Angular 2 screencast

Ari Lerner & the Fullstack.io team

Contents

1 Angular 2 screencast

- Welcome to the Angular 2 screencast to the angular 2.0 course
- In this screencast, we'll work through building a complex Angular application, starting from the very beginning.

We'll end up with a complex, tested application that will be deployed and publicly accessible.

I'm Ari Lerner, of Fullstack.io and I will be your guides on this adventure of learning Angular 2.

In order to get going, we're going to need a development environment.

1.1 Text editor

As we're going to be writing code throughout this course, we'll need a text editor capable of writing code.

We suggest the following text editors:

- Atom
- Sublime Text 2/3
- Visual Studio Code

We'll be using Atom in the screencast, but use what you feel most comfortable with.

1.2 Web browser

We're going to need a web browser to develop our application. Of course, when we develop a web application, we'll need to be able to check it out in the environment our users will be using it.

For our purposes, we're going to use the Chrome web browser. We *highly* suggest downloading it and making it your default web browser.

1.3 NodeJS

We're building our Angular application using tools that are made available through the NodeJS development environment. Download and install NodeJS if you don't already have it installed.

> If you're on a mac, you can install it using Homebrew.

1.4 Terminal

Finally, we're going to be using a terminal application throughout this course as well. Access to your terminal environments for each platform are as follows:

- Mac Open the Terminal.app application at /Applications/Utilities/Terminal.app
- Windows We suggest using Cygwin https://cygwin.com
- Unix You should already be in a terminal window

2 Bootstrapping our application

We're going to be using the `angular-cli` tool available through Node Package Manager (we'll refer to the Node Package Manager throughout this course as `npm`). We can install the `angular-cli` with the following command in our Terminal application:

```
1 npm install --global angular-cli
```

> We'll use the `--global` flag to tell `npm` we want the installed binary to be available anywhere in our development environment. The rest of the course, we won't use the `--global` flag.

We'll be using the cli throughout the entire course. The `angular-cli` bootstrap command sets up an application with an folder structure with a build system out of the box. We can use the `angular-cli` command using the binary `ng`. Make

sure this is in your $PATHenvironment.Ifit'snotinyour$'PATH', check the documents with this course.

Installing the cli can take quite a while, so be patient as we download it. A command-line interface (cli) gives us a common structure for our angular 2 applications. It also allows us to completely bypass custom-building our own build-tooling systems, which any experienced developer will tell you… it's the worst.

Once the `angular-cli` package has been installed, we can use it in our terminal to bootstrap our Angular 2 application. The `angular-cli` tool will take a few minutes to create the application structure, including writing our first sample application files. It gives us an excuse to be lazy with our application generation.

Let's bootstrap our application with the following commands:

```
1 ng new reddit-clone
2 cd reddit-clone
```

the `ng new` command will take another few minutes to create the structure and install our dependencies automatically.

2.1 Launching our applications

The `angular-cli` package comes with a few commands we'll be using throughout the course. The `serve` command boots up a built-in webserver. It will package our application up and boot it on our local computer and make it available at http://localhost:4200. Let's check out our initial application in Chrome.

Look at that! We have a running application.

Let's look through the files that the `angular-cli new` command created for us.

Looking through the directories, we will spend most of our time in the `src/` directory. The `src/` directory contains all of our application files, our configuration, our tests, and templates. We'll also be editing the `angular-cli.json` and later, spend time with the `karma.conf.js` and `protractor.conf.js` as well as the `e2e` directory when we get to testing our application.

The `src/` directory contains a few directories where we'll spend almost all of our time. The `app/` directory contains all of our application files, the `assets/` directory will contain assets we'll include, and the `environments/` directory contains

different environment values for our different deployment environments. When we start to write tests for our application, we'll dive deeper into the `environments/` directory.

2.2 Our generated application

We're going to use the default language for building our Angular 2.0 application of TypeScript. If you have never used TypeScript before, do not worry as we'll explain any intricasies we need to worry about. TypeScript provides a lot of convenient safety checks in our web application and is pretty similar to ES6, in syntax.

