ng-book 2

The Complete Book on AngularJS 2

Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

FULLSTACK.io
gistia

ng-book 2

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

Contents

Book Revision

Revision 12 - Covers up to Angular 2 (2.0.0-alpha.46, 2015-11-13)

Prerelease

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

Bug Reports

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

Chat With The Community!

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

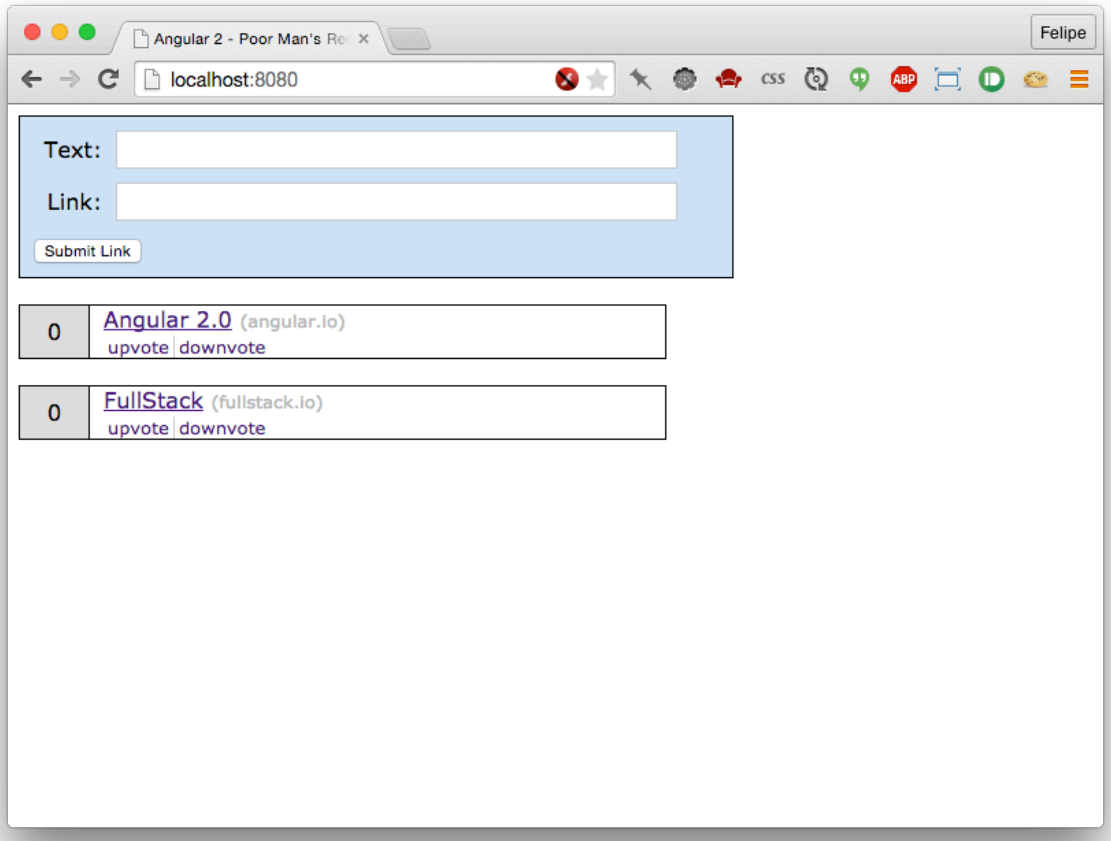
[1] mailto:us@fullstack.io?Subject=ng-book%20feedback
[2] https://gitter.im/ng-book/ng-book

Writing your First Angular2 Web Application

Poor Man's Reddit Clone

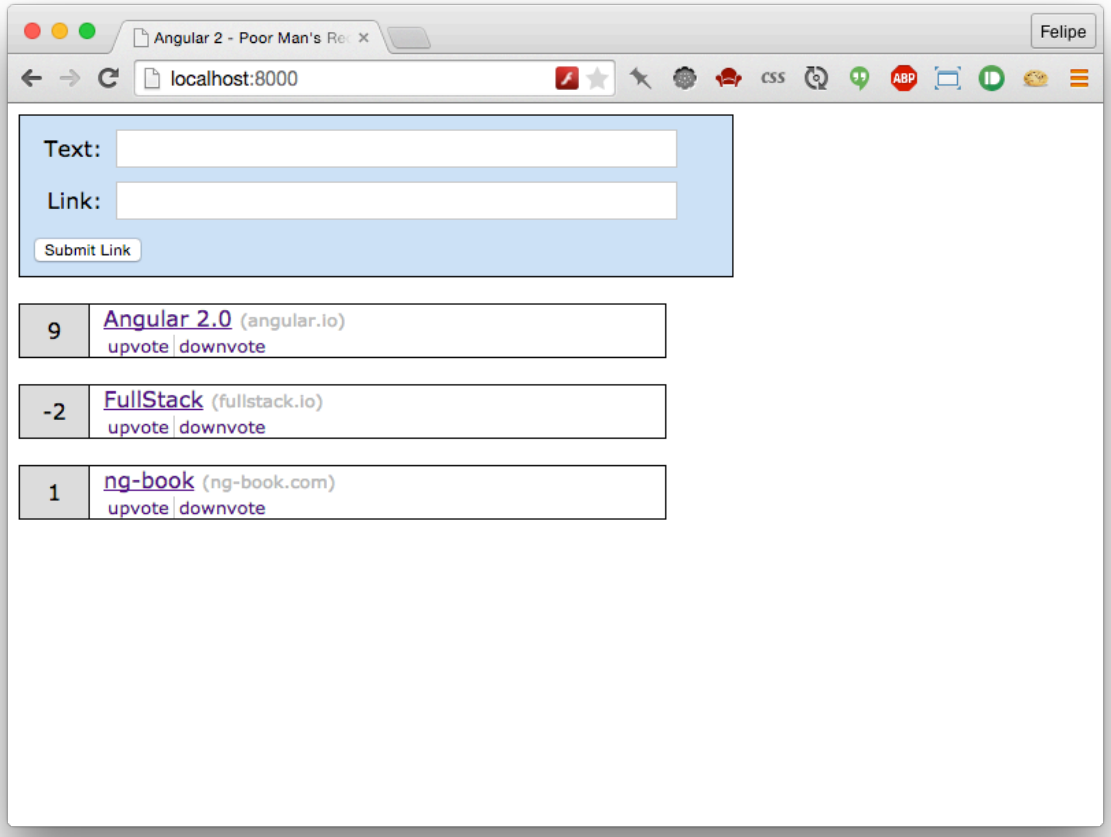
In this chapter we're going to write a toy application that allows the user to submit a new article with a title and an URL. You can think of it like a reddit-clone. By putting together a whole app we're going to touch on most of the parts of Angular 2.

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



Completed application

First, a user will submit a new link and after submitting the users will be able to upvote or downvote each article. Each link will have a score that we will keep track of.



App with new article

For this example, 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/ES6 you should be able to follow along without any problems. We'll go over TypeScript more in depth in the next chapter.

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/[3] .

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 going to be 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 1.6 or greater. To install it, run the following `npm` command:

```
1 $ npm install -g 'typescript@1.6.2'
```

`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[4] as it will let you run commands just as we have them written out in this book.

Example Project

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

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

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

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

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

[3] https://nodejs.org/download/
[4] https://www.cygwin.com/

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

```
$ npm install
```

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

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
  </head>
  <body>
  </body>
</html>
```

Angular 2 itself is a javascript file. So we need to add a `script` tag to this document to include it. But our setup with Angular/TypeScript has some dependencies itself:

Traceur

Traceur is a JavaScript compiler that backports ES6 code into ES5 code, so you can use all ES6 features with browsers that only understand ES5.

Even though we're using TypeScript, we're using the Traceur runtime to polyfill ("backport") some ES6 features to our ES5 code

The Angular team is working to remove the Traceur runtime as a dependency of Angular 2

We've vendored Traceur in the sample code, so to add Traceur to our app we add the following script tag:

```
<script src= "vendor/traceur-runtime.js"></script>
```

For more information, check the Traceur GitHub repository[5].

SystemJS

We also need to add SystemJS. SystemJS is a dynamic module loader. It helps simplify creating modules and requiring our code in our web app. It allows you to require the modules you need in the right order.

We've also vendored SystemJS in the example code so to add SystemJS we need to add this:

[5] https://github.com/google/traceur-compiler

```
    <script src="vendor/system.js"></script>
```

Angular2

And of course, we also need to add Angular itself. We have a version of Angular 2 in the code sample already, so to use it we can simply put this script tag:

```
    <script src="vendor/angular2.dev.js"></script>
```

Writing a hello world application

Here's how our code should look now:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Libraries -->
    <script src="vendor/traceur-runtime.js"></script>
    <script src="vendor/system.js"></script>
    <script src="vendor/angular2.dev.js"></script>
  </head>
  <body>
  </body>
</html>
```

We also need some CSS so our application will look good. Lets include a stylesheet as well:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Libraries -->
    <script src="vendor/traceur-runtime.js"></script>
    <script src="vendor/system.js"></script>
    <script src="vendor/angular2.dev.js"></script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css" href="styles.css"> <!-- <- here -->
  </head>
  <body>
  </body>
</html>
```

Let's now create our first TypeScript file. Create a new file called `app.ts` on 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 read TypeScript files, only Javascript files. We'll compile our `.ts` to a `.js` file in just a few minutes.

```
1  /// <reference path="typings/angular2/angular2.d.ts" />
2
3  import {
4    Component,
5    View,
6    bootstrap,
7  } from "angular2/angular2";
8
9  @Component({
10   selector: 'hello-world'
11 })
12 @View({
13   template: `<div>Hello world</div>`
14 })
15 class HelloWorld {
16 }
17
18 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 three things: `Component`, `View`, and `bootstrap`. We're `importing` it `from "angular2/angular2"`. The `"angular2/angular2"` portion tells our program where to find the dependencies that we're looking for.

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 just making these dependencies available to 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 defines the weather? Or what if we wanted to have a `<login>` tag that defines where we should put the login?

That is the idea behind components.

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

So 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:

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

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

1. A `Component` annotation
2. A `View` annotation
3. A definition class

Let's take these one at a time.

If you've been programming in javascript for a while then this next statement is a little weird to see in javascript:

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

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`.

You might be asking, what's the difference between decorating with an annotation and subclassing? Why use annotations instead of regular class code? How can I write my own annotations? We'll deal with these questions in later chapters.

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`.

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

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 simply 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.

Making a `View`

Similar to `@Component`, the `@View` annotation indicates that `HelloWorld` also has a `View`. This `View` defines an HTML `template` that will be rendered when this component is rendered.

```
@View({
  template: `<div>Hello world</div>`
})
```

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. We could have written the markup above as:

```
@View({
  template: `
  <div>
    Hello world
  </div>
`
})
```

Using backticks for multiline strings is **fantastic** and makes it way easier 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 just adds a lot of overhead. When you have to switch between a lot of files it adds overhead to your development. Personally, if my templates are smaller than a page I much prefer having the templates alongside the code. I can see both the logic and the view together and it's really easy to understand how they interact

The biggest drawback to putting your views inlined with your code is that many editors don't yet support syntax highlighting of the internal strings. 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 `HelloWorld`.

Once it 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!

Putting all the pieces together

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:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Libraries -->
    <script src="vendor/traceur-runtime.js"></script>
    <script src="vendor/system.js"></script>
    <script src="vendor/angular2.dev.js"></script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <script>
      System.import('app.js');
    </script>

    <hello-world></hello-world>

  </body>
</html>
```

We have one problem though: on the new `System.import('app.js')` line, we're using SystemJS to import the module defined in the `app.js` file. But, as you can see, we don't have an `app.js` file yet.

Compiling the code

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`:

```
$ 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:

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.

```
$ ls app.js
app.js
```

We have one more step to test our application. We need to run a test web-server to serve our app from. Let's install a NodeJS static http server called **live-server**. This server is nice because it automatically reloads the page when you make changes to your code.

To install, run the following command:

```
$ npm install -g live-server
```

After it finishes, you can run it with the `live-server` command:

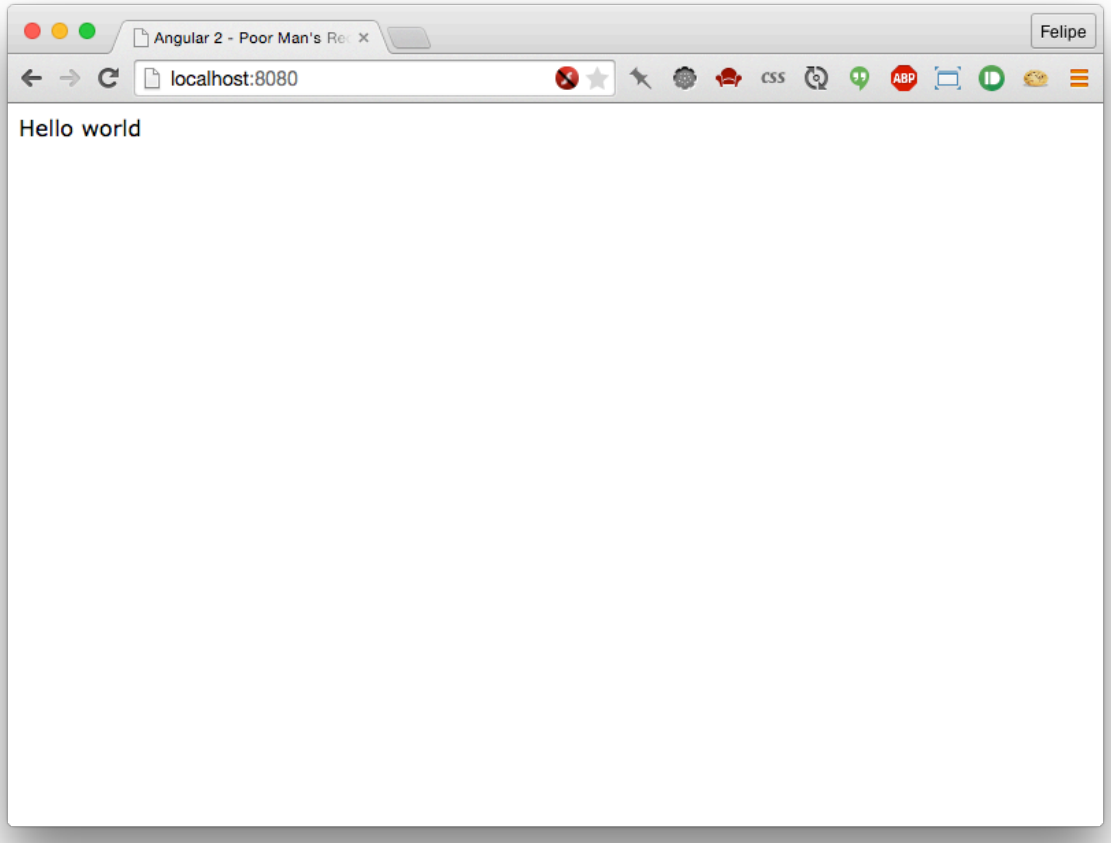
```
$ live-server
Serving "/Users/fcoury/code/angular2-reddit" at http://127.0.0.1:8080
```

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

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

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

Open your browser and type http://localhost:8080. If everything worked correctly, you should see the following:



Completed application

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

1. Make sure that your `app.js` file was created from the Typescript compiler `tsc`
2. Make sure that your webserver was started in the same directory as your `app.js` file
3. Make sure that your `index.html` file matches our code example above
4. Try opening the page in Chrome, right click, and pick "Inspect Element". Then click the "Console" tab and check for any errors.
5. If all else fails, join us here to chat on Gitter![6]

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:

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

Adding data to a component

Our component right now isn't very interesting. Most components will have data that make the component dynamic.

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:

```
@Component({
  selector: 'hello-world'
})
@View({
  template: `<article>Hello {{ name }}</article>`
})
class HelloWorld {
  name: string;

```

[6] https://gitter.im/ng-book/ng-book

```
10    constructor() {
11      this.name = 'Felipe';
12    }
13  }
```

Here we changed three things:

1. `name` Property On the `HelloWorld` class we added a *property*. Notice that the syntax is new relative to ES5 javascript. We say `name: string;`. That 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

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.

We use our `name` property by using `this.name`

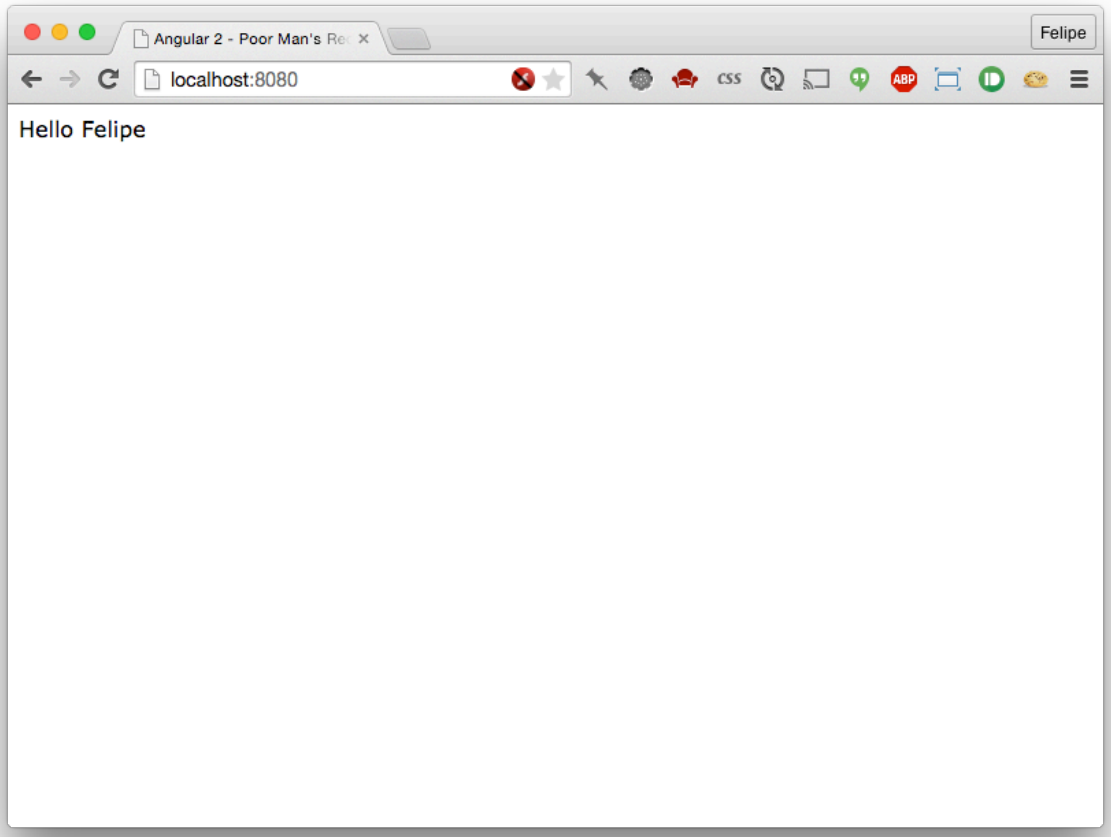
When we write:

```
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 `View` 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 a template. Here, because the View is *bound* to our Component, the `name` will expand to `this.name` e.g. `'Felipe'` in this case.

Try it out After making these changes, reload the page. We should see `"Hello Felipe"`



Application with Data

At this point, you might be wondering what indicates that the `View` is bound to this component declaration. The way annotation works is they affect the declaration following the annotation itself.

So when we have code like:

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

```
@Component({
  selector: 'reddit-article',
  inputs: ['article']


})
@View({
  template: `
  <article>
    <div class="votes">{{ article.votes }}</div>
```

```
11      <div class="main">
12        <h2>
13          <a href="{{ article.link }}">{{ article.title }}</a>
14          <span>({{ article.domain() }})</span>
15        </h2>
16        <ul>
17          <li><a href (click)="voteUp()">upvote</a></li>
18          <li><a href (click)="voteDown()">downvote</a></li>
19        </ul>
20      </div>
21    </article>
22    `
23  })
24  class RedditArticle {
```

We are *annotating* the `RedditArticle` class with the preceding annotations `Component` and `View`.

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   /// <reference path="typings/angular2/angular2.d.ts" />
2
3   import {
4     Component,
5     NgFor,
6     View,
7     bootstrap,
8   } from "angular2/angular2";
9
10  @Component({
11    selector: 'hello-world'
12  })
13  @View({
14    directives: [NgFor],
```

```
15    template: `
16    <ul>
17      <li *ng-for="#name of names">Hello {{ name }}</li>
18    </ul>
19    `
20  })
21  class HelloWorld {
22    names: Array<string>;
23
24    constructor() {
25      this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
26    }
27  }
28
29  bootstrap(HelloWorld);
```

The first change to point out is the new `Array<string>` property on our `HelloWorld` class.

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

We added a new property for our `@View` annotation: `directives: [NgFor]`. Unlike Angular 1 where every directive was available in a global namespace, Angular 2 requires that you explicitly state which directives you want to use. This makes the view *require* the `NgFor` directive.

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

The `*ng-for` syntax says we want to use the `NgFor` directive on this attribute.

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 *ng-for="#foobar of names">Hello {{ foobar }}</li>
```

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

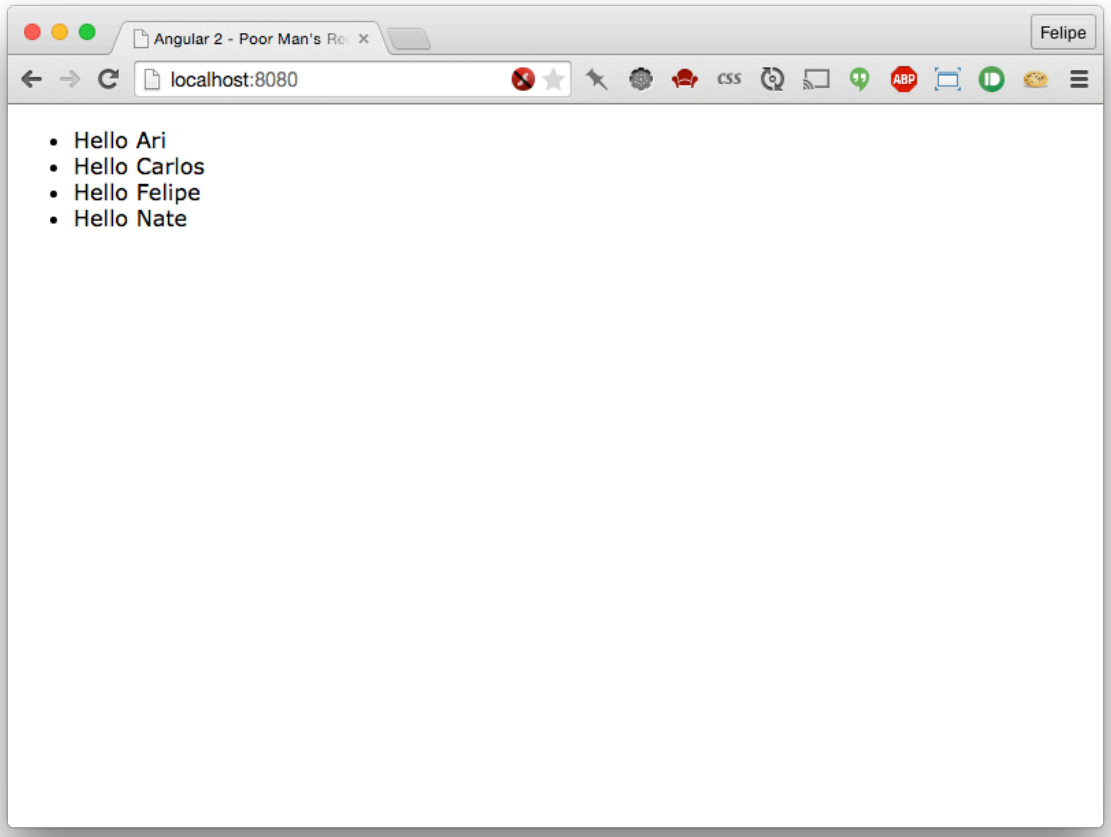
```
1   <li *ng-for="#name of foobar">Hello {{ name }}</li>
```

We'd get an error because `foobar` isn't a property on the component. You can think of this directive like a `for each` loop.

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 ng-for directive here[7]

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

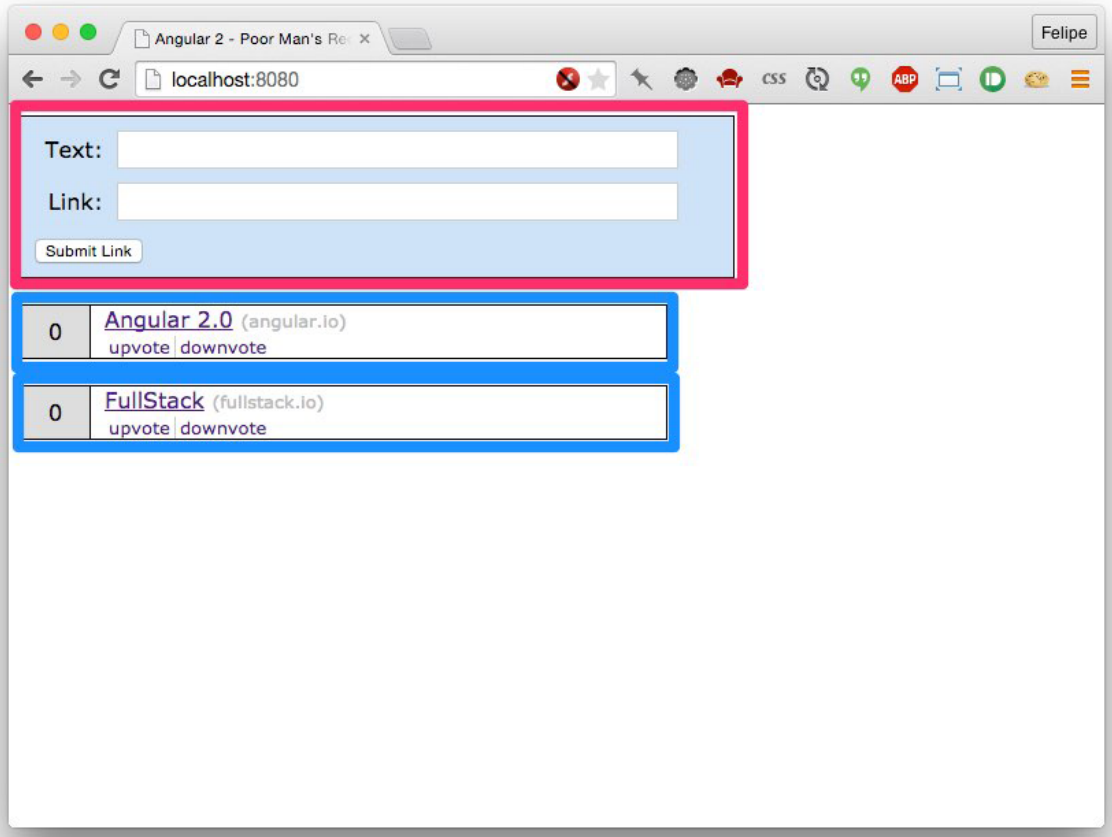
[7] https://github.com/angular/angular/blob/master/modules/angular2/src/common/directives/ng_for.ts



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 form used to submit new articles would be one component (marked in red in the picture)
2. Each article would be another component (marked in blue).

The form component

Let's start building the form component. For that, we'll change our `app.ts`. 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:

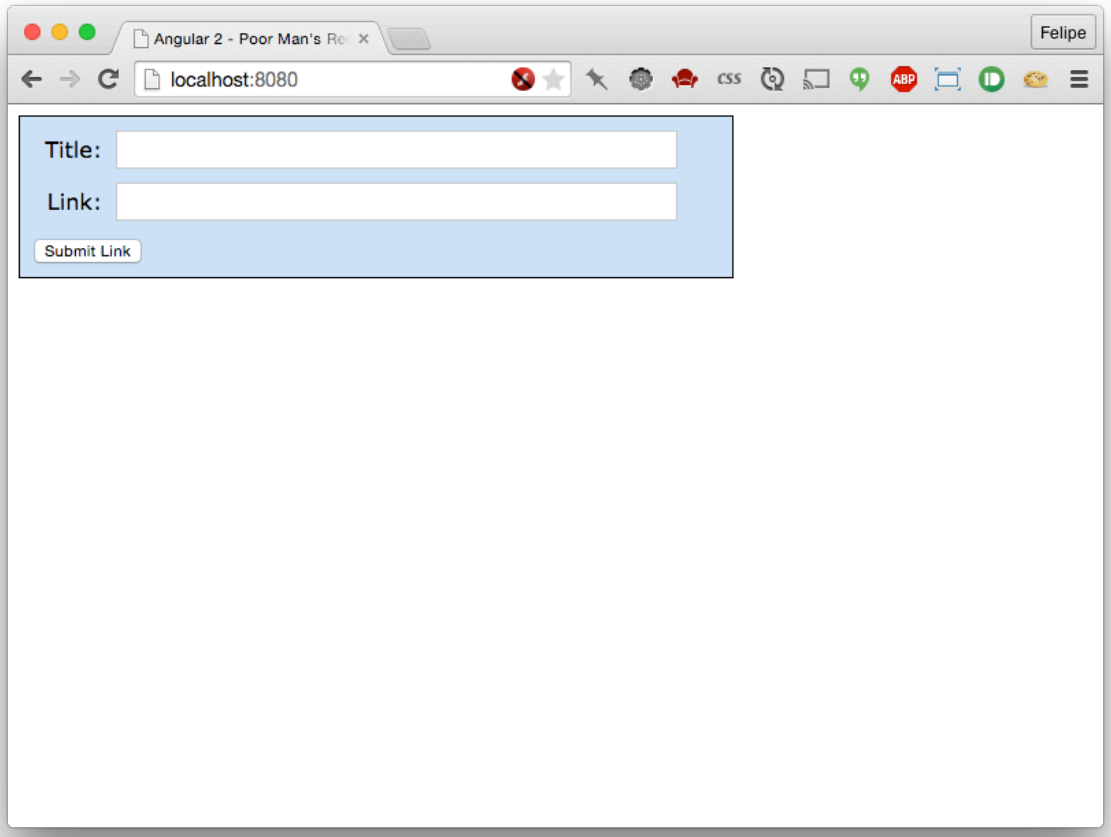
```
/// <reference path="typings/angular2/angular2.d.ts" />

import {
  Component,
  NgFor,
  View,
  bootstrap,
} from "angular2/angular2";

@Component({
  selector: 'reddit'
})
@View({
  template: `
    <section class="new-link">
      <div class="control-group">
        <div><label for="title">Title:</label></div>
        <div><input name="title"></div>
      </div>
      <div class="control-group">
        <div><label for="link">Link:</label></div>
        <div><input name="link"></div>
      </div>

      <button>Submit Link</button>
    </section>
  `
})
class RedditApp {
}

bootstrap(RedditApp);
```

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

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

After updating your `app.ts`, use our new component by changing your `index.html` and replacing the `<hello-world></hello-world>` tag with `<reddit></reddit>`.

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



Form

Adding Interaction

Now, if you click the submit button, nothing will happen because we haven't added any behavior to our application.

Let's add some interaction:

```
@Component({
  selector: 'reddit'
})
@View({
  template: `
    <section class="new-link">
      <div class="control-group">
        <div><label for="title">Title:</label></div>
        <div><input name="title" #newtitle></div>
      </div>
      <div class="control-group">
        <div><label for="link">Link:</label></div>
        <div><input name="link" #newlink></div>
      </div>

      <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
    </section>
  `,

  directives: [NgFor]
})
class RedditApp {
  addArticle(title, link) {
    console.log("Adding article with title", title.value, "and link", link.value\
);
  }
}
```

To add interaction to our application we need to do two or three things:

1. Create a method in our Component class (`addArticle` in this case) that contains the logic of our action
2. Bind the action to an event in our view (here on our `button` tag)
3. Bind values in inputs to variables that can be used in our action

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

Binding `input`s to values

Notice in our first input tag we have the following:

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

The new syntax here is the *reference* to `#newtitle`. This markup tells angular to bind this input to the variable `newtitle`. The effect is that this makes the variable `newtitle` available to the actions within this view. `newtitle` is now an object that represents this `input` tag and we can 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 these things come from?

1. `addArticle` is a function on our Component definition class `RedditApp`
2. `newtitle` comes from the resolve (`#newtitle`) on our `input` for `title`
3. `newlink` comes from the resolve (`#newlink`) on our `input` for `link`

All together:

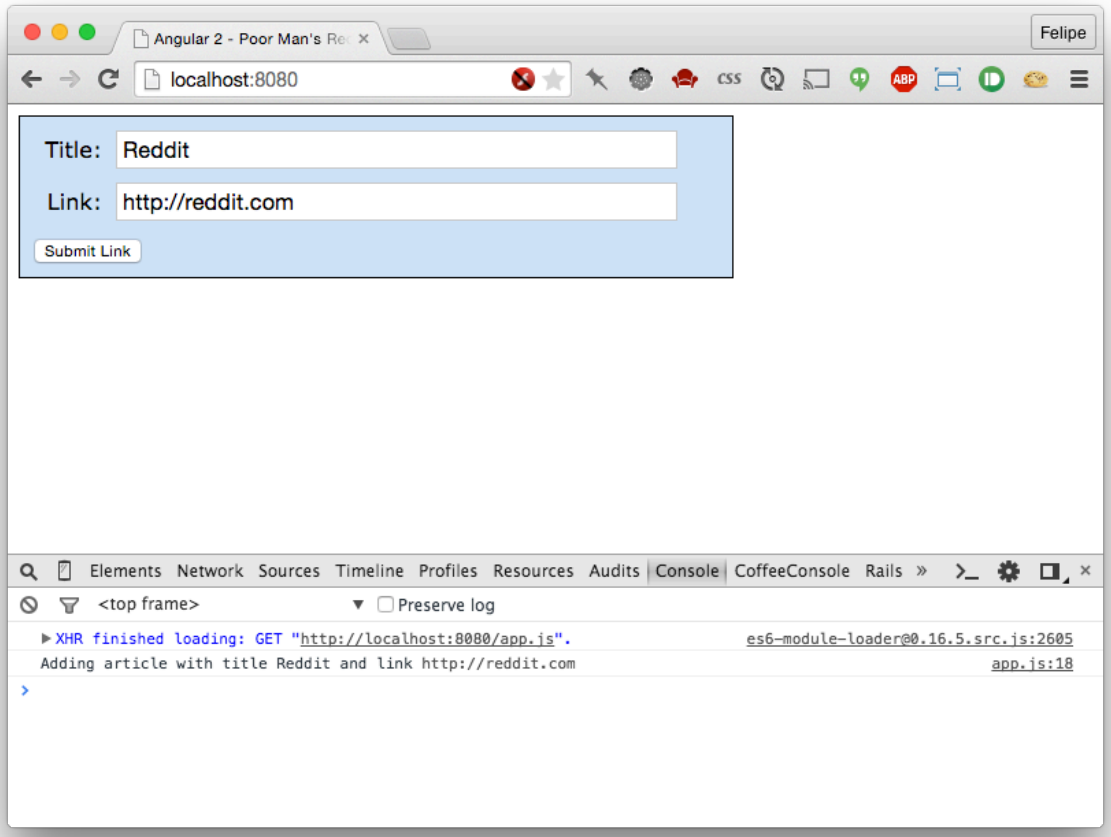
```
<button (click)="addArticle(newtitle, newlink)">Submit Link</button>
```

Defining the Action Logic

On our `class RedditApp` we define a new function called `addArticle`. It takes two arguments: `title` and `link`. For now, we're just going to `console.log` out those arguments.

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:

```
@Component({
  selector: 'reddit-article'
})
@View({
  template: `
  <article>
    <div class="votes">{{ votes }}</div>
    <div class="main">
      <h2>
        <a href="{{ link }}">{{ title }}</a>
      </h2>
      <ul>
        <li><a href (click)='voteUp()'>upvote</a></li>
        <li><a href (click)='voteDown()'>downvote</a></li>
      </ul>
    </div>
  </article>
  `
})
class RedditArticle {
  votes: number;
  title: string;
  link: string;

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

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

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

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 by annotating the class with `@View`
3. Creating a component-definition class (e.g. `RedditArticle`) which houses our component logic

Creating the `reddit-article` Component First, we define a new Component with `@Component`. We're saying that this component is placed by using the tag `<reddit-article>` (i.e. the selector is a tag name).

