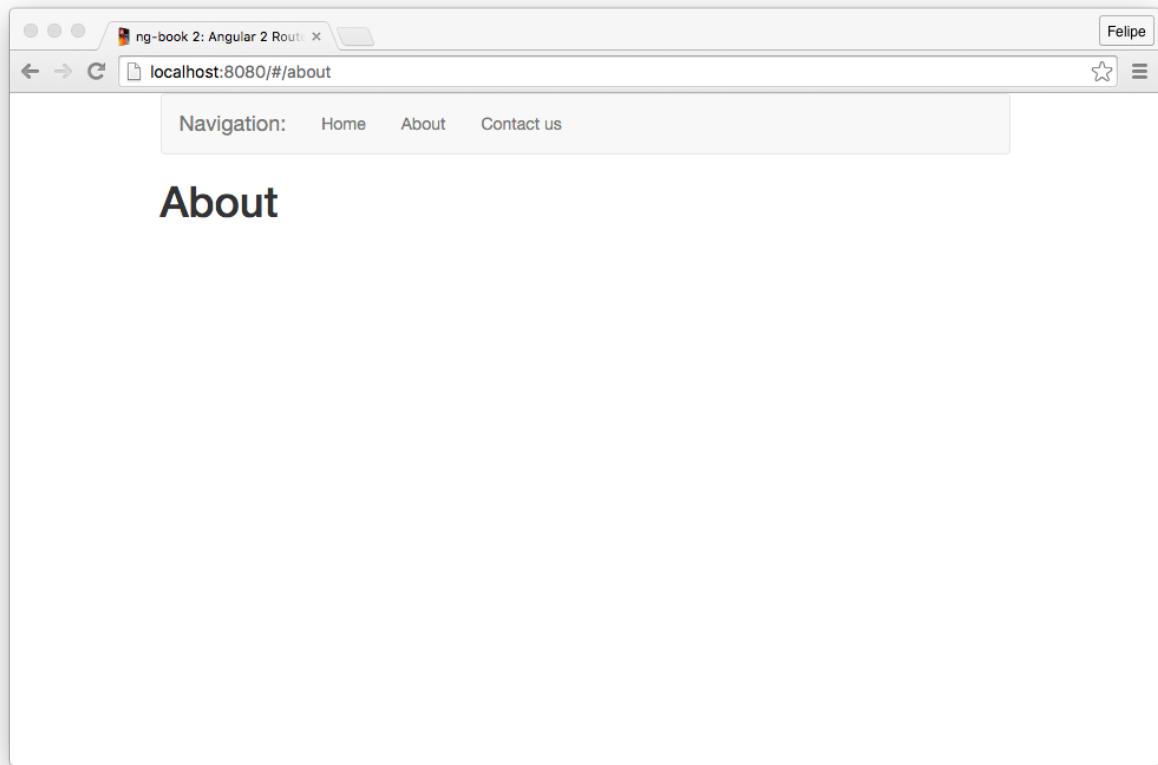
When you type `http://localhost:8080/` into your browser you should see the home route rendered:



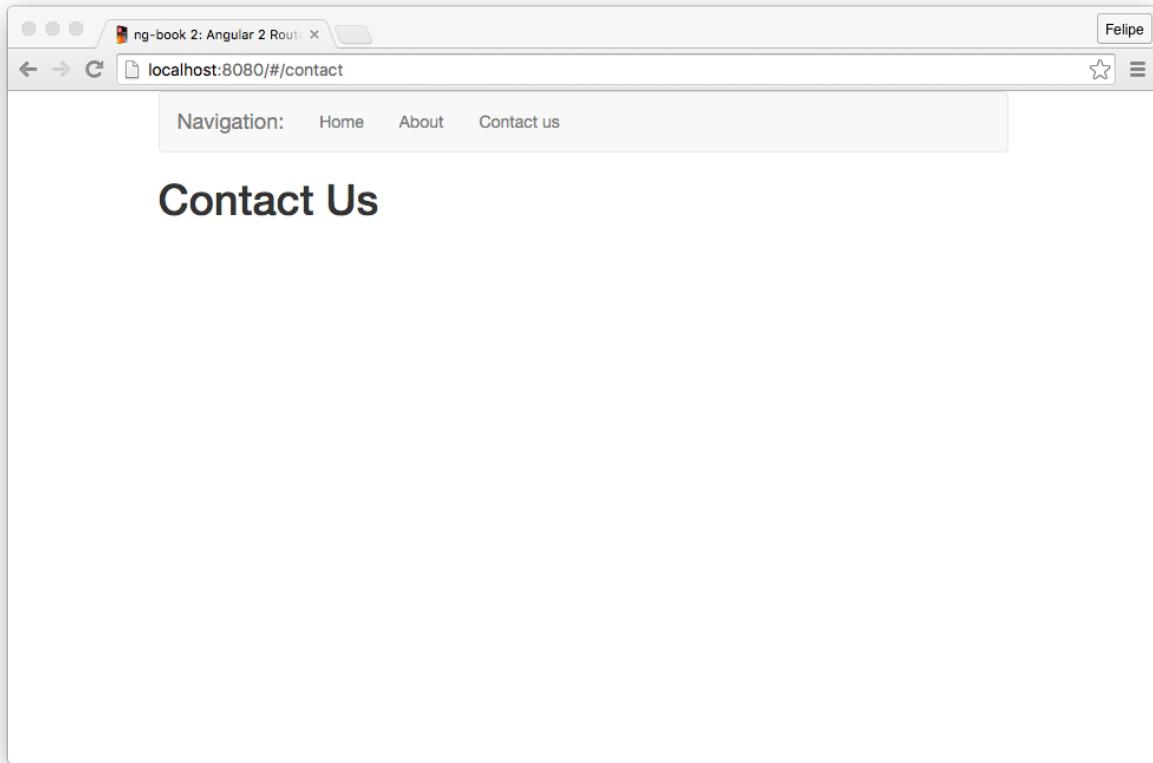
Home Route

Notice that the URL in the browser got redirected to `http://localhost:8080/`

Now clicking each link will render the appropriate routes:



About Route



Contact Us Route

Route Parameters

In our apps we often want to navigate to a specific resource. For instance, say we had a news website and we had many articles. Each article may have an ID, and if we had an article with ID 3 then we might navigate to that article by visiting the URL:

/articles/3

And if we had an article with an ID of 4 we would access it at

/articles/4

and so on.

Obviously we're not going to want to write a route for each article, but instead we want to use a variable, or *route parameter*. We can specify that a route takes a parameter by putting a colon : in front of the path segment like this:

/route/:param

So in our example news site, we might specify our route as:

```
/articles/:id
```

To add a parameter to our router configuration, we specify the route path like this:

```
code/routes/music/app/ts/app.ts
```

```
46 @RouteConfig([
47   { path: '/', name: 'root', redirectTo: [ 'Search' ] },
48   { path: '/search', name: 'Search', component: SearchComponent },
49   { path: '/artists/:id', name: 'Artists', component: ArtistComponent },
50   { path: '/tracks/:id', name: 'Tracks', component: TrackComponent },
51   { path: '/albums/:id', name: 'Albums', component: AlbumComponent },
52 ])
```

When we visit the route `/artist/123`, the `123` part will be passed as the `id` route parameter to our route.

But how can we retrieve the parameter for a given route? That's where we use route parameters.

RouteParams

In order to use route parameters, we need to first import `RouteParams`:

```
1 import {RouteParams} from "angular2/router";
```

Next, we inject the `RouteParams` into the constructor. Let's say we have a `RouteConfig` that specifies the following:

```
1 @RouteConfig([
2   { path: "/articles/:id", name: "articles", component: ArticleComponent }
3 ])
```

Then when we write the `ArticleComponent`, we add the `RouteParams` as one of the constructor arguments:

```
1 export class ArticleComponent {  
2   id: string;  
3  
4   constructor(private routeParams: RouteParams) {  
5     this.id = routeParams.get("id");  
6   }  
7 }
```

And then when we visit `/articles/230`, our component's `id` attribute should receive 230.