In any web application, the first spot to check for the startup location is the `index.html`. This `index.html` contains the instructions for Angular to start it's application. The `ng` command takes care of loading the appropriate JavaScript files in the browser automatically. The interesting part of the `index.html` file is the tag `<app-root></app-root>`. This is the `selector` of our root Angular component.

Let's skip over all the intermediate files that deal with loading our app and jump right to the first component in the `src/app/app.component.ts` file. The `src/app/app.component.ts` file contains our first Angular 2 component, which is what we currently see running in the browser.

/ / TODO: Define this For instance, if we could write a component for the `<h1 />` tag, it might look similar to the following (pretend that the `<b />` tag means big and bold):

```
1 @Component({
2   selector: 'h1',
3   template: '<b>{{ content }}</b>'
4 })
5 export class H1Component {}
```

2.3 What is a component

Angular 2 is built on the idea of components, but it's not unique to Angular 2. Components are a way for us to define the content and business logic of how a browser should treat an HTML tag.

> If you're familiar with Angular 1, you can think about components as being a mashup between controllers, `$scope`, and directives.

We can think about an Angular component like we're building new functionality into the browser. We can define dynamic functionality in our components, as we'll see in building our application.

2.4 The basics

Seeing TypeScript for the first time might feel jolting, but don't worry, we're here to help. Let's look at the first component `app.component.ts`.

The first line uses the `import-from` keyword. Just like it sounds, this `imports` (or requires) the `Component` that's *exported* by the `@angular/core` package. High-level, this means something called `Component` is provided by the `@angular/core` package. In fact, we can see that in action in our `app.component.ts` file.

The `import` syntax is similar to the `require` function in ES5 Node-style JavaScript. Using the `require()` function, the equivalent functionality looks like:

```
1 var Angular = require('@angular/core');
2 var Component = Angular.Component;
```

The `app.component.ts` file already defines an export of the class `AppComponent`. If we wanted to import this `AppComponent` class from another file, we can use the same syntax:

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

The `@` sign refers to a decorator. Although it sounds scary, a decorator is just a function that *adds/changes* functionality of a JS object (such as functions or objects).

2.4.1 Example of a decorator

> Decorators are just functions which accept an object definition as it's first argument.
>
> For instance, let's say that we have a function that says hello in pure JavaScript:

```
function sayHello() {
  return "Hello";
};
```

Let's add a decorator that appends a dynamic name to the return value of the function. We could define it like so:

```
function decorator (fn) {
  return function(name) {
    return fn() + " " + name;
  }
}
```

We can decorate the `sayHello()` function with the decorator by passing the decorator function the `sayHello` function.

```
const decorated = decorator(sayHello);
// using
decorated("Ari"); // "Hello Ari"
```

All of the decorators we'll use in ng2 work like this, where the input (the `fn`) is the class and the decorator adds functionality to the object.

We can define a component by using the `@Component()` decorator before our class definition. The `@Component()` decorator accepts a single argument, which is expected to be an object where the keys are the name of functionality we want the `@Component()` to use and the values are the inputs to these features.

The minimum requirements a `@Component` needs to define in the object argument are:

- `selector` – the string HTML tag where we want the component to be booted
- `template`/`templateUrl` - the HTML content that defines the content of the component.

Let's update our `app.component.ts` file slightly to simplify our setup for the time being. Rather than use a `templateUrl`, let's use the `template` key.