Creating the `reddit-article` View Second, we define the view with `@View`. There are three ideas worth noting here:

1. 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 `RedditArticle` class.
2. We can also use template strings in property values like we do in the a `href="{{ link }}"`. The value of the `href` will be dynamically populated with the value of `link` from the component class
3. 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 `RedditArticle` class (and downvote respectively).

Creating the `reddit-article` RedditArticle Definition Class Here we create three properties for each `RedditArticle`:

1. `votes` - a `number` representing the sum of all upvotes, minus the sum of 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:

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

And we define two functions for voting, one for voting up and one for voting down.

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

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

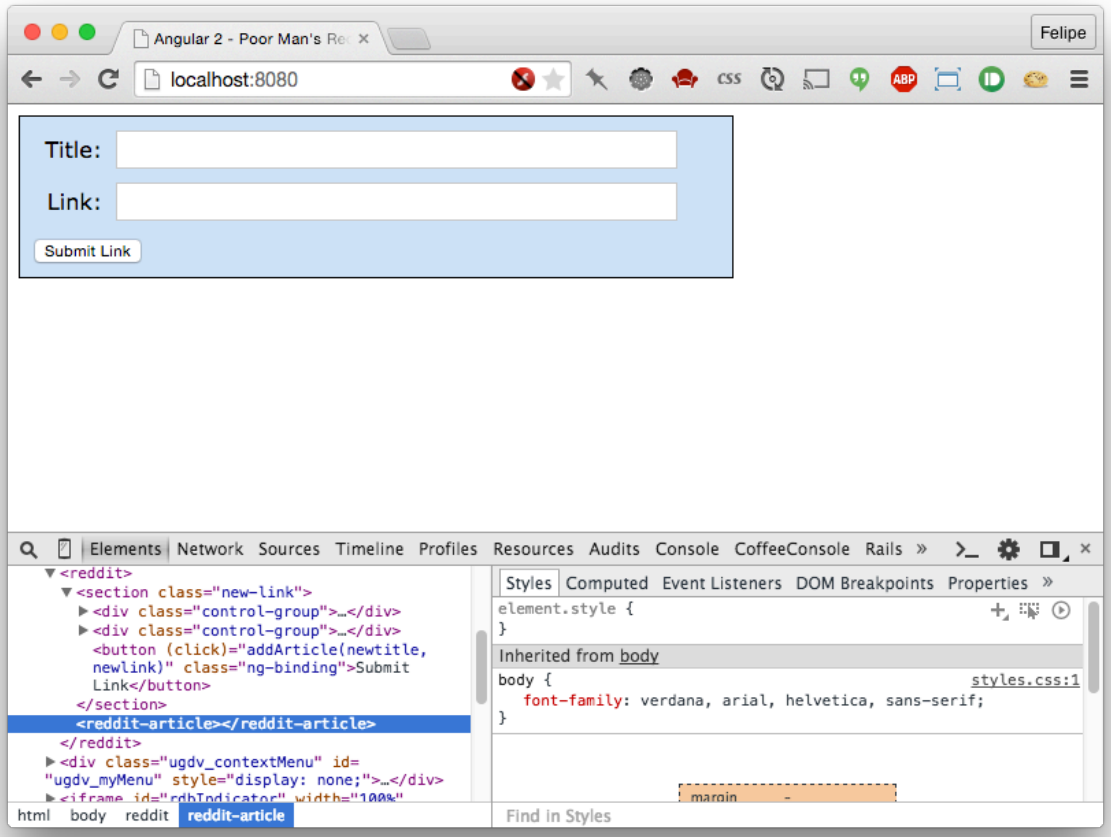
In `voteUp` we increment `this.votes` by one. Similarly we decrement for `voteDown`.

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 that component code. First we will add the `<reddit-article>` tag to the `RedditApp`'s template:

```
template: `
  <section class="new-link">
    <div class="control-group">
      <div><label for="title">Title:</label></div>
      <div><input name="title" #newtitle></div>
    </div>
    <div class="control-group">
      <div><label for="link">Link:</label></div>
      <div><input name="link" #newlink></div>
    </div>

    <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
  </section>


  <reddit-article></reddit-article>
`
```

If we try to reload the browser now, you will see that the `<reddit-article>` tag wasn't compiled, as can be seen inspecting the DOM here:



Unexpanded tag when inspecting the DOM

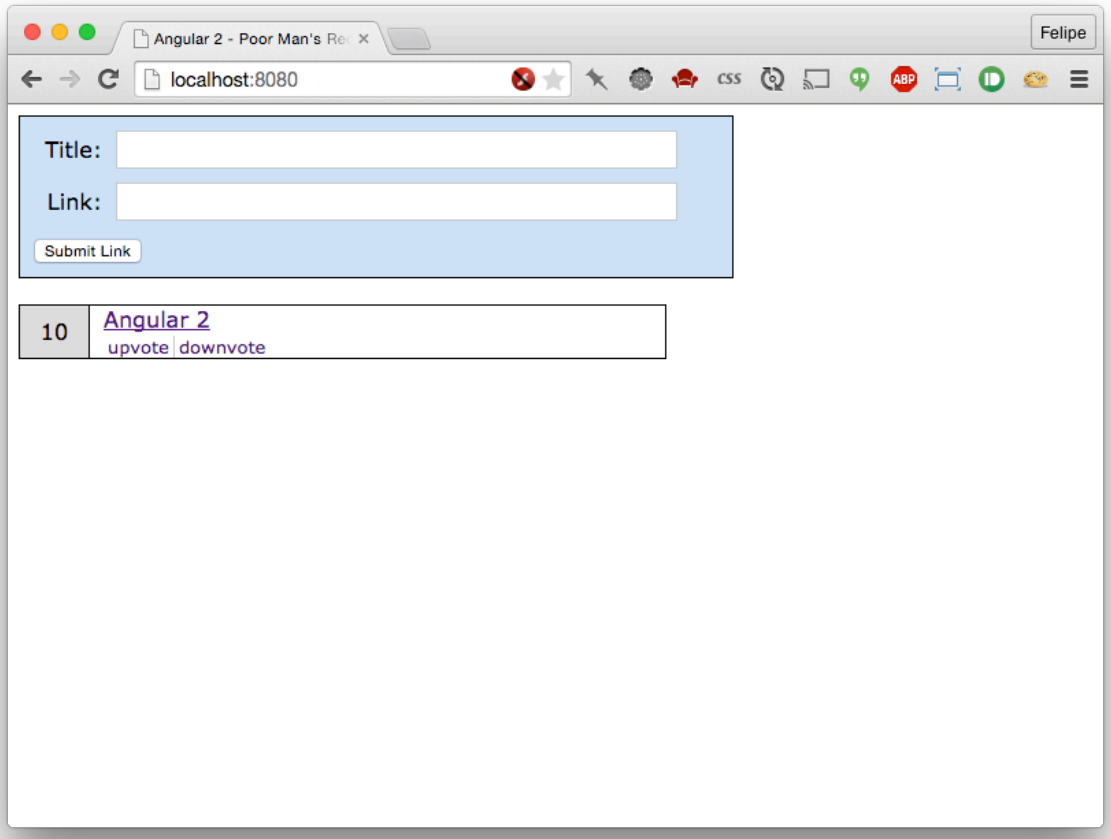
That happens because the `RedditApp` component doesn't know about the `RedditArticle` component yet!

To fix that, we just have to declare the `RedditArticle` component on the `directive` property of the `RedditApp`'s view annotation, just like we did when we used `For` before:

```
1  template: `
2    <section class="new-link">
3      <div class="control-group">
4        <div><label for="title">Title:</label></div>
5        <div><input name="title" #newtitle></div>
6      </div>
7      <div class="control-group">
8        <div><label for="link">Link:</label></div>
9        <div><input name="link" #newlink></div>
10     </div>
```

```
11
12      <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
13    </section>
14
15    <reddit-article></reddit-article>
16  `,
17
18  directives: [RedditArticle]
```

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



Rendered RedditArticle 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 just 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() {
2    this.votes += 1;
3    return false;
4  }
5
6  voteDown() {
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 components

Creating an Article class

Right now we only have one article in the page and there's no way to render more, unless we create add a new `<reddit-article>` tag. And even if we do, all the articles would have the same content, so it wouldn't be very interesting.

A good practice when writing Angular2 code is to try to isolate the data structures you are using from the component code. In order to accomplish that, and before making any further changes to the component, let's create a data structure that would represent one article. Add the following code before the `RedditArticle` component code:

```
1  class Article {
2    title: string;
3    link: string;
4    votes: number;
5
6    constructor(title, link) {
7      this.title = title;
8      this.link = link;
9      this.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`, and we also assume the recently submitted article has zero votes.

Now let's change the `RedditArticle` code to use our new `Article` class. Instead of storing the properties directly on the `RedditArticle` component, instead we're storing the properties on the `Article` class and simply storing the array of `Articles` in our component:

```
@Component({
  selector: 'reddit-article'
})
@View({
  template: `
      <article>
        <div class="votes">{{ article.votes }}</div>
        <div class="main">
          <h2>
            <a href="{{ article.link }}">{{ article.title }}</a>
          </h2>
          <ul>
            <li><a href (click)='voteUp()'>upvote</a></li>
            <li><a href (click)='voteDown()'>downvote</a></li>
          </ul>
        </div>
      </article>
      `
})
class RedditArticle {
  article: Article;

  constructor() {
    this.article = new Article('Angular 2', 'http://angular.io');
  }

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

  voteDown() {
    this.article.votes -= 1;
    return false;
```

```
35    }
36  }
```

Notice that we substitute all of the properties with the `article` instead. We can also use our new class `Article` as a type for the `article` property!

If you reload the browser, you're going to see everything works the same way. That's good but something in our code is still a little off: our `voteUp` and `voteDown` methods break the encapsulation of the `Article` class by changing their internal properties directly.

`voteUp` and `voteDown` current break the Law of Demeter[8] 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".

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

The problem is that our `RedditArticle` component knows too much about the `Article` class internals. To fix that, let's also move the methods to the `Article` class, changing `RedditArticle` to cope with the change:

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

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

```
11
12    voteUp() {
13      this.votes += 1;
14      return false;
15    }
16
17    voteDown() {
18      this.votes -= 1;
19      return false;
20    }
21  }
22
23  @Component({
24    selector: 'reddit-article'
25  })
26  @View({
27    template: `
28        <article>
29          <div class="votes">{{ article.votes }}</div>
30          <div class="main">
31            <h2>
32              <a href="{{ article.link }}">{{ article.title }}</a>
33            </h2>
34            <ul>
35              <li><a href (click)='article.voteUp()'>upvote</a></li>
36              <li><a href (click)='article.voteDown()'>downvote</a></li>
37            </ul>
38          </div>
39        </article>
40        `
41  })
42  class RedditArticle {
43    article: Article;
44
45    constructor() {
46      this.article = new Article('Angular 2', 'http://angular.io');
47    }
48  }
```

Checkout our `RedditArticle` component definition now: it's so short! We've moved a lot of logic **out** of our component and into our models. An analogous MVC guideline would be Fat Models, Skinny Controllers[9]. 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, 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:

```
class RedditApp {
  articles: Array<Article>;

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

  addArticle(title, link) {
    console.log("Adding article with title", title.value, "and link", link.value\
);
  }
}
```

Notice that our `RedditApp` has the line:

```
  articles: Array<Article>;
```

The `Array<Article>` might look a little funny if you're not used to typing your javascript. The word for this pattern is *generics*. It's a concept seen in Java, among others, and 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 that list by setting `this.articles` in the constructor:

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

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

Configuring a `RedditArticle` Component with properties

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

Here we're introducing a new attribute of `Component` called **`inputs`**.

We can configure out `Component` with `inputs` that are passed to it from it's parent.

Previously we had our `RedditArticle` Component class defined like this:

```
class RedditArticle {
  article: Article;

  constructor() {
    this.article = new Article('Angular 2', 'http://angular.io');
  }
}
```

The problem here is that we've hard coded a particular `Article` in the constructor.

What we would really like to do is be able to configure the `Article` we want to display in markup, like this:

```
<reddit-article article="article1"></reddit-article>
<reddit-article article="article2"></reddit-article>
```

In order to use an attribute in the HTML as an input to our component we use the `inputs` option of `Component`.

It looks like this:

```
@Component({
  selector: 'reddit-article',
  inputs: ['article'],
})
@View({
  // same ...
})
class RedditArticle {
  article: Article;
}
```

Notice that `inputs` is an `Array` where you enumerate all the properties your component can receive.

So our full listing of `RedditArticle` looks like this:

```
@Component({
  selector: 'reddit-article',
  inputs: ['article'],
})
@View({
  template: `
      <article>
        <div class="votes">{{ article.votes }}</div>
        <div class="main">
          <h2>
            <a href="{{ article.link }}">{{ article.title }}</a>
          </h2>
          <ul>
            <li><a href (click)='article.voteUp()'>upvote</a></li>
            <li><a href (click)='article.voteDown()'>downvote</a></li>
          </ul>
        </div>
      </article>
      `
})
class RedditArticle {
  article: Article;
}
```

Here we added a new `inputs` property to the `@Component` annotation. This allows the component to receive an `article` property from the DOM, while making it available as a `article` within the template.

What's great here is that we've totally eliminated the functions in the class definition `RedditArticle`! This helps make this component be a bit easier to reason about.

Rendering a list of Articles

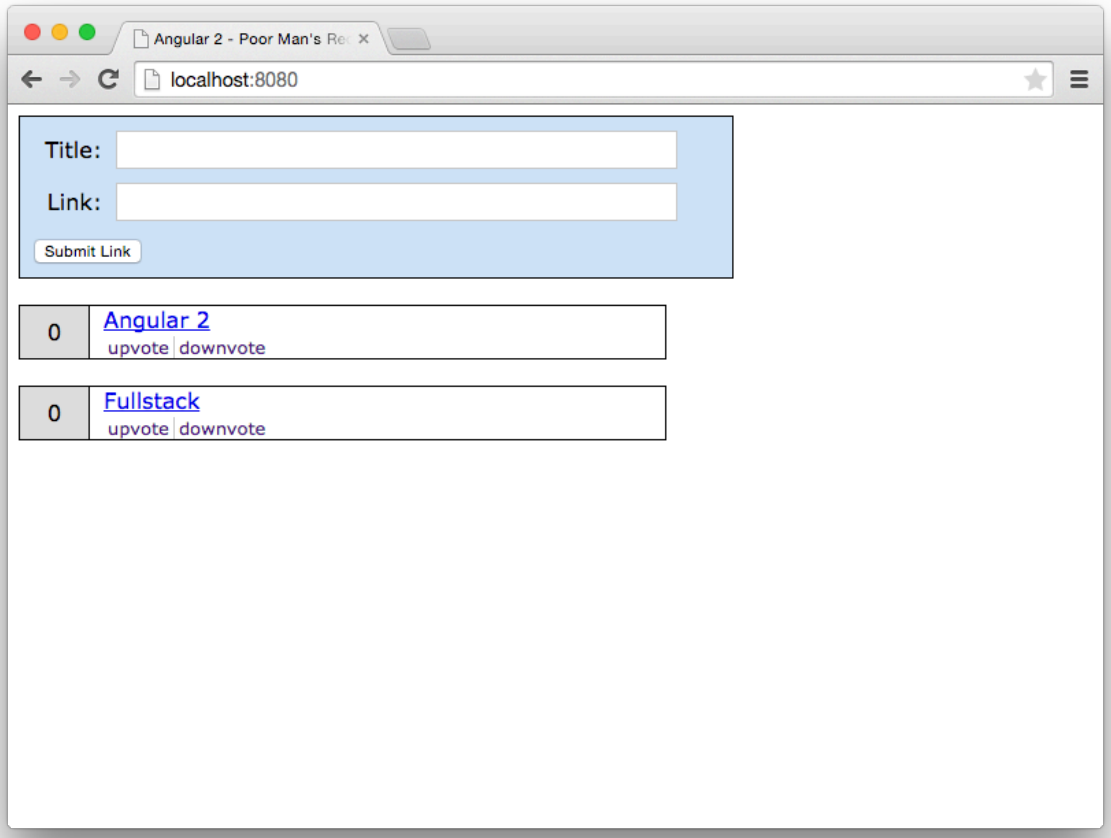
Let's configure `RedditApp` to render all the `articles`. To do so, instead of having the `<reddit-article>` tag alone, we are going to use the `For` directive again, to render multiple tags:

```
@Component({
  selector: 'reddit'
})
@View({
  template: `
    <section class="new-link">
      <div class="control-group">
        <div><label for="title">Title:</label></div>
        <div><input name="title" #newtitle></div>
      </div>
      <div class="control-group">
        <div><label for="link">Link:</label></div>
        <div><input name="link" #newlink></div>
      </div>

      <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
    </section>

    <reddit-article
       *ng-for="#article of articles"
       [article]="article">
    </reddit-article>
  `,

  directives: [RedditArticle, NgFor]
})
class RedditApp {
  addArticle(title, link) {
    console.log("Adding article with title", title.value, "and link", link.value\
);
  }
}
```

Remember when we rendered a list of names as a bullet list using the `NgFor` directives? Well, that also works for rendering multiple components. The `*ng-for="#article of articles"` syntax will iterate through the list of articles and creating the local variable `article`.

To indicate the article we want to render on each iteration, we use the `[article]="article"` notation. The square brackets indicate that we are setting the component's `article` variable to receive the template's `article` local variable we declared inside our `*ng-for` clause. Phew.

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



Multiple articles being rendered

The only thing left now is to change the `addArticle` method to add a new `Article` when you click the submit button:

```
addArticle(title, link) {
  this.articles.push(new Article(title.value, link.value));
  title.value = '';
  link.value = '';
}
```

This will:

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

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

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

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

```
domain() {
  var link = this.link.split('//')[1];
  return link.split('/')[0];
}
```

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

```
@View({
  template: `
  <article>
    <div class="votes">{{ article.votes }}</div>
    <div class="main">
      <h2>
        <a href="{{ article.link }}">{{ article.title }}</a>
        <span>({{ article.domain() }})</span>
      </h2>
      <ul>
        <li><a href (click)='article.voteUp()'>upvote</a></li>
        <li><a href (click)='article.voteDown()'>downvote</a></li>
      </ul>
    </div>
  </article>
  `
})
```

And now when we reload the browser, we should see the completed application.

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

Onward!

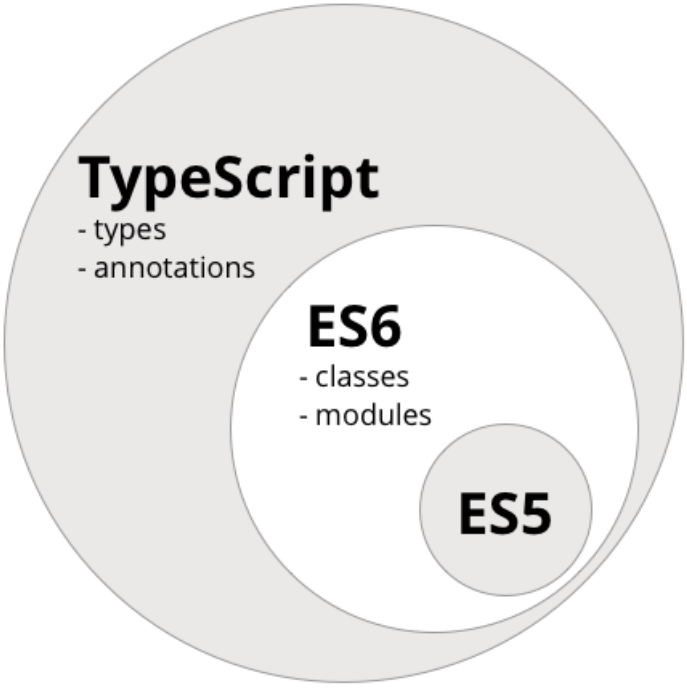
TypeScript

Angular 2 is built in TypeScript

Angular 2 is built in a Javascript-like language called TypeScript[10].

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.

[10] 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**[11] by Google and **babel**[12] 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.

[11] https://github.com/google/traceur-compiler
[12] 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 it's name:

```
var name: string;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Trying it out with a REPL

To play with the examples on this chapter, let's install a nice little utility called **TSUN**[13] (TypeScript Upgraded Node):

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

Now start tsun:

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

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

>
```

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

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

Built-in types

String

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

```
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`:

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

Boolean

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

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

Array

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

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

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

Or similarly with a `number`:

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

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

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

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

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

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

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

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

Any

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

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

Void

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

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

Classes

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

A number of good practices have been adopted by the JavaScript community to compensate the lack of classes. A good summary of those good practices can be found in Mozilla Developer Network's JavaScript Guide[14], and you can find a good overview on the Introduction to Object-Oriented Javascript[15] page.

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

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

```
class Vehicle {
}
```

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

Properties

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

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

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

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

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

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

Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    // set initial age
    p.age = 6;

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

    // -> 18
```

Constructors

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

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

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

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

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

Is the same as:

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

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

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

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

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

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

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

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

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

It makes our previous example a little easier to write:

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

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

Inheritance

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

If you want to have a deeper understanding of how inheritance used to work in ES5, take a look at the Mozilla Developer Network article about it: Inheritance and the prototype chain[16].

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:

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

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

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

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

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

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

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

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

Running this should show:

```
First line
Second line
```

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

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

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

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

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

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

Utilities

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

- fat arrow function syntax
- template strings

Fat Arrow Functions

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

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

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

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

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

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

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

Or as a statement:

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

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

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

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

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

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

Template Strings

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

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

Variables in strings

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

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

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

console.log(greeting);
```

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

Multiline strings

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

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

// do something with `template`
```

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

Wrapping up

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

- Interfaces
- Generics
- Importing and Exporting Modules
- 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 in Angular 2. The idea is that by taking a step back 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. In the next chapter, 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, ng2 Components have some significant advantages over ng1 directives and we'll talk about that below. First, let's start at the top: the Application.

Application

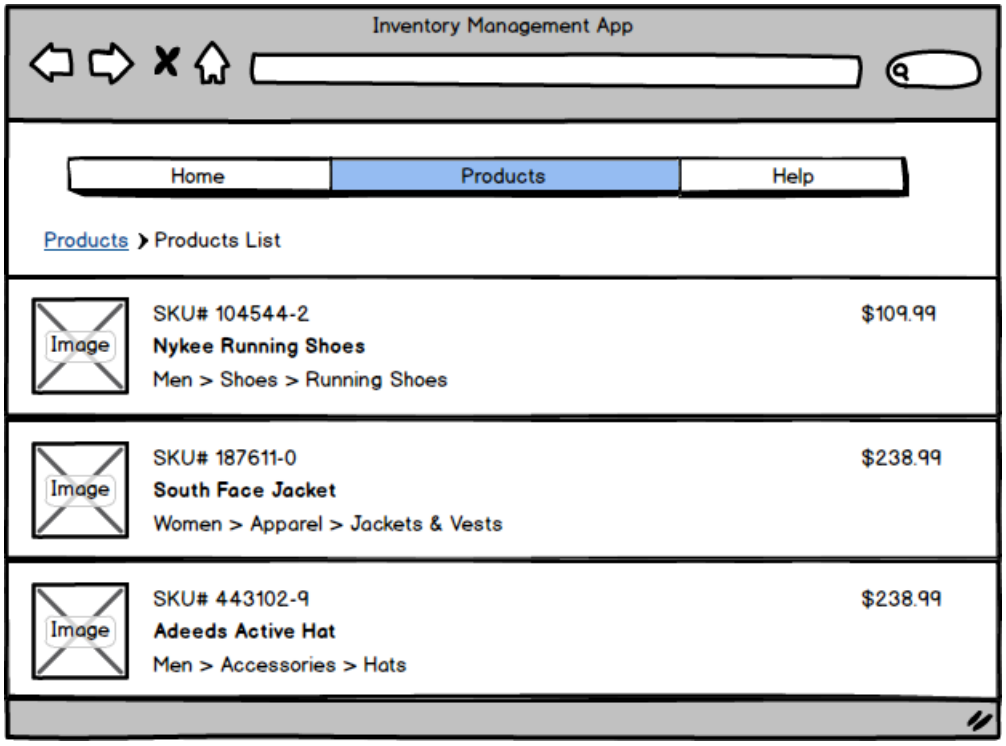
An ng2 Application is nothing more than a tree of Components.

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 talk about an imaginary 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'd to do is split it up into individual components (organized into a tree).

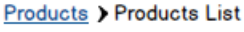
In this example, we could group the page into three high level components

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

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

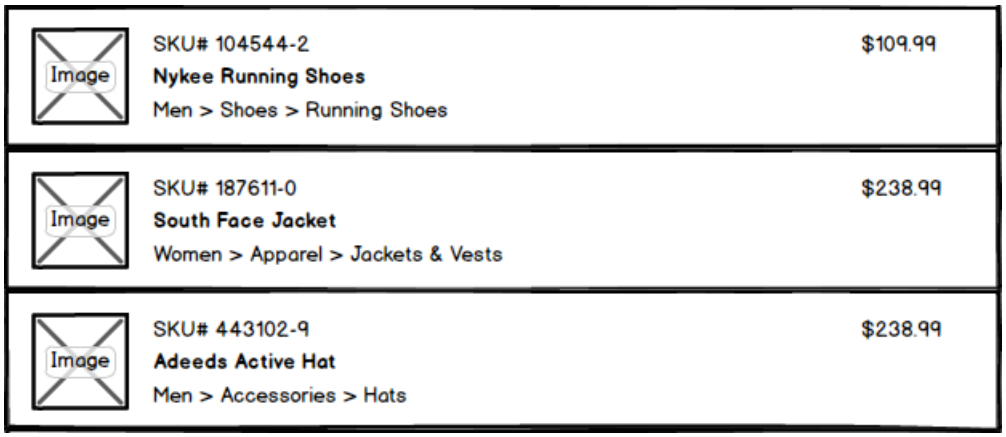
Navigation Component

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



Breadcrumbs Component

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



Product List Component

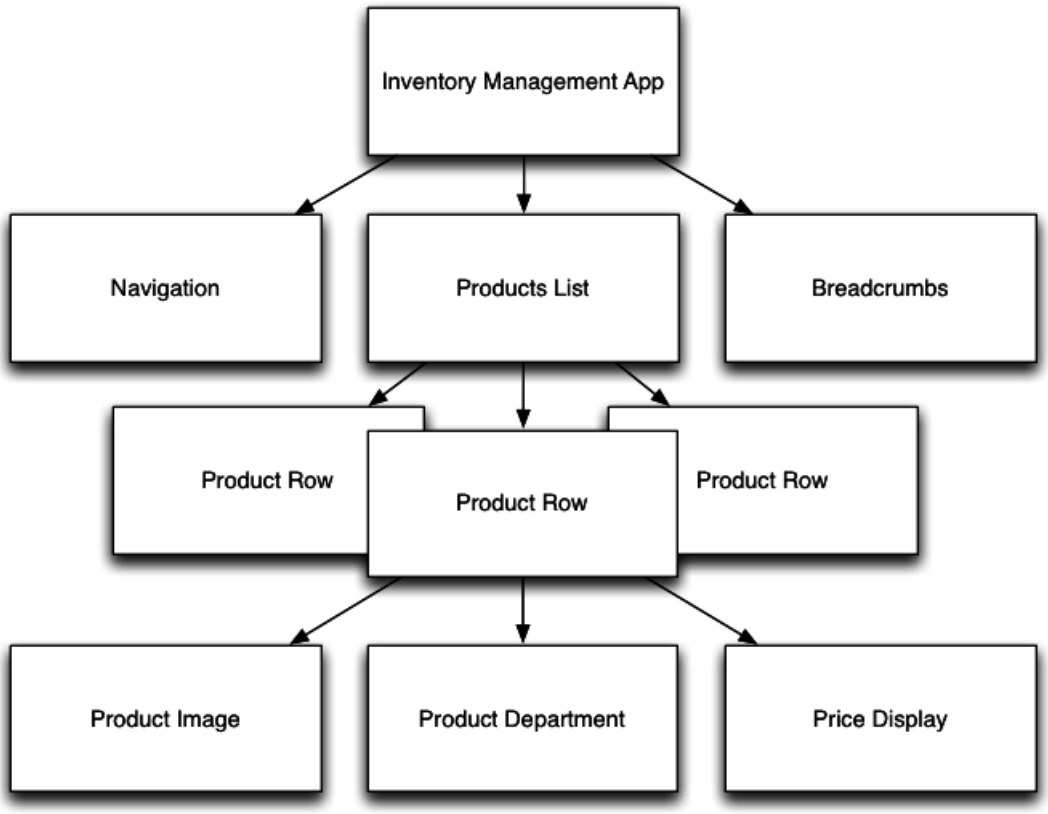
Breaking this component down into the next level of smaller components, we could say that the Product List is composed of multiple Product Rows.

Product Row Component

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

- the **Product Image** component, that would be responsible to render a product image, given its image name (imagine that the component knows how to get the image using a proper Amazon S3 URL, for instance);
- the **Product Department** would have the responsibility of, given a department id, render the whole department tree, like *Men > Shoes > Running Shoes*;
- the **Price Display** would be a more generic component, that could be reused accross the application, to properly render a 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.

One important thing to note is that each application can only have one top level component.

Components

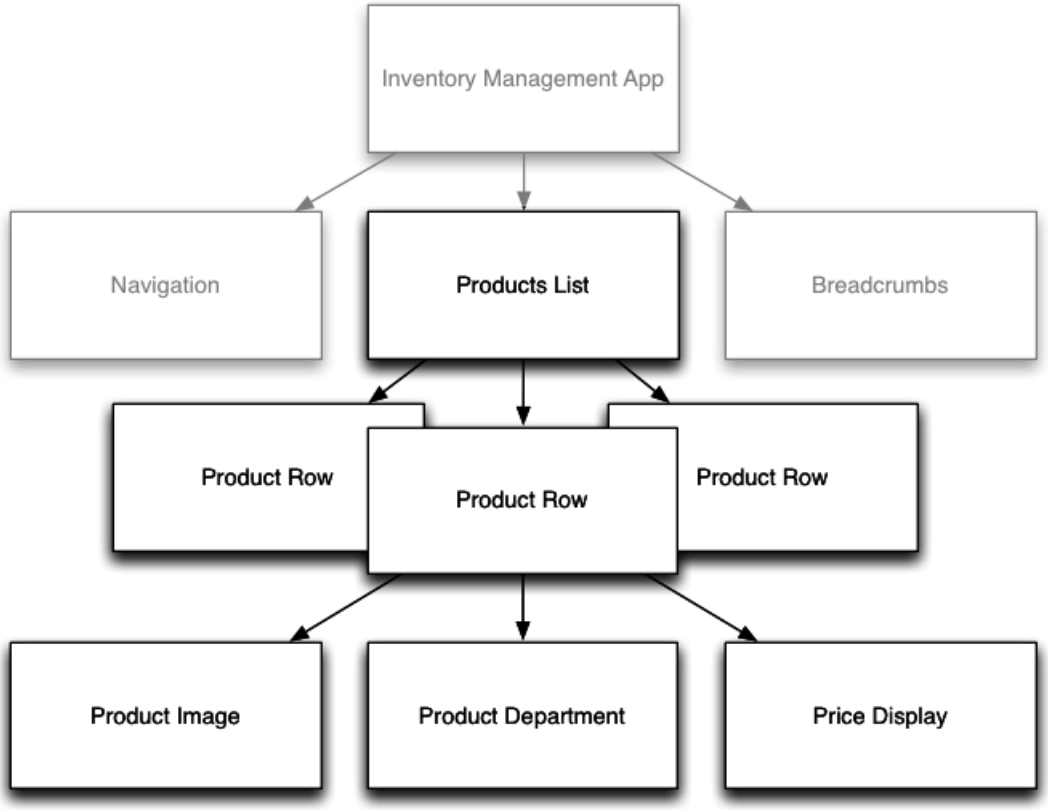
As you can see, Components are the fundamental building block of Angular 2 applications. We break our application into more granular child components.

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

Each components is composed of three parts:

- Component *Decorator*
- A View
- A Controller

To illustrate the key concepts we need to understand about components, let's focus on the **Products List** its child components:



Products List Component

An implementation of the Products List component could be:

```
1     // The "component decorator"
2     @Component({
3       selector: 'products-list',
4       inputs: ['products']
5     })
6
7     // The view annotation
8     @View({
9       directives: [ProductRow, NgFor],
10      template: `
11      <div class="products-list">
12        <div product-row *ng-for="#product in products" [product]="product">
13        </div>
```

```
14        </div>
15        `
16      })
17
18      // The "controller" class
19      class ProductsList {
20        products: Array<Product>;
21      }
```

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:

Both the **component decorator** and the **view definition** sections are represented by **annotations**, however the **controller** is represented by a **class**.

Not sure how annotations work? Checkout the Annotations Section in the Typescript Chapter

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

Component Decorator

The component decorator is where you declare 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. Here 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, say, 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: 'products-list'`, we're saying that in our HTML we want to match the `products-list` tag, that is, we're defining a new tag that has built in functionality whenever we use it. E.g. when we put this in our HTML:

```
1  <products-list></products-list>
```

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

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

```
<div products-list></div>
```

Component `inputs`

Our `@Component` annotation also has another key: `inputs`. With `inputs` we're describing the configurable parameters we expect our component to receive. The idea here is that we can pass in an `Array` of `Products` which this component will render.

You declare all the parameters your component expects to *receive* from other components in the `inputs` key.

`inputs` takes an array of strings which specifies all "inputs" to this component. They strings can simply be each property name or have the format `'componentProperty: exposedProperty'`.

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

```
  @Component({
    //...
    inputs: ['name', 'age', 'enabled']
    //...
  })
  class MyComponent {
    name: string;
    age: number;
    enabled: boolean;
  }
```

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

```
  @Component({
    //...
    inputs: [
      'name: name',
      'age: age',
      'isEnabled: enabled'
    ]
    //...
  })
  class MyComponent {
    name: string;
    age: number;
    isEnabled: boolean;
  }
```

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

```
@Component({
  //...
  inputs: ['name', 'age', 'isEnabled: enabled']
  //...
})
class MyComponent {
  name: string;
  age: number;
  isEnabled: boolean;
}
```

In the `inputs` array, when the strings are on 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**.

In this case, for `name` and `age` we can use a simple string . However, for the property `enabled` we chose different values: `enabled` would be referred on the (incoming) view as `enabled` but it would be translated into a controller `isEnabled` property. We'll see more examples of this shortly.

How the inputs are exposed to the world is done on the View part of the component, as we'll start learning now.

Controller

The controller of a component is a class that **holds all the component inputs** and contains the **implementation of behavior** that the component should have.

```
class ProductsList {
  products: Array<Product>;
}
```

In this case, we're specifying an instance variable of `products` which is an `Array` of `Product` objects. But in this example we're not defining any behavior (we will before long).

Views

You can think of the view as the visual part of the component. It is represented by the `@View` annotation and it declares the HTML template that the component will have.

```
// The view annotation
@View({
  directives: [ProductRow, NgFor],
  template: `
  <div class="products-list">
    <div product-row *ng-for="#product in products" [product]="product">
    </div>
  </div>
  `
})
```

This `@View` annotation has many possible configuration options, but we're using only two here:

- `directives`: 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 `ProductRow` directive.
- `template`: This specifies the HTML template that we want to use for rendering this component. Notice that we're using TypeScript's backtick "` multi-line string syntax. Also you've probably noticed there is a lot of syntax in that view. We'll go over each part in detail.

Inputs and Ouputs

There are two more major concepts we need to be aware of to use components: **input bindings** and **output bindings**.

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

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

For "reading" data Data in your component are available to the View by using input bindings. In our example above, we're exposing our `products` array to the view by defining it as an input.

For "writing" data Our application isn't going to be read-only. There are also going to be inputs by the user. To handle an input from a user and send information out of this component we use **outputs**.

For example, if we want to add some behavior when a button is clicked or a form is submitted, we *declare* that intention on the View. That is, we add output bindings in our view. However, the actual implementation of handling that event takes place in the controller.

When talking about *inputs*, the component decorator knows *what* inputs exist and the view defines *how* they are rendered. When talking about *outputs, it's the other way around: the view knows _what* outputs will be triggered and the controller knows *how* they are handled.

In an early version of Angular 2, `inputs` and `outputs` were called `properties` and `events`. While the new naming is better, the original naming gives some insight into how they were originally used: `inputs` are "public component properties" and `outputs` are *events* emitted out of the component.

Data binding

Let's take a look at how our view can *bind* to the component data by using inputs. There are two ways our view can use inputs.

Private Properties

The first way is to just inline the expression, for example:

```
1     @View({
2       template: `
3       The next number is {{ myNumber + 1 }}
4       `
5     })
6     class SomeComponent {
7       myNumber: number;
8
9       constructor() {
10        this.myNumber = 2;
11      }
12    }
```

Here we are using the `myNumber` property that the controller declared within an expression `{{ myNumber + 1 }}`. That expression is then replaced when rendering the component. In our case it would render the `The next number is 3` text.

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.

This previous example is an instance of a *controller-only property*. You can think of it as a private property for your component. A "private", or controller-only, property is only visible within the component itself.

Public Inputs

The second way to use an expression is by using a *component input*.

Component inputs are declared **on the component decorator** and therefore made visible outside the component. You can think of the `inputs` as "public properties" of the component.

```
    @Component({
      selector: 'some-component',
      inputs: ['number']
    })

    @View({
      template: `
      The next number is {{ number + 1 }}
      `
    })
    class SomeComponent {
      number: number;
    }
```

In this example, we're exposing a public input `number`, which means we allow a value to be passed in to this component from an external component.

But now that we've exposed this input, how do we actually use it?

When using components you *bind* values to inputs using the `[input]="expression"` notation.

So say we wanted to pass a number to `SomeComponent` in a view, we could do the following:

```
  <some-component [number]='41'>
```

And then our component would render `The next number is 42`.

The syntax for setting inputs on components is the same whether we are using custom components or built-in components.

For instance, if we wanted our view to render an `input` tag with a `value` of the number, we could use the same property-setting notation:

```
@View({
  template: `
  <input [value]='number + 1'>
  `
})
class SomeComponent {
  number: number;

  constructor() {
    this.number = 2;
  }
}
```

Note here that there is a distinction between the `inputs` of an Angular component vs. the `input` tag in HTML. An `input` tag is the everyday form field you use to get data from the browser. `inputs` is metadata describing the properties that can be passed in to a component.

Output binding

In the last section we put data *in* to our component by using input binding. In this section, let's look at how we get data *out* of a component using outputs.

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

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

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

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

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

  constructor() {
    this.value = 1;
  }

  increase() {
    this.value++;
  }

  decrease() {
    this.value--;
  }
}
```

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 for 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 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. We'll do that in the next example.

Putting it all together

Lets change the application example we used earlier in the chapter and allow our `ProductsList` component know when we click a given row. We could use this event to redirect to an individual product's page.

In order for us to do that, let's first address the interaction of the `ProductsList` component and its children `ProductRow` components.

We'll start by writing the `ProductRow` component:

code/how_angular_works/inventory_app/app.ts

```
@Component({
  selector: 'product-row',
  inputs: ['product'],
  outputs: ['click']
})
@View({
  directives: [ProductImage, ProductDepartment, PriceDisplay],
  template: `
  <div class="product-row cf" (click)="clicked()">
    <product-image [product]="product"></product-image>
    <div class="product-info">
      <div class="product-sku">SKU #{{ product.sku }}</div>
      <div class="product-name">{{ product.name }}</div>
      <product-department [product]="product"></product-department>
    </div>
    <price-display [price]="product.price"></price-display>
  </div>
  `
})
class ProductRow {
```

```
  product: Product;
  click: EventEmitter;

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

  clicked() {
    this.click.next(this.product);
  }
}
```

```
@Component({
  selector: 'product-row',
  inputs: ['product'],
  outputs: ['click']
})
@View({
  template: `
  <div class="product-row" (click)="clicked()">
    <div product-image [product]="product">
    </div>
    <div product-department [department]="product.department_id">
    </div>
    <div price-display [price]="product.price">
    </div>
  </div>
  `,
  directives: [ProductImage, ProductDepartment, PriceDisplay]
})
class ProductRow {
  product: Product;
  click: EventEmitter;

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

  clicked() {
    this.click.next(this.product);
  }
}
```

There are a few things on `ProductRow` that looks new. Notice that we have a new `outputs` property on the component decorator: this is how you declare which custom events your component will trigger.

We are also now binding to the `click` event of our `product-row` `div` in the `@View`, using `(click)="clicked()"`.

When this div is clicked, Angular 2 is going to call our controller's `clicked()` method.

The missing piece here is how do we tell the ouside world that our component has been clicked. That's where the `EventEmitter` class comes in.

As the name says, whenever our component wants to emit an event to the outside world, we will use this class.

So if you check our controller's `clicked` method:

```
1  clicked() {
2    this.click.next(this.product);
3  }
```

You can see that we are calling the `next` method of the `EventEmitter`, and passing the product this `ProductRow` holds. What we are doing is forwarding the event on to any one who is listening. This way, if we had another component that used ours, we could write:

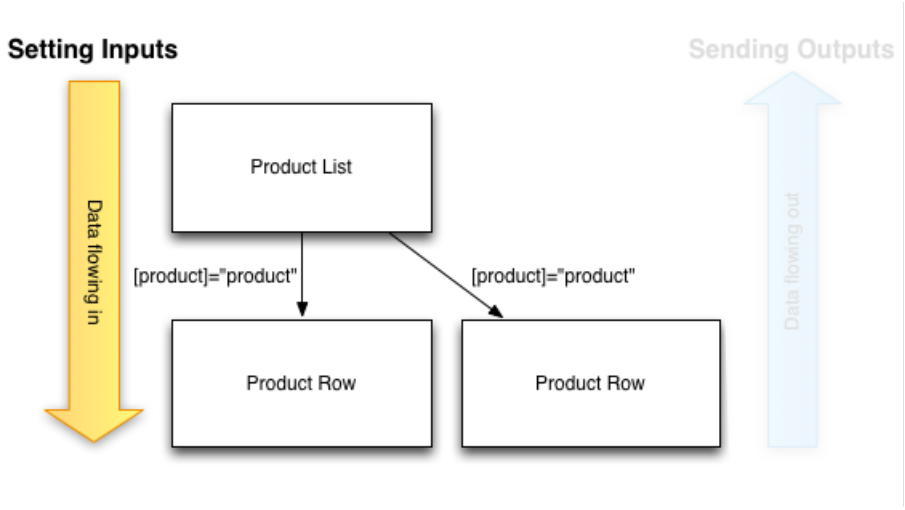
```
1  <div product-row [product]="product" (click)="display(product)">
2  </div>
```

On the `[product]="product"` snippet, we're setting the `ProductRow`'s `product` input to receive the calling component's `product` variable.

Then, this component controller's `display` method would be called when the `ProductRow` is clicked and the product would be passed to that method. Pretty neat.

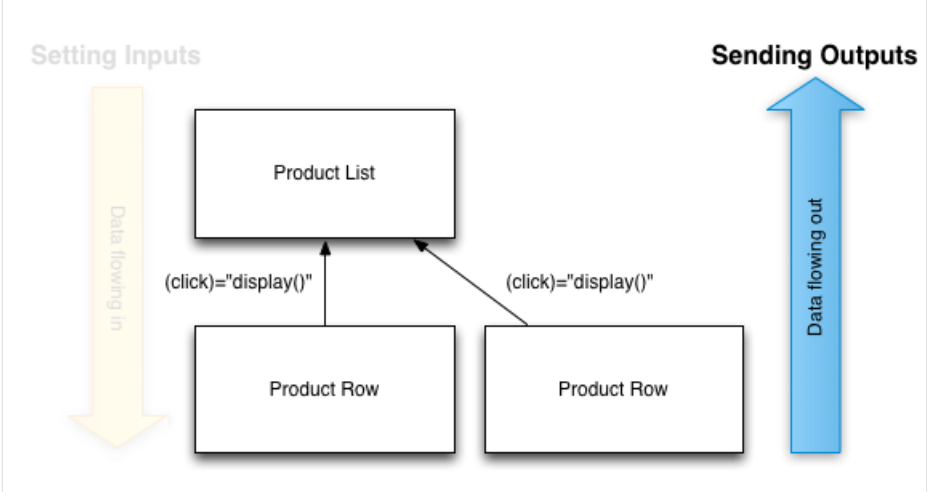
Data Flow

So we can say that data would flow **from** the `ProductsList` and **in** to our `ProductRow` component by setting its `product` attribute:



Data Flowing Into ProductRow

And data would flow out of the `ProductRow` component and into the `ProductsList` by having the `ProductsList` component bind to the `ProductRow`'s click event like below:



Data Flowing Out from ProductRow

It's important to know that data *binding* is one way: from parent to child. This binding is automatic, but **one directional** (by default). This differs from Angular 1, where data binding was bi-directional. It makes our program **much easier** to reason about. We'll talk more about how to manage data flow in further chapters.

Summary

In this chapter we learned that the core concept of Angular 2 is components. Any Angular 2 application is one component, composed of other smaller, more granular components.

Components are composed of three parts, which are the Component Decorator, a View and a Controller. The Component Decorator describes the configurable options of a component. In the View, we describe how the component will be rendered and which other components we'll use while rendering. Finally, on the Controller is where you create the component behavior, or the business logic behind it.

In Angular 2 all bindings are one way only. This is a big difference from Angular 1, where all the bindings are bidirectional.

For this reason, components should receive data through inputs and send data by using outputs. This way, other components can bind to those outputs and respond to them.

Now that we learned how Angular 2 works, let's start looking into components in more detail.

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.

NgIf

The `ng-if` 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:

```
<div [ng-if]="false"></div>          <!-- never displayed -->
<div [ng-if]="a > b"></div>          <!-- displayed if a is more than b -->
<div [ng-if]="str == 'yes'"></div>   <!-- displayed if str holds the string "yes"\
 -->
<div [ng-if]="myFunc()"></div>       <!-- displayed if myFunc returns a true valu\
e -->
```

If you have experience with Angular 1, you probably used `ng-if` 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 `ng-if` several times like this:

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

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

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

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

```
<div class="container">
  <div *ng-if="myVar == 'A'">Var is A</div>
  <div *ng-if="myVar == 'B'">Var is B</div>
  <div *ng-if="myVar == 'C'">Var is C</div>
  <div *ng-if="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something el\
se</div>
</div>
```

For cases like this, Angular 2 introduces the `ng-switch` 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 `ng-switch-when` directive
- Handle all the other unknown cases with `ng-switch-default`

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

```
<div class="container" [ng-switch]="myVar">
  <div *ng-switch-when="'A'">Var is A</div>
  <div *ng-switch-when="'B'">Var is B</div>
  <div *ng-switch-default>Var is something else</div>
</div>
```

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

```
<div class="container" [ng-switch]="myVar">
  <div *ng-switch-when="'A'">Var is A</div>
  <div *ng-switch-when="'B'">Var is B</div>
  <div *ng-switch-when="'C'">Var is C</div>
  <div *ng-switch-default>Var is something else</div>
</div>
```

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

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

You also can declare the same `*ng-switch-when` value for different elements. Here's an example:

code/built_in_components/ng_switch/app.ts

```
@View({
  directives: [NgSwitch, NgSwitchWhen, NgSwitchDefault],
  template: `
    <div>Current choice is: {{ choice }}</div>

    <ul [ng-switch]="choice">
      <li *ng-switch-when="1">First choice</li>
      <li *ng-switch-when="2">Second choice</li>
      <li *ng-switch-when="3">Third choice</li>
      <li *ng-switch-when="4">Fourth choice</li>
      <li *ng-switch-when="2">Second choice, again</li>
      <li *ng-switch-default>Default choice</li>
    </ul>

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

Another nice feature of `ng-switch-when` 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 `ng-style` 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

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

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

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

code/built_in_components/ng_style/app.ts

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

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

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

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

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

In our example, we are defining two input boxes:

code/built_in_components/ng_style/app.ts

```
    <div><input type="text" name="color" value="{{color}}" #colorinput></div>
    <div><input type="text" name="fontSize" value="{{fontSize}}" #fontinput></di\
v>
```

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

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

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

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

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

code/built_in_components/ng_style/app.ts

```
    <div>
      <span [ng-style]="{color: color}">{{ color }} text</span>
    </div>

    <div [style.background-color]="color" style="color: white;">{{ color }} back\
ground</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

```
  apply(color, fontSize) {
    this.color = color;
    this.fontSize = fontSize;
  }
```

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

NgClass

The `NgClass` directive, represented by a `ng-class` 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 `ng-class` 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

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

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

code/built_in_components/ng_class/app.ts

```
    <div [ng-class]="{bordered: false}">This is never bordered</div>
    <div [ng-class]="{bordered: true}">This is always bordered</div>
```

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

Simple class directive usage

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

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

code/built_in_components/ng_class/app.ts

```
    <div [ng-class]="{bordered: isBordered}">
      This is a div with object literal. Border is {{ isBordered ? "ON" : "OFF" \
}}
    </div>
```

Alternatively, we can define the object in our component:

code/built_in_components/ng_class/app.ts

```
  toggleBorder() {
    this.isBordered = !this.isBordered;
    this.classesObj = {
      bordered: this.isBordered
    };
  }
```

And use the object directly:

code/built_in_components/ng_class/app.ts

```
    <div [ng-class]="classesObj">
      This is a div with object var. Border is {{ classesObj.bordered ? "ON" : "\
OFF" }}
    </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:

```
<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

```
    <div class="base" [ng-class]="['blue', 'round']">
      This will always have a blue background and
      round corners
    </div>
```

Or declare an array variable in our component:

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

And passing it in:

code/built_in_components/ng_class/app.ts

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

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

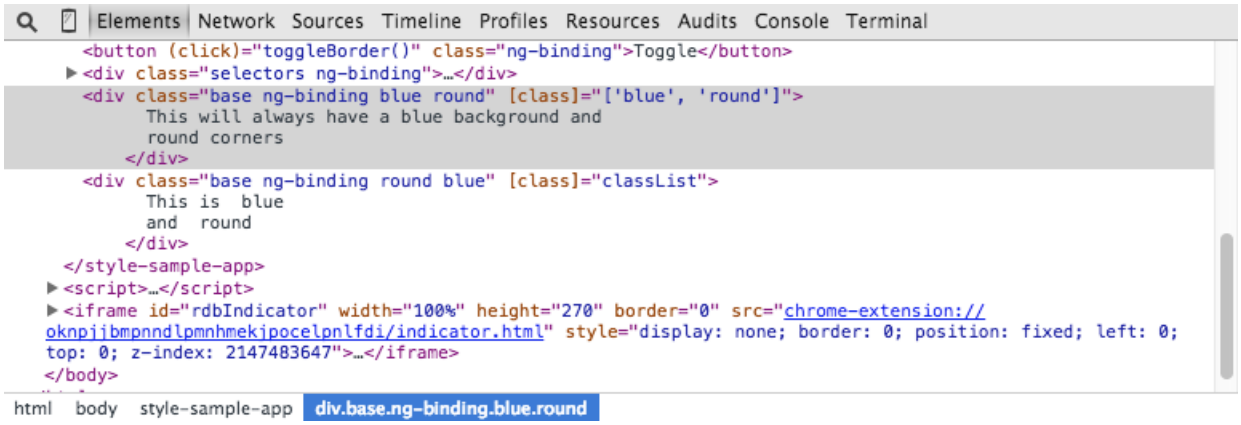
The resulting classes added to the element will always be the set of the classes provided by usual `class` HTML attribute and the result of the evaluation of the `[class]` directive.

In this example:

code/built_in_components/ng_class/app.ts

```
    <div class="base" [ng-class]="['blue', 'round']">
      This will always have a blue background and
      round corners
    </div>
```

The element will have all three classes: `base` from the `class` HTML attribute and also `blue` and `round` from the `[class]` assigment (along with the `ng-binding` class that Angular 2 uses internally):



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 `*ng-for="#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:

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

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

```
<div *ng-for="#c of cities">{{ c }}</div>
```

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

Miami
Sao Paulo
New York

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

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

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

code/built_in_components/ng_for/app.ts

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tr *ng-for="#p of people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
  </tr>
</table>
```

Getting the following result:

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

```
    this.peopleByCity = [
      {
          city: 'Miami',
          people: [
              { name: 'John', age: 12 },
              { name: 'Angel', age: 22 }
          ]
      },
      {
          city: 'Sao Paulo',
          people: [
              { name: 'Anderson', age: 35 },
```

```
            { name: 'Felipe', age: 36 }
          ]
      }
    ];
```

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

code/built_in_components/ng_for/app.ts

```
  <div *ng-for="#item of peopleByCity">
    <div>{{ item.city }}</div>
```

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

code/built_in_components/ng_for/app.ts

```
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Age</th>
        </tr>
      </thead>
      <tr *ng-for="#p of item.people">
        <td>{{ p.name }}</td>
        <td>{{ p.age }}</td>
      </tr>
    </table>
```

Resulting in the following template code:

code/built_in_components/ng_for/app.ts

```
  <div *ng-for="#item of peopleByCity">
    <div>{{ item.city }}</div>

    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Age</th>
        </tr>
```

```
10        </thead>
11        <tr *ng-for="#p of item.people">
12          <td>{{ p.name }}</td>
13          <td>{{ p.age }}</td>
14        </tr>
15      </table>
```

And it would render a table for each city:

Miami

Name	Age
John	12
Angel	22

Sao Paulo

Name	Age
Anderson	35
Felipe	36

Rendering nested arrays

Getting an index

There are times that we need the index of each item when we're iterating an array.

We can get the index by appending the syntax `#idx = index` to the value of our `ng-for` 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...

Let's change our first example to have the following markup:

code/built_in_components/ng_for/app.ts

```
1   <div *ng-for="#c of cities; #num = index">
2     {{ num+1 }} - {{ c }}
3   </div>
```

It will add the position of the city before the name, like this:

1 - Miami
2 - Sao Paulo
3 - New York

Using an index

NgNonBindable

We use `ng-non-bindable` 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 `ng-non-bindable` 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    {{ content }}
3    <span ng-non-bindable>
4      <- This is what {{ content }} rendered
5    </span>
6  </div>
```

And with that `ng-non-bindable` attribute, ng2 will not compile with that `span`'s context, leaving it intact:

Some text <- This is what {{ content }} rendered

Result of using ng-non-bindable

`ng-non-bindable` is built-in to the compiler, so you don't need to inject it as a directive.

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 our 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 is it?

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 ng2 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:

```
// create a new Control with the value "Nate"
var nameControl = new Control("Nate");

var name = nameControl.value; // -> Nate

// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
// etc.
```

To build up forms we create `Control`s (and groups of `Control`s) 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 (`ng-control`, in this case). For instance, we might have the following in our form:

```
<!-- part of some bigger form -->
<input type="text" ng-control="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 `Control`s. If we wanted to check the validity of our form, it's cumbersome to iterate over an array of `Control`s and check each `Control` for validity. `ControlGroup`s solve this issue by providing a wrapper interface around a collection of `Control`s.

Here's how you create a `ControlGroup`:

```
var personInfo = new ControlGroup({
    firstName: new Control("Nate"),
    lastName: new Control("Murray"),
    zip: new Control("90210")
})
```

`ControlGroup` and `Control` have a common ancestor (`AbstractControl`[17]). That means we can check the `status` or `value` of `personInfo` just as easily as a single `Control`:

[17] https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/model.ts

```
personInfo.value; // -> {
//   firstName: "Nate",
//   lastName: "Murray",
//   zip: "90210"
// }

// now we can query this control group for certain values, which have sensible
// values depending on the children Control's values:
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty  // -> false
personInfo.valid  // -> true
// etc.
```

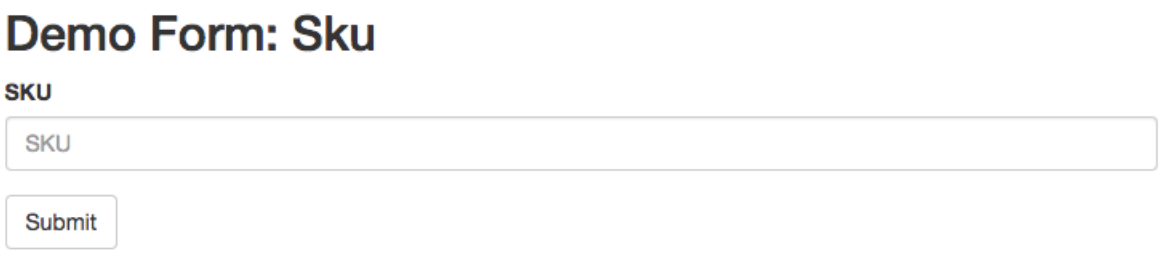
Notice that when we tried to get the `value` from the `ControlGroup` we received an **object** with key-value pairs. This is a really handy way to get the full set of values from our form without having to iterate over each `Control` individually.

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:



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 a component:

- `@Component()` annotation
- `@View()` annotation
- The Component definition class

Let's take these in turn:

Simple SKU Form: @Component Annotation

code/forms/app/forms/demo_form_sku.ts

```
1 @Component({
2   selector: 'demo-form-sku'
3 })
```

Here we just pass one option to the `@Component` annotation: `selector`. `selector` tells Angular what elements this component will bind to. In this case we can use this component by having a `demo-form-sku` tag like so:

```
1 <demo-form-sku></demo-form-sku>
```

Simple SKU Form: @View Annotation

Let's look at our `@View`:

code/forms/app/ts/forms/demo_form_sku.ts

```
@View({
  directives: [FORM_DIRECTIVES],
  template: `
  <div>
    <h2>Demo Form: Sku</h2>
    <form #f="form"
          (submit)="onSubmit(f.value)">

      <div class="form-group">
        <label for="skuInput">SKU</label>
        <input type="text"
               class="form-control"
               id="skuInput"
               placeholder="SKU"
               ng-control="sku">
      </div>

      <button type="submit" class="btn btn-default">Submit</button>
    </form>
  </div>
  `
})
```

We start by injecting `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:

- `ng-control`
- `ng-control-group`
- `ng-form`
- `ng-model`

... and several more[18]. We haven't talked about how to use these directives or what they do, but we will shortly. Just know for now that by injecting `FORM_DIRECTIVES`, that means we can use any of the directives in that list in our view.

[18] https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/directives.ts

`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 **non-obvious**: it includes the `form` tag in its selector (instead of requiring you to explicitly add `ng-form` 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 `form`
2. A **`(submit)`** action

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

```
1     <form #f="form"
2           (submit)="onSubmit(f.value)">
```

First we have `#f="form"`. 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 `*ng-for`. As in `*ng-for="#product in products"`).

Here we're creating an alias to `form`, for this view, bound to the variable `#f`. Where did `form` come from in the first place? It came from the `NgForm` directive.

And what type of object is `form`? 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 `(submit)` action.

We bind to the `submit` action of our form by using the syntax: `(submit)="onSubmit(f.value)"`.

- `(submit)` - comes from `NgForm`
- `onSubmit()` - will be implemented in our component definition class (below)
- `f.value` - `f` is the `form` `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 I want to touch on before we talk about `NgControl`:

code/forms/app/ts/forms/demo_form_sku.ts

```
      <div class="form-group">
        <label for="skuInput">SKU</label>
        <input type="text"
               class="form-control"
               id="skuInput"
               placeholder="SKU"
               ng-control="sku">
      </div>
```

- `class="form-group"` and `class="form-control"` - these two classes are totally optional. They come from the CSS framework Bootstrap[19]. 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 match.
- We set a `placeholder` of "SKU", which is just a hint for what this `input` should say when it is blank

The `NgControl` directive specifies a `selector` of `ng-control`. This means we can attach it to our `input` tag by adding this sort of attribute: `ng-control="whatever"`. In this case, we say: `ng-control="sku"`.

`NgControl` vs. `ng-control`: what's the difference? Generally, when we use CamelCase, like `NgControl`, we're specifying the class, or directive and referring to the object as it's defined in code. The lower case with dashes (a.k.a. kebab-case), like `ng-control`, 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 `ng-control` is concretely defined in the class `NgControlName` (not `NgControl`). You can find the definition here[20]

`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"`.

[19] http://getbootstrap.com/css/

[20] https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/directives/ng_control_name.ts

`ng-control` must be used as a child of `NgForm` (or `NgFormModel`, which we talk about below). Don't try to use `ng-control` 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

```
export class DemoFormSku {
  onSubmit(value) {
    console.log('you submitted value: ', value);
  }
}
```

Here our class only 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!

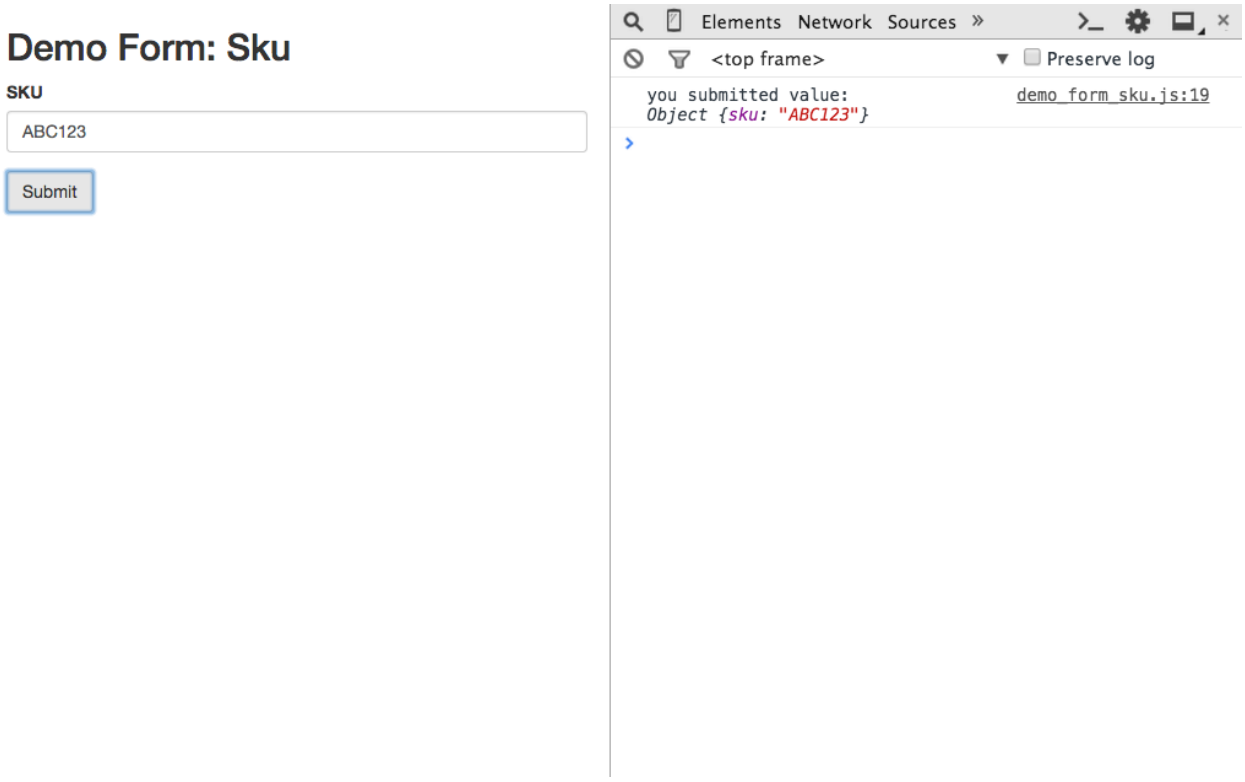
Putting it all together, here's what our code listing looks like:

code/forms/app/ts/forms/demo_form_sku.ts