Music Search App

Let's now work on a more complex application. We will build a music search application that has the following features:

1. Search for tracks that match a given term
2. Show matching tracks in a grid
3. Show singer details when the singer name is clicked
4. Show album details and show a list of tracks when the album name is clicked
5. Show song details allow the user to play a preview when the song name is clicked

Sportify music for active people

Search

rhapsody in blue

Results



George Gershwin
Rhapsody in Blue

Gershwin: Rhapsody in Blue/An American in Paris



George Gershwin
Rhapsody in Blue

Gershwin Plays Gershwin: The Piano Rolls



George Gershwin
Rhapsody in Blue

Gershwin: Rhapsody in Blue / An American in Paris



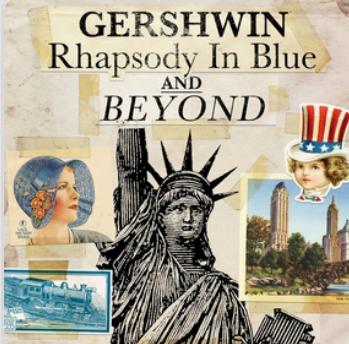
George Gershwin
Rhapsody in Blue

Gershwin: Piano Concerto in F, Rhapsody in



George Gershwin
Rhapsody in Blue

Gershwin: Rhapsody in Blue; Piano Concerto in



George Gershwin
Rhapsody in Blue

Gerswin - Rhapsody in Blue and Beyond

The Search View of our Music App

The routes we will need for this application will be:

- /search - search form and results
- /artists/:id - artist info, represented by a Spotify ID

- /albums/: id - album info, with a list of tracks using the Spotify ID
- /tracks/: id - track info and preview, also using the Spotify ID



Sample Code The complete code for the examples in this section can be found in the routes/music folder of the sample code. That folder contains a README.md, which gives instructions for building and running the project.

Try running the code while reading this section and feel free play around to get a deeper insight about how it all works.

We will use the [Spotify API⁷³](#) to get information about tracks, artists and albums.

First Steps

The first file we need work on is app.ts. Let's start by importing classes we'll use from Angular:

code/routes/music/app/ts/app.ts

```
1  /*
2   * Angular
3   */
4  import {
5    Component,
6    provide
7  } from 'angular2/core';
8  import {bootstrap} from 'angular2/platform/browser';
9  import {HTTP_PROVIDERS} from 'angular2/http';
10 import {
11   APP_BASE_HREF,
12   ROUTER_DIRECTIVES,
13   ROUTER_PROVIDERS,
14   ROUTER_PRIMARY_COMPONENT,
15   HashLocationStrategy,
16   LocationStrategy,
17   Router,
18   RouteConfig,
19 } from 'angular2/router';
```

Now that we have the imports there, let's think about the components we'll use for each route.

⁷³<https://developer.spotify.com/web-api>

- For the `Search` route, we'll create a `SearchComponent`. This component will talk to the Spotify API to perform the search and then display the results on a grid.
- For the `Artists` route, we'll create an `ArtistComponent` which will show the artist's information
- For the `Albums` route, we'll create an `AlbumComponent` which will show the list of tracks in the album
- For the `Tracks` route, we'll create a `TrackComponent` which will show the track and let us play a preview of the song

Since this new component will need to interact with the Spotify API, it seems like we need to build a service that uses the `http` module to call out to the API server.

Everything in our app depends on the data, so let's build the `SpotifyService` first.

The SpotifyService



You can find the full code for the `SpotifyService` in the `routes/music/app/ts/services` folder of the sample code.

The first method we'll implement is `searchByTrack` which will search for track, given a search term. One of the endpoints documented on Spotify API docs is [the Search endpoint⁷⁴](#).

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 how a first version of the service would look like:

```
1 class SpotifyService {  
2   constructor(public http: Http) {  
3   }  
4  
5   searchByTrack(query: string) {  
6     let params: string = [  
7       `q=${query}`,  
8       `type=track`  
9     ].join("&");  
10    let queryURL: `https://api.spotify.com/v1/search?${params}`;  
11    return this.http.request(queryURL).map(res => res.json());  
12  }  
13 }
```

⁷⁴<https://developer.spotify.com/web-api/search-item/>

This code performs an HTTP GET request to the URL <https://api.spotify.com/v1/search>⁷⁵, passing our query as the search term and type hardcoded to track.

This http call returns an Observable. We are going one step further and using the RxJS function map to transform the result we would get (which is an http module's Response object) and parsing it as JSON, resulting on an object.

Any function that calls searchByQuery then will have to use the Observable API to subscribe to the response like this:

```
1 service
2   .searchTrack('query')
3   .subscribe((res: any) => console.log('Got object', res))
```

The SearchComponent

Now that we have a service that will perform track searches, we can start coding the SearchComponent.

Again, we start with an import section:

code/routes/music/app/ts/components/SearchComponent.ts

```
1 /*
2  * Angular
3  */
4
5 import {Component, OnInit} from 'angular2/core';
6 import {
7   Router,
8   RouterLink,
9   RouteParams,
10 } from 'angular2/router';
11
12 /*
13  * Services
14  */
15 import {SpotifyService} from 'services/SpotifyService';
```

Here we're importing, among other things, the SpotifyService class we just created.

We then start coding the component. We're using search as the selector, importing a few directives and using the following template:

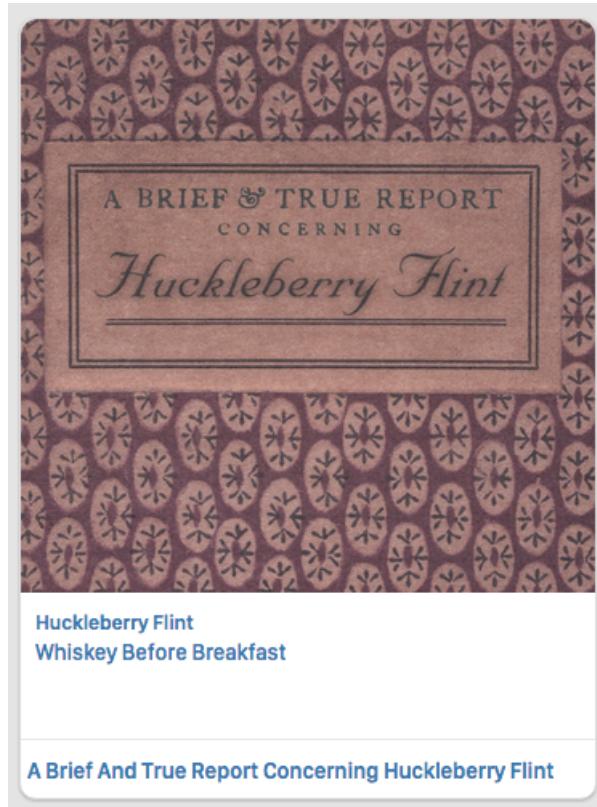
⁷⁵<https://api.spotify.com/v1/search>

code/routes/music/app/ts/components/SearchComponent.ts

```
1 @Component({
2   selector: 'search',
3   directives: [RouterLink],
4   template: `
5     <h1>Search</h1>
6
7     <p>
8       <input type="text" #newquery
9         [value]="query"
10        (keydown.enter)="submit(newquery.value)">
11       <button (click)="submit(newquery.value)">Search</button>
12     </p>
13
14     <div *ngIf="results">
15       <div *ngIf="!results.length">
16         No tracks were found with the term '{{ query }}'
17       </div>
18
19     <div *ngIf="results.length">
20       <h1>Results</h1>
21
22       <div class="row">
23         <div class="col-sm-6 col-md-4" *ngFor="#t of results">
24           <div class="thumbnail">
25             <div class="content">
26               
27               <div class="caption">
28                 <h3>
29                   <a [routerLink]=["'/Artists', {id: t.artists[0].id}]">
30                     {{ t.artists[0].name }}
31                   </a>
32                 </h3>
33                 <br>
34                 <p>
35                   <a [routerLink]=["'/Tracks', {id: t.id}]">
36                     {{ t.name }}
37                   </a>
38                 </p>
39               </div>
40             <div class="attribution">
41               <h4>
```

```
42         <a [routerLink]=["/Albums", {id: t.album.id}]>
43             {{ t.album.name }}
44         </a>
45     </h4>
46     </div>
47     </div>
48     </div>
49     </div>
50     </div>
51     </div>
52     </div>
53     `-
54 })
```

The goal here is to render each resulting track side by side on a card like below:



Music App Card

The Search Field

Let's break down the HTML template a bit.