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Hello {{ title }}</h1>
  `
})
export class AppComponent {
  title = 'Ari';
}
```

2.5 High-level approach

Let's take a step back and look at what we are building. We have a sidebar component and a main articles pane. Inside the main articles pane, we have another component, the article.

All in all, we'll have 3 major components with their own functionality requirements. We'll look at each of these when we get to them. For now, let's get started building our application.

For the time being, let's focus on building an initial frame of our application. Rather than focus on building our application correctly the first time, let's just get to building the functionality of our application and then coming back to break up our application into multiple files with tests. We can shrug off the idea that we have to build the perfect application and just start using Angular 2. However, we will come back and update our application file structure.

> Note that we are *specifically not* following the Angular StyleGuide for the first part of our journey with Angular 2. We'll come back and update our code to match the styleguide a little bit later.
>
> We've found that reducing the amount of work we need to learn when starting out learning/using Angular 2 is more efficient than being perfect. But don't worry, we'll come back to writing our application using the styleguide guidelines, so by the end of this, you'll be writing applications with the intentions set out by the styleguide.

Let's get on with writing some code. We're going to make a reddit-clone application. Let's find our `src/app/app.component.ts` file generated by the generator and make a small update that will make it a bit easier to view all of our work in this single file.

Rather than use the `templateUrl` that was automatically generated for us, let's set the `template` option on our component and write our markup inline. We'll come back later and move them back to their own templates, but for now it'll be easier to see the template inside the same location as our definition of the component.

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

@Component({
  selector: 'app-root',
  template: `
    <div>Hello from the AppComponent</div>
  `
})
export class AppComponent {}
```

> Note that we're using the backtick (') instead of the single/double quote mark. This tells our TypeScript compiler that we are writing a multi-line string, instead of needing to manually concatenate our strings together.

We can run the `angular-cli` built-in webserver using the `ng serve` and any changes we make to our `AppComponent` will be automatically reloaded so we can see the changes update in real-time. When we need to kill the server and restart, we'll mention it, but we like to keep it running while we're editing.

To start building our application, let's update our template to include a block pane with a sidebar as well as the main content article listing:

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <div id="sidebar">
```

```
      Sidebar will go here
    </div>
    <div id="content">
      <div>
        Article list will go here
      </div>
    </div>
  </div>
  `
})
export class AppComponent {}
```

Ignoring the `id`s in the source code (just using them for clarity about the intention of the elements in our template), we have a basic application that contains a completely unstyled set of text that says we have the two texts in the browser.

By placing all of our markup inside a single component, we are limiting our ability to scale our application. Even though we are defining all of our code in this single file, using one component to define our entire application is kind of crazy.

Let's define our second component, which will be our sidebar. In order to create a component, we'll need to decorate a class with the `@Component` decorator. Let's define the class we are going to decorate called `SidebarComponent`. We can define this in the same `src/app/app.component.ts` file:

```
@Component({
  selector: 'app-sidebar',
  template: `
    <div id="sidebar">
      Sidebar will go here
    </div>
  `
})
class SidebarComponent {}
```

In order to actually use the new `SidebarComponent`, we'll place a tag with the name of the selector, called: `app-sidebar` in the `AppComponent` template:

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <app-sidebar></app-sidebar>
    <div id="content">
      <div>
        Article list will go here
      </div>
    </div>
  </div>
  `
})
export class AppComponent {}
```

Unfortunately, when we run this in the browser, we'll see an error in the Chrome console. The reason we'll see this error is that although the component is defined in the same file as the `AppComponent`, Angular doesn't know that we want to use the `SidebarComponent` in our app.

In order to tell Angular we want to use the `SidebarComponent` in our application, we'll need to *declare* to Angular that we want to use it. We can *declare* this in the `src/app/app.module.ts` file, which contains the `@NgModule` created by the generator.

In order to declare the `SidebarComponent` in the `app.module.ts` file, we'll need to *export* the class from the file so we can reference it from another file.

Let's update the definition of the `SidebarComponent` with the `export` keyword:

```
@Component({
  selector: 'app-sidebar',
  template: `
    <div id="sidebar">
      Sidebar will go here
    </div>
  `
})
export class SidebarComponent {}
```

Let's now open up the `src/app/app.module.ts` file and *import* this `SidebarComponent` from the `src/app/app.component.ts`. We can see the `AppComponent` already being imported from the file. We just need to add a second entry to *also* pull over our `SidebarComponent`.

```
// ...
import {
  AppComponent,
  SidebarComponent
} from './app.component';

@NgModule({
  declarations: [
    AppComponent,
    SidebarComponent
  ],
  // ...
})
export class AppModule { }
```

Refreshing our page again (this will happen *automatically* for us by the `ng serve` webserver), we'll see we no longer see the same error and our component shows up in the browser.