```
/// <reference path="../../typings/app.d.ts" />
import {Component, bootstrap, View} from "angular2/angular2";
import {FORM_DIRECTIVES, FormBuilder, ControlGroup} from "angular2/angular2";

@Component({
  selector: 'demo-form-sku'
})
@View({
  directives: [FORM_DIRECTIVES],
  template: `
  <div>
    <h2>Demo Form: Sku</h2>
    <form #f="form"
          (submit)="onSubmit(f.value)">

      <div class="form-group">
        <label for="skuInput">SKU</label>
```

```
18          <input type="text"
19                 class="form-control"
20                 id="skuInput"
21                 placeholder="SKU"
22                 ng-control="sku">
23        </div>
24
25        <button type="submit" class="btn btn-default">Submit</button>
26      </form>
27    </div>
28    `
29  })
30  export class DemoFormSku {
31    onSubmit(value) {
32      console.log('you submitted value: ', value);
33    }
34  }
```

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 `ng-form` and `ng-control` 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:

Wait what? Okay, 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 components chapter, so go there if you'd like to learn more about it in depth.

At a high level, Dependency Injection is a way to tell Angular what dependencies this component needs to function properly.

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
1  export class DemoFormSkuBuilder {
2    myForm: ControlGroup;
3
4    constructor(fb: FormBuilder) {
5      this.myForm = fb.group({
6        "sku": ["ABC123"]
7      });
8    }
9
10   onSubmit(value) {
11     console.log('you submitted value: ', value);
12   }
13 }
```

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 `ng-form` is applied for us automatically when we used `FORM_DIRECTIVES`. We also mentioned that `ng-form` creates it's 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

```
1     <form [ng-form-model]="myForm"
2           (submit)="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 `ng-form-model`.

If you're curious, the `selector` for `NgForm` is:

```
1  form:not([ng-no-form]):not([ng-form-model]),ng-form,[ng-form]
```

This means you *could* have a form that doesn't get `NgForm` applied by using the `ng-no-form` 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 **`ng-control` 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

```
1         <input type="text"
2                class="form-control"
3                id="skuInput"
4                placeholder="SKU"
5                [ng-form-control]="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

```
/// <reference path="../../typings/app.d.ts" />
import {Component, bootstrap, View} from "angular2/angular2";
import {FORM_DIRECTIVES, FormBuilder, ControlGroup} from "angular2/angular2";

@Component({
  selector: 'demo-form-sku-builder'

})
@View({
  directives: [FORM_DIRECTIVES],
  template: `
  <div>
    <h2>Demo Form: Sku with Builder</h2>
    <form [ng-form-model]="myForm"
          (submit)="onSubmit(myForm.value)">

      <div class="form-group">
        <label for="skuInput">SKU</label>
        <input type="text"
               class="form-control"
               id="skuInput"
               placeholder="SKU"
               [ng-form-control]="myForm.controls['sku']">
      </div>

      <button type="submit" class="btn btn-default">Submit</button>
    </form>
  </div>
  `
})
export class DemoFormSkuBuilder {
  myForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      "sku": ["ABC123"]
    });
  }

  onSubmit(value) {
    console.log('you submitted value: ', value);
```

```
42    }
43  }
```

Remember:

To create a new `ControlGroup` and `Controls` implicitly use:

- `ng-form` and
- `ng-control`

But to bind to an existing `ControlGroup` and `Controls` use:

- `ng-form-model` and
- `ng-form-control`

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  var 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

```
    this.myForm = fb.group({
      "sku":  ["", Validators.required]
    });
```

Troubleshooting: If you see `error TS2339: Property 'required' does not exist on type 'typeof Validators'.` You may need to add the following to your `angular2.d.ts` file

```
 class Validators {
   static required: any;
   static compose(validators: List<Function>): Function;
 }
```

See also: this Stack Overflow[21]

Now we need to use our validation in the view. There's two ways we can access the validation value in the view:

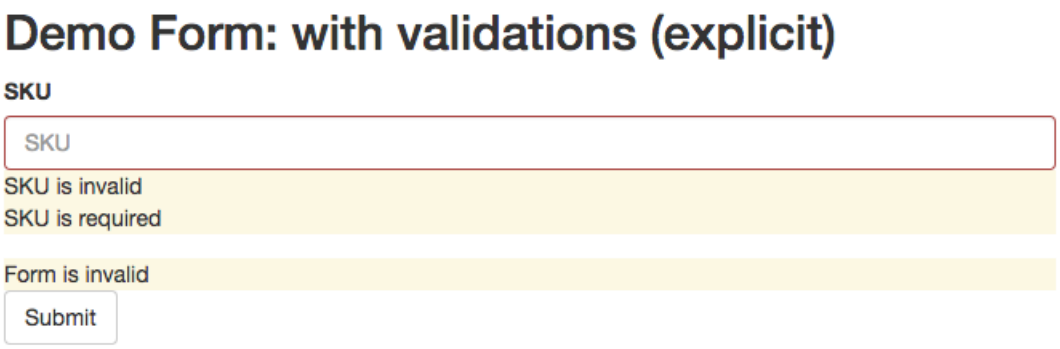
1. We can explicitly assign the `Control` `sku` to an instance variable of the class - which is more verbose, but gives us easy access to the `Control` in the view.
2. We can lookup the `Control` `sku` from `myForm` in the view. This requires less work in the component definition class, but is slightly 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:

[21] http://stackoverflow.com/a/31132953/110644



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

```
1  export class DemoFormWithValidationsExplicit {
2    myForm: ControlGroup;
3    sku: AbstractControl;
4
5    constructor(fb: FormBuilder) {
6      this.myForm = fb.group({
7        "sku":  ["", Validators.required]
8      });
9
10     this.sku = this.myForm.controls['sku'];
11   }
```

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

```
1       <div *ng-if="!myForm.valid"
2         class="bg-warning">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

```
1         <div *ng-if="!sku.valid"
2           class="bg-warning">SKU is invalid</div>
```

Field coloring

I'm using the Bootstrap CSS Framework's `.form-group` class[22] so that means if I add the class `.has-error` to the `.form-group` 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

```
1       <div class="form-group"
2           [class.has-error]="!sku.valid && sku.touched">
```

Notice here that we have two conditions for setting the `.has-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.

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:

[22] http://getbootstrap.com/css/#forms-control-validation

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
        <div *ng-if="sku.hasError('required')"
          class="bg-warning">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:

```
    <div *ng-if="myForm.hasError('required', 'sku')"
      class="bg-warning">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

```
/// <reference path="../../typings/app.d.ts" />
import {Component, bootstrap, View} from "angular2/angular2";
import {FORM_DIRECTIVES, FormBuilder, ControlGroup, Control, NgIf} from "angular\
2/angular2";
import {Validators} from 'angular2/angular2';

@Component({
  selector: 'demo-form-with-validations-explicit'

})
@View({
  directives: [FORM_DIRECTIVES, NgIf],
  template: `
  <div>
    <h2>Demo Form: with validations (explicit)</h2>
    <form [ng-form-model]="myForm"
          (submit)="onSubmit(myForm.value)">

      <div class="form-group"
          [class.has-error]="!sku.valid && sku.touched">
        <label for="skuInput">SKU</label>
        <input type="text"
              class="form-control"
              id="skuInput"
```

```
25                placeholder="SKU"
26                [ng-form-control]="sku">
27           <div *ng-if="!sku.valid"
28             class="bg-warning">SKU is invalid</div>
29           <div *ng-if="sku.hasError('required')"
30             class="bg-warning">SKU is required</div>
31         </div>
32
33         <div *ng-if="!myForm.valid"
34           class="bg-warning">Form is invalid</div>
35         <button type="submit" class="btn btn-default">Submit</button>
36       </form>
37     </div>
38     `
39 })
40 export class DemoFormWithValidationsExplicit {
41   myForm: ControlGroup;
42   sku: AbstractControl;
43
44   constructor(fb: FormBuilder) {
45     this.myForm = fb.group({
46       "sku":  ["", Validators.required]
47     });
48
49     this.sku = this.myForm.controls['sku'];
50   }
51
52   onSubmit(value) {
53     console.log('you submitted value: ', value);
54   }
55 }
```

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

```
export class DemoFormWithValidationsShorthand {
  myForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      "sku":  ["", Validators.required]
    });
  }

  onSubmit(value) {
    console.log('you submitted value: ', value);
  }
}
```

Notice that we have removed the `sku: AbstractControl`. Moving on.

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 `ng-form-control` 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

```
      <div class="form-group"
          [class.has-error]="!myForm.find('sku').valid && myForm.find('sku').tou\
ched">
```

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 `form` 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 `form`. 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

```
1         <input type="text"
2                class="form-control"
3                id="skuInput"
4                placeholder="SKU"
5                #sku="form"
6                [ng-form-control]="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="form"`, `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

```
1          <div *ng-if="!sku.control.valid"
2            class="bg-warning">SKU is invalid</div>
3          <div *ng-if="sku.control.hasError('required')"
4            class="bg-warning">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

```
    // this won't work
    <div class="form-group"
       [class.has-error]="!sku.control.valid && sku.control.touched">
```

Why not? Because this `.form-group` is a parent of the `input` element that declares the reference.

Putting it togther

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

```
/// <reference path="../../typings/app.d.ts" />
import {Component, bootstrap, View} from "angular2/angular2";
import {FORM_DIRECTIVES, FormBuilder, ControlGroup, Control, NgIf} from "angular\
2/angular2";
import {Validators} from 'angular2/angular2';

@Component({
  selector: 'demo-form-with-validations-shorthand'

})
@View({
  directives: [FORM_DIRECTIVES, NgIf],
  template: `
  <div>
    <h2>Demo Form: with validations (shorthand)</h2>
    <form [ng-form-model]="myForm"
          (submit)="onSubmit(myForm.value)">

      <div class="form-group"
          [class.has-error]="!myForm.find('sku').valid && myForm.find('sku').tou\
ched">
        <label for="skuInput">SKU</label>
        <input type="text"
               class="form-control"
               id="skuInput"
               placeholder="SKU"
               #sku="form"
               [ng-form-control]="myForm.controls['sku']">
         <div *ng-if="!sku.control.valid"
           class="bg-warning">SKU is invalid</div>
```

```
31            <div *ng-if="sku.control.hasError('required')"
32              class="bg-warning">SKU is required</div>
33          </div>
34
35          <div *ng-if="!myForm.valid"
36            class="bg-warning">Form is invalid</div>
37          <button type="submit" class="btn btn-default">Submit</button>
38        </form>
39      </div>
40      `
41  })
42  export class DemoFormWithValidationsShorthand {
43    myForm: ControlGroup;
44
45    constructor(fb: FormBuilder) {
46      this.myForm = fb.group({
47        "sku":  ["", Validators.required]
48      });
49    }
50
51    onSubmit(value) {
52      console.log('you submitted value: ', value);
53    }
54  }
```

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

```
function skuValidator(control) {
  if (!control.value.match(/^123/)){
    return {invalidSku: true};
  }
}
```

This validator will return an error code `invalidSku` if the input (the `control.value`) does not begin with `123`.

Assigning the Validator to the `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

```
    this.myForm = fb.group({
      "sku":  ["", Validators.compose([
        Validators.required, skuValidator])]
    });
```

`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

```
        <div *ng-if="sku.hasError('invalidSku')"
          class="bg-warning">SKU must begin with <tt>123</tt></div>
```

Note that for 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 try out 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 here[23]

To watch for changes on a control we:

1. get access to the `EventEmitter` by calling `control.valueChanges`. Then we
2. add an *observer* using the `.observer` method

Here's an example:

code/forms/app/ts/forms/demo_form_with_events.ts

```
1     this.sku.valueChanges.subscribe(
2       (value) => {
3         console.log("sku changed to: ", value);
4       }
5     );
6
7     this.myForm.valueChanges.subscribe(
8       (value) => {
9         console.log("form changed to: ", value);
10      }
11    );
```

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:

[23] https://github.com/jhusain/observable-spec

```
sku changed to:  k
form changed to:  Object {sku: "k"}
sku changed to:  kj
form changed to:  Object {sku: "kj"}
```

As you can see each keystroke causes the control to change, so our observable is triggered. When we observe the individual `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"}`).

ng-model

`NgModel` is a special directive: it binds a model to a form. `ng-model` is special in that it does implement it's own 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 `ng-model` right away. There are good reasons to avoid two-way data binding. Of course, `ng-model` can be really handy, but know that we don't necessarily rely on 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 `ng-model` 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

```
export class DemoFormNgModel {
  myForm: ControlGroup;
  productName: string;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      "productName":  ["", Validators.required]
    });
  }
```

Notice that we're simply storing `productName: string` as an instance variable.

Next, let's use `ng-model` on our `input` tag:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
                class="form-control"
                id="productNameInput"
                placeholder="Product Name"
                [ng-form-control]="myForm.find('productName')"
                [(ng-model)]="productName">
```

Now notice something - the syntax for `ng-model` is funny: we are using both brackets and parenthesis around the `ng-model` attribute! The idea this is intended to invoke is that we're using both the *property* `[]` brackets and the *action* `()` parenthesis. It's an indication of the two-way bind.

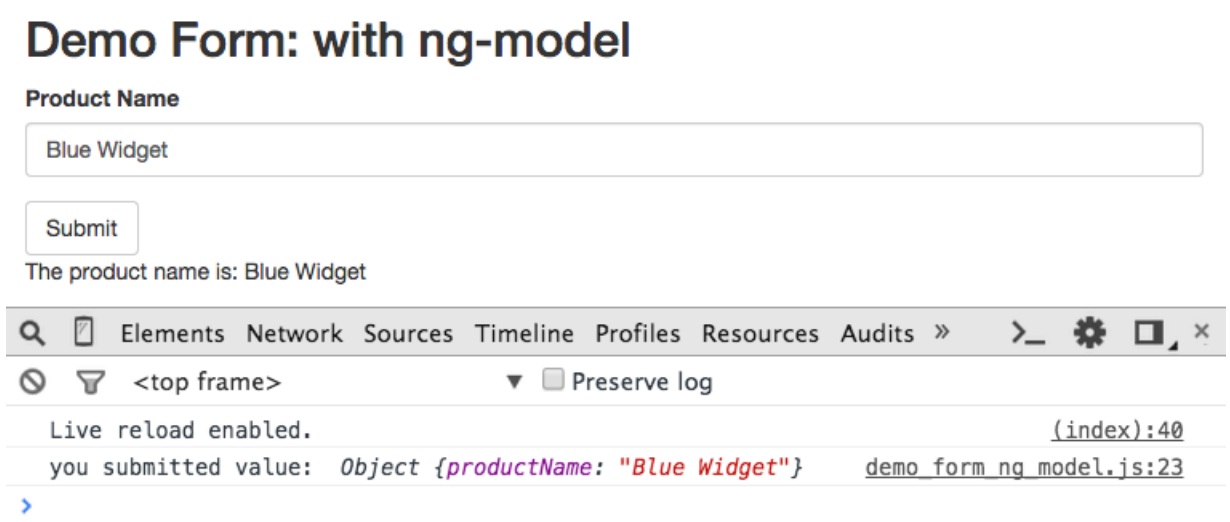
Notice something else here: we're still using `ng-form-control` to specify that this `input` should be bound to the `Control` on our form. We do this because `ng-model` is only binding the `input` to the instance variable - the `Control` is completely separate. But because we still want to validate this value and submit it as part of the form, we keep the `ng-form-control` directive.

Last, let's display our `productName` value in the view:

code/forms/app/ts/forms/demo_form_ng_model.ts

```
    <div>
      The product name is: {{productName}}
    </div>
```

Here's what it looks like:



Demo Form with ng-model

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:

- **MVW / Two-way data binding**: *Model-View-Whatever* is a term used[24] 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**[25]: uses a unidirectional data flow. In Flux, Stores hold data, Views render what's in the Store, and Actions change the data in the Store. There is a bit more ceremony to setup Flux, but the idea is that because data only flows in one direction, it's easier to reason about.
- **Observables**: Observables give us streams of data. We subscribe to the streams and then perform operations to react to changes. RxJs[26] is the most popular reactive streams library for

[24] See: Model View Whatever

[25] https://facebook.github.io/flux/

[26] 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.
- Falcor[27] 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.

[27] http://netflix.github.io/falcor/

Data Architecture with Observables - Part 1: Services

Observables and RxJS

In Angular, we can structure our application to use Observables as the backbone of our data architecture. Using Observables to structure our data is called *Reactive Programming*.

But what are Observables, and Reactive Programming anyway? Reactive Programming is a way to work with asynchronous streams of data. Observables are the main data structure we use to implement Reactive Programming. But I'll admit, those terms may not be that clarifying. So we'll look at concrete examples through the rest of this chapter that should be more enlightening.

Note: Some RxJS Knowledge Required

I want to point out **this book is not primarily about Reactive Programming**. There are several other good resources that can teach you the basics of Reactive Programming and you should read them. We've listed a few below.

Consider this chapter a tutorial on how to work with RxJS and Angular rather than an exhaustive introduction to RxJS and Reactive Programming.

In this chapter, I'll **explain in detail the RxJS concepts and APIs that we encounter**. But know that you may need to supplement the content here with other resources if RxJS is still new to you.

Use of Underscore.js in this chapter

Underscore.js[28] is a popular library that provides functional operators on Javascript data structures such as `Array` and `Object`. We use it a bunch in this chapter alongside RxJS. If you see the _ in code, such as `_.map` or `_.sortBy` know that we're using the Underscore.js library. You can find the docs for Underscore.js here[29].

Learning Reactive Programming and RxJS

If you're just learning RxJS I recommend that you read this article first:

[28] http://underscorejs.org/

[29] http://underscorejs.org/

- The introduction to Reactive Programming you've been missing[30] 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?[31]
- Which instance operators to use on streams?[32]
- RxMarbles[33] - 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.

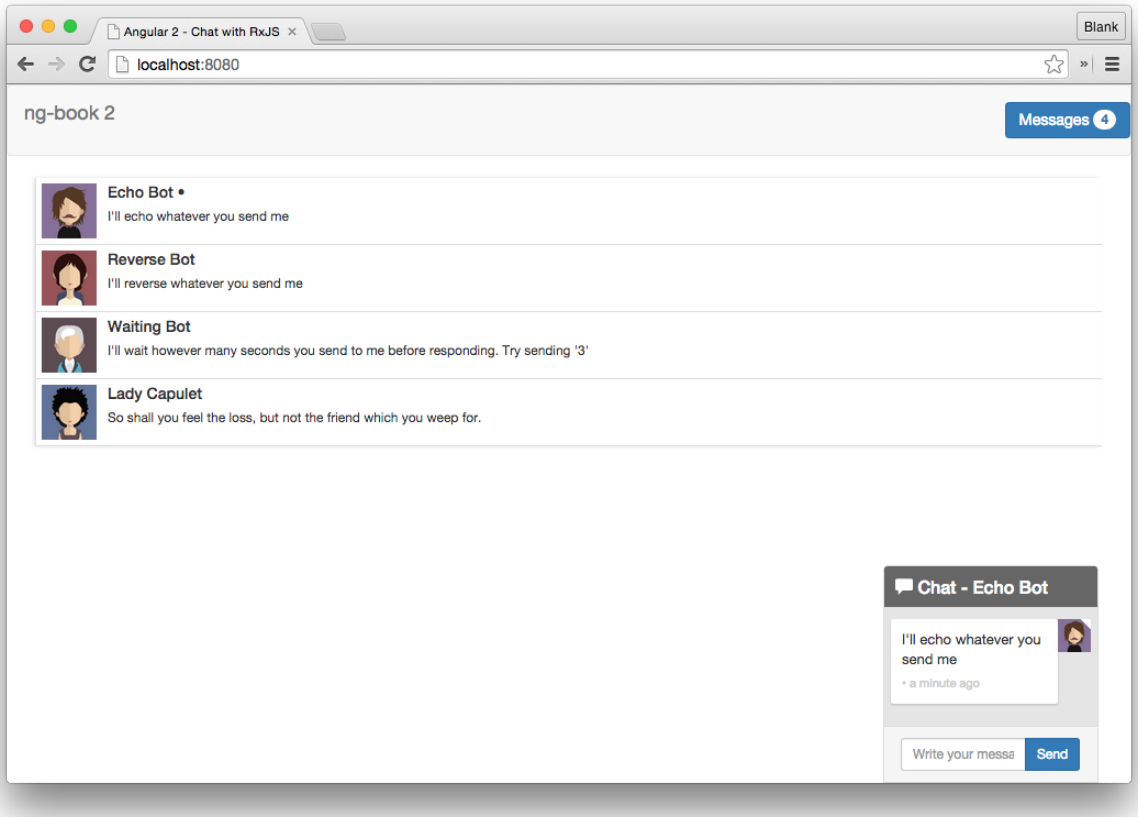
I want to give you fair warning: learning RxJS can be a bit mind-bending at first. But trust me, you'll get the hang of it and it's worth it. Here's a few big ideas about streams that you might find helpful:

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

[30] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
[31] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md
[32] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md
[33] 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

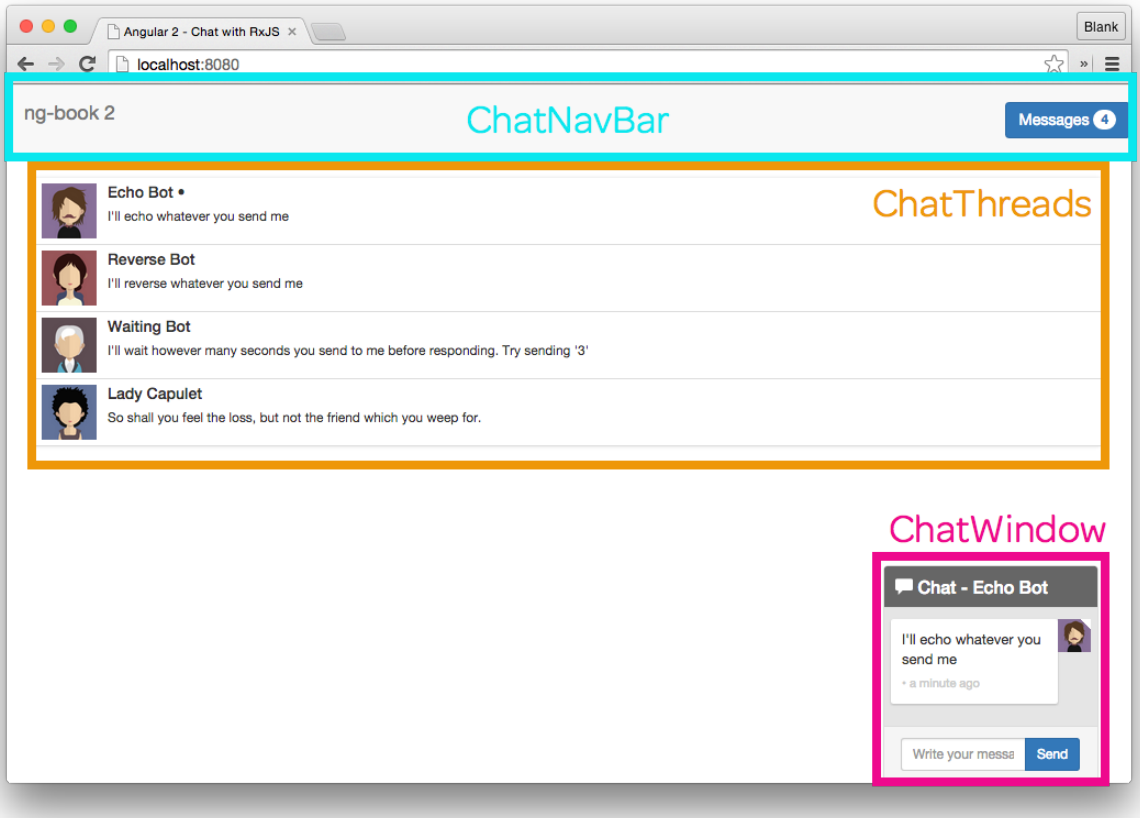
Let's look at an overview of how this app is constructed. We have

- 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:

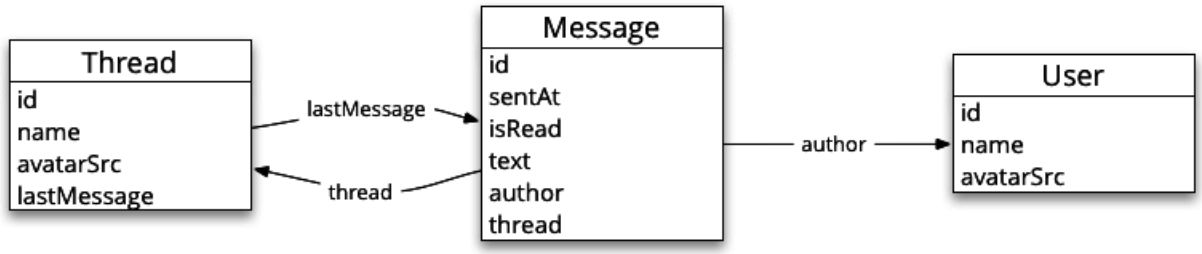


Chat Top-Level Components

- `ChatNavBar` - contains the unread messages count
- `ChatThreads` - shows a clickable list of threads, along with the most recent message and the conversation avatar
- `ChatWindow` - shows the messages in the current thread with an input box to send new messages

Models

This application also has three models:



Chat Models

- `User` - stores information about a chat participant
- `Message` - stores an individual message
- `Thread` - stores a collection of `Messages` as well as some data about the conversation

Services

In this app, each of our models has a corresponding *service*. The services are singleton objects that play two roles:

1. **Provide streams** of data that our application can subscribe to
2. **Provide operations** to add or modify data

For instance, the `UserService`:

- publishes a stream that emits the current user and
- offers a `setCurrentUser` function which will set the current user (that is, emit the current user from the `currentUser` stream)

Summary

At a high level, the application data architecture is straightforward:

- The **services** maintain streams which emit models (e.g. `Messages`)
- The **components** subscribe to those streams and render according to the most recent values

For instance, the `ChatThreads` component listens for the most recent list of threads from the `ThreadService` and the `ChatWindow` subscribes for the most recent list of messages.

In the rest of this chapter, we're going to go in-depth on how we implement this using Angular 2 and RxJS. We'll start by implementing our models, then look at how we create Services to manage our streams, and then finally implement the Components.

Implementing the Models

Let's start with the easy stuff and take a look at the models.

User

Our `User` class is straightforward. We have an `id`, `name`, and `avatarSrc`.

code/rxjs/chat/app/ts/models.ts

```
export class User {
  id: string;

  constructor(public name: string,
              public avatarSrc: string) {
    this.id = uuid();
  }
}
```

Notice above that we're using a TypeScript shorthand in the constructor. When we say `public name: string` we're telling TypeScript that 1. we want `name` to be a public property on this class and 2. assign the argument value to that property when a new instance is created.

Thread

Similarly, `Thread` is also a straightforward TypeScript class:

code/rxjs/chat/app/ts/models.ts

```
export class Thread {
  id: string;
  lastMessage: Message;
  name: string;
  avatarSrc: string;

  constructor(id?: string,
              name?: string,
              avatarSrc?: string) {
    this.id = id || uuid();
    this.name = name;
    this.avatarSrc = avatarSrc;
  }
}
```

Note that we store a reference to the `lastMessage` in our `Thread`. This 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

```
export class Message {
  id: string;
  sentAt: Date;
  isRead: boolean;
  author: User;
  text: string;
  thread: Thread;

  constructor(obj?: any) {
    this.id             = obj && obj.id             || uuid();
    this.isRead         = obj && obj.isRead         || false;
    this.sentAt         = obj && obj.sentAt         || new Date();
    this.author         = obj && obj.author         || null;
    this.text           = obj && obj.text           || null;
    this.thread         = obj && obj.thread         || null;
```

```
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`[34] inherits from both `Observable`[35] and `Observer`[36]

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.

[34] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/subject.md
[35] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md
[36] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md

`BehaviourSubject`[37] 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:

```
userService.subscribe((newUser) => {
  console.log('New User is: ', newUser.name);
})

// => New User is: originalUserName

let u = new User('Nate', 'anImgSrc');
userService.currentUser.onNext(u);

// => 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:

[37] 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

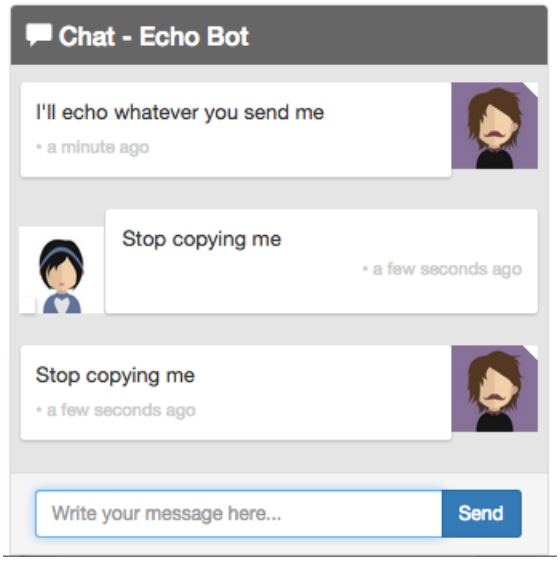
```
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Rx.Subject<Message> = new Rx.Subject<Message>();
```

If we want, we can define a helper method to add `Messages` to this stream:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  addMessage(message: Message): void {
    this.newMessages.onNext(message);
  }
```

It would also be helpful to have a stream that will get all of the messages from a thread that are not from a particular user. For instance, consider the Echo Bot:



Real mature, Echo Bot

When we are implementing the Echo Bot, we don't want to enter an infinite loop and repeat back the bot's messages to itself.