This first section will have the search field:

code/routes/music/app/ts/components/SearchComponent.ts

```

1  <p>
2    <input type="text" #newquery
3      [value]="query"
4      (keydown.enter)="submit(newquery.value)"
5    <button (click)="submit(newquery.value)">Search</button>
6  </p>

```

Here we have the input field and we're binding its DOM element `value` property the `query` property of our component.

We also give this element a template variable named `#newquery`. We can now access the value of this input within our template code by using `newquery.value`.

The button will trigger the `submit` method of the component, passing the value of the input field as a parameter.

We also want to trigger `submit` when the user hits “Enter” so we bind to the `keydown.enter` event on the input.

Search Results and Links

The next section displays the results. We're relying on the `NgFor` directive to iterate through each track from our results object:

code/routes/music/app/ts/components/SearchComponent.ts

```

39   <div class="col-sm-6 col-md-4" *ngFor="#t of results">
40     <div class="thumbnail">

```

For each track, we display the artist name:

code/routes/music/app/ts/components/SearchComponent.ts

```

45       <a [routerLink]=["'/Artists', {id: t.artists[0].id}]">
46         {{ t.artists[0].name }}
47       </a>

```

Notice how we're using the `RouterLink` directive to redirect to `['/artists', {id: t.artists[0].id}]`.

This is how we set *router parameters* on a given route. Say we have an artist with an id abc123. When this link is clicked, the app would then redirect to `/artist/abc123` (where `abc123` is the `:id` parameter).

You'll see further down how we can retrieve this value inside the proper component that handles this route.

Now we're displaying the track:

code/routes/music/app/ts/components/SearchComponent.ts

```
51      <a [routerLink]="/Tracks", {id: t.id}>
52        {{ t.name }}
53      </a>
```

And the album:

code/routes/music/app/ts/components/SearchComponent.ts

```
58      <a [routerLink]="/Albums", {id: t.album.id}>
59        {{ t.album.name }}
60      </a>
```

SearchComponent Class

Let's take a look at the constructor first:

code/routes/music/app/ts/components/SearchComponent.ts

```
71 export class SearchComponent implements OnInit {
72   query: string;
73   results: Object;
74
75   constructor(public spotify: SpotifyService, public router: Router,
76             public routeParams: RouteParams) {
77 }
```

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 RouteParams and making them public properties of our class.

search

In our SearchComponent we will call out to the SpotifyService and render the results. There are two cases when we want to run a search:

We want to run a search when the user:

- enters a search query and submits the form
- navigates to this page with a given URL in the query parameters (e.g. someone shared a link or bookmarked the page)

To perform the actual search for both cases, we create the `search` method:

code/routes/music/app/ts/components/SearchComponent.ts

```
88  search(): void {
89      this.query = this.routeParams.get('query');
90      if (!this.query) {
91          return;
92      }
93
94      this.spotify
95          .searchTrack(this.query)
96          .subscribe((res: any) => this.renderResults(res));
97 }
```

Maybe we expected this method to take a parameter to indicate which search term we'd be searching for.

Instead, what we want to do is use the route parameter `query` to retrieve that information.

We then subscribe to the `searchTrack` resulting Observable and whenever we get new results, we're calling `renderResults`.

code/routes/music/app/ts/components/SearchComponent.ts

```
99  renderResults(res: any): void {
100      this.results = null;
101      if (res && res.tracks && res.tracks.items) {
102          this.results = res.tracks.items;
103      }
104  }
```

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 anyone with a URL that includes a search to be able to jump straight into the results.

To do that, we are going to implement a hook Angular router provides that kicks in whenever our component is initialized.



But isn't that what constructors are for? Well, yes and no. Yes, constructors are used to initialize values, but if you want to write good, testable code, you want to minimize the side effects of *constructing* an object. So keep in mind that you should put your component's initialization logic always on a hook like below.

Here's the implementation of the `ngOnInit` method:

`code/routes/music/app/ts/components/SearchComponent.ts`

```
79  ngOnInit(): void {  
80      this.search();  
81  }
```

To use `ngOnInit` we imported the `OnInit` class and declared that our component implements `OnInit`.

As you can see, we're just performing the search here. Since the term we're searching for comes from the URL, we're good.

`submit`

Now let's see what we do when the user submits the form.

`code/routes/music/app/ts/components/SearchComponent.ts`

```
83  submit(query: string): void {  
84      this.router.navigate(['/Search', {query: query}]);  
85      this.search();  
86  }
```

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 gonna see the same search result rendered. We can say that we're persisting the search term on the URL.

Putting it all together

Here's the full listing for the `SearchComponent` class:

code/routes/music/app/ts/components/SearchComponent.ts

```
1  /*
2   * Angular
3   */
4
5  import {Component, OnInit} from 'angular2/core';
6  import {
7    Router,
8    RouterLink,
9    RouteParams,
10 } from 'angular2/router';
11
12 /*
13  * Services
14 */
15 import {SpotifyService} from 'services/SpotifyService';
16
17 @Component({
18   selector: 'search',
19   directives: [RouterLink],
20   template: `
21     <h1>Search</h1>
22
23     <p>
24       <input type="text" #newquery
25         [value]="query"
26         (keydown.enter)="submit(newquery.value)"
27       <button (click)="submit(newquery.value)">Search</button>
28     </p>
29
30     <div *ngIf="results">
31       <div *ngIf="!results.length">
32         No tracks were found with the term '{{ query }}'
33       </div>
34
35       <div *ngIf="results.length">
36         <h1>Results</h1>
37
38         <div class="row">
39           <div class="col-sm-6 col-md-4" *ngFor="#t of results">
40             <div class="thumbnail">
41               <div class="content">
```

```
42      
43      <div class="caption">
44          <h3>
45              <a [routerLink]=[ '/Artists', {id: t.artists[0].id}]>
46                  {{ t.artists[0].name }}
47              </a>
48          </h3>
49          <br>
50          <p>
51              <a [routerLink]=[ '/Tracks', {id: t.id}]>
52                  {{ t.name }}
53              </a>
54          </p>
55      </div>
56      <div class="attribution">
57          <h4>
58              <a [routerLink]=[ '/Albums', {id: t.album.id}]>
59                  {{ t.album.name }}
60              </a>
61          </h4>
62      </div>
63  </div>
64  </div>
65  </div>
66  </div>
67  </div>
68  </div>
69  `

70 })
71 export class SearchComponent implements OnInit {
72     query: string;
73     results: Object;
74
75     constructor(public spotify: SpotifyService, public router: Router,
76                 public routeParams: RouteParams) {
77     }
78
79     ngOnInit(): void {
80         this.search();
81     }
82
83     submit(query: string): void {
```

```
84     this.router.navigate(['/Search'], {query: query});
85     this.search();
86 }
87
88 search(): void {
89     this.query = this.routeParams.get('query');
90     if (!this.query) {
91         return;
92     }
93
94     this.spotify
95         .searchTrack(this.query)
96         .subscribe((res: any) => this.renderResults(res));
97 }
98
99 renderResults(res: any): void {
100    this.results = null;
101    if (res && res.tracks && res.tracks.items) {
102        this.results = res.tracks.items;
103    }
104 }
105 }
```

Trying the search

Now that we have completed the code for the search, let's try it out:

Sportify music for active people

[Home](#)[Add](#)