2.5.1 Familial relationships between components

The relationship between the `AppComponent` and the `SidebarComponent` is in a heirarchy. We say that the `SidebarComponent` is a *child* of the `AppComponent`. If we use another component in the `AppComponent`, the `SidebarComponent` will be a *sibling* of the `SidebarComponent` and a child of the `AppComponent`.

2.6 Dynamic data

Currently, we've worked directly with static data inside of our `AppComponent`. In order to set ourselves up to work with dynamic data in our application, let's create another component which we'll use to display a single Article.

Let's create the `ArticleComponent`.

We'll need to define a class called `ArticleComponent` which we'll need to `export` as we'll want to pull this component out and add it to the `@NgModule` of our app, as we did with the `SidebarComponent`.

In order to tell Angular we want the class called `ArticleComponent` to be an Angular component, we need to decorate it with the `@Component` decorator. We'll add the selector and template along with this `ArticleComponent` class.

```
@Component({
  selector: 'app-article',
  template: `
    <div>Article will go here</div>
  `
})
export class ArticleComponent {}
```

Let's update our `src/app/app.module.ts` file to include and declare the `ArticleComponent`:

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

@NgModule({
  declarations: [
    AppComponent,
    SidebarComponent,
    ArticleComponent
  ],
  // ...
})
export class AppModule { }
```

With our component registered with the module, we can now use the component. Let's add a few article components inside our `AppComponent` template:

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <app-sidebar></app-sidebar>
    <div id="content">
      <app-article></app-article>
      <app-article></app-article>
      <app-article></app-article>
      <app-article></app-article>
      <app-article></app-article>
    </div>
  </div>
  `
})
export class AppComponent {}
```

When the page refreshes, we'll see that we have 5 individual instances of the `ArticleComponent` in our page… that is, we'll have 5 copies of the template from the `ArticleComponent`, one for each instance that appears in the template.

We are still only using static data in our application, however. In order to make our application use dynamic data, we'll need to create a JavaScript object to use as the data variable. We'll create a JavaScript variable inside our app that we'll pass along to our components. This way we can pass along *dynamic* objects along to our components and let the templates work regardless of the underlying data.

Let's add a variable called `article` to our `AppComponent`.

```
@Component({
  selector: 'app-root',
  // ...
})
export class AppComponent {
  article: Object;
}
```

Now any instance of our `AppComponent` will contain the `article` variable (object). Right now, the `article` will be undefined for all of our instances. We can set the value of the variable when we create the instance.

We can depend upon TypeScript to call the `constructor()` method on our clsses when it creates an instance of our class. We can use the `constructor()` to define the value of this `article` variable for each class.

```
@Component({
  selector: 'app-root',
  // ...
})
export class AppComponent {
  article: Object;

  constructor() {
    this.article = {
      title: 'The Angular 2 screencast',
      description: 'The easiest way to learn Angular 2'
    }
  }
}
```

To *pass* this article instance into an instance of the `ArticleComponent`, we'll need to take the following steps:

1. Define the input on the component class
2. Pass the variable to the instance in our template.

In order to define the input on the class, we can use the `@Input` decorator provided by the `@angular/core` package. Let's import the decorator from the `@angular/core` scoped package:

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

We can *decorate* a variable in the `ArticleComponent` by prepending it with the `@Input()` decorator and reference it in the template of our component:

```
@Component({
  selector: 'app-article',
  template: `
    <div>
      <h2>{{ article.title }}</h2>
    </div>
  `
})
export class ArticleComponent {
  @Input() article: Object;
}
```

This takes care of step 1. For step 2, all we need to do to pass a value of a variable is pass the `article` variable with the square brackets (`[]`) where the "name" inside the square brackets matches the `@Input` variable in the `ArticleComponent` and it's value equals the name of the instance variable in the `AppComponent`:

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <app-sidebar></app-sidebar>
    <div id="content">
      <app-article
        [article]="article"></app-article>
    </div>
  </div>
  `
})
export class AppComponent {
  article: Object;
  // ...
}
```

With the input passed in our `AppComponent` instance, we now get a rendered story on the the page.

2.7 Multiple dynamic variables

Right now, we only have one dynamic article variable being added to our page. Often times, we'll want to add multiple dynamic articles to the page, rather than one.

Angular 2 gives us a built-in way of handling an iterable object to add multiple copies of a component on screen. We can accept a *list* object and iterate over the object, creating a new component for each one.

Let's update our implementation to show multiple articles, instead of just one. First, let's change the definition of the `article` variable in the `AppComponent` class to a list of Objects and change the name from `article` to `articles`:

```
@Component({
  selector: 'app-root',
  // ...
})
export class AppComponent {
  articles: Object[];
  // ...
}
```

2.7.1 Arrays in TypeScript

In TypeScript, we can annotate a variable as a list by adding the square brackets (`[]`) to the end of the definition. This note tells TypeScript that we are expecting a list of the type denoted.

Next, let's update the instantiation of the article to be a list of articles:

```
@Component({
  selector: 'app-root',
  // ...
})
export class AppComponent {
  articles: Object[];

```

```
  constructor() {
    this.articles = [{
      title: 'The Angular 2 screencast',
      description: 'The easiest way to learn Angular 2'
    }, {
      title: 'Fullstack React',
      description: 'Wanna learn React as well?'
    }];
  }
}
```

In order to iterate through our list, we'll need to use a loop for each article item in the `articles` list. In Angular 2, we have the `ngFor` directive.

We'll need to update our template to account for the fact that we are now sending a list of objects where we want to create a new component instance for each one.

We can add the directive `*ngFor` to loop through our articles and create a component instance for each variable in the list:

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <app-sidebar></app-sidebar>
    <div id="content">
      <app-article
        *ngFor="let article of articles"
        [article]="article"></app-article>
    </div>
  </div>
  `
})
export class AppComponent {
  articles: Object[];
  // ...
}
```

The ngFor directive takes the expression form of:

`let [localVariable] of [iteratable variable]`

where Angular will loop through the list and create a variable named what we place in `[localVariable]` from each of the `[iterable variable]`. Therefore, in the above example, Angular sets the local variable for each of the articles in the `articles` variable to `article` for each copy.

We *still* have to pass the `article` input to each `app-article`, with just adding the `ngFor` directive.

This creates a copy of the `app-article` component for every article.

3 Models

Often times we'll want to add additional functionality and validation on our objects. One feature TypeScript gives us to handle this is defining types.

When we create an object type, we can define it's attributes and their type such that we can create a class around the type of object. With a class, we can add methods, validations, etc.

Instead of passing a simple `Object` in our list of articles of our `AppComponent`, let's create a class we'll call `Article` to encapsulate the functionality of an article.

```
1 class Article {
2   title: string;
3   description: string;
4 }
```

Just like we did with our `AppComponent`, we can use the `constructor()` method to set the instance values.

```
1 class Article {
2   title: string;
3   description: string;
4
5   constructor(title: string, description: string) {
6     this.title = title;
7     this.description = description;
```

```
  }
}
```

Since this is such a common pattern, TypeScript gives us the ability to define the two steps in one. Rather than define both the property and set the instance property, we can do those in the same line by updating our description to define and set the properties in the constructor as follows:

```
class Article {
  constructor(
    private title: string,
    private description: string) {
  }
}
```

Instead of setting the list of articles on the `AppComponent` as a plain object now, we can update it to be a list of instances of our `Article` model.

```
@Component({
  selector: 'app-root',
  template: `
  <div id="container">
    <app-sidebar></app-sidebar>
    <div id="content">
      <app-article
        *ngFor="let article of articles"
        [article]="article"></app-article>
    </div>
  </div>
  `
})
export class AppComponent {
  articles: Article[];

  constructor() {
    this.articles = [
      new Article(
```

```
        'The Angular 2 screencast',
        'The easiest way to learn Angular 2'
      ),
      new Article(
        'Fullstack React',
        'Wanna learn React as well?'
      )
    ];
  }
}
```

Our app still renders as we expect *and* it now we have a way of adding functionality we can use in our template. For instance, let's add a `date()` function to the `Article` to demonstrate adding this functionality.