To implement this we can subscribe to the `newMessages` stream and filter out all messages that are 1. part of this thread and 2. not written by the bot. You can think of this as saying, for a given `Thread` I want a stream of the messages that are "for" this `User`.

code/rxjs/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

```
interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}
```

Let's define our `updates` stream:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  updates: Rx.Subject<any> =
    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

```
  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan(initialMessages, (messages: Message[],
              operation: IMessagesOperation) => {
        return operation(messages);
      })
```

This code introduces a new stream function: `scan`[38]. 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:

[38] 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`[39] method like so:

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan(initialMessages, (messages: Message[],
              operation: IMessagesOperation) => {
        return operation(messages);
      })
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .shareReplay(1);
```

Adding `Messages` to the `messages` Stream

Now we could add a `Message` to the `messages` stream like so:

```
var myMessage = new Message(/* params here... */);

updates.onNext( (messages: Message[]): Message[] => {
  return messages.concat(myMessage);
})
```

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`.

[39] 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:

```
addMessage(newMessage: Message) {
  updates.onNext( (messages: Message[]): Message[] => {
    return messages.concat(newMessage);
  })
}

// somewhere else

var myMessage = new Message(/* params here... */);
MessagesService.addMessage(myMessage);
```

This is a little bit better, but it's not "the reactive way". In part, because this action of creating a message isn't composable with other streams. (Also this method is circumventing our `newMessages` stream. More on that later.)

A reactive way of creating a new message would be **to have a stream that accepts `Messages` to add to the list**. Again, this can be a bit new if you're not used to thinking this way. Here's how you'd implement it:

First we make an "action stream" called `create`. (The term "action stream" is only meant to describe its role in our service. The stream itself is still a regular `Subject`):

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  // action streams
  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

```
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
```

The `map`[40] 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`[41].

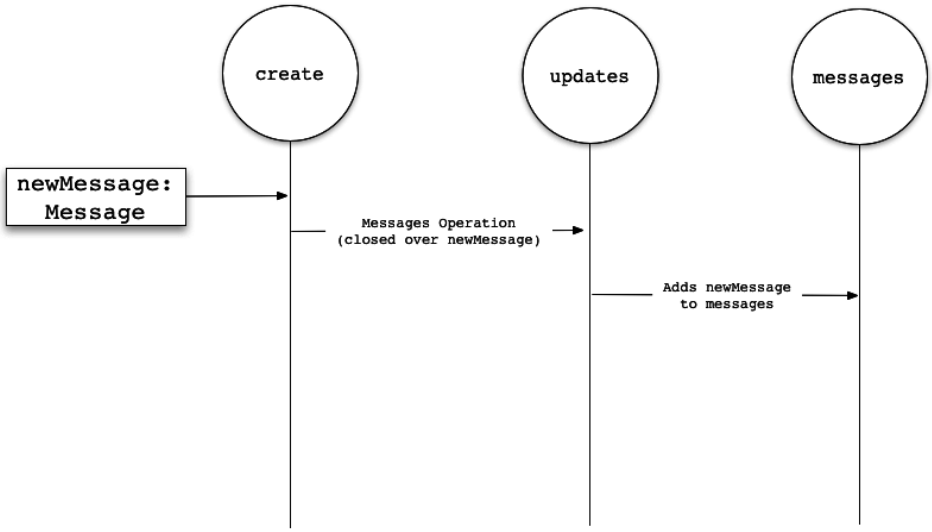
code/rxjs/chat/app/ts/services/MessagesService.ts

```
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
      .subscribe(this.updates);
```

What we're doing here is *subscribing* the `updates` stream to listen to the `create` stream. This means that if `create` receives a `Message` it will emit an `IMessagesOperation` that will be received by `updates` and then the `Message` will be added to `messages`.

Here's a diagram that shows our current situation:

[40] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/select.md
[41] 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`)

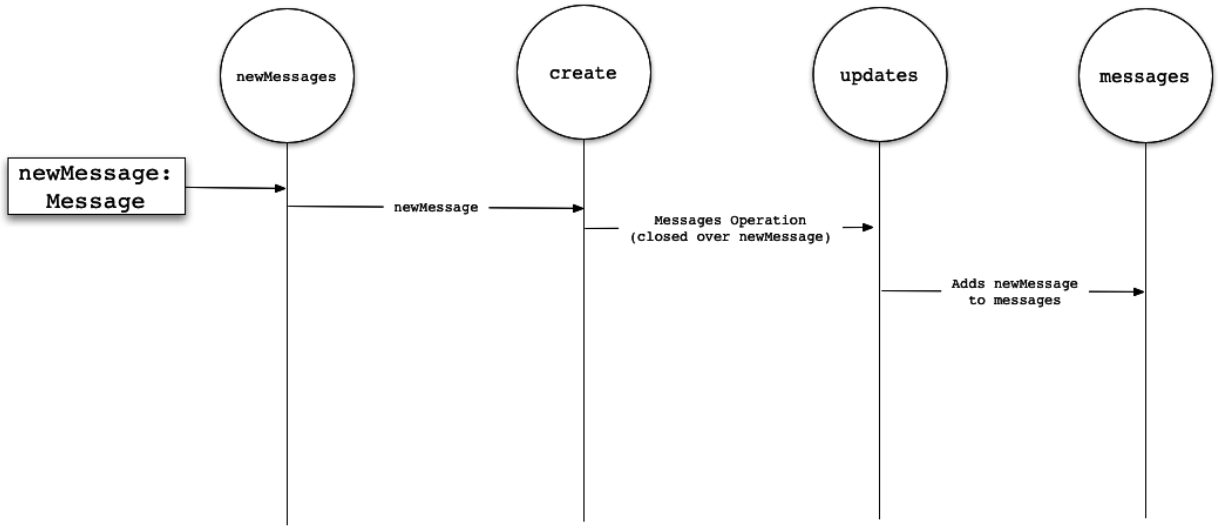
Anywhere in our code, if we want to get the most current list of messages, we just have to go to the `messages` stream. But we have a problem, **we still haven't connected this flow to the `newMessages` stream.

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

```
let initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Rx.Subject<Message> = new Rx.Subject<Message>();

  // `messages` is a stream that emits an array of the most up to date messages
  messages: Rx.Observable<Message[]>;

  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  updates: Rx.Subject<any> =
    new Rx.Subject<any>();

  // action streams
  create: Rx.Subject<Message> = new Rx.Subject<Message>();
  markThreadAsRead: Rx.Subject<any> = new Rx.Subject<any>();

  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan(initialMessages, (messages: Message[],
              operation: IMessagesOperation) => {
        return operation(messages);
      })
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .shareReplay(1);

    // `create` takes a Message and then puts an operation (the inner function)
    // on the `updates` stream to add the Message to the list of messages.
    //
    // That is, for each item that gets added to `create` (by using `onNext`)
    // 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    }
81
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 indivdually 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` were going to define four streams that emit respectively:

1. A map of the current set of `Threads` (in `threads`)
2. A chronological list of `Threads`, newest-first (in `orderedthreads`)
3. The currently selected `Thread` (in `currentThread`)
4. The list of `Messages` for the currently selected `Thread` (in `currentThreadMessages`)

Let's walk through how to build each of these streams, and we'll learn a little more about RxJS along the way.

A map of the current set of `Threads` (in `threads`)

Let's start by defining our `ThreadsService` class and the instance variable that will emit the `Threads`:

code/rxjs/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 }>;
```

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
```

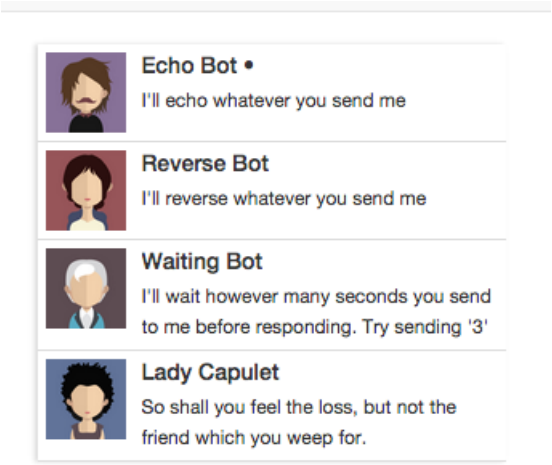
Recall that each time a new `Message` is added to the steam, `messages` will emit an array of the current `Messages`. We're going to look at each `Message` and we want to return a unique list of the `Threads`.

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread it in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;
```

Notice above that each time we will create a new list of `threads`. The reason for this is because we might delete some messages down the line (e.g. leave the conversation). Because we're recalculating the list of threads each time, we naturally will "delete" a thread if it has no messages.

In the threads list, we want to show a preview of the chat by using the text of the most recent `Message` in that `Thread`.



List of Threads with Chat Preview

In order to do that, we'll store the most recent `Message` for each `Thread`. We know which `Message` is newest by comparing the `sentAt` times:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
          // Cache the most recent message for each thread
          let messagesThread: Thread = threads[message.thread.id];
          if (!messagesThread.lastMessage ||
              messagesThread.lastMessage.sentAt < message.sentAt) {
            messagesThread.lastMessage = message;
          }
        });
        return threads;
      })
```

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

```
    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread it in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;

          // Cache the most recent message for each thread
          let messagesThread: Thread = threads[message.thread.id];
          if (!messagesThread.lastMessage ||
              messagesThread.lastMessage.sentAt < message.sentAt) {
            messagesThread.lastMessage = message;
          }
        });
        return threads;
      })
      // share this stream across multiple subscribers and makes sure everyone
      // receives the current list of threads when they first subscribe
      .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

```
    let nate: User = new User("Nate Murray", "");
    let felipe: User = new User("Felipe Coury", "");

    let t1: Thread = new Thread("t1", "Thread 1", "");
    let t2: Thread = new Thread("t2", "Thread 2", "");

    let m1: Message = new Message({
      author: nate,
      text: "Hi!",
      thread: t1
    });

    let m2: Message = new Message({
      author: felipe,
      text: "Where did you get that hat?",
      thread: t1
    });

    let m3: Message = new Message({
      author: nate,
      text: "Did you bring the briefcase?",
      thread: t2
    });
```

Now let's create an instance of our services:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
    let messagesService = new MessagesService();
    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.

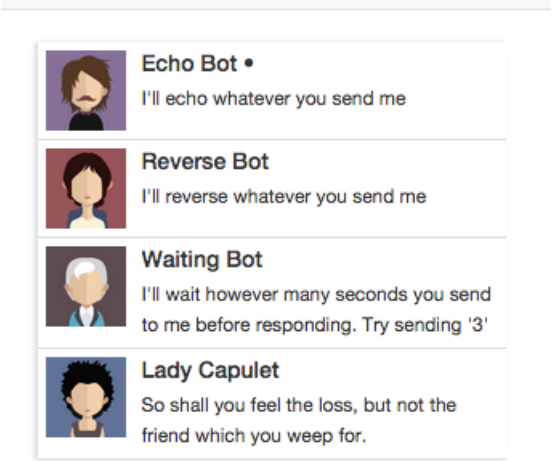
Let's subscribe to `threads` and log out what comes through:

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
    threadsService.threads
      .subscribe( (threadIdx) => {
        var threads = _.values(threadIdx);
        var threadNames = _.map(threads, 'name').join(', ')
        console.log(`=> threads (${threads.length}): ${threadNames} `);
      })

    messagesService.addMessage(m1);
    messagesService.addMessage(m2);
    messagesService.addMessage(m3);

    // => threads (1): Thread 1
    // => threads (1): Thread 1
    // => threads (2): Thread 1, Thread 2

  });
});
```

A chronological list of `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:

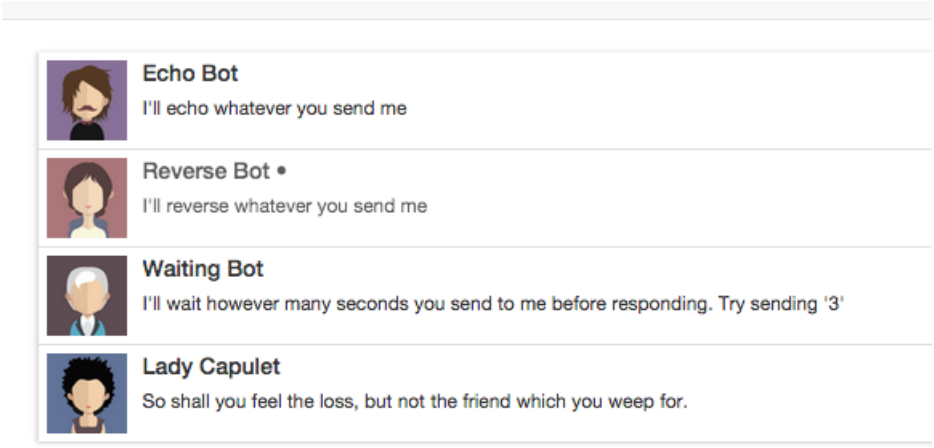
code/rxjs/chat/app/ts/services/ThreadsService.ts

```
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 'â€¢' 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`

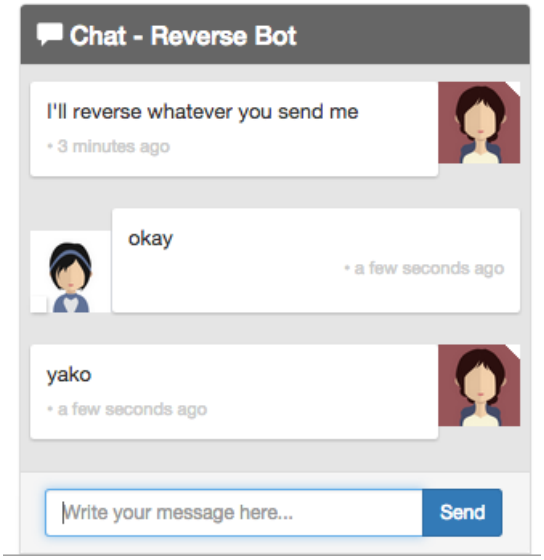
So all we need to do is hook them together:

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:

```
var theCurrentThread: Thread;

this.currentThread.subscribe((thread: Thread) => {
  theCurrentThread = thread;
})

this.currentThreadMessages.map(
  (mesages: Message[]) => {
    return _.filter(messages,
      (message: Message) => {
        return message.thread.id == theCurrentThread.id;
      })
  })
```

What's wrong with this approach? Well, if the `currentThread` changes, `currentThreadMessages` won't know about it and so we'll have an outdated list of `currentThreadMessages`!

What if we reversed it, and stored the current list of messages in a variable and subscribed to the changing of `currentThread`? We'd have the same problem only this time we would know when the thread changes but not when a new message came in.

How can we solve this problem?

It turns out, RxJS has a set of operators that we can use to **combine multiple streams**. In this case we want to say "if *either* `currentThread` **or** `messagesService.messages` changes, then we want to emit something." For this we use the `combineLatest`[42] 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 that the messages for the current thread, this is a convenient area to mark these messages as read.

[42] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
            .value();
```

Whether or not we should be marking messages as read here is debatable. The biggest drawback is that we're mutating objects in what is, essentially, a "read" thread. i.e. this is a read operation with a side effect, which is generally a Bad Idea. That said, in this application the `currentThreadMessages` only applies to the `currentThread` and the `currentThread` should always have 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

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
        if (currentThread && messages.length > 0) {
          return _.chain(messages)
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
            .value();
        } else {
          return [];
        }
      })
      .shareReplay(1);
```

Our Completed `ThreadsService`

Here's what our `ThreadService` looks like:

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list of threads
  threads: Rx.Observable<{ [key: string]: Thread }>;

  // `orderedThreads` contains a newest-first chronological list of threads
  orderedThreads: Rx.Observable<Thread[]>;

  // `currentThread` contains the currently selected thread
  currentThread: Rx.Subject<Thread> =
    new Rx.BehaviorSubject<Thread>(new Thread());

  // `currentThreadMessages` contains the set of messages for the currently
  // selected thread
  currentThreadMessages: Rx.Observable<Message[]>;

  constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread it in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;

          // Cache the most recent message for each thread
          let messagesThread: Thread = threads[message.thread.id];
          if (!messagesThread.lastMessage ||
              messagesThread.lastMessage.sentAt < message.sentAt) {
            messagesThread.lastMessage = message;
          }
        });
        return threads;
      })
      // share this stream across multiple subscribers and makes sure everyone
      // receives the current list of threads when they first subscribe
      .shareReplay(1);

    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

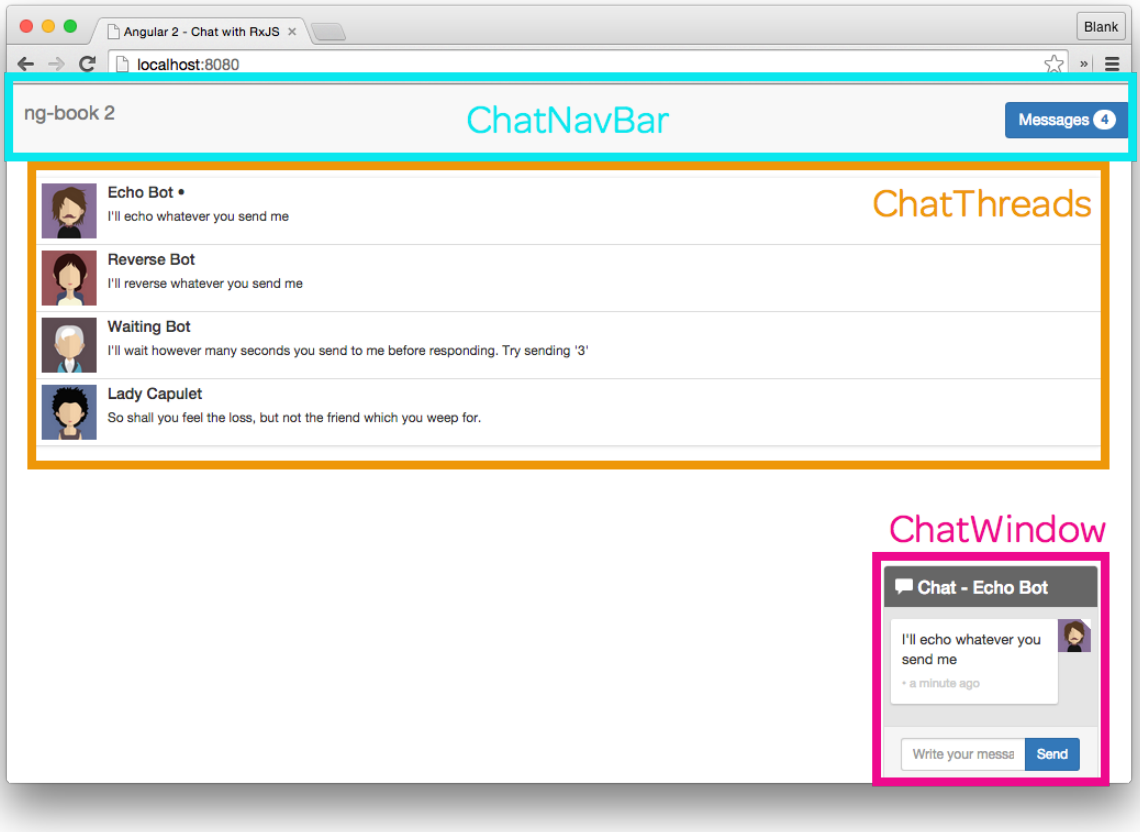
Building Our Views: The `ChatApp` Top-Level Component

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 your 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

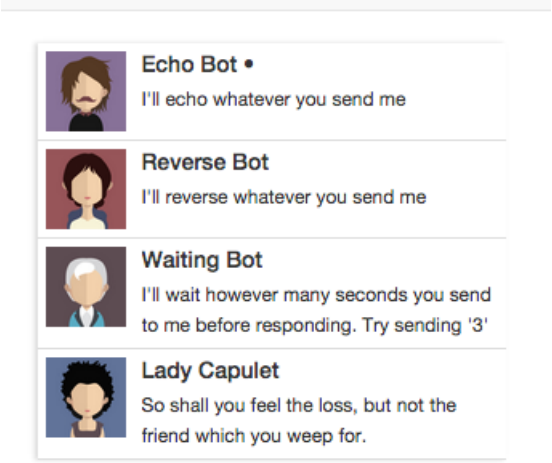
```
@Component({
  selector: "chat-app"
})
@View({
  directives: [ChatNavBar,
               ChatThreads,
               ChatWindow],
  template: `
  <div>
    <nav-bar></nav-bar>
    <div class="container">
      <chat-threads></chat-threads>
      <chat-window></chat-window>
    </div>
  </div>
  `
})
class ChatApp {
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public userService: UserService) {
    ChatExampleData.init(messagesService, threadsService, userService);

  }
}
```

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

```
@Component({
  selector: "chat-threads"
})
```

`ChatThreads` Controller

Take a look at our component controller `ChatThreads`:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
export class ChatThreads {
  threads: Rx.Observable<any>;

  constructor(public threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

Here we're injecting `ThreadsService` and then we're keeping a reference to the `orderedThreads`.

`ChatThreads` `@View`

Lastly, let's look at the `@View`:

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@View({
  directives: [NgFor, ChatThread],
  pipes: [RxPipe],
  template: `
    <!-- conversations -->
    <div class="row">
      <div class="conversation-wrap">

        <chat-thread
             *ng-for="#thread of threads | rx"
             [thread]="thread">
        </chat-thread>

      </div>
    </div>
  `
})
```

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

```
@Component({
  selector: "chat-threads"
})
@View({
  directives: [NgFor, ChatThread],
  pipes: [RxPipe],
  template: `
    <!-- conversations -->
    <div class="row">
      <div class="conversation-wrap">

        <chat-thread
             *ng-for="#thread of threads | rx"
             [thread]="thread">
        </chat-thread>

      </div>
    </div>
  `
})
export class ChatThreads {
  threads: Rx.Observable<any>;

  constructor(public threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

`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 efficently.

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

```
class ChatThread {
  thread: Thread;
  selected: boolean = false;

  constructor(public threadsService: ThreadsService) {
  }

  onInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```

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

```
@View({
  directives: [NgIf],
  template: `
  <div class="media conversation">
    <div class="pull-left">
      <img class="media-object avatar"
           src="{{thread.avatarSrc}}">
    </div>
    <div class="media-body">
      <h5 class="media-heading contact-name">{{thread.name}}
        <span *ng-if="selected">&bull;</span>
      </h5>
      <small class="message-preview">{{thread.lastMessage.text}}</small>
    </div>
    <a (click)="clicked($event)" class="div-link">Select</a>
  </div>
  `
})
```

Notice we've got some straight-forward bindings like `{{thread.avatarSrc}}`, `{{thread.name}}`, and `{{thread.lastMessage.text}}`.

We've got an `*ng-if` which will show the `&bull;` 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:

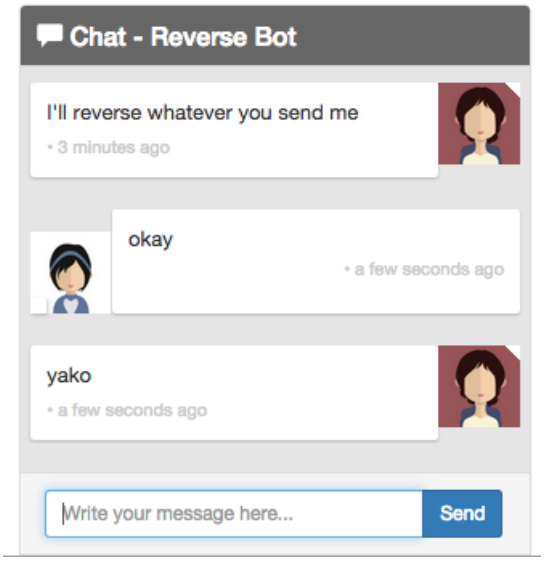
code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  lifecycle: [ LifecycleEvent.onInit ],
  properties: ["thread"],
  selector: "chat-thread"
})
@View({
  directives: [NgIf],
  template: `
  <div class="media conversation">
    <div class="pull-left">
      <img class="media-object avatar"
           src="{{thread.avatarSrc}}">
    </div>
    <div class="media-body">
      <h5 class="media-heading contact-name">{{thread.name}}
        <span *ng-if="selected">&bull;</span>
      </h5>
      <small class="message-preview">{{thread.lastMessage.text}}</small>
    </div>
    <a (click)="clicked($event)" class="div-link">Select</a>
  </div>
  `
})
class ChatThread {
  thread: Thread;
  selected: boolean = false;

  constructor(public threadsService: ThreadsService) {
  }

  onInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
```

```
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  })
```

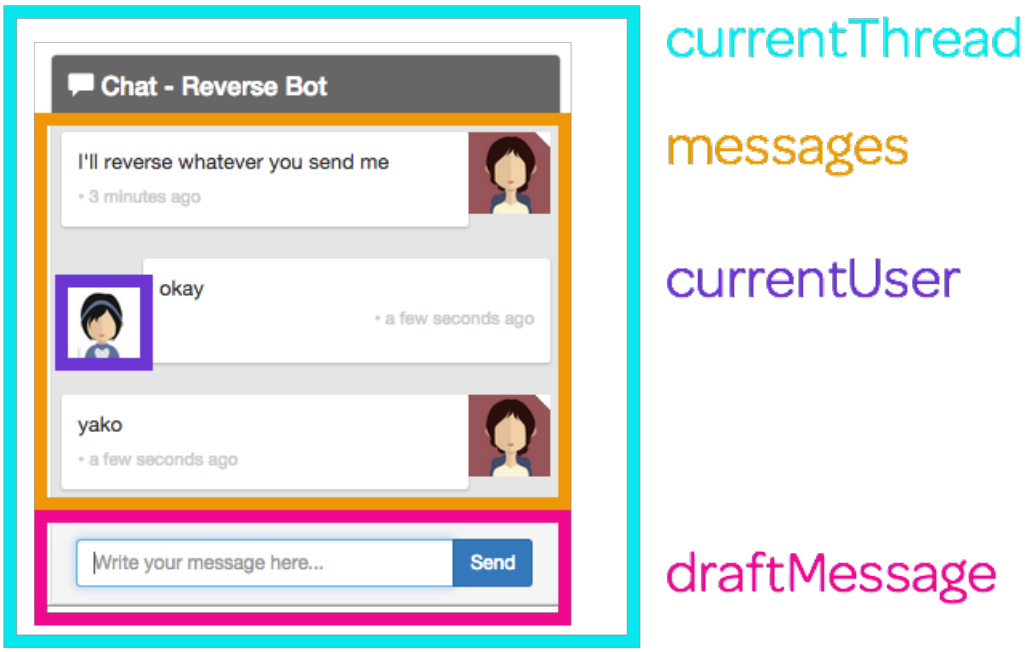
ChatWindow Properties

Our Chat window has four properties:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
export class ChatWindow {
  messages: Rx.Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;
```

Here's a diagram of where each one is used:



Chat Window Properties

In our `constructor` we're going to inject four things:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public userService: UserService,
              public el: ElementRef) {
  }
```

The first three are our services. The fourth, `el` is an `ElementRef` which we can use to get access to the host DOM element. We'll use that when we scroll to the bottom of the chat window when we create and receive new messages.

Remember: by using `public messagesService: MessagesService` in the constructor, we are not only injecting the `MessagesService` but setting up an instance variable that we can use later in our class via `this.messagesService`

`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

```
  sendMessage(): void {
    let m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
  }
```

The `sendMessage` function above takes the `draftMessage`, sets the `author` and `thread` using our component properties. Every message we send has "been read" already (we wrote it) so we mark it as read.

Notice here that we're not updating the `draftMessage` text. That's because we're going to bind the value of the messages text in the view in a few minutes.

After we've updated the `draftMessage` properties we send it off to the `messagesService` and then **create a new `Message`** and set that new `Message` to `this.draftMessage`. We do this to make sure we don't mutate an already sent message.

`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

```
  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }
```

`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

```
  scrollToBottom(): void {
    let scrollPane: any = this.el
      .nativeElement.querySelector(".msg-container-base");
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }
```

Now that we have a function that will scroll to the bottom, we have to make sure that we call it at the right time. Back in `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

```
    this.messages
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
```

Why do we have the `setTimeout`?

If we call `scrollToBottom` immediately when we get a new message then what happens is we scroll to the bottom before the new message is rendered. By using a `setTimeout` we're telling Javascript that we want to run this function when it is finished with the current execution queue. This happens **after** the component is rendered, so it does what we want.

`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

```
@View({
  directives: [NgFor,
               ChatMessage,
               FORM_DIRECTIVES],
  pipes: [RxPipe],
```

We start by defining some markup and the panel header:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>
```

Next we show the list of messages. Here we use `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

```
            <div class="panel-body msg-container-base">
              <chat-message
                   *ng-for="#message of messages | rx"
                   [message]="message">
              </chat-message>
            </div>
```

Lastly we have the message input box and closing tags:

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
            <div class="panel-footer">
              <div class="input-group">
                <input type="text"
                       class="chat-input"
                       placeholder="Write your message here..."
                       (keydown.enter)="onEnter($event)"
                       [(ng-model)]="draftMessage.text" />
                <span class="input-group-btn">
                  <button class="btn-chat"
                     (click)="onEnter($event)"
                     >Send</button>
                </span>
              </div>
            </div>

          </div>
        </div>
      </div>
    </div>
  `
```

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

```
<input type="text"
       class="chat-input"
       placeholder="Write your message here..."
       (keydown.enter)="onEnter($event)"
       [(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 it's 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

```
<input type="text"
       class="chat-input"
       placeholder="Write your message here..."
       (keydown.enter)="onEnter($event)"
       [(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

```
<span class="input-group-btn">
  <button class="btn-chat"
     (click)="onEnter($event)"
     >Send</button>
</span>
```

The Entire `ChatWindow` Component

Here's the code listing for the entire `ChatWindow` Component:

code/rxjs/chat/app/ts/components/ChatWindow.ts

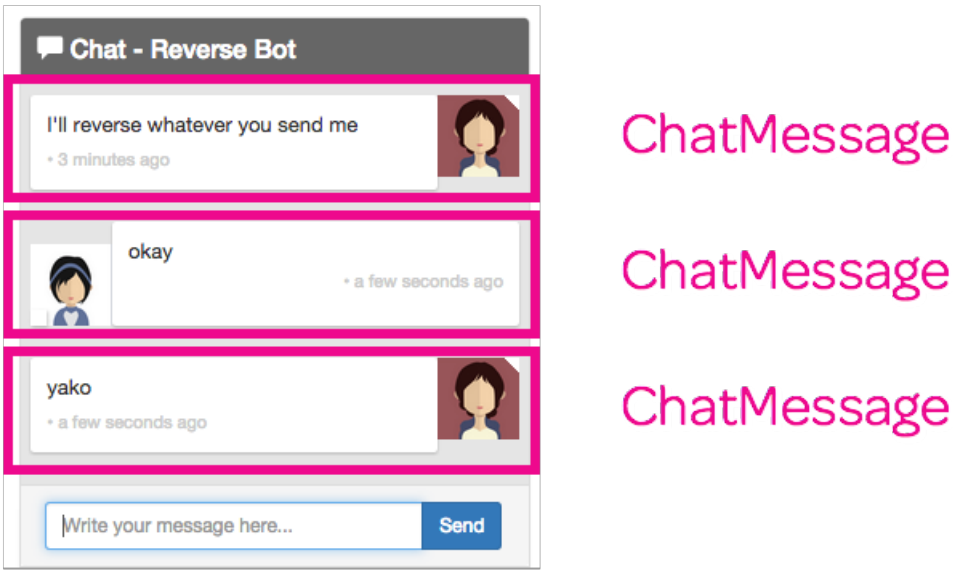
```
@Component({
  lifecycle: [ LifecycleEvent.onInit ],
  selector: "chat-window"
})
@View({
  directives: [NgFor,
               ChatMessage,
               FORM_DIRECTIVES],
  pipes: [RxPipe],
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>

            <div class="panel-body msg-container-base">
              <chat-message
                   *ng-for="#message of messages | rx"
                   [message]="message">
              </chat-message>
            </div>

            <div class="panel-footer">
              <div class="input-group">
                <input type="text"
                       class="chat-input"
                       placeholder="Write your message here..."
                       (keydown.enter)="onEnter($event)"
                       [(ng-model)]="draftMessage.text" />
```

```
                <span class="input-group-btn">
                  <button class="btn-chat"
                     (click)="onEnter($event)"
                     >Send</button>
                </span>
              </div>
            </div>

          </div>
        </div>
      </div>
    </div>
  `
})
export class ChatWindow {
  messages: Rx.Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;

  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public userService: UserService,
              public el: ElementRef) {
  }

  onInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    this.draftMessage = new Message();

    this.threadsService.currentThread.subscribe(
      (thread: Thread) => {
        this.currentThread = thread;
      });

    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
        });

```

```
    this.messages
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
  }

  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }

  sendMessage(): void {
    let m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
  }

  scrollToBottom(): void {
    let scrollPane: any = this.el
      .nativeElement.querySelector(".msg-container-base");
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }

}
```

The `ChatMessage` Component

Each `Message` is rendered by the `ChatMessage` component.



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

```
@Component({
  lifecycle: [ LifecycleEvent.onInit ],
  properties: ["message"],
  selector: "chat-message"
})
```

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

```
export class ChatMessage {
  message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(public userService: UserService) {
  }

  onInit(): void {
    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
        });
  }

}
```

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

```
@View({
  directives: [NgIf,
               NgClass],
  pipes: [FromNowPipe],
  template: `
  <div class="msg-container"
       [ng-class]="{'base-sent': !incoming, 'base-receive': incoming}">

```

```
    <div class="avatar"
         *ng-if="!incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>

    <div class="messages"
      [ng-class]="{'msg-sent': !incoming, 'msg-receive': incoming}">
      <p>{{message.text}}</p>
      <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
    </div>

    <div class="avatar"
         *ng-if="incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>
  </div>
  `
})
```

The `FromNowPipe` is a pipe that casts our `Messages` sent-at time to a human-readable "x seconds ago" message. You can see that we use it by: `{{message.sentAt | fromNow}}`

`FromNowPipe` uses the excellent `moment.js`[43] 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:

```
    [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:

[43] http://momentjs.com/

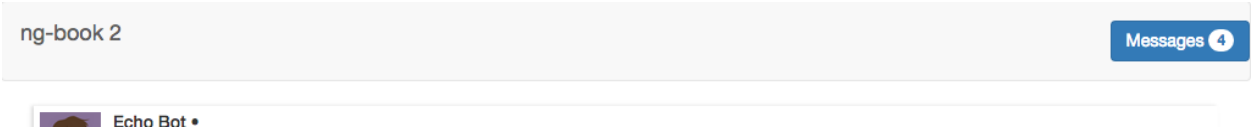
code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  lifecycle: [ LifecycleEvent.onInit ],
  properties: ["message"],
  selector: "chat-message"
})
@View({
  directives: [NgIf,
               NgClass],
  pipes: [FromNowPipe],
  template: `
  <div class="msg-container"
       [ng-class]="{'base-sent': !incoming, 'base-receive': incoming}">

    <div class="avatar"
         *ng-if="!incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>

    <div class="messages"
      [ng-class]="{'msg-sent': !incoming, 'msg-receive': incoming}">
      <p>{{message.text}}</p>
      <time>{{message.sender}} • {{message.sentAt | fromNow}}</time>
    </div>

    <div class="avatar"
         *ng-if="incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>
  </div>
  `
})
export class ChatMessage {
  message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(public userService: UserService) {
  }

  onInit(): void {
    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 Unread Count in the `ChatNavBar` Component

The best way to try out the unread messages count is to use the "Waiting Bot". If you haven't already, try sending the message '3' to the Waiting Bot and then switch to another window. The Waiting Bot will then wait 3 seconds before sending you a message and you will see the unread messages counter increment.