Search

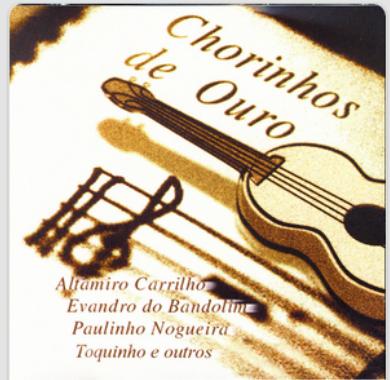
Results



Bando De Macambira
André do Sapato Novo

[Chorinho](#)

Ordinarius
André de Sapato Novo / Tico Tico no Fubá

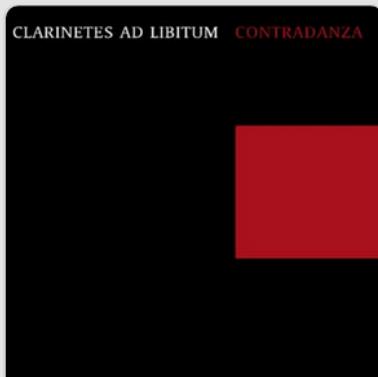
[Rio de Choro](#)

Affamiro Carrilho
Evandro do Bandolim
Paulinho Nogueira
Toquinho e outros

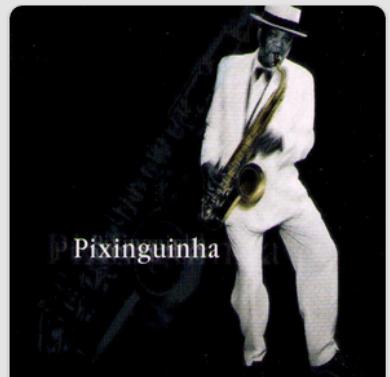
Evandro Do Bandolim
André De Sapato Novo

[Chorinhos De Ouro](#)

Pixinguinha
André de Sapato Novo

[Benedito Lacerda E Pixinguinha](#)

Clarinetes Ad Libitum
André de Sapato Novo

[Contradanza](#)

Pixinguinha
André De Sapato Novo

[Latin Jazz Roots](#)

Trying out Search

We can click the artist, track or album links to navigate to the proper route.

TrackComponent

For the track route, we use the TrackComponent. It basically displays the track name, the album cover image and allow the user to play a preview using an HTML5 audio tag:

code/routes/music/app/ts/components/TrackComponent.ts

```
27 <p>
28   <audio controls src="{{ track.preview_url }}"></audio>
29 </p>
```

Like we did for the search before, we're going to use the Spotify API. Let's refactor the method `searchTrack` and extract two other useful methods we can reuse:

code/routes/music/app/ts/services/SpotifyService.ts

```
19 query(URL: string, params?: Array<string>): Observable<any[]> {
20   let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;
21   if (params) {
22     queryURL = `${queryURL}?${params.join('&')}`;
23   }
24
25   return this.http.request(queryURL).map((res: any) => res.json());
26 }
27
28 search(query: string, type: string): Observable<any[]> {
29   return this.query(`/search`, [
30     `q=${query}`,
31     `type=${type}`
32   ]);
33 }
```

Now that we extracted those, notice how simpler `searchTrack` become:

code/routes/music/app/ts/services/SpotifyService.ts

```
35 searchTrack(query: string): Observable<any[]> {
36   return this.search(query, 'track');
37 }
```

Now let's create a method to allow the component we're building retrieve track information, based in the track ID:

code/routes/music/app/ts/services/SpotifyService.ts

```
39  getTrack(id: string): Observable<any[]> {
40    return this.query(`tracks/${id}`);
41  }
```

And now we can now use `getTrack` from a new `ngOnInit` method:

code/routes/music/app/ts/components/TrackComponent.ts

```
44  ngOnInit(): void {
45    this.spotify
46      .getTrack(this.id)
47      .subscribe((res: any) => this.renderTrack(res));
48  }
```

The other components work on a very similar way, using `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

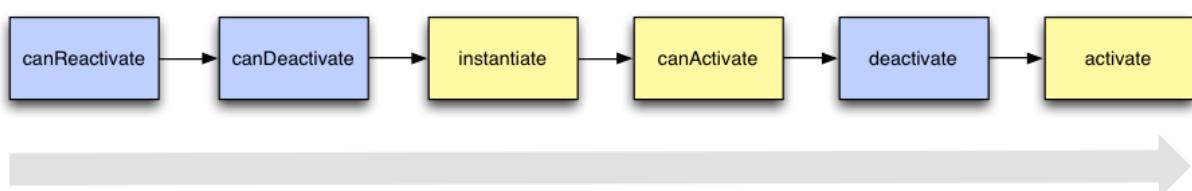
Router Lifecycle Hooks

There are times that we may want to do some action when changing routes. A classical example of that is authentication. Let's say we have a **login** route and a **protected** route.

We want to only allow the app to go to the protected route if the correct username and password were provided on the login page.

In order to do that, we need to hook into the lifecycle of the router and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

There are 6 different hooks Angular 2 router provides:



Router Lifecycle Hooks

The hooks are called on the illustrated sequence, from left to right.

The ones filled in light blue are triggered on the component that is being deactivated (or the one whose route the app is leaving), while the ones in yellow are triggered on the component that is being activated (the one whose the app is entering).

So if you're navigating from `LoginComponent` to `ProtectedComponent` routes, the triggering sequence is the following:

1. `LoginComponent.canReactivate` *if return is false, stops;*
2. `LoginComponent.canDeactivate` *if return is false, stops;*
3. `ProtectedComponent.instantiate;`
4. `ProtectedComponent.canActivate` *if return is false, stops;*
5. `LoginComponent.deactivate;`
6. `ProtectedComponent.activate;`

For all those hooks, except `canActivate`, all you need to do is declare a method with the hook name, like:

```
1 class MyComponent {  
2     canReactivate() {  
3         // ...  
4     }  
5 }
```

And the framework will call it at the right time. The `canActivate` hook is the only one that needs to be declared differently, as we'll start to see now.

Going back to the authentication problem, what we would need to do in this scenario is using the `ProtectedComponent.canActivate` hook and return `false` in case the authentication fails.

Let's revisit our initial application, adding login and password input fields and a new protected route that only works if we provide a certain username and password combination.



Sample Code The complete code for the examples in this section can be found in the `routes/auth` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

Try running the code while reading this section and feel free play around to get a deeper insight about how it all works.

AuthService

Let's create a very simple and minimal implementation of a service, responsible for authentication and authorization of resources:

code/routes/auth/app/ts/services/AuthService.ts

```
1 import {Injectable, provide} from 'angular2/core';
2
3 @Injectable()
4 export class AuthService {
5   login(user: string, password: string): boolean {
6     if (user === 'user' && password === 'password') {
7       localStorage.setItem('username', user);
8       return true;
9     }
10
11   return false;
12 }
```

This method will return true if the provided user/password pair matches. Also, when it is matched, it's gonna use `localStorage` to save the username. This will also serve as a flag indicating whether there is an active logged user or not.



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 MDN⁷⁶](#)

code/routes/auth/app/ts/services/AuthService.ts

```
1 }
2
3 getUser(): any {
```

The `logout` method just clears the `username` value:

code/routes/auth/app/ts/services/AuthService.ts

```
14 logout(): any {
15   localStorage.removeItem('username');
16 }
```

And the final two methods:

- `getUser` returns the `username` or null
- `isLoggedIn` uses `getLogged` to return true if we have a user

⁷⁶<https://developer.mozilla.org/en-US/docs/Web/API/Storage>

code/routes/auth/app/ts/services/AuthService.ts

```
18  getUser(): any {
19      return localStorage.getItem('username');
20  }
21
22  isLoggedIn(): boolean {
23      return this.getUser() !== null;
24  }
```

The last thing we do is export an AUTH_PROVIDERS, so it can be injected into our app:

code/routes/auth/app/ts/services/AuthService.ts

```
27  export var AUTH_PROVIDERS: Array<any> = [
28      provide(AuthService, {useClass: AuthService})
29  ];
```

LoginComponent

This component will either show a login form, for the case when there is no logged user, or display a little banner with user information along with a logout link.