Let's open up the `Article` class and add the `date()` function to return the current date:

```
class Article {
  constructor(
    private title: string,
    private description: string) {
  }

  date() :Date {
    return new Date();
  }
}
```

> The `date()` function will run *everytime* the template renders, so it will constantly update everytime we change anything on the screen. We'll come back to update this later.

With the `date()` function on the `Article`, we can add the result to the template of the `ArticleComponent`.

```
@Component({
  selector: 'app-article',
```

```
  template: `
    <div>
      <h2>{{ article.title }}</h2>
      <p>{{ article.date() }}</p>
    </div>
  `
})
export class ArticleComponent {
  @Input() article: Object;
}
```

Now the `date` of our `Article` is present in the view!

4 Styling

At this point, let's take a step back and make our application look a little better. It's currently pretty ugly. In this section, we're going to look at how to integrate stying in our application.

When we talk about styling, we'll be working through setting up our application to use global styling, local styling, and inline styling. With Angular 2, there are several different types of approaches we can take when styling our components. Let's look at these several different ways we can style our components.

4.1 Global styling

Global styling is regular css styling with nothing extra applied or added. When we build global styling, we're talking about adding a Cascading StyleSheet (CSS, for short) to the document of our application and allowing the native CSS browser handling to manage the styles.

We'll talk about two types of adding styling into our application at the global level:

- Adding a stylesheet to the page template
- Adding styles at the global level from a component

4.1.1 Adding a stylesheet to the page template

Let's add a stylesheet to the page template first. We can add a stylesheet to our application in the same way that we would for any web application. We can add a `<link />` tag to the `<head />` element of our page and let the browser fetch it normally.

Let's say we wanted to add the CDN version of semantic-ui at `https://cdnjs.cloudflare.com/ajax/li ui/2.2.4/semantic.min.js`. We can add the `<link />` tag to our `src/index.html` just like we would with any other web application.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>RedditClone</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <!-- Our stylesheet -->
  <link rel="stylesheet"
  href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.2.4/semantic.min.js" />
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

However, since we're using the `angular-cli` tool, we can utilize it's template generation to add a stylesheet to our application. We can direct the angular-cli tool to add a stylesheet link through configuring the angular cli.

In order to configure our angular-cli webserver, we'll open the generated `angular-cli.json` in the root folder where our application is set.

The `angular-cli.json` file is used by the cli to determine which files to load where and when. To add a style, we'll update the `styles` key in the first entry in the `apps` json key.

Let's install the `semantic-ui-css` package from npm into our application. We'll also want to install jQuery as well as it's a dependency for semantic-ui.

> We'll use the semantic-ui framework throughout our application. The framework is similar to bootstrap web framework, but much more verbose.

```
npm install --save semantic-ui-css jquery
```

Let's update our `styles` key in the `angular-cli.json` file to include the `semantic.css` file from the npm module:

```
{
  "apps": [
    {
      "root": "src",
      // ...
      "styles": [
        "../node_modules/semantic-ui-css/semantic.css",
        "styles.css"
      ]
    }
  ],
  // ...
}
```

Since we'll also be using the semantic-ui JS library, let's also add the required JavaScript files to our `angular-cli.json` by updating the `scripts` key. We'll need to list the `jquery.js` dependency before we add the `semantic.js` library as `jquery` is a dependency of `semantic.js`:

```
{
  "apps": [
    {
      "root": "src",
      // ...
      "scripts": [
        "../node_modules/jquery/dist/jquery.js",
        "../node_modules/semantic-ui-css/semantic.js"
```

```
9         ],
10      }
11    ],
12    // ...
13  }
```

When we start our server up again (or create a production deployment), our styles will be added directly to the html delivered to the browser with the request.