The `ChatNavBar` `@Component`

First we define a pretty plain `@Component`:

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

```
export class ChatNavBar {
  unreadMessagesCount: number;

  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService) {
  }

  onInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _.reduce(
            messages,
            (sum: number, m: Message) => {
              let messageIsInCurrentThread: boolean = m.thread &&
                currentThread &&
                (currentThread.id === m.thread.id);
              if (m && !m.isRead && !messageIsInCurrentThread) {
                sum = sum + 1;
              }
              return sum;
            },
            0);
      });
  }
}
```

If you're not an expert in TypeScript you might find the above syntax a little bit hard to parse. In the `combineLatest` callback function we're returning an array with `currentThread` and `messages` as 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

```
@View({
  directives: [],
  template: `
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="https://ng-book.com/2">
          <img src="${require("images/logos/ng-book-2-minibook.png")}"></img> ng\
-book 2
        </a>
      </div>
      <p class="navbar-text navbar-right">
        <button class="btn btn-primary" type="button">
          Messages <span class="badge">{{unreadMessagesCount}}</span>
        </button>
      </p>
    </div>
  </nav>


  `
})
```

The Completed `ChatNavBar`

Here's the full code listing for `ChatNavBar`:

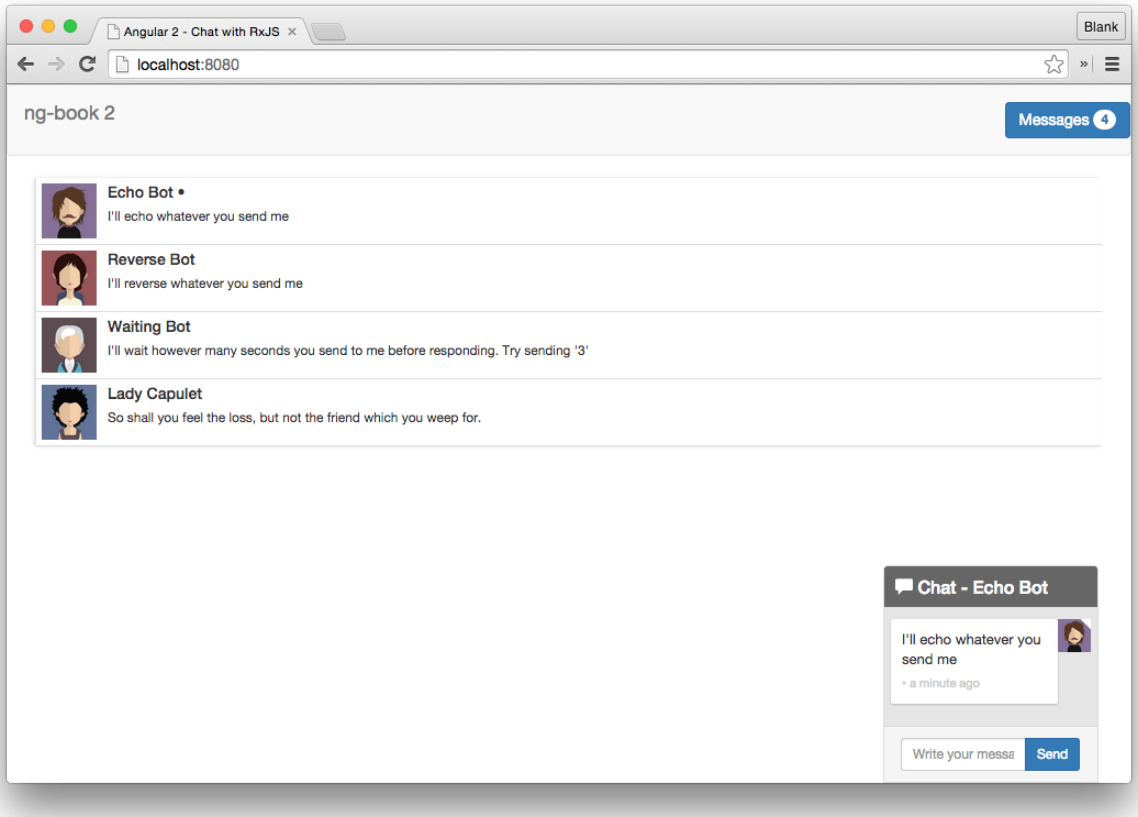
code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
@Component({
  lifecycle: [ LifecycleEvent.onInit ],
  selector: "nav-bar"
})
@View({
  directives: [],
  template: `
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="https://ng-book.com/2">
          <img src="${require("images/logos/ng-book-2-minibook.png")}"></img> ng\
-book 2
        </a>
      </div>
      <p class="navbar-text navbar-right">
        <button class="btn btn-primary" type="button">
          Messages <span class="badge">{{unreadMessagesCount}}</span>
        </button>
      </p>
    </div>
  </nav>

  `
})
export class ChatNavBar {
  unreadMessagesCount: number;

  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService) {
  }

  onInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        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:

```
let rev: User     = new User("Reverse Bot", require("images/avatars/female-avata\
r-4.png"));
let tRev: Thread     = new Thread("tRev", rev.name, rev.avatarSrc);
```

code/rxjs/chat/app/ts/ChatExampleData.ts

```
    // reverse bot
    messagesService.messagesForThreadUser(tRev, rev)
      .forEach( (message: Message) => {
        messagesService.addMessage(
          new Message({
            author: rev,
            text: message.text.split("").reverse().join(""),
            thread: tRev
```

```
9           })
10        );
11      });
```

Above you can see that we've subscribed to the messages for the "Reverse Bot" by using `messagesForThreadUser`. Try writing a few bots of your own.

Next Steps

Some ways to improve this chat app would be to become stronger at RxJS and then hook it up to an actual API. We'll talk about how to make API requests in the HTTP Chapter. For now, enjoy your fancy chat application!

HTTP

Introduction

Angular comes with it's 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 our `index.html` we simply put it in a `script` tag like this:

code/http/app/index.html

```
    <script src="/_/vendor/angular2.dev.2.0.0-alpha.37.js"></script>
    <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

```
/*
 * Include types from npm
 */

/// <reference path="../../node_modules/angular2/bundles/typings/angular2/angula\
r2.d.ts" />
/// <reference path="../../node_modules/angular2/bundles/typings/angular2/http.d\
.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

```
import {Component, bootstrap, View} from "angular2/angular2";
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.

```
bootstrap(HttpApp, [HTTP_BINDINGS]);
```

Now we can inject the `Http` service into our components (or anywhere we use DI, actually).

```
class MyFooComponent {
  constructor(public http: Http) {
  }

  makeRequest(): void {
    // do something with this.http ...
  }
}
```

A Basic Request

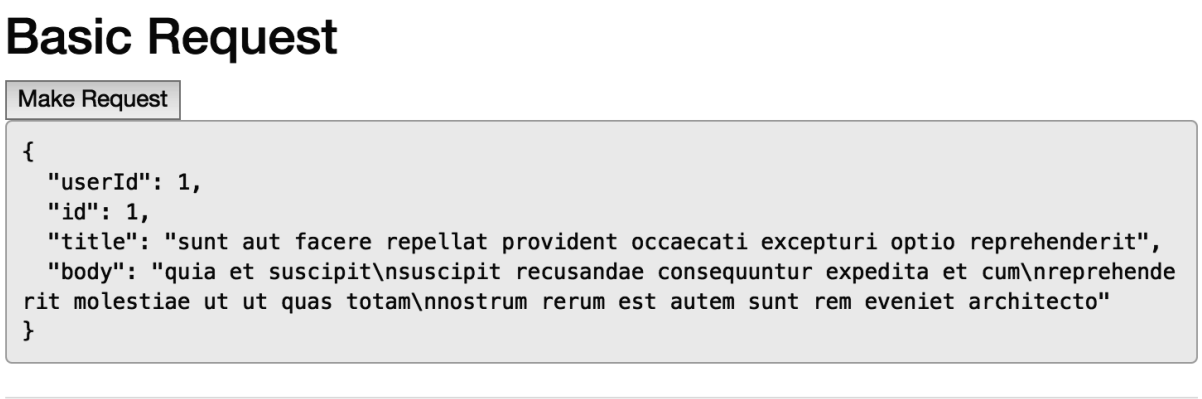
The first thing we're going to do is make a simple `GET` request to the `jsonplaceholder` API[44].

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:

[44] http://jsonplaceholder.typicode.com



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 #2794[45]

When we call `toRx()` we get an `Rx.Observable`. We can subscribe to changes (akin to using `then` from a Promise) using `subscribe`.

[45] https://github.com/angular/angular/issues/2794

code/http/app/ts/components/SimpleHTTPComponent.ts

```
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
```

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

```
import {Component, View, NgIf} from "angular2/angular2";
import {Http, Response} from "angular2/http";

@Component({
  selector: "simple-http"
})
@View({
  directives: [NgIf],
  template: `
  <h2>Basic Request</h2>
  <button type="button" (click)="makeRequest()">Make Request</button>
  <div *ng-if="loading">loading...</div>
  <pre>{{data | json}}</pre>
`
})
export class SimpleHTTPComponent {
  data: Object;
  loading: boolean;

```

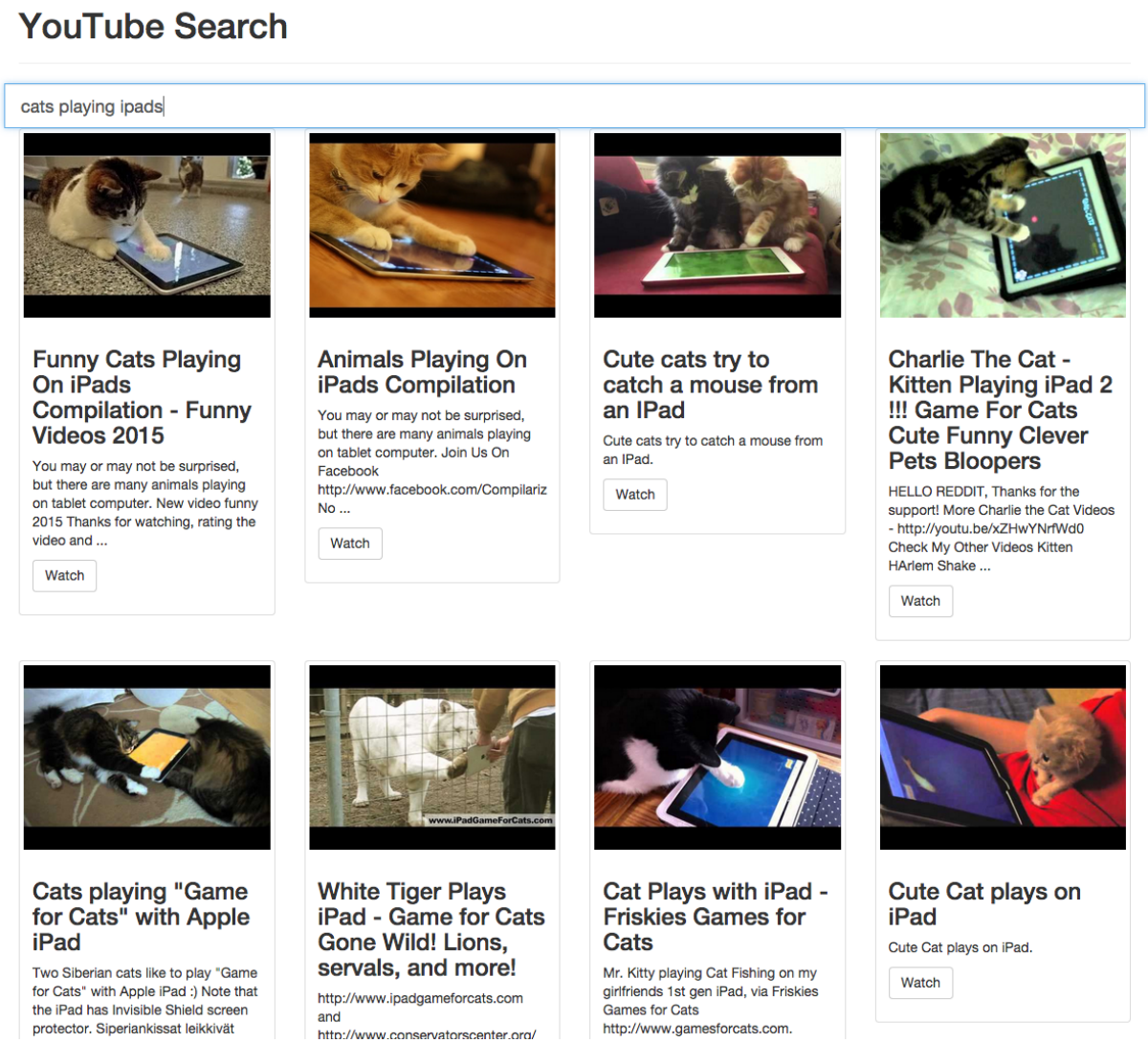
```
20    constructor(public http: Http) {
21    }
22
23    makeRequest(): void {
24      this.loading = true;
25      this.http.request("http://jsonplaceholder.typicode.com/posts/1")
26        .toRx()
27        .subscribe((res: Response) => {
28          this.data = res.json();
29          this.loading = 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":



Can I get my cat to write Angular 2?

For this example we're going to write several things:

1. A `SearchResult` object that will hold the data we want from each result
2. A `YouTubeService` which will manage the API request to YouTube and convert the results to a stream of `SearchResult[]`
3. A `SearchBox` component which will call out to the `YouTube` service as the user types
4. A `SearchResultComponent` which will render a specific `SearchResult`
5. A `YouTubeSearchComponent` which will encapsulate our whole YouTube searching app and render the list of results

Let's handle each part one at a time.

Patrick Stapleton has an excellent repository named angular2-webpack-starter[46]. 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

```
class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id           = obj && obj.id           || null;
    this.title        = obj && obj.title        || null;
    this.description  = obj && obj.description  || null;
    this.thumbnailUrl = obj && obj.thumbnailUrl || null;
    this.videoUrl     = obj && obj.videoUrl     ||
                        `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

This pattern of taking an `obj?: any` lets us simulate keyword arguments. The idea is that we can create a new `SearchResult` and just pass in an object containing the keys we want to specify.

The only thing to point out here is that we're constructing the `videoUrl` using a hard-coded URL format. You're welcome to change this to a function which takes more arguments, or use the video `id` directly in your view to build this URL if you need to.

Writing the `YouTubeService`

The API

For this example we're going to be using the YouTube v3 search API[47].

[46] https://github.com/angular-class/angular2-webpack-starter

[47] 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[48]. 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:

```
let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

Eventually we're going to want to test our app. One of the things we find when testing is that we don't always want to test against production - we often want to test against staging or a development API.

To help with this environment configuration, one of the things we can do is **make these constants injectable**.

Why should we inject these constants instead of just using them in the normal way? Because if we make them injectable we can

1. have code that injects the right constants for a given environment at deploy time and
2. replace the injected value easily at test-time

By injecting these values, we have a lot more flexibility about their values down the line.

In order to make these values injectable, we use the `bind(...).toValue(...)` syntax like this:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
export var youTubeServiceInjectables: Array<any> = [
  bind(YouTubeService).toClass(YouTubeService),
  bind(YOUTUBE_API_KEY).toValue(YOUTUBE_API_KEY),
  bind(YOUTUBE_API_URL).toValue(YOUTUBE_API_URL)
];
```

Here we're specifying that we want to bind `YOUTUBE_API_KEY` "injectably" to the value of `YOUTUBE_API_KEY`. (Same for `YOUTUBE_API_URL`, and we'll define `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`

[48] https://developers.google.com/youtube/registering_an_application#Create_API_Keys

```
// http/app.ts
import {youTubeServiceInjectables} from "components/YouTubeSearchComponent";

// ....
// further down

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

```
@Injectable()
export class YouTubeService {
  constructor(public http: Http,
              @Inject(YOUTUBE_API_KEY) private apiKey: string,
              @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }
```

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

```
  search(query: string): Rx.Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join("&");
    let queryUrl: string = `${this.apiUrl}?${params}`;
```

We're building the `queryUrl` in a manual way here. We start by simply putting the query params in the `params` variable. (You can find the meaning of each of those values by reading the search API docs[49].)

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

```
    return this.http.get(queryUrl)
      .toRx() // convert to Rx stream
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
```

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`.

[49] 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

```
@Injectable()
export class YouTubeService {
  constructor(public http: Http,
              @Inject(YOUTUBE_API_KEY) private apiKey: string,
              @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }

  search(query: string): Rx.Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join("&");
    let queryUrl: string = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
      .toRx() // convert to Rx stream
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
```

```
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
  }
}
```

Writing the 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

```
@Component({
  events: ["loading", "results"],
  selector: "search-box"
})
```

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:

```
<search-box
 (results)="updateResults($event)"
 (loading)="loading = $event"
  ></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

```
@View({
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
```

`SearchBox` Controller Definition

Our `SearchBox` controller is a new class:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
class SearchBox implements OnInit {
  loading: EventEmitter = new EventEmitter();
  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

```
  constructor(public youtube: YouTubeService,
              private el: ElementRef) {
  }
```

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

```
  onInit(): void {
    // convert the `keyup` event into an observable stream
    (<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" 1[50]. 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:

[50] Reactiveexpertswillnotethatl'mhandwavinghere.'flatMapLatest'hasamorespecifictechnicaldefinitionwhichyoucan[readaboutintheRxJSdocshere](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/flatmaplatest.md).

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  onInit(): void {
    // convert the `keyup` event into an observable stream
    (<any>Rx).Observable.fromEvent(this.el.nativeElement, "keyup")
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounce(250)                             // only once every 250ms
      .do(() => this.loading.next(true))         // enable loading
      // search, discarding old events if new input comes in
      .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

```
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.next(false);
          this.results.next(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.next(false);
        },
        () => { // on completion
          this.loading.next(false);
        }
      );
```

The first argument specifies what we want to do when the stream emits a regular event. Here we emit an event on both of our `EventEmitters`:

1. We call `this.loading.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

```
@Component({
  events: ["loading", "results"],
  selector: "search-box"
})
@View({
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
class SearchBox implements OnInit {
  loading: EventEmitter = new EventEmitter();
  results: EventEmitter = new EventEmitter();

  constructor(public youtube: YouTubeService,
              private el: ElementRef) {
  }

  //
  // Note:
  // The (<any>Rx) syntax below is a temporary hack to disable type checking
  // because the Typescript definition files bundled with angular for rx don't
  // include `fromEvent`. See:
  // http://stackoverflow.com/questions/23217334/how-do-i-extend-a-typescript-cl\
ass-definition-in-a-separate-definition-file
  //
  onInit(): void {
    // convert the `keyup` event into an observable stream
    (<any>Rx).Observable.fromEvent(this.el.nativeElement, "keyup")
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounce(250)                             // only once every 250ms
      .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 sucesss
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          <img src="{{result.thumbnailUrl}}">
10         <div class="caption">
11           <h3>{{result.title}}</h3>
12           <p>{{result.description}}</p>
13           <p><a href="{{result.videoUrl}}"
14                 class="btn btn-default" role="button">Watch</a></p>
```

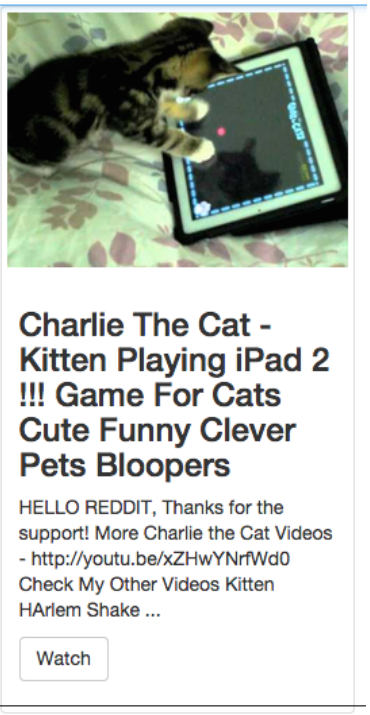
```
15          </div>
16        </div>
17      </div>
18    `
19  })
20  export class SearchResultComponent {
21    result: SearchResult;
22  }
```

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`.



Single Search Result Component

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  })
```

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

```
export class YouTubeSearchComponent {
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

This component holds one instance variable: `results` which is an array of `SearchResults`.

We also define one function: `updateResults`. `updateResults` simply takes whatever new `SearchResult[]` it's given and sets `this.results` to the new value.

We'll use both `results` and `updateResults` in our `@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

```
@View({
  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

```
template: `
<div class='container'>
    <div class="page-header">
      <h1>YouTube Search
        <img
          style="float: right;"
          *ng-if="loading"
          src=${loadingGif} />
      </h1>
    </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`[51].

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

```
    <div class="row">
      <div class="input-group input-group-lg col-md-12">
        <search-box
          (loading)="loading = $event"
          (results)="updateResults($event)"
           ></search-box>
      </div>
    </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`.

[51] 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

```
      <div class="row">
        <search-result
          *ng-for="#result of results"
          [result]="result">
        </search-result>
      </div>
  </div>
  `
})
```

`YouTubeSearchComponent` Full Listing

Here's the full listing for the `YouTubeSearchComponent`:

code/http/app/ts/components/YouTubeSearchComponent.ts

```
@Component({
  selector: "youtube-search"
})
@View({
  directives: [CORE_DIRECTIVES, SearchBox, SearchResultComponent],
  template: `
  <div class='container'>
      <div class="page-header">
        <h1>YouTube Search
          <img
            style="float: right;"
            *ng-if="loading"
            src=${loadingGif} />
        </h1>
      </div>

      <div class="row">
        <div class="input-group input-group-lg col-md-12">
          <search-box
             (loading)="loading = $event"
```

```
            (results)="updateResults($event)"
             ></search-box>
        </div>
      </div>

      <div class="row">
        <search-result
          *ng-for="#result of results"
          [result]="result">
        </search-result>
      </div>
  </div>
  `
})
export class YouTubeSearchComponent {
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

There we have it! A functional search-as-you-type implemented for YouTube videos! Try running it from the code examples if you haven't already.

angular/http API

Of course, all of the HTTP requests we've made so far have simply been `GET` requests. It's important that we know how we can make other requests too.

Making a `POST` request

Making `POST` request with `angular/http` is very much like making a `GET` request except that we have one additional parameter: a body.

jsonplaceholder API[52] also provides a convent URL for testing our POST requests, so let's use it for a `POST`:

[52] http://jsonplaceholder.typicode.com

code/http/app/ts/components/MoreHTTPRequests.ts

```
  makePost(): void {
    this.loading = true;
    this.http.post(
      "http://jsonplaceholder.typicode.com/posts",
      JSON.stringify({
        body: "bar",
        title: "foo",
        userId: 1
      }))
      .toRx()
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

Notice in the second argument we're taking an `Object` and converting it to a JSON string using `JSON.stringify`.

PUT / PATCH / DELETE / HEAD

There are a few other fairly common HTTP requests and we call them in much the same way.

- `http.put` and `http.patch` map to `PUT` and `PATCH` respectively and both take a URL and a body
- `http.delete` and `http.head` map to `DELETE` and `HEAD` respectively and both take a URL (no body)

Here's how we might make a `DELETE` request:

code/http/app/ts/components/MoreHTTPRequests.ts

```
  makeDelete(): void {
    this.loading = true;
    this.http.delete("http://jsonplaceholder.typicode.com/posts/1")
      .toRx()
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

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[53]

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 }
```

[53] https://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.js[54] you might write something like this:

```
var express = require('express');
var router = express.Router();

// define the about route
router.get('/about', function(req, res) {
  res.send('About us');
});
```

Or with Ruby on Rails[55] you might have:

```
# routes.rb
get '/about', to: 'pages#about'

# PagesController.rb
class PagesController < ActionController::Base
  def about
    render
  end
end
```

The pattern varies per framework, but in both of these cases you have a **server** that accepts a request and *routes* to a **controller** and the controller runs a specific **action**, depending on the path and parameters.

Client-side routing is very similar in concept but different in implementation. With client-side routing **we're not necessarily making a request to the server** on every URL change. With our Angular apps, we refer to them as "Single Page Apps" (SPA) because our server only gives us a single page and it's our JavaScript that renders the different pages.

So how can we have different routes in our JavaScript code?

[54] http://expressjs.com/guide/routing.html
[55] 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

```
@RouteConfig([
  { path: "/", redirectTo: "/home" },
  { path: "/home",    as: "Home",    component: HomeComponent },
  { path: "/about",   as: "About",   component: AboutComponent },
  { path: "/contact", as: "Contact", component: ContactComponent }
])
```

Notice a few things about this code:

- every route has a `path` attribute that specifies the URL this route will handle
- routes can have a name, specified by the `as` attribute. The name has to be camel cased, like `Search` or `MySearchRoute`
- routes must always declare the component that will be rendered via the `component` attribute
- routes can redirect to another URL by using the `redirectTo` attribute

Each route definition ties a given URL to a given component. When we visit the specified path the configured component will be rendered.

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.

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

```
@View({
  directives: [ROUTER_DIRECTIVES],
  template: `
  <div>
    <nav>
      <a>Navigation:</a>
      <ul>
        <li><a [router-link]="['/Home']">Home</a></li>
        <li><a [router-link]="['/About']">About</a></li>
        <li><a [router-link]="['/Contact']">Contact us</a></li>
      </ul>
    </nav>

    <router-outlet></router-outlet>
  </div>
  `
})
```

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:

```
<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

```
        <li><a [router-link]="['/Home']">Home</a></li>
        <li><a [router-link]="['/About']">About</a></li>
        <li><a [router-link]="['/Contact']">Contact us</a></li>
```

We can see on the left-hand side the `[router-link]` 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 `router-link` 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.

If we ran our app now, we would get an error:

```
Error: No base href set. Either provide a binding to "appBaseHrefToken" or add a\
 base element.
    at new BaseException (lang.js:39)
    at new Location (location.js:38)
    at ReflectionCapabilities.factory (reflection_capabilities.js:15)
    at Injector._instantiate (injector.js:587)
    at Injector._new (injector.js:542)
    at InjectorDynamicStrategy.getObjByKeyId (injector.js:306)
    at Injector._getByKeyDefault (injector.js:721)
    at Injector._getByKey (injector.js:672)
    at Injector._getByDependency (injector.js:658)
    at Injector._instantiate (injector.js:552)_
```

That's because we need to tell Angular what the *base URL* is for our application.

We do that by importing the `APP_BASE_HREF` constant:

```
import {
  APP_BASE_HREF,
  // ...
} from "angular2/router";
```

And then binding a string with the base path to it:

```
bootstrap(RoutesDemoApp, [
  // ...
  provide(APP_BASE_HREF, {useValue: '/'})
]);
```

If we used `/app` instead of `/`, the resulting links would be like `/app/home` for the `home` route, for instance.

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.

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

```
/// <reference path="../../typings/app.d.ts" />

/*
 * Angular
 */
import {Component, View} from "angular2/angular2";

@Component({
  selector: "home"
})
@View({
  template: `<h1>Welcome!</h1>`
})
export class HomeComponent {
}
```

AboutComponent

Similarly, the `AboutComponent` will just have a basic `h1`:

code/routes/basic/app/ts/components/AboutComponent.ts

```
/// <reference path="../../typings/app.d.ts" />

/*
 * Angular
 */
import {Component, View} from "angular2/angular2";

@Component({
  selector: "about"
})
@View({
  template: `<h1>About</h1>`
})
export class AboutComponent {
}
```

ContactComponent

And, likewise with `AboutComponent`:

code/routes/basic/app/ts/components/ContactComponent.ts

```
/// <reference path="../../typings/app.d.ts" />

/*
 * Angular
 */
import {Component, View} from "angular2/angular2";

@Component({
  selector: "contact"
})
@View({
  template: `<h1>Contact Us</h1>`
})
export class ContactComponent {
}
```

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 by adding a reference to the definitions file we're using:

code/routes/basic/app/ts/app.ts

```
/// <reference path="../typings/app.d.ts" />
```

And then we add the imports we'll need, both from `angular2` and `router` bundles:

code/routes/basic/app/ts/app.ts

```
/*
 * Angular
 */
import {provide, Component, bootstrap, View} from "angular2/angular2";
import {
  APP_BASE_HREF,
  ROUTER_DIRECTIVES,
  ROUTER_PROVIDERS,
  ROUTER_PRIMARY_COMPONENT,
  HashLocationStrategy,
  LocationStrategy,
  Router,
  RouteConfig,
  RouteParams,
} from "angular2/router";
```

Next step is to import the three components we created above:

code/routes/basic/app/ts/app.ts

```
/*
 * Components
 */
import {HomeComponent} from "components/HomeComponent";
import {AboutComponent} from "components/AboutComponent";
import {ContactComponent} from "components/ContactComponent";
```

Now let's get to the real component code. We start with the declaration of the component selector:

code/routes/basic/app/ts/app.ts

```
@Component({
  selector: "router-app"
})
```

Followed by the view declaration:

code/routes/basic/app/ts/app.ts

```
@View({
  directives: [ROUTER_DIRECTIVES],
  template: `
  <div>
    <nav>
      <a>Navigation:</a>
      <ul>
        <li><a [router-link]="['/Home']">Home</a></li>
        <li><a [router-link]="['/About']">About</a></li>
        <li><a [router-link]="['/Contact']">Contact us</a></li>
      </ul>
    </nav>

    <router-outlet></router-outlet>
  </div>
  `
})
```

Notice how we declare we're using `ROUTER_DIRECTIVES`. For this component, we're using both the `RouterOutlet` and the `RouterLink` directives that as we saw, are part of the `ROUTER_DIRECTIVES` import.

Like we mentioned above, `RouterOutlet` is used within our template to indicate where the route contents will be placed, represented by the `<router-outlet></router-outlet>` snippet.

The `RouterLink` directive is used by individual navigation links:

code/routes/basic/app/ts/app.ts

```
        <li><a [router-link]="['/Home']">Home</a></li>
        <li><a [router-link]="['/About']">About</a></li>
        <li><a [router-link]="['/Contact']">Contact us</a></li>
```

This means that when the anchor tag is clicked, Angular will load the routes, as we mentioned before.

The next section is where we define the routes by using an annotation:

code/routes/basic/app/ts/app.ts

```
@RouteConfig([
  { path: "/", redirectTo: "/home" },
  { path: "/home",    as: "Home",    component: HomeComponent },
  { path: "/about",   as: "About",   component: AboutComponent },
  { path: "/contact", as: "Contact", component: ContactComponent }
])
```

And finally, we declare the `RoutesDemoApp` class:

code/routes/basic/app/ts/app.ts

```
class RoutesDemoApp {
  constructor(public router: Router) {
  }
}
```

Notice that we are *injecting* a `Router` instance and making it publicly available by writing `public router: Router` on the constructor. This is good practice because it allows your component methods to manipulate the route configuration programmatically and listen to any router events.

In the last section of the `app.ts` file, we bootstrap the application:

code/routes/basic/app/ts/app.ts

```
bootstrap(RoutesDemoApp, [
  ROUTER_PROVIDERS,
  provide(ROUTER_PRIMARY_COMPONENT, {useValue: RoutesDemoApp}),
  provide(APP_BASE_HREF, {useValue: '/'})
]);
```

Like before, we are now bootstrapping the app and telling that `RoutesDemoApp` is the root component.

The difference now is that we're injecting two different things into the component:

- The `ROUTING_PROVIDERS` constant is an array with all the important pieces of Angular 2 routing and holds things like the route registry, the `Location` class, among others.
- Next, we're binding our main application component `RoutesDemoApp` to the `ROUTER_PRIMARY_COMPONENT` constant. This tells Angular2 which component to render when bootstrapping the application;
- Finally, we inject the result of the binding expression: `bind(APP_BASE_HREF).toValue('/')`. This basically sets the base URL for the application to `/`.