The relevant code here is the login and logout methods:

code/routes/auth/app/ts/components/LoginComponent.ts

```
48  this.message = '';
49  if (!this.authService.login(username, password)) {
50      this.message = 'Incorrect credentials.';
51      /* tslint:disable */
52      setTimeout(function() {
53          this.message = '';
54      }.bind(this), 2500);
55      /* tslint:enable */
56  }
57  return false;
58 }
59
60 logout(): boolean {
61     this.authService.logout();
62     return false;
63 }
```

Once our service validates the credentials, we log the user in.

The component template has two snippets that are displayed based on whether the user is logged in or not.

The first is a login form, protected by `*ngIf="!authService.getUser()"`:

code/routes/auth/app/ts/components/LoginComponent.ts

```
18 <form class="form-inline" *ngIf="!authService.getUser()">
19   <div class="form-group">
20     <label for="username">User:</label>
21     <input class="form-control" name="username" #username>
22   </div>
23
24   <div class="form-group">
25     <label for="password">Password:</label>
26     <input class="form-control" type="password" name="password" #password>
27   </div>
28
29   <a class="btn btn-default" (click)="login(username.value, password.value)">
30     Submit
31   </a>
32 </form>
```

And the information banner, containing the logout link, protected by the inverse - `*ngIf="authService.getUser()`

code/routes/auth/app/ts/components/LoginComponent.ts

```
34 <div class="well" *ngIf="authService.getUser()">
35   Logged in as <b>{{ authService.getUser() }}</b>
36   <a href (click)="logout()">Log out</a>
37 </div>
```

There's another snippet of code that is displayed when we have an authentication error:

code/routes/auth/app/ts/components/LoginComponent.ts

```
13   template: `
14   <div class="alert alert-danger" role="alert" *ngIf="message">
15     {{ message }}
16   </div>
```

Now that we can handle the user login, let's create a restrict resource.

ProtectedComponent

This component will be only accessible to logged in users. But how can we do that?

The answer is to use a router hook. More specifically, we'll use the `CanActivate` hook.

This hook will be called when the router tries to change from one route to the other. By returning false, we will be basically telling the router to abort this attempt to change the route and therefore not allowing the user to see this new route.

To implement the hook, we have to first import the `CanActivate` annotation:

```
1 import {CanActivate} from 'angular2/router';
```

And annotate our component with it:

code/routes/auth/app/ts/components/ProtectedComponent.ts

```
16 @CanActivate(  
17   (nextInstr: any, currInstr: any) => {  
18     let injector: any = Injector.resolveAndCreate([AuthService]);  
19     let authService: AuthService = injector.get(AuthService);  
20     return authService.isLoggedIn();  
21   }  
22 )
```

As you can see, the `CanActivate` annotation takes a callback function.

When called, this function will receive two arguments. They both represent a route **instruction**. This is how Angular names a route descriptor and it includes information like route parameters, paths, query parameters and a lot more.

The first parameter describes the route the user is trying to visit (or the *next* route) and the second describe the current route.

Notice how you can use all that granular information to decide whether or not the user can be navigating each route your application expose.

In our case, since we're protecting the component as a whole, we just need to return false when there's no logged user:



If we add logs to the CanActivate function:

```
1 console.log('nextInstr', nextInstr);
2 console.log('currInstr', currInstr);
```

We can see what information an Instruction carry:

```
Developer Tools - http://localhost:8080/
Elements Network Sources Timeline Profiles Resources Audits Console Terminal
<top frame> ▾ Preserve log
Filter □ Regex □ Hide network messages All Errors Warnings Info Logs Debug Handled
nextInstr
  ▾ ComponentInstruction {urlPath: "protected", urlParams: Array[0], _recognizer: PathRecognizer, params: Object, reuse: false} ↴ ProtectedComponent.ts:22
    ► _recognizer: PathRecognizer
    componentType: (...)

  ► params: Object
    reuse: false
    specificity: (...)

  ► terminal: (...)

  ► urlParams: Array[0]
    urlPath: "protected"

  ► __proto__: ComponentInstruction
currInstr
  ▾ ComponentInstruction {urlPath: "home", urlParams: Array[0], _recognizer: PathRecognizer, params: Object, reuse: false} ↴ ProtectedComponent.ts:23
    ► _recognizer: PathRecognizer
    componentType: (...)

    ► params: Object
      reuse: false
      specificity: (...)

    ► terminal: (...)

    ► urlParams: Array[0]
      urlPath: "home"

    ► __proto__: ComponentInstruction
> |
```

Output of Instruction Debug

If you want to dig further, you can check Angular source code here:

<https://github.com/angular/angular/blob/b2bc50dbd14ba2db7a640d8c3c3c149c13db87a1/modules/angular2/src/router/instruction.ts>

Back to the component, let's investigate what the code is doing:

code/routes/auth/app/ts/components/ProtectedComponent.ts

```
18 let injector: any = Injector.resolveAndCreate([AuthService]);
19 let authService: AuthService = injector.get(AuthService);
20 return authService.isLoggedIn();
```

We're first asking the Injector class to give us a concrete injector that is aware of only once class: AuthService.

Next, we're asking this injector to give us an instance of that class.

⁷⁷<https://github.com/angular/angular/blob/b2bc50dbd14ba2db7a640d8c3c3c149c13db87a1/modules/angular2/src/router/instruction.ts>

Now that we have an AuthService to call, we call the `isLogged()` method which will then decide if the user is allowed to continue.

To make our application aware of our components and the AuthService, let's add the following imports to `app.ts`:

code/routes/auth/app/ts/app.ts

```
15  /*
16   * Components
17  */
18 import { LoginComponent } from 'components/LoginComponent';
19 import { HomeComponent } from 'components/HomeComponent';
20 import { AboutComponent } from 'components/AboutComponent';
21 import { ContactComponent } from 'components/ContactComponent';
22 import { ProtectedComponent } from 'components/ProtectedComponent';
23
24 /*
25  * Services
26 */
27 import { AUTH_PROVIDERS } from 'services/AuthService';
```

And adding it to bootstrapping:

code/routes/auth/app/ts/app.ts

```
74 bootstrap(RoutesDemoApp, [
75   ROUTER_PROVIDERS,
76   AUTH_PROVIDERS,
77   provide(LocationStrategy, { useClass: HashLocationStrategy })
78 ]);
```

Let's now add the new `/protected` path to the router:

code/routes/auth/app/ts/app.ts

```
62 @RouteConfig([
63   { path: '/',           name: 'root',      redirectTo: [ 'Home' ] },
64   { path: '/home',       name: 'Home',       component: HomeComponent },
65   { path: '/about',      name: 'About',      component: AboutComponent },
66   { path: '/contact',    name: 'Contact',    component: ContactComponent },
67   { path: '/protected',  name: 'Protected',  component: ProtectedComponent },
68 ])
```

We now have to add:

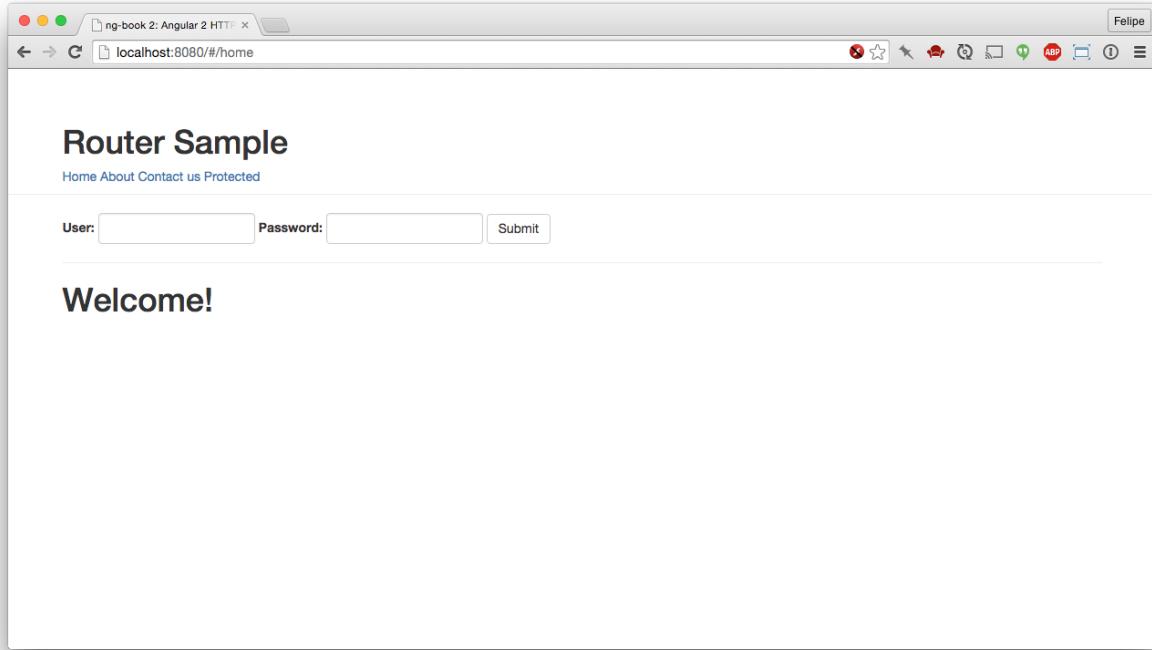
1. LoginComponent to the list of directives
2. a new link to the protected route
3. a <login> tag to the template, to render the new component

Here's how it should look like:

code/routes/auth/app/ts/app.ts

```
34  @Component({
35    selector: 'router-app',
36    directives: [ROUTER_DIRECTIVES, LoginComponent],
37    template: `
38      <div class="page-header">
39        <div class="container">
40          <h1>Router Sample</h1>
41          <div class="navLinks">
42            <a [routerLink]="/Home">Home</a>
43            <a [routerLink]="/About">About</a>
44            <a [routerLink]="/Contact">Contact us</a>
45            <a [routerLink]="/Protected">Protected</a>
46          </div>
47        </div>
48      </div>
49
50      <div id="content">
51        <div class="container">
52
53          <login></login>
54
55          <hr>
56
57          <router-outlet></router-outlet>
58        </div>
59      </div>
60    `
61  })
```

Now when we open the application on the browser, we can see the new login form and the new protected link:

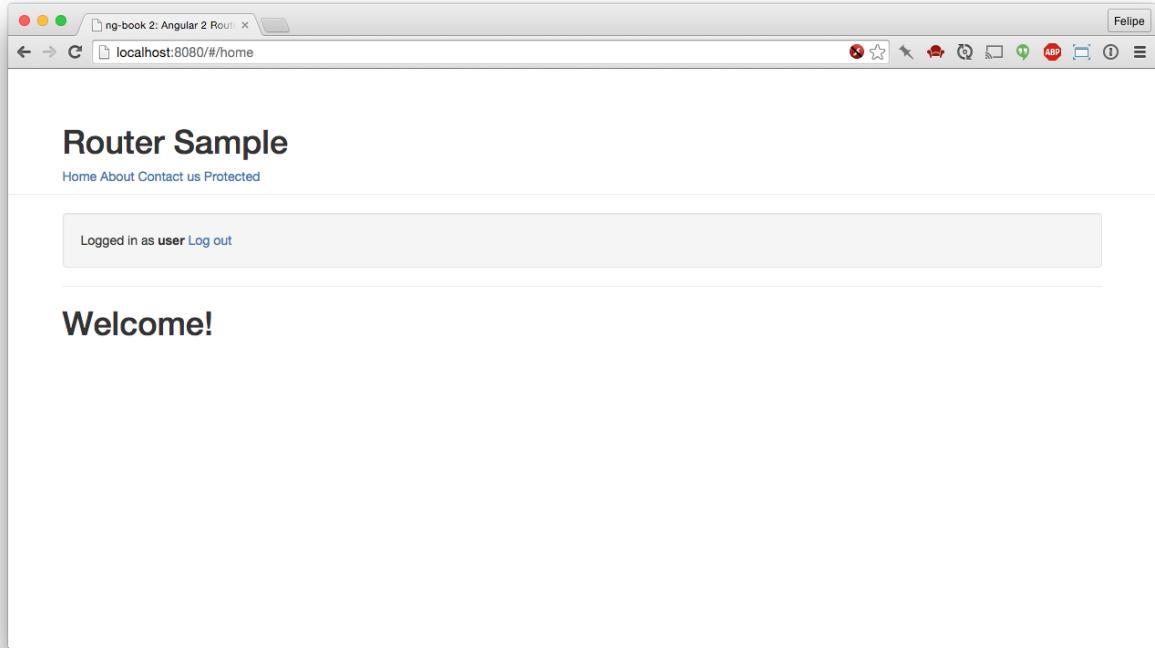


Auth App - Initial Page

If you click the Protected link, you'll see nothing happens. The same happens if you try to manually visit <http://localhost:8080/#/protected>⁷⁸.

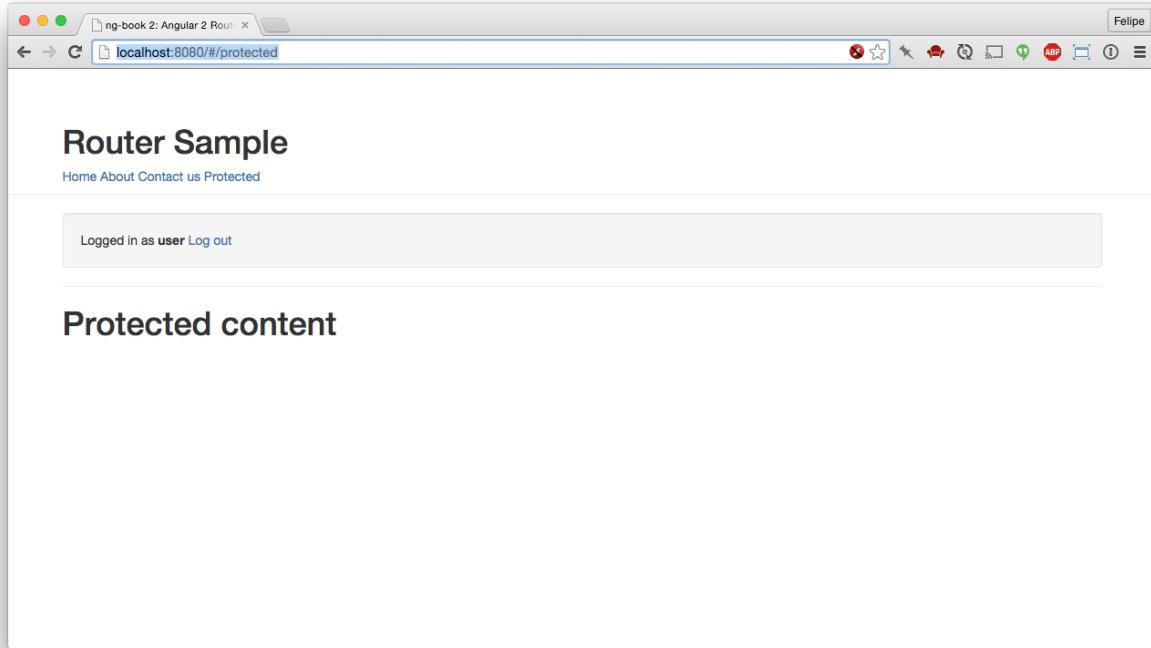
Now enter *user* and *password* on the form and click **Submit**. You'll see that we now get the current user displayed on a banner:

⁷⁸<http://localhost:8080/#/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 don't necessarily keep anyone from viewing the javascript pages behind those routes.

Nested routes

Nested routes is the concept of allowing routes within routes. Let's say we have a website with one area to allow users to know our team, called **Who we are?** and another one for our **Products**.

We could think that the perfect route for **Who we are?** would be /about and for products /products.

And we're happily displaying all our team and all our products when visiting this areas.

What happens when the website grows and we now need to display individual information about each person in our team and also for each product we sell?

In order to support scenarios like these, the router allows the user to define nested routes.