4.1.2 Adding styles at the global level from a component

Adding styles in Angular 2 is straight-forward on a component level, but it can be a bit tricky to understand why these styles don't cascade naturally down to child components.

Angular 2 treats components as their own element on the page and any styles associated with the component will be generated for that component and not cascade down to their children. This approach solves one of the biggest painpoints of writing CSS where global styles can clobber child styles, where it can often be trouble to find the root of one style.

Sometimes, however we do want to write a style that cascades from one parent component to it's child component. In order to accomplish this, we need to tell Angular we want to change the way it handles the styles for that component. For this reason, Angular 2 allows us to manipulate this *view encapsulation* property.

4.2 View encapsulation

In Angular 2, we get to design components that define their own view without affecting the rest of the application's components. However, sometimes we want to affect the rest of our components. View Encapsulation is the term that defines Angular's behavior of how it treats view elements belonging to one component over the next.

There are 3 modes of encapsulation that we can use to set our component's view encapsulation on the `ViewEncapsulation` object from angular core:

```
1  import { ViewEncapsulation } from '@angular/core';
```

- `ViewEncapsulation.Emulated` (the default encapsulation if we don't otherwise define one) emulates the behavior of the Shadow DOM preprocessing to handle scoping CSS styles.
- `ViewEncapsulation.Native` uses the browser's native Shadow DOM implementation to attach the component's host element, which includes it's styles.
- `ViewEncapsulation.None` disables the view encapsulation feature in Angular 2.

Defining the view encapsulation definition is easy using the component definition object key of `encapsulation`.

For instance, in our previous example, let's say we have a component which represens a document reader to show document and we want to ensure all links to urls include an underline of all `<a />` tags.

```
@Component({
  templateUrl: 'document-reader.component.html',
  styleUrls: ['./document-reader.component.css'],
  // Optional as the component defines this for us if we don't set an encapsulation prc
  encapsulation: ViewEncapsulation.Emulated
})
class DocumentReader {
  // ...
}
```

We can set the `encapsulation` property to `None`, which tells Angular to add the styles to the `<head />` of the document:

```
@Component({
  templateUrl: 'document-reader.component.html',
  styleUrls: ['./document-reader.component.css'],
  encapsulation: ViewEncapsulation.None
})
class DocumentReader {
  // ...
}
```

If we want to use the browser's native Shadow DOM, we can set the `DocumentReader` Component's encapsulation as follows:

```
@Component({
  templateUrl: 'document-reader.component.html',
  styleUrls: ['./document-reader.component.css'],
  encapsulation: ViewEncapsulation.Native
})
class DocumentReader {
  // ...
}
```

For most cases, the `ViewEncapsulation.Emulated` is just right as the Shadow DOM isn't fully implemented in most browsers and Emulated was written to help solve this problem.

4.3 Local styling

As we just discussed with *view encapsulation*, we can add styles to a component to a component and it naturally will only apply to the element we associated them with. When we write a CSS style on a component it will naturally *only apply to the component we associate it with*.

There are multiple ways we can add styles to a component. In using the angular-cli, we've worked through at least one method of adding custom styles to our Components, thanks to the angular-cli generator.

We can add styles directly with the `styles` component object key directly as CSS code:

```
@Component({
  templateUrl: 'list-item.component.html',
  styles: [
    '.item { border: 1px solid red; }'
  ]
})
class ListItem {}
```

By applying our styles directly to the component, all the selectors here will only be applied to this component (via *view encapsulation*).