Here's the complete code for our `app.ts` file:

code/routes/basic/app/ts/app.ts

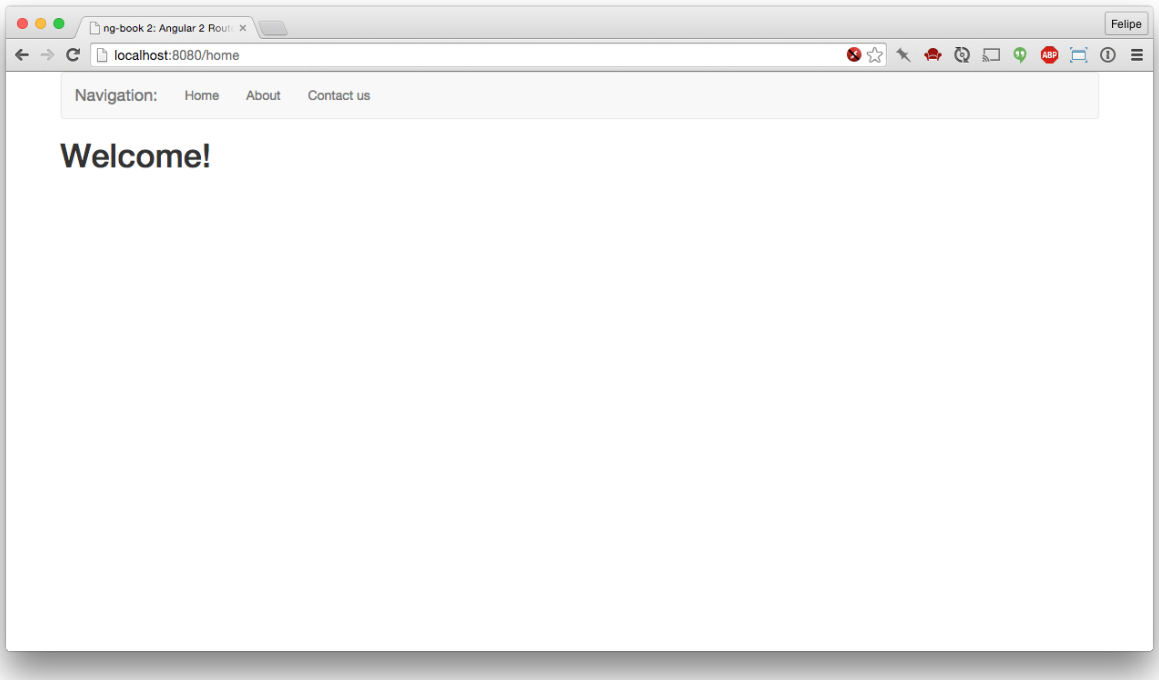
```
/// <reference path="../typings/app.d.ts" />

/*
 * Angular
 */
import {provide, Component, bootstrap, View} from "angular2/angular2";
import {
  APP_BASE_HREF,
  ROUTER_DIRECTIVES,
  ROUTER_PROVIDERS,
  ROUTER_PRIMARY_COMPONENT,
  HashLocationStrategy,
  LocationStrategy,
  Router,
  RouteConfig,
  RouteParams,
} from "angular2/router";

/*
 * Components
 */
import {HomeComponent} from "components/HomeComponent";
import {AboutComponent} from "components/AboutComponent";
import {ContactComponent} from "components/ContactComponent";

/*
 * Webpack
 */
require("css/styles.scss");

@Component({
  selector: "router-app"
})
@View({
  directives: [ROUTER_DIRECTIVES],
  template: `
  <div>
    <nav>
      <a>Navigation:</a>
      <ul>
        <li><a [router-link]="['/Home']">Home</a></li>
```

```
        <li><a [router-link]="['/About']">About</a></li>
        <li><a [router-link]="['/Contact']">Contact us</a></li>
      </ul>
    </nav>

    <router-outlet></router-outlet>
  </div>
  `
})
@RouteConfig([
  { path: "/", redirectTo: "/home" },
  { path: "/home",    as: "Home",    component: HomeComponent },
  { path: "/about",   as: "About",   component: AboutComponent },
  { path: "/contact", as: "Contact", component: ContactComponent }
])
class RoutesDemoApp {
  constructor(public router: Router) {
  }
}

bootstrap(RoutesDemoApp, [
  ROUTER_PROVIDERS,
  provide(ROUTER_PRIMARY_COMPONENT, {useValue: RoutesDemoApp}),
  provide(APP_BASE_HREF, {useValue: '/'})
]);
```

Running the application

You can now go into the application root folder (`code/routes`) and run `npm run server` to boot the application.

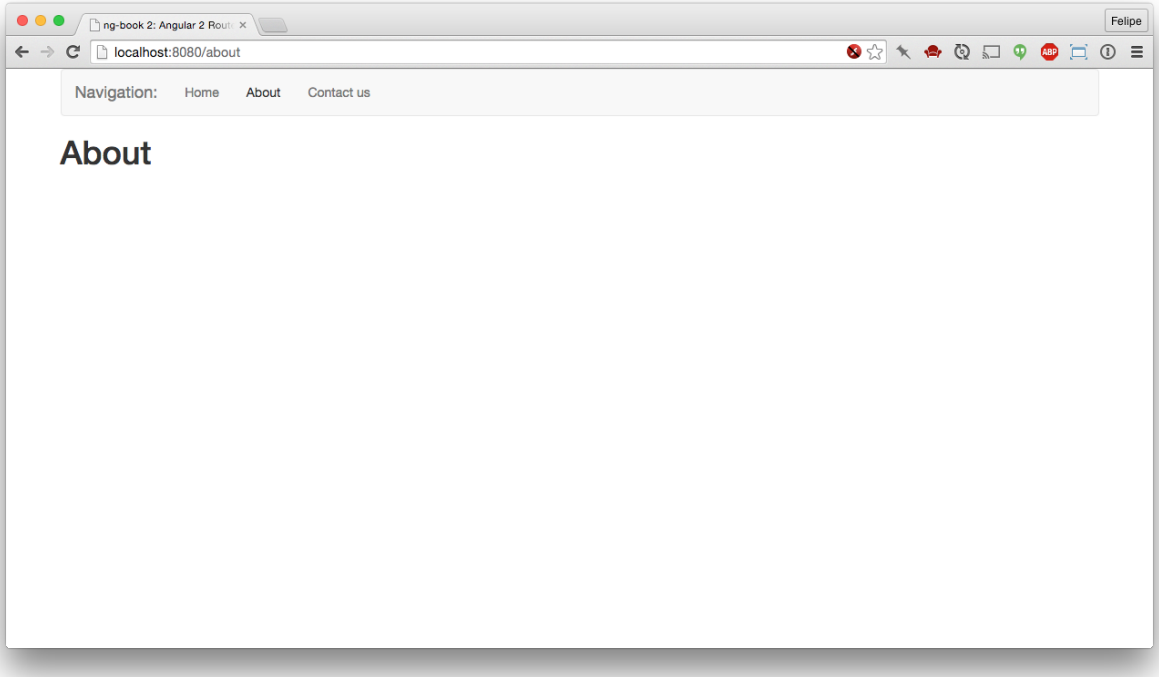
When you type into your browser you should see the home route rendered:



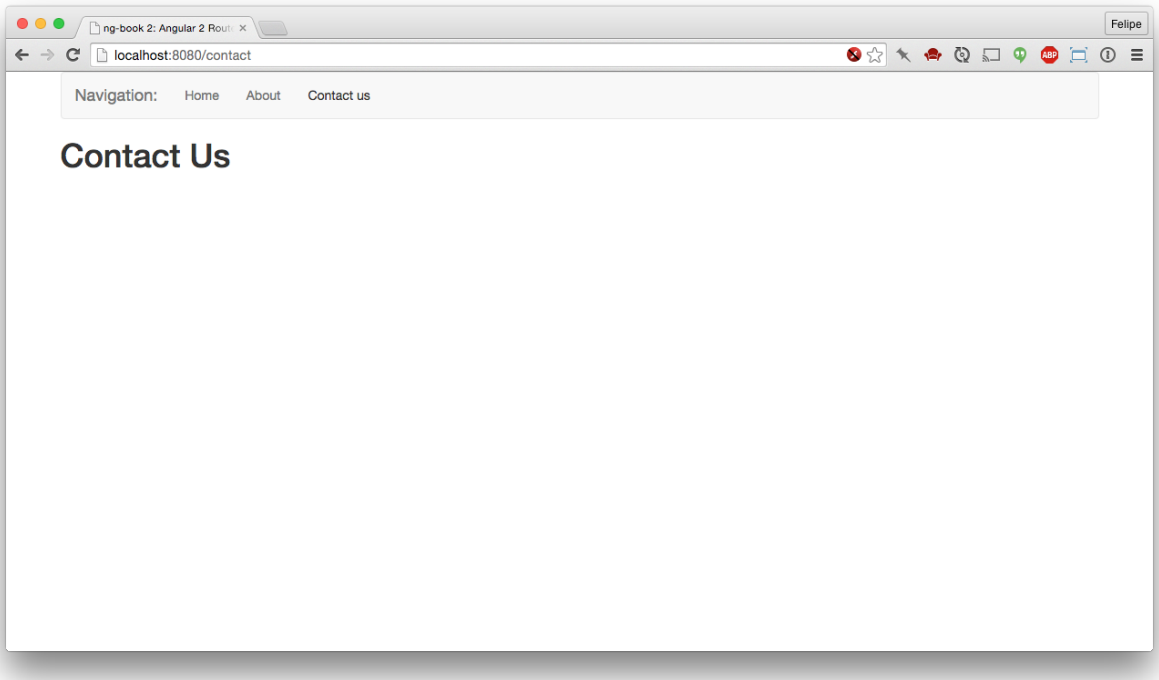
Home Route

Notice that the URL in the browser got redirected to .

Now clicking each link will render the appropriate routes:


ng-book 2: Angular 2 Rout
Felipe
localhost:8080/about
Navigation:
Home
About
Contact us
About

About Route



Contact Us Route

Routing strategies

Now if you change routes and refresh the page, you'll notice something bad happens: we get a 404! Why is that?

The way the application parses and creates paths from and to route (what we called *routing modes* in Angular 1) is now called *location strategy* in Angular 2.

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`.

Because our URLs are regular paths (that is, not using hash/anchor tags) when we refresh the page, instead of asking the server for the root URL, instead we're asking for `/about` or `/contact`. Because there's no page at `/about` the server returns a 404.

We can change the location strategy used for the application by binding the `LocationStrategy` class to a new strategy class.

Instead of using the default `PathLocationStrategy` we can also use the `HashLocationStrategy`.

This new strategy works with hash based paths, like `/#/home` or `/#/contact`. (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`.

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.hash.ts

```
1  import {
2    APP_BASE_HREF,
3    ROUTER_PROVIDERS,
4    ROUTER_PRIMARY_COMPONENT,
5    HashLocationStrategy, // <-- added
6    LocationStrategy,     // <-- added
7    Router,
8    RouterOutlet,
9    RouteConfig,
10   RouterLink
11 } from "angular2/router";
```

and then change the bootstrap section to add new binding:

code/routes/basic/app/ts/app.hash.ts

```
1 bootstrap(RoutesDemoApp, [
2   ROUTER_PROVIDERS,
3   provide(ROUTER_PRIMARY_COMPONENT, {useValue: RoutesDemoApp}),
4   provide(APP_BASE_HREF, {useValue: '/'}),
5   provide(LocationStrategy, {useClass: HashLocationStrategy}) // <-- added
6 ]);
```

In our sample application, you'll find a file called `app/ts/app.hash.ts`. If you want to play with the `HashLocationStrategy`, you can rename the file so it overwrites the `app/ts/app.ts` file and reload your application and you'll see hash based version of the URLs of the routes.

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.

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

```
@RouteConfig([
  { path: "/", redirectTo: "/search" },
  { path: "/search", as: "Search", component: SearchComponent },
  { path: "/artists/:id", as: "Artists", component: ArtistComponent },
  { path: "/tracks/:id", as: "Tracks", component: TrackComponent },
  { path: "/albums/:id", as: "Albums", component: AlbumComponent },
])
```

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`:

```
import {RouteParams} from "angular2/router";
```

Next, we inject the RouteParams into the constructor. Let's say we have a `RouteConfig` that specifies the following:

```
@RouteConfig([
  { path: "/articles/:id", as: "articles", component: ArticleComponent }
])
```

Then when we write the `ArticleComponent`, we add the `RouteParams` as one of the constructor arguments:

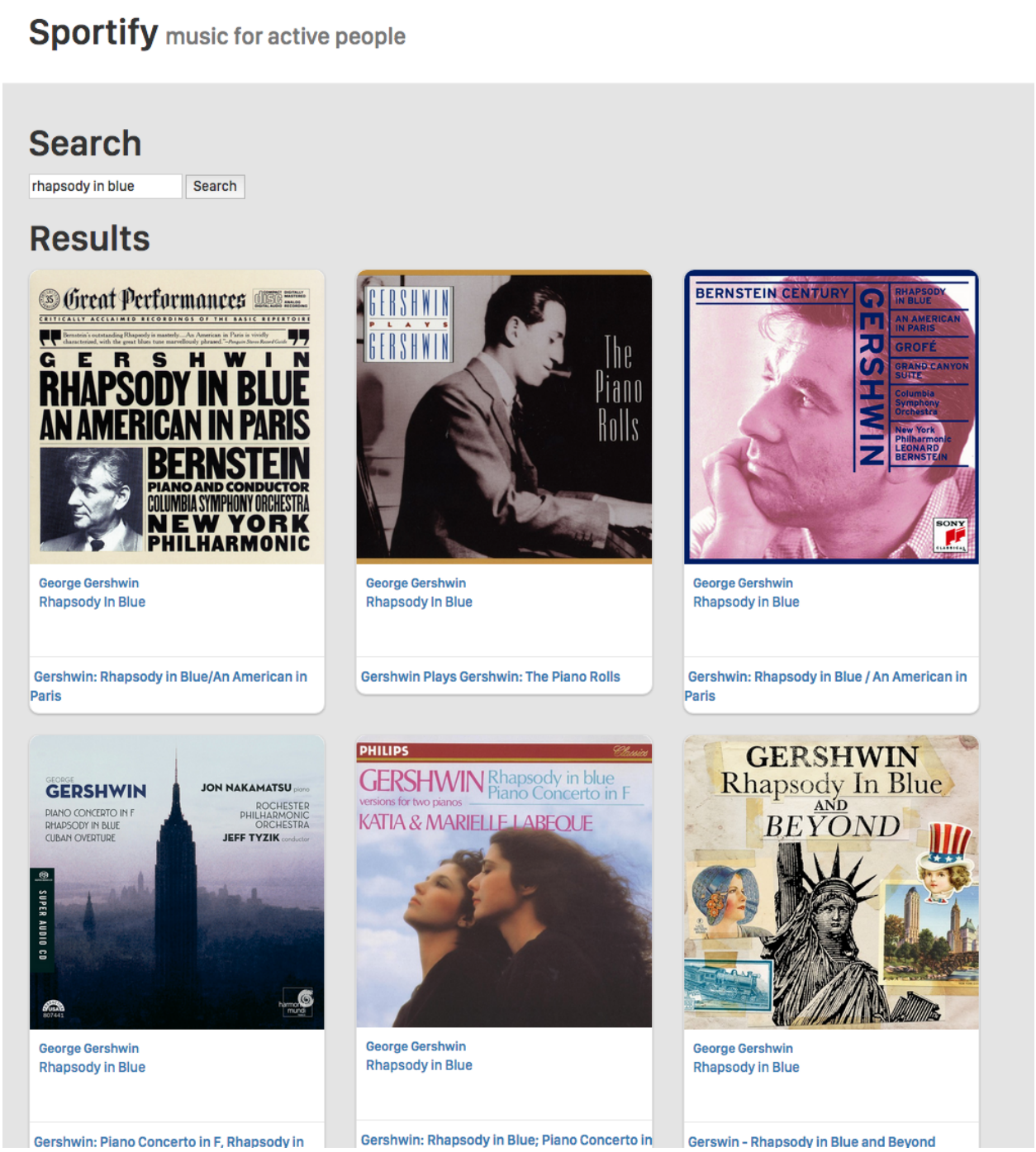
```
export class ArticleComponent {
  id: string;

  constructor(private routeParams: RouteParams) {
    this.id = routeParams.get("id");
  }
}
```

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



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 API[56] 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  /// <reference path="../typings/app.d.ts" />
2
3  /*
4   * Angular
5   */
6  import {
7    Component,
8    View,
9    provide,
10   bootstrap
11 } from "angular2/angular2";
12 import {HTTP_PROVIDERS} from "angular2/http";
13 import {
14   APP_BASE_HREF,
15   ROUTER_DIRECTIVES,
16   ROUTER_PROVIDERS,
17   ROUTER_PRIMARY_COMPONENT,
18   HashLocationStrategy,
19   LocationStrategy,
20   Router,
21   RouteConfig,
22 } from "angular2/router";
```

[56] https://developer.spotify.com/web-api

Now that we have the imports there, let's think about the components we'll use for each route.

- For the `Search` route, we'll create a `SearchComponent`. This component will talk to the Spotify API to perform the search and then display the results on a grid.
- For the `Artists` route, we'll create an `ArtistComponent` which will show the artist's information
- For the `Albums` route, we'll create an `AlbumComponent` which will show the list of tracks in the album
- For the `Tracks` route, we'll create a `TrackComponent` which will show the track and let us play a preview of the song

Since this new component will need to interact with the Spotify API, it seems like we need to build a service that uses the `http` module to call out to the API server.

Everything in our app depends on the data, so let's build the `SpotifyService` first.

The `SpotifyService`

You can find the full code for the `SpotifyService` in the `routes/music/app/ts/services` folder of the sample code.

The first method we'll implement is `searchByTrack` which will search for track, given a search term.

If we look into the Spotify API docs, we'll find the Search endpoint[57] that does exactly what we want: it takes a query (`q`) and a `type` parameters. Query is the search term and since we're searching for songs, we should use `type=track`.

Here's what this service will 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.get(queryURL).toPromise();
12   }
13 }
```

[57] https://developer.spotify.com/web-api/search-item/

This will make an HTTP `GET` request to the URL https://api.spotify.com/v1/search[58], passing our `query` as the search term and `type` hardcoded to `track`. The result will come in form of the Angular2 `http` module's `Response` object (when the promise is resolved). We'll then extract the JSON contents using the `Response`'s `.json()` method.

The `SearchComponent`

With the `SpotifyService` in place, we can now build our `SearchComponent`. As usual, we start with an import section:

code/routes/music/app/ts/components/SearchComponent.ts

```
/// <reference path="../../typings/app.d.ts" />

/*
 * Angular
 */
import {Component, View, OnInit, NgFor, NgIf} from "angular2/angular2";
import {Response} from "angular2/http";
import {
  Router,
  RouterLink,
  RouteParams,
} from "angular2/router";

/*
 * Services
 */
import {SpotifyService} from "services/SpotifyService";
```

Notice that we import the `SpotifyService` class we just created.

Next step is to declare the component annotations. We're going to use `search` as selector, so let's add the `Component` annotation like so:

code/routes/music/app/ts/components/SearchComponent.ts

```
@Component({
  selector: "search"
})
```

After that, let's move on to the `View` annotation. As you'll notice on the template, we will be using the `NgIf`, `NgFor` and `RouterLink` directives:

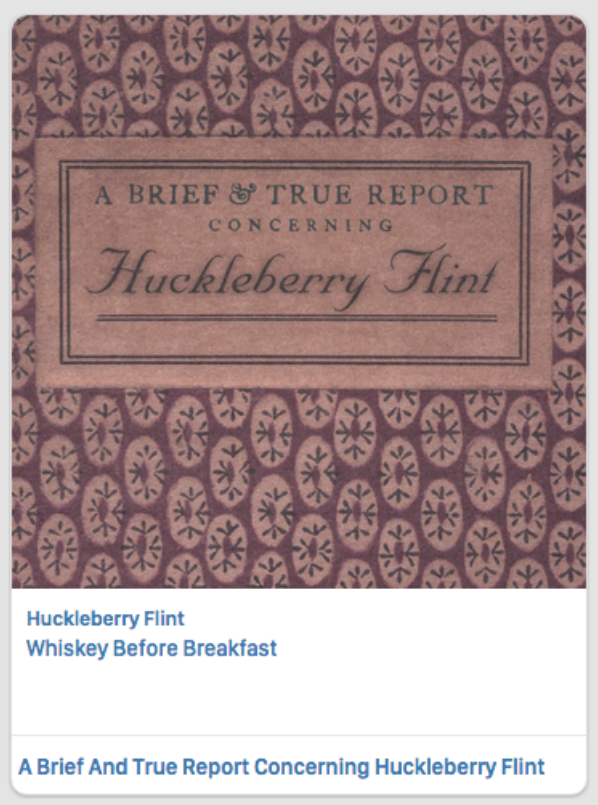
[58] https://api.spotify.com/v1/search

code/routes/music/app/ts/components/SearchComponent.ts

```
@View({
  directives: [NgIf, NgFor, RouterLink],
  template: `
  <p>
    <h1>Search</h1>
  </p>

  <p>
    <input type="text" #newquery
      [value]="query"
      (keydown.enter)="submit(newquery.value)">
    <button (click)="submit(newquery.value)">Search</button>
  </p>

  <p>
    <div *ng-if="results">
      <h1>Results</h1>

      <div class="row">
        <div class="col-sm-6 col-md-4" *ng-for="#t of results.tracks.items">
          <div class="thumbnail">
            <div class="content">
              <img src="{{ t.album.images[0].url }}" class="img-responsive">
              <div class="caption">
                <h3>
                  <a [router-link]="['/Artists', {id: t.artists[0].id}]">
                    {{ t.artists[0].name }}
                  </a>
                </h3>
                <br>
                <p>
                  <a [router-link]="['/Tracks', {id: t.id}]">
                    {{ t.name }}
                  </a>
                </p>
              </div>
              <div class="attribution">
                <h4>
                  <a [router-link]="['/Albums', {id: t.album.id}]">
                    {{ t.album.name }}
                  </a>
```

```
                </h4>
              </div>
            </div>
          </div>
        </div>
      </div>
    </p>
    `
})
```

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. The first area is responsible for displaying the search field:

code/routes/music/app/ts/components/SearchComponent.ts

```
<p>
  <h1>Search</h1>
</p>

<p>
  <input type="text" #newquery
    [value]="query"
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</p>
```

This will render an input field with its value set to the `query` component property. We also give this element a template variable `#newquery`. This means we can access the value of this input in our template 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.

Search Results and Links

The next section handles the results. You can see that we're using the `NgFor` directive to iterate through each resulting track:

code/routes/music/app/ts/components/SearchComponent.ts

```
        <div class="col-sm-6 col-md-4" *ng-for="#t of results.tracks.items">
          <div class="thumbnail">
```

Then, for each track we display the artist name:

code/routes/music/app/ts/components/SearchComponent.ts

```
                  <a [router-link]="['/Artists', {id: t.artists[0].id}]">
                    {{ t.artists[0].name }}
                  </a>
```

Notice that we use 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.

Similarly, we can link to the track itself:

code/routes/music/app/ts/components/SearchComponent.ts

```
                    <a [router-link]="['/Tracks', {id: t.id}]">
                      {{ t.name }}
                    </a>
```

And the album:

code/routes/music/app/ts/components/SearchComponent.ts

```
                    <a [router-link]="['/Albums', {id: t.album.id}]">
                      {{ t.album.name }}
                    </a>
```

`SearchComponent` Controller Class

Let's take a look at the `SearchComponent` controller class:

code/routes/music/app/ts/components/SearchComponent.ts

```
export class SearchComponent implements OnInit {
  query: string;
  results: Object;

  constructor(public spotify: SpotifyService, public router: Router,
              public routeParams: RouteParams) {
  }
```

Here we're declaring two properties:

- `query` will hold the current search term and
- `results` will hold the search results

On the `constructor` we're injecting the `SpotifyService` (that we created above), `Router`, and `RouteParams`. We're then making them a public property by prepending the keyword `public` when defining the constructor parameters.

`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:

1. When the user enters a search query and submits the form
2. When the user navigates to this page with a given URL in the query parameters (e.g. someone shared a link or bookmarked the page)

To handle each of these cases, we create a `search` function that *takes no arguments* as parameters, but instead, reads from the `routeParams` to get the query:

code/routes/music/app/ts/components/SearchComponent.ts

```
1  search(): void {
2    this.query = this.routeParams.get("query");
3    if (!this.query) {
4      return;
5    }
6    this.spotify.searchTrack(this.query).then(this.saveResults.bind(this));
7  }
```

This method returns a promise and we call the `saveResults` method when the promise returns:

code/routes/music/app/ts/components/SearchComponent.ts

```
1  saveResults(res: Response): void {
2    this.results = res.json();
3  }
```

Because `results` is a property on the component that is used in the view, Angular's change detection will detect that `results` changed and re-render the view.

Searching on Page Load If someone navigates to this page with a query in the URL then we want to run a search immediately. To do this, we use the `onInit` function, which will be called when this component is initialized (that is, after the first change detection check):

code/routes/music/app/ts/components/SearchComponent.ts

```
1  onInit(): void {
2    this.search();
3  }
```

> To use `onInit` we imported the `OnInit` class and declared that our component `implements OnInit`.

Inside this method, we call the `search` method, to perform the search based on the `query` parameter.

`submit` We also want to search when the user submits the form. `submit` takes a search query as a parameter:

code/routes/music/app/ts/components/SearchComponent.ts

```
  submit(query: string): void {
    this.router.navigate(["/Search", {query: query}]);
    this.search();
  }
```

Here we're doing two things:

1. Navigating the browser using the `Router`'s `navigate` method. This method takes the same parameters as a `RouterLink`. In our case, navigating to `['/Search', {query: query}]` will change the browser's URL to `/search?query=<search term>`. This is important in a way that if we reload the browser, the search term gets persisted. Also, if you bookmark this URL, it will take you straight back to the search term if you visit it again.
2. Performing the actual search;

Putting it Together Here's the full listing for the `SearchComponent` class:

code/routes/music/app/ts/components/SearchComponent.ts

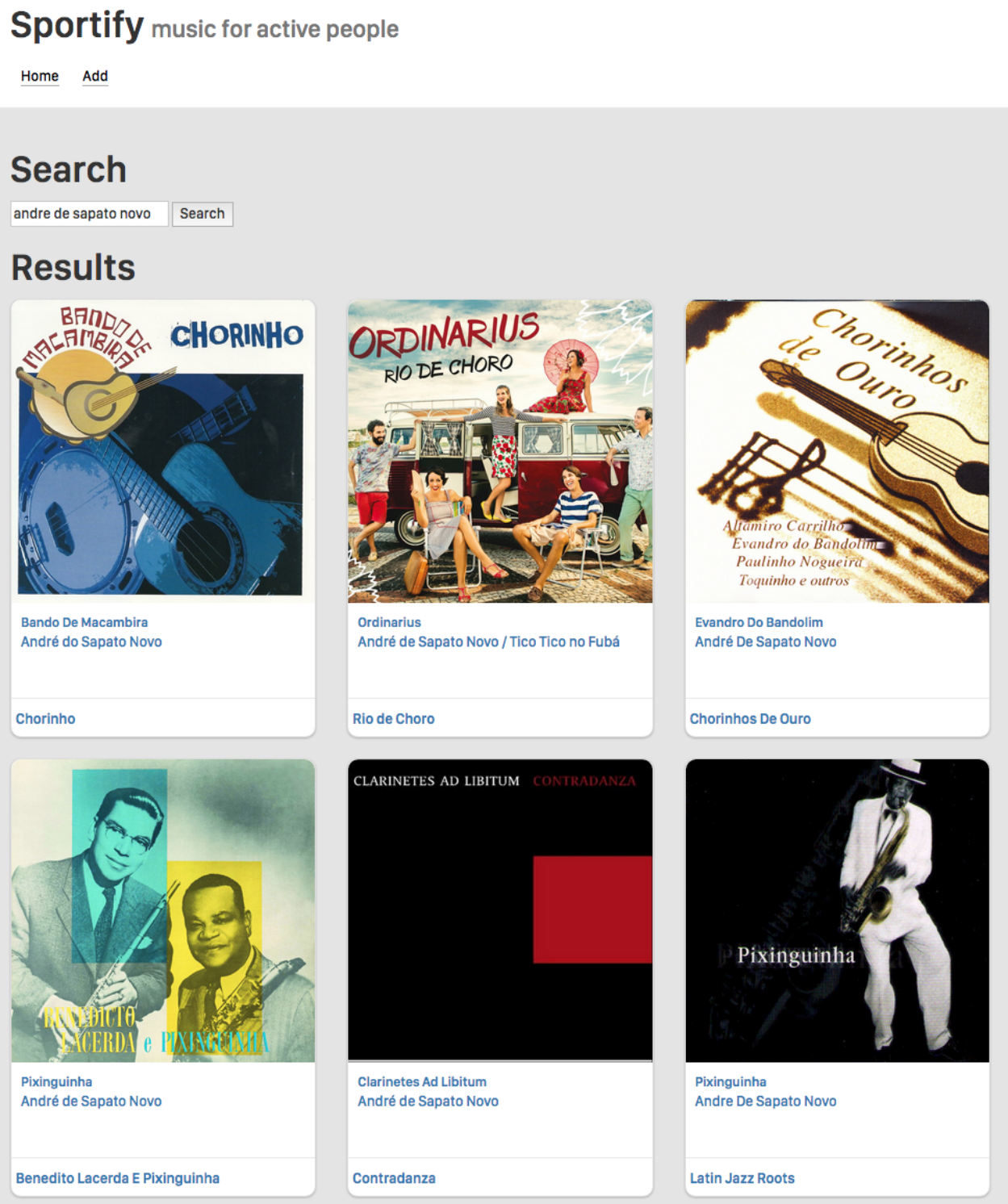
```
import {Component, View, OnInit, NgFor, NgIf} from "angular2/angular2";
import {Response} from "angular2/http";
import {
  Router,
  RouterLink,
  RouteParams,
} from "angular2/router";

/*
 * Services
 */
import {SpotifyService} from "services/SpotifyService";

@Component({
  selector: "search"
})
@View({
  directives: [NgIf, NgFor, RouterLink],
  template: `
```

```
20    <p>
21      <h1>Search</h1>
22    </p>
23
24    <p>
25      <input type="text" #newquery
26        [value]="query"
27        (keydown.enter)="submit(newquery.value)">
28      <button (click)="submit(newquery.value)">Search</button>
29    </p>
30
31    <p>
32      <div *ng-if="results">
33        <h1>Results</h1>
34
35        <div class="row">
36          <div class="col-sm-6 col-md-4" *ng-for="#t of results.tracks.items">
37            <div class="thumbnail">
38              <div class="content">
39                <img src="{{ t.album.images[0].url }}" class="img-responsive">
40                <div class="caption">
41                  <h3>
42                    <a [router-link]="['/Artists', {id: t.artists[0].id}]">
43                      {{ t.artists[0].name }}
44                    </a>
45                  </h3>
46                  <br>
47                  <p>
48                    <a [router-link]="['/Tracks', {id: t.id}]">
49                      {{ t.name }}
50                    </a>
51                  </p>
52                </div>
53                <div class="attribution">
54                  <h4>
55                    <a [router-link]="['/Albums', {id: t.album.id}]">
56                      {{ t.album.name }}
57                    </a>
58                  </h4>
59                </div>
60              </div>
61            </div>
```

```
62          </div>
63        </div>
64      </div>
65    </p>
66    `
67  })
68  export class SearchComponent implements OnInit {
69    query: string;
70    results: Object;
71
72    constructor(public spotify: SpotifyService, public router: Router,
73                public routeParams: RouteParams) {
74    }
75
76    onInit(): void {
77      this.search();
78    }
79
80    submit(query: string): void {
81      this.router.navigate(["/Search", {query: query}]);
82      this.search();
83    }
84
85    search(): void {
86      this.query = this.routeParams.get("query");
87      if (!this.query) {
88        return;
89      }
90      this.spotify.searchTrack(this.query).then(this.saveResults.bind(this));
91    }
92
93    saveResults(res: Response): void {
94      this.results = res.json();
95    }
96  }
```

Trying Out Search

Now we have enough functionality that we can try out our search:



Trying out Search

When we click either the artist, the track or the album links, we will navigate to the appropriate

route.