To do that, you can have multiple, nested `router-outlet`. So each area of our application can have their own child components, that also have its own `router-outlets`.

Let's work on an example to clear things up.

In this example, we'll have a products section where the user will be able to view two highlighted products by visiting a nice URL. For all the other products, the routes will use the product ID.



Sample Code The complete code for the examples in this section can be found in the `routes/auth` 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 this section and feel free play around to get a deeper insight about how it all works.

Configuring Routes

We will start by describing two top-level routes on the `app.ts` file:

`code/routes/nested/app/ts/app.ts`

```
48 { path: '/home', name: 'Home', component: HomeComponent, useAsDefault: true },
49 { path: '/products/...', name: 'Products', component: ProductsComponent },
```

The `home` route looks very familiar, but what's with this `/...` on `products`?

This is the way the router expects the routes that have child routes to be declared. It's saying that we're expecting more things to go after `/products`.

ProductsComponent

This component will have, by its turn, its own route configuration:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
39 @RouteConfig([
40   { path: '/main', name: 'Main', component: MainComponent, useAsDefault: true },
41   { path: '/:id', name: 'ById', component: ByIdComponent },
42   { path: '/interest', name: 'Interest', component: InterestComponent },
43   { path: '/sportify', name: 'Sportify', component: SportifyComponent },
44 ])
```

The two last lines here are also straightforward. Those are the two products we want to visit by URL, **Sportify** and **Interest**.

For the first route, the only new thing is `useAsDefault: true`. What is that?

If we didn't have that, what would happen if we visited the `/products` path, with no child route?

As a matter of fact, we would get an error because we're trying to link to a route defined by `['Products']` on `app.ts`, but there isn't one.

Now with this attribute set, when we visit `/products`, we'll be redirected to the **Main** route.

The last route we need to understand is **ById**. For that route, we're using the `/:id` path.

That's how you tell the router that a route expects to receive a parameter. In this case, when the user visits something *that doesn't match any other route*, it will fallback to this route.

On top of that everything that is passed after `/` will be extracted to a parameter of the route, called `:id`. If we used `:product_id` instead, for instance, that's the name the route parameter would get.

Now on the component template, we'll have a link to each of those static child routes:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
27 <a [routerLink]="['./Main']">Main</a> |
28 <a [routerLink]="['./Interest']">Interest</a> |
29 <a [routerLink]="['./Sportify']">Sportify</a> |
```

You can see that the route links are all in the format `['./Main']`, with a preceding `./`. This indicates that you want to navigate the **Main** route from 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:

code/routes/nested/app/ts/components/ProductsComponent.ts

```
30   Enter id: <input #id size="6">
31   <button (click)="goToProduct(id.value)">Go</button>
```

The goToProduct method would be then defined like so:

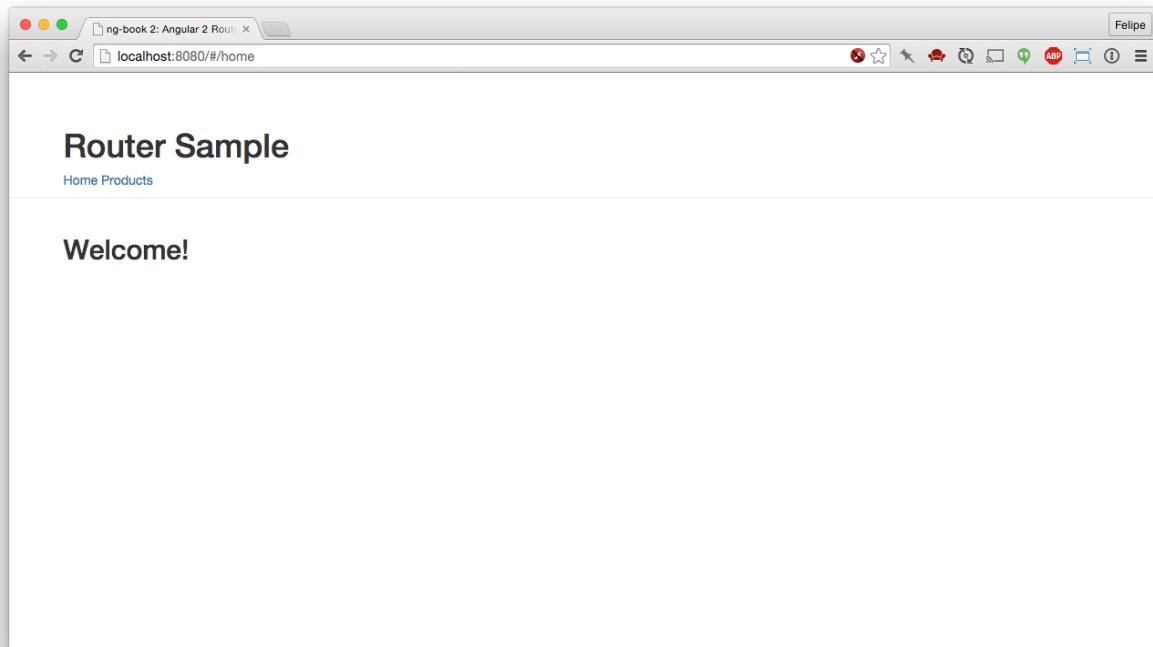
code/routes/nested/app/ts/components/ProductsComponent.ts

```
46   constructor(public router: Router) {
47   }
48
49   goToProduct(id: string): void {
50     this.router.navigate(['./ById', {id: id}]);
51 }
```

First on the constructor we're declaring a public instance of the Router, since we're gonna use that instance to navigate to the product by id.

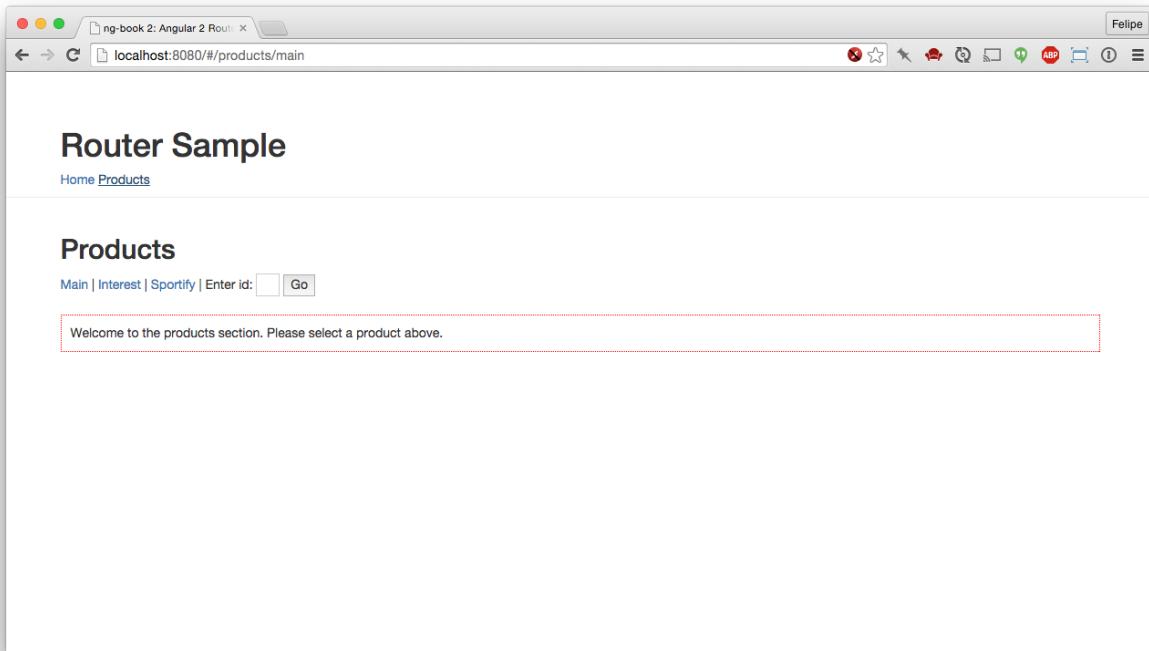
We do that by calling the router's navigate method and providing the route name and an object with route parameters. In our case we're just passing the id.