It's not always convenient to write our styles directly in the code (especially where syntax highlighting is misaligned). We can *also* apply external stylesheets using the `styleUrls`:

```
@Component({
  templateUrl: 'list-item.component.html',
  styleUrls: ['list-item.component.css']
})
class ListItem {}
```

4.3.1 Local styling

It's also possible to directly embed styles in our template markup. Just as though we are adding styles in our template (we are), we can set them in the view style tag:

```
// list-item.component.html
<style>
.item { border: 1px solid blue; }
</style>
<ul>
  <li *ngFor='let i of items'>
    <!-- ... -->
  </li>
</ul>
```

There are a lot of other options we have available to us to handle styles, thanks to the view encapsulation feature.

5 Making our app production ready

We've only been using a single file up through this point in our app. Although this is great for learning, we can do much better and make it easier to extend by using the angular cli generator to create new elements of our application.

Let's divide up our components into multiple files. Let's look at the components we already have and copy and paste our original implementation in the single file.

The components we have thus far:

- Sidebar
- Article
- ArticleList
- AppComponent

Since we don't need to create a new component for the `AppComponent` (we already have been working in the `app.component.ts` file), let's create the 3 components we don't have files for using the `angular-cli` generate command:

```
ng generate component sidebar
```

The angular-cli accepts a command as the first argument. Here, we're calling the `generate` command. Each argument after is defined by the action we are taking. In this case, we're using the `generate` command to create a new component called `sidebar`.

Running this sidebar function will create a few files for us in the `src/app/sidebar` directory of our app:

Let's repeat this process for the next two components:

```
ng generate component article
ng generate component article_list
```

With our new files generated, let's pull out all of our functionality from the `src/app/app.component.ts` file into each of the new files.

6 Fetching from a server

The `http` service from Angular 2 returns an observable by default. When we turned our app to using a promise instead of an observable, we were creating a

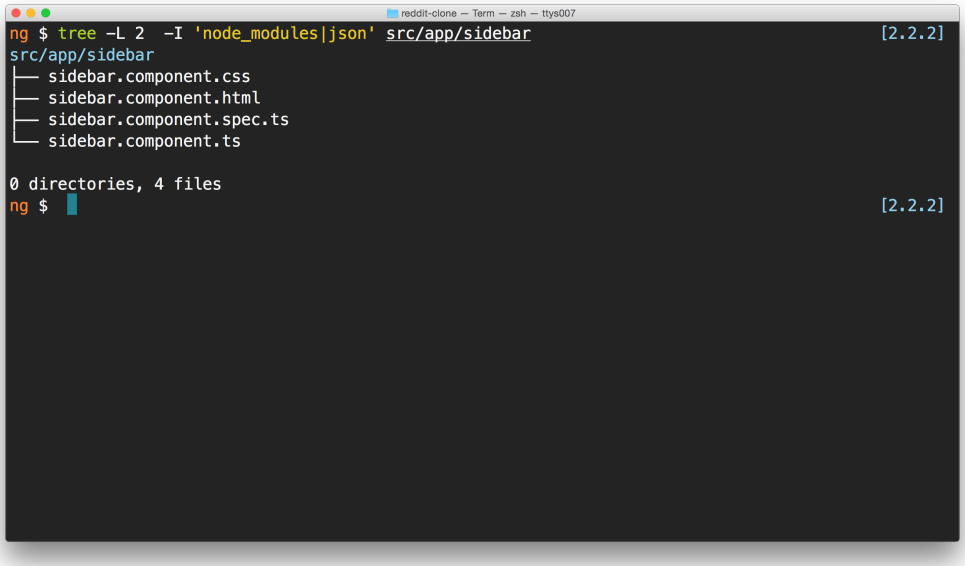

Figure 1:

`Promise` object to handle it's result from an observable sequence. We can scrap that approach entirely and return an observable sequence instead.

As now we're going to use a live server to handle this operation, let's update our approach and take a quick detour to talk about environment handling in our Angular 2 app.

When we set up our application to use live servers, we often want to deal with different environment settings for different deployments. For instance, we might want to use a development server for our development, a mock-testing one for test environments, and a completely different one in production. In the twelve-factor app approach, we want to use different variables for different environments. This is set up automatically for us by the angular-cli.