Improving `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

```
    <p>
      <audio controls src="{{ track.preview_url }}"></audio>
    </p>
```

Also, in order to retrieve the track information we need the Spotify API again. To make the search method more generic, we created a generic `query` and `search` methods to perform a search:

code/routes/music/app/ts/services/SpotifyService.ts

```
  query(URL: string, params?: Array<string>): Promise<Response> {
    let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join("&")}`;
    }

    return this.http.get(queryURL).toPromise();
  }

  search(query: string, type: string): Promise<Response> {
    return this.query(`/search`, [
      `q=${query}`,
      `type=${type}`
    ]);
  }
```

With that, we can now refactor the existing `searchTrack` method:

code/routes/music/app/ts/services/SpotifyService.ts

```
  searchTrack(query: string): Promise<Response> {
    return this.search(query, "track");
  }
```

And implement the new `getTrack` method:

code/routes/music/app/ts/services/SpotifyService.ts

```
  getTrack(id: string): Promise<Response> {
    return this.query(`/tracks/${id}`);
  }
```

Back to the `TrackComponent`, we can now use `getTrack` on the initialization method `onInit`:

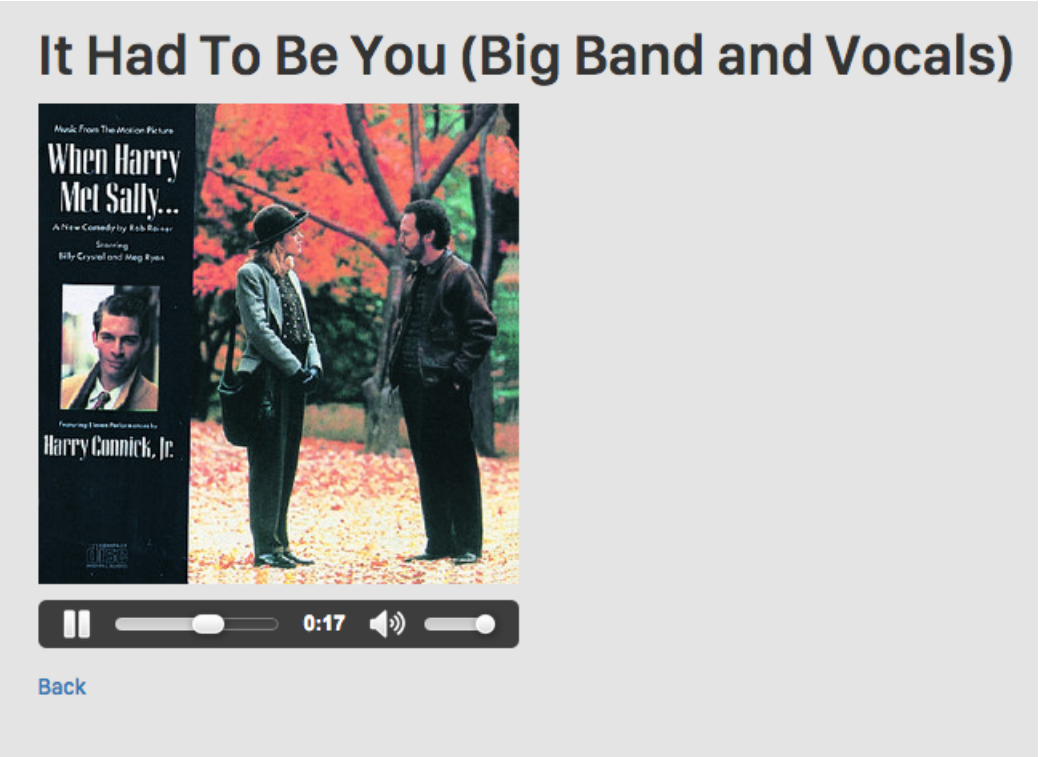
code/routes/music/app/ts/components/TrackComponent.ts

```
  onInit(): void {
    this.spotify.getTrack(this.id).then(this.renderTrack.bind(this));
  }
```

The other components work on a very similar way.

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

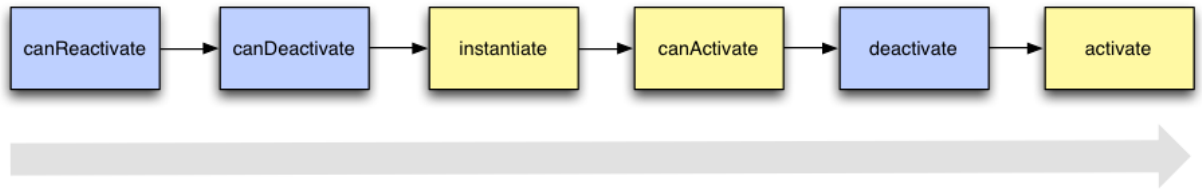
Router Lifecycle Hooks

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:

```
class MyComponent {
  canReactivate() {
    // ...
  }
}
```

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

Before changing the application components, let's introduce a new service `AuthService` that will be responsible for authentication and authorization.

code/routes/auth/app/ts/services/AuthService.ts

```
import {Injectable, provide} from "angular2/angular2";

@Injectable()
export class AuthService {
  login(user: string, password: string): boolean {
    if (user === "user" && password === "password") {
      localStorage.setItem("username", user);
      return true;
    }

    return false;
  }
```

The `login` method receives the credentials and, if it matches, we set an `username` value using the browser's `localStorage`.

code/routes/auth/app/ts/services/AuthService.ts

```
  logout() {
    localStorage.removeItem("username");
  }
```

The `logout` method just clears the `username` value.

code/routes/auth/app/ts/services/AuthService.ts

```
  getUser() {
    return localStorage.getItem("username");
  }

  isLogged() {
    return this.getUser() !== null;
  }
```

Finally, `getUser` returns the current logged user name and `isLogged` returns true if we have a logged user.

We also export an `AUTH_PROVIDERS`, so it can be injected into our app:

code/routes/auth/app/ts/services/AuthService.ts

```
export var AUTH_PROVIDERS: Array<any> = [
  provide(AuthService, {useClass: AuthService})
];
```

LoginComponent

Next, we'll create a new `LoginComponent` that will render either a login form when the user is not logged in or a banner with user information and a logout link otherwise.

The important parts of this component are the `login` and `logout` methods:

code/routes/auth/app/ts/components/LoginComponent.ts

```
export class LoginComponent {
  message: string;

  constructor(public authService: AuthService) {
    this.message = "";
  }

  login(username: string, password: string): boolean {
    this.message = "";
    if (!this.authService.login(username, password)) {
      this.message = "Incorrect credentials.";
      setTimeout(function() {
        this.message = "";
      }.bind(this), 2500);
    }
    return false;
  }

  logout(): boolean {
    this.authService.logout();
    return false;
  }
```

We are relying on our `AuthService` to validate the credentials and actually *log the user in*. On the component template, you can also notice that we are asking the service whether the user is logged in or not. When not logged in, we'll see a form:

code/routes/auth/app/ts/components/LoginComponent.ts

```
  <form class="form-inline" *ng-if="!authService.getUser()">
    <div class="form-group">
      <label for="username">User:</label>
      <input class="form-control" name="username" #username>
    </div>

    <div class="form-group">
      <label for="password">Password:</label>
      <input class="form-control" type="password" name="password" #password>
    </div>

    <a class="btn btn-default" (click)="login(username.value, password.value)">
```

```
13        Submit
14      </a>
15    </form>
```

And when logged in, we'll see information about the user, along with the logout link:

code/routes/auth/app/ts/components/LoginComponent.ts

```
1   <div class="well" *ng-if="authService.getUser()">
2     Logged in as <b>{{ authService.getUser() }}</b>
3     <a href (click)="logout()">Log out</b>
4   </div>
```

Finally, we also display an error message if the credentials are wrong:

code/routes/auth/app/ts/components/LoginComponent.ts

```
1   <div class="alert alert-danger" role="alert" *ng-if="message">
2     {{ message }}
3   </div>
```

ProtectedComponent

The other new component we'll introduce is `ProtectedComponent`. The idea is that only logged users should be able to see this component.

In order to *protect* the component from being seen by guests, we have to rely on the hooks Angular 2 router provides us. To do that, we need to add a `CanActivate` annotation to our component, and ask the `AuthService` if we have a current user:

code/routes/auth/app/ts/components/ProtectedComponent.ts

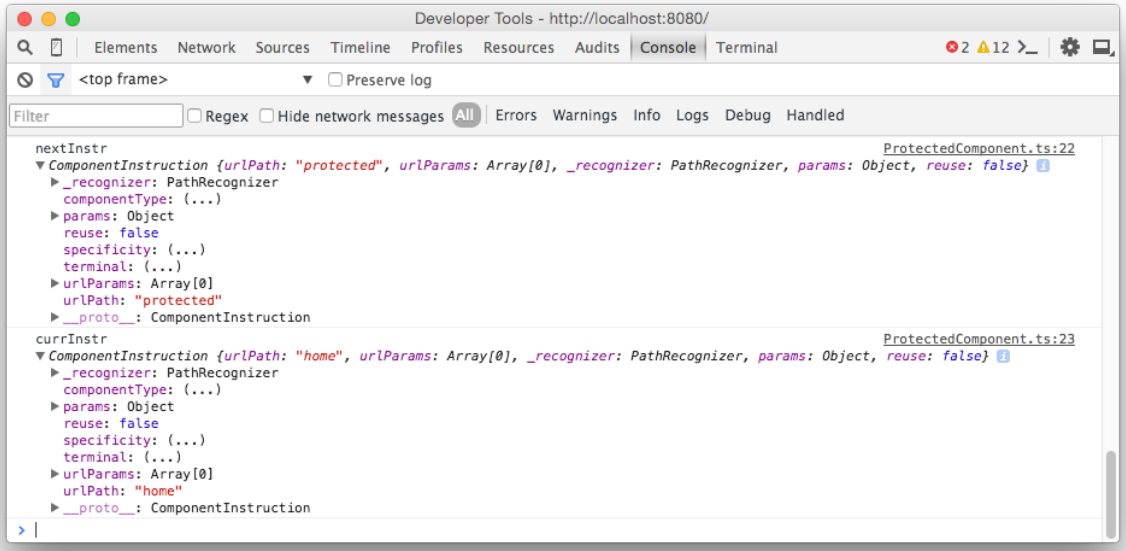
```
1   @CanActivate(
2     (nextInstr, currInstr) => {
3       let injector = Injector.resolveAndCreate([AuthService]);
4       let authService: AuthService = injector.get(AuthService);
5       return authService.isLogged();
6     }
7   )
```

The `CanActivate` annotation takes a function that receives two **Instruction**s. **Instructions** include all the route information, like route parameters, paths, URL parameters and a lot more. They are designed to help you determine if a given route change can be completed or not.

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:



Output of Instruction Debug

For more information check out the angular2 source code, that as of the writing of this chapter can be found at https://github.com/angular/angular/blob/master/modules/angular2/src/router/instruction.ts[59]

Let's break down the code a bit.

code/routes/auth/app/ts/components/ProtectedComponent.ts

```
1     let injector = Injector.resolveAndCreate([AuthService]);
2     let authService: AuthService = injector.get(AuthService);
3     return authService.isLogged();
```

The first line is creating a new injector (using `resolveAndCreate`) that only knows the `AuthService` class. Then the following line uses that injector to obtain an instance of `AuthService`.

Finally, we return the result of the `isLogged` method that, if we remember, returns true if we have an username stored on the browser's local storage.

[59] https://github.com/angular/angular/blob/master/modules/angular2/src/router/instruction.ts

Going back to the `app.ts` code, we need to import both the new `LoginComponent` and the `AUTH_PROVIDERS` we declared before:

code/routes/auth/app/ts/app.ts

```
/*
 * Components
 */
import {LoginComponent} from "components/LoginComponent";
import {HomeComponent} from "components/HomeComponent";
import {AboutComponent} from "components/AboutComponent";
import {ContactComponent} from "components/ContactComponent";
import {ProtectedComponent} from "components/ProtectedComponent";

/*
 * Services
 */
import {AUTH_PROVIDERS} from "services/AuthService";
```

And injecting it into the app:

code/routes/auth/app/ts/app.ts

```
bootstrap(RoutesDemoApp, [
  ROUTER_PROVIDERS,
  AUTH_PROVIDERS,
  provide(ROUTER_PRIMARY_COMPONENT, {useValue: RoutesDemoApp}),
  provide(APP_BASE_HREF, {useValue: '/'}),
  provide(LocationStrategy, {useClass: HashLocationStrategy})
]);
```

Let's also add a new `/protected` path to the router:

code/routes/auth/app/ts/app.ts

```
@RouteConfig([
  { path: "/", redirectTo: "/home" },
  { path: "/home",      as: "Home",      component: HomeComponent },
  { path: "/about",     as: "About",     component: AboutComponent },
  { path: "/contact",   as: "Contact",   component: ContactComponent },
  { path: "/protected", as: "Protected", component: ProtectedComponent },
])
```

And change a few things on the view:

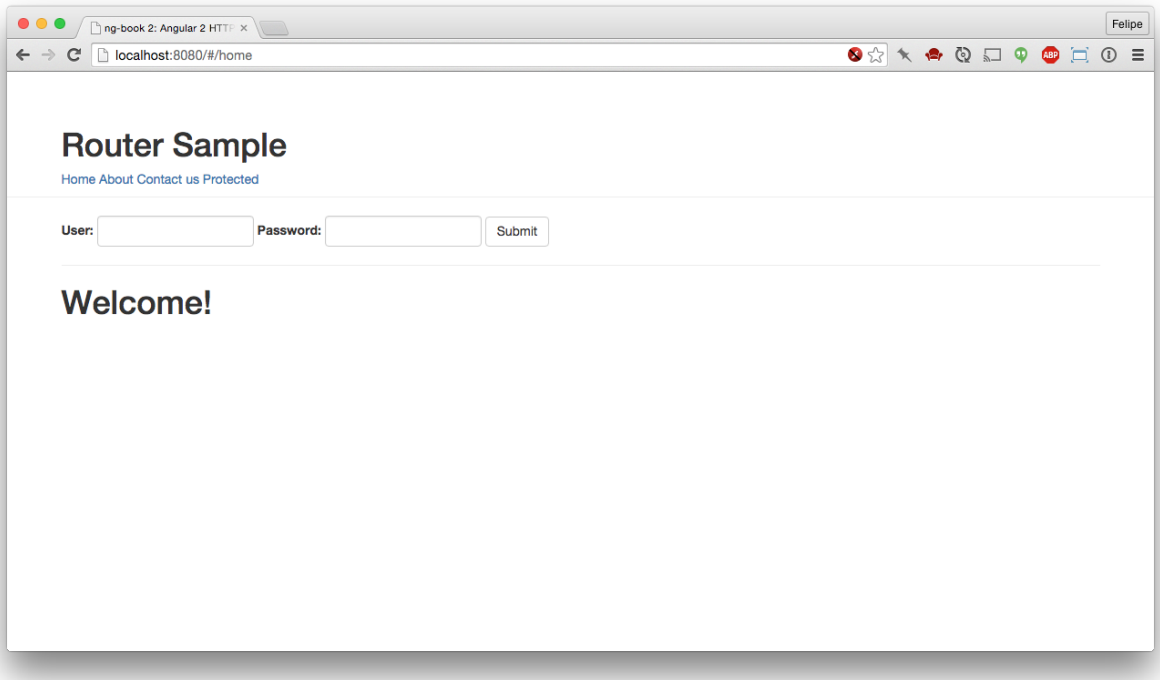
1. add the new `LoginComponent` as a directive;
2. add a link to the new protected route and;
3. add the `<login>` tag to the markup, so we render the component.

The new `View` should look like this:

code/routes/auth/app/ts/app.ts

```
@View({
  directives: [ROUTER_DIRECTIVES, LoginComponent],
  template: `
  <div class="page-header">
    <div class="container">
      <h1>Router Sample</h1>
      <div class="navLinks">
        <a [router-link]="['/Home']">Home</a>
        <a [router-link]="['/About']">About</a>
        <a [router-link]="['/Contact']">Contact us</a>
        <a [router-link]="['/Protected']">Protected</a>
      </div>
    </div>
  </div>

  <div id="content">
    <div class="container">

      <login></login>

      <hr>

      <router-outlet></router-outlet>
    </div>
  </div>
  `
})
```

Now when we open the application on the browser, we can see the new login form and the new protected link:

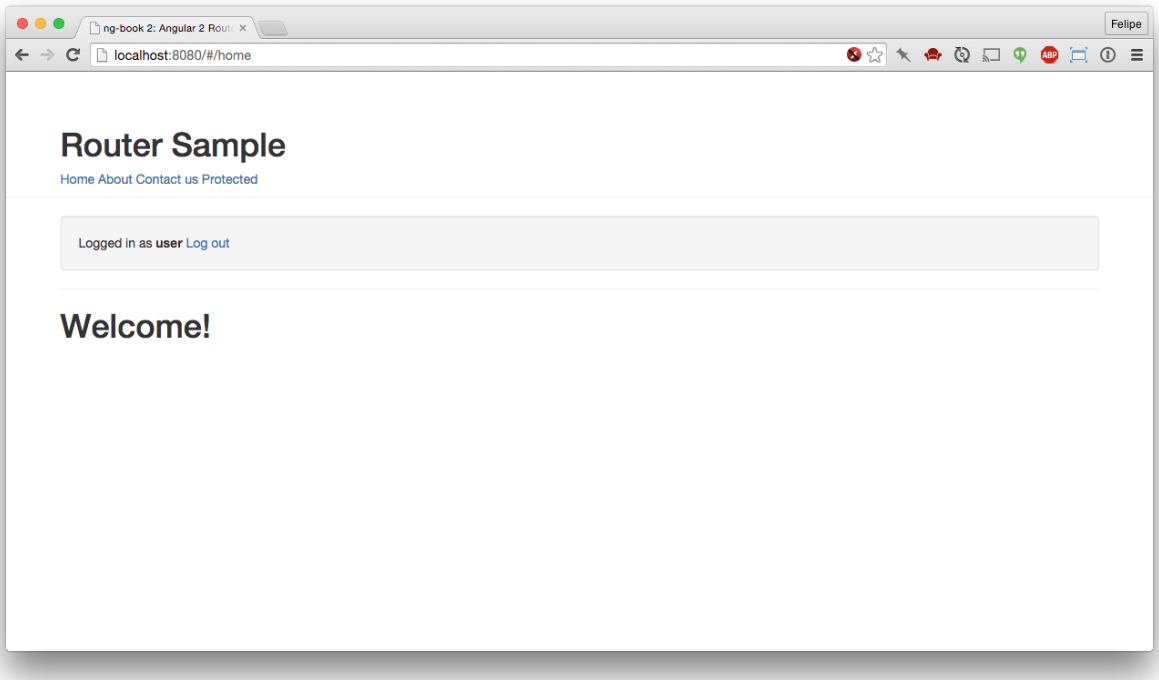


Auth App - Initial Page

If you click the Protected link, you'll see nothing happens. The same happens if you try to manually visit http://localhost:8080/#/protected[60].

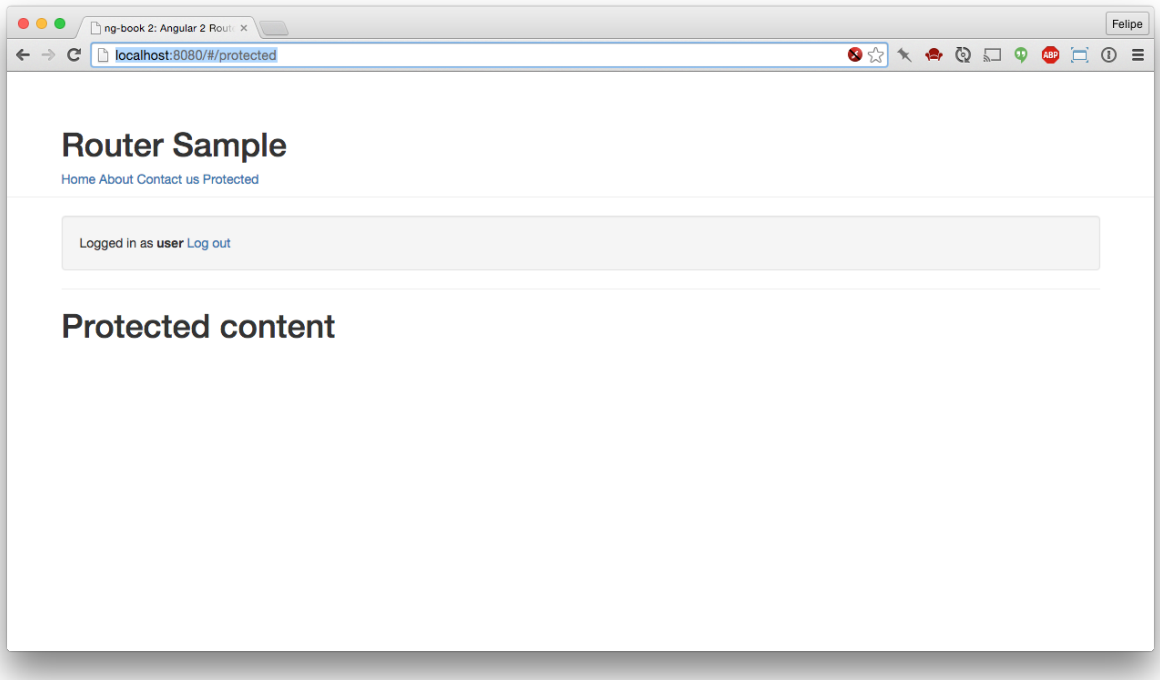
Now enter *user* and *password* on the form and click **Submit**. You'll see that we now get the current user displayed on a banner:

[60] 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

Angular's router allows you to have multiple `router-outlets`, which means we can have "nested routes".

The idea behind nested routes is that we can have different "sections" of our app, each which have their own child components.

Nested routes are clearer with an example, so let's work on another app that has two sections: the home section and the products section.

The products section will allow the user to view two products by name and also enter the id of a product.

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

On the `app.ts` file, we describe the two routes:

code/routes/nested/app/ts/app.ts

```
  { path: "/", redirectTo: "/home" },
  { path: "/home",          as: "Home",     component: HomeComponent },
  { path: "/products/...", as: "Products", component: ProductsComponent },
])
```

As you can see, for the `products` route, we're using a path like `/products/...`. This indicates that the `ProductsComponent` will also define its own routes, that will be rendered inside its own `router-outlet`.

If we look into `ProductComponent`, we find the `RouteConfig`, like below:

code/routes/nested/app/ts/components/ProductsComponent.ts

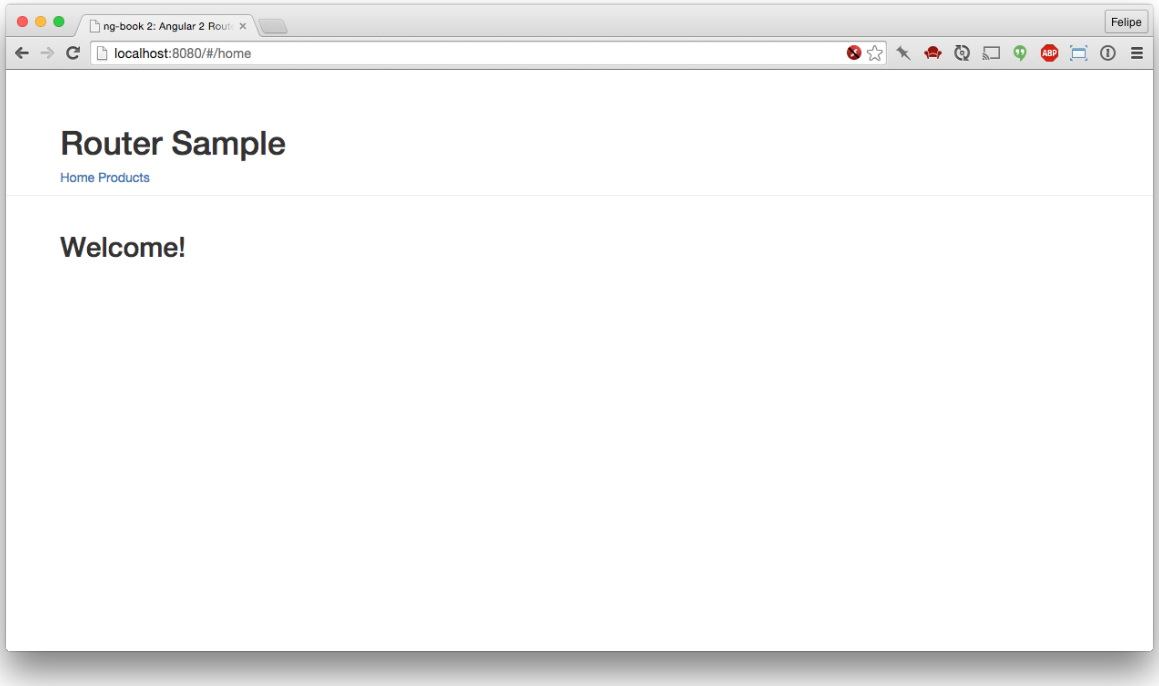
```
@RouteConfig([
  { path: "/",         redirectTo: "main" },
  { path: "/:id",      as: "ById",     component: ByIdComponent },
  { path: "/main",     as: "Main",     component: MainComponent },
  { path: "/interest", as: "Interest", component: InterestComponent },
  { path: "/sportify", as: "Sportify", component: SportifyComponent },
])
```

We will support 5 routes:

- `/products/` will redirect to `/main`;

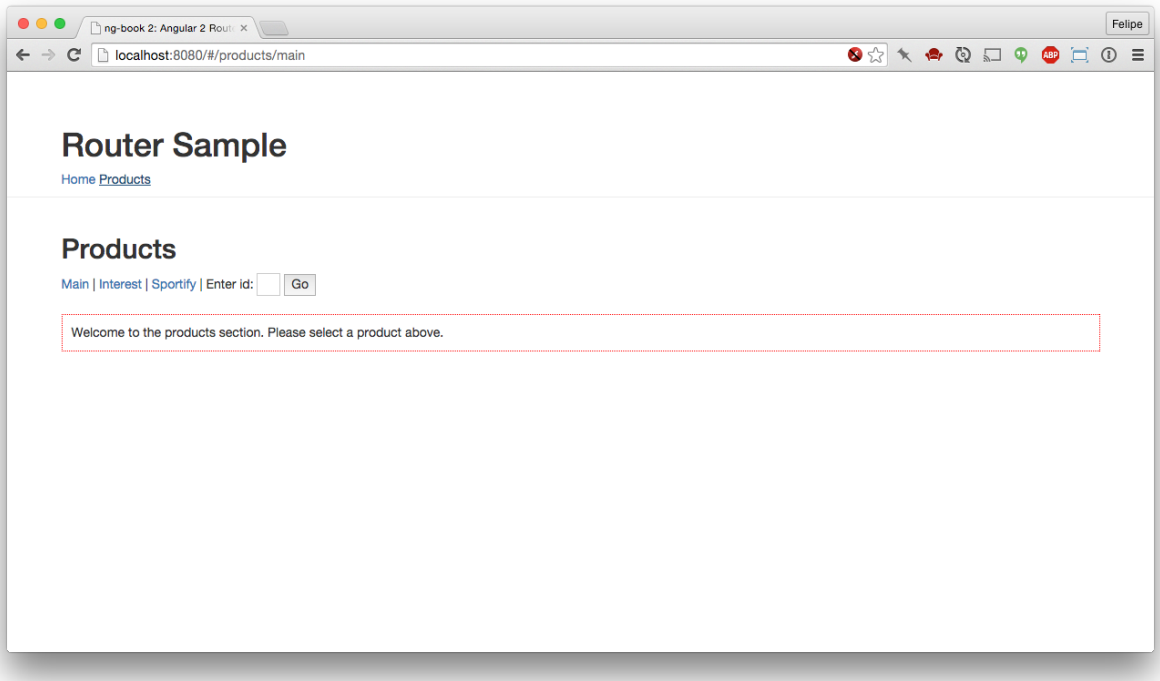
- `/products/main` will render to the `MainComponent`;
- `/products/interest` will render to the `InterestComponent`;
- `/products/sportify` will render to the `SportifyComponent`;
- anything else will render the `ByIdComponent`;

When you open the application in your browser, you'll see the welcome message of the `home` route:



Nested Routes App

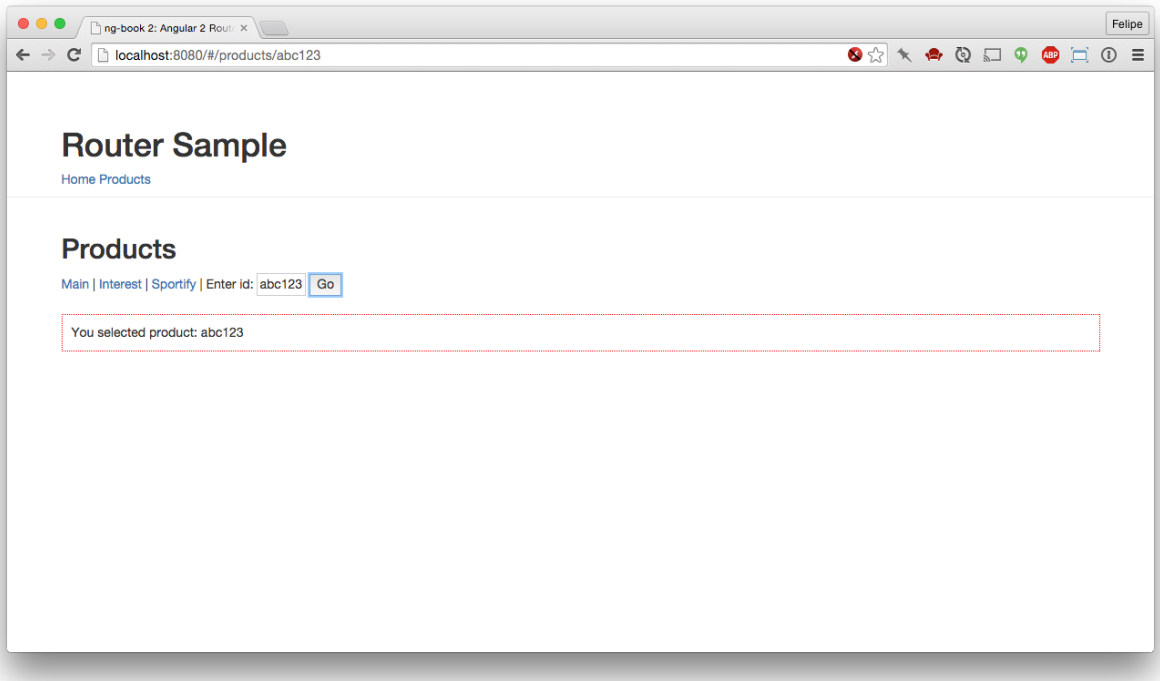
If you click on the Products link, you'll be redirected to `/products/main` that will render as follows:



Nested Routes App - Products Section

Everything below the grey line on the top is being rendered inside the main application's `router-outlet`. And everything inside the dotted red line is being rendered inside the `ProductComponent`'s `router-outlet`. That's how you nest one route within another.

Now if we visit one of the product links or if we enter anything on the textbox and click Go, you'll see new content being 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/sporitfy`) over the parameterized ones (like `/products/123`). This way `/products/sportify` will never be handled by the more generic, catch-all route `/products/:id`.

Summary

As we can see, the new Angular router is very powerful and flexible. Now go out and route your apps!

Changelog

Revision 12 - 2015-11-16

- "Routing" Chapter
  - Fixed `ROUTER_DIRECTIVES` typo - Wayne R.
- "First App" Chapter
  - Updated example to `angular-2.0.0-alpha-46`
  - Fixed some bolding around `NgFor` to clarify the code example - Henrique M.
  - Fixed `Duplicate identifier 'Promise'.` errors due to a bad `tsconfig.json` in `angular2-reddit-base/` - Todd F.
  - Fixed language typos caught by Steffen G.
  - "Forms" Chapter
    * Updated example to `angular-2.0.0-alpha-46`
      · Fixes the method of subscribing to Observables in the "Form with Events" section
    * Fixed a few typos and language issues - Christopher C., Travis P.
  - "TypeScript" Chapter
    * Fixed some unclear language about `enum` - Frede H.
  - "Built-in Components" Chapter
    * Fixed a typo where `[class]` needed to be `[ng-class]` - Neal B.
  - "How Angular Works" Chapter

    * Fixed language typos - Henrique M.

Revision 11 - 2015-11-09

- Fixed explanation of TypeScript benefits - Thanks Don H!
- Fixed tons of typos found by Wayne R - Thanks Wayne!
- "How Angular Works" Chapter
  - Fixed typos - Jegor U.
  - Converted a component to use `inputs/outputs` - Jegor U.
  - Fixed `number` to `myNumber` typo - Wayne R.
- "Built-in Components" Chapter
  - Fixed language typos - Wayne R., Jek C., Jegor U.
  - Added a tip-box explaining object keys with dashes - Wayne R.
  - Use controller view value for ng-style `color` instead of the form field value - Wayne R.
- "Forms" Chapter

    - Fixed language typos - Wayne R., Jegor U.
- "Data Architecture in Angular 2"
    - Was accidentally part of "Forms" and is now promoted to 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