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:

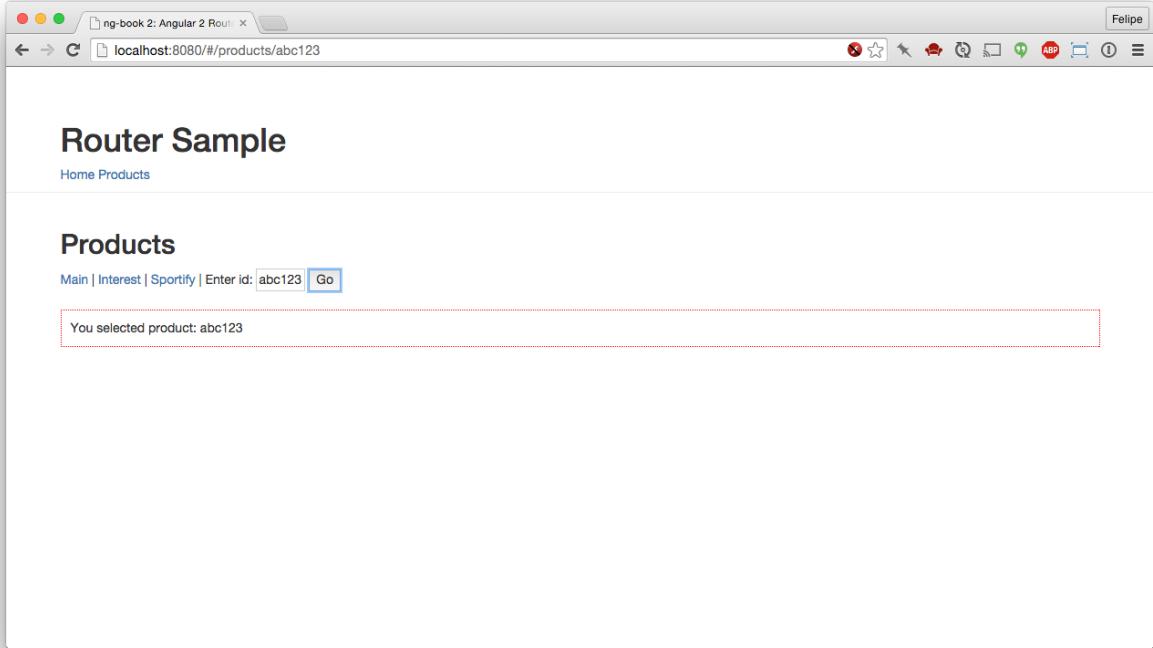


Nested Routes App - Products Section

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!

Changelog

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 UI⁷⁹ styles
- “Built-in Components” Chapter
 - Updated built-in components sample apps to `angular-2.0.0-beta.0`
 - Added Semantic UI

Revision 12 - 2015-11-16

- “Routing” Chapter
 - Fixed ROUTER_DIRECTIVES typo - Wayne R.
- “First App” Chapter
 - Updated example to `angular-2.0.0-alpha-46`
 - Fixed some bolding around NgFor to clarify the code example - Henrique M.

⁷⁹<http://semantic-ui.com>

- Fixed Duplicate identifier 'Promise'. errors due to a bad `tsconfig.json` in `angular2-reddit-base/` - Todd F.
- Fixed language typos caught by Steffen G.
- “Forms” Chapter
 - * Updated example to `angular-2.0.0-alpha-46`
 - Fixes the method of subscribing to Observables in the “Form with Events” section
 - * Fixed a few typos and language issues - Christopher C., Travis P.
- “TypeScript” Chapter
 - * Fixed some unclear language about `enum` - Frede H.
- “Built-in Components” Chapter
 - * Fixed a typo where `[class]` needed to be `[ng-class]` - Neal B.
- “How Angular Works” Chapter
 - * Fixed language typos - Henrique M.

Revision 11 - 2015-11-09

- Fixed explanation of TypeScript benefits - Thanks Don H!
- Fixed tons of typos found by Wayne R - Thanks Wayne!
- “How Angular Works” Chapter
 - Fixed typos - Jegor U.
 - Converted a component to use `inputs/outputs` - Jegor U.
 - Fixed `number` to `myNumber` typo - Wayne R.
- “Built-in Components” Chapter
 - Fixed language typos - Wayne R., Jek C., Jegor U.
 - Added a tip-box explaining object keys with dashes - Wayne R.
 - Use controller view value for `ng-style color` instead of the form field value - Wayne R.
- “Forms” Chapter
 - Fixed language typos - Wayne R., Jegor U.
- “Data Architecture in Angular 2”
 - Was accidentally part of “Forms” and is now promoted to it’s proper place as an introduction mini-chapter - Wayne R.
- “RxJS Pt 1.” Chapter
 - Fixed language typos - Wayne R.
- “RxJS Pt 2.” Chapter
 - Fixed Unicode problem - Birk S.
 - Clarified language around `combineLatest` return value - Birk S.
- “Typescript” Chapter
 - Fixed language typo - Travis P., Don H.
- “Routing” Chapter

- Fixed language typos - Jegor U., Birk S.
- “First App” Chapter
 - Fixed link to `ng_for` - Mickey V.
- “HTTP” Chapter
 - Fixed language typos - Birk S.
 - Clarified `ElementRef` role in `YouTubeSearchComponent`
 - Fixed link to `RequestOptions` - Birk S.

Revision 10 - 2015-10-30

- Upgraded Writing your First Angular2 Web Application chapter to `angular-2.0.0-alpha.44`
- Upgraded Routing chapter to `angular-2.0.0-alpha.44`
- Fixed ‘pages#about’ on the rails route example. - Thanks Rob Y!

Revision 9 - 2015-10-15

- Added Routing Chapter

Revision 8 - 2015-10-08

- Upgraded chapters 1-5 to `angular-2.0.0-alpha.39`
- properties and events renamed to inputs and outputs
- Fixed an issue in the First App chapter that said `#newtitle` bound to the value of the input (it’s really binding to the Control object) - Danny L
- `CSSClass` renamed to `NgClass`
- `ng-non-bindable` is now built-in so you don’t need to inject it as a directive
- Updated the forms chapter as there were several changes to the forms API
- Fixed `NgFor` source url in First App chapter - Frede H.

Revision 7 - 2015-09-23

- Added HTTP Chapter
- Fixed `For` -> `NgFor` typo - Sanjay S.

Revision 6 - 2015-08-28

- Added RxJS Chapter Data Architecture with Observables - Part 1 : Services
- Added RxJS Chapter Data Architecture with Observables - Part 2 : View Components

Revision 5

- Finished built-in components chapter

Revision 4

- Added built-in components chapter draft
- Added a warning about linewrapping of long URLs - Thanks Kevin B!
- Explained how annotations are bound to components on the First App chapter - thanks Richard M. and others
- Copy typo fixes - thanks Richard M.!
- Fixed TypeScript using `integer` instead of `number` - Richard M. and Roel V.
- Fixed “var nate =” listings require a comma to be a valid JS object - thanks Roel V.
- Renamed a few “For” directive mentions to “NgFor” - thanks Richard M.
- Fixed type on “RedditArticle” - thanks Richard M.
- Explained how annotations are bound to components on the First App chapter (thanks Richard M. and others)
- Typos and grammar improvements on First App chapter (thanks Kevin B)
- Typos and code improvements on How Angular Works (thanks Roel V.)

Revision 3

- Added forms chapter

Revision 2

- Updated `For` directive to `NgFor` accross all chapters and examples (templates changed from `*for=` to `*ng-for=` as well)
- Changed the suggested static web server from `http-server` to `live-server` so the execution command is valid both in OSX/Linux and Windows
- Changed the `@Component`’s `properties` property to match the latest AngularJS 2 format
- Updated `angular2.dev.js` bundle to latest version for all examples
- Updated `typings` folder with latest version for all examples

Revision 1

Initial version of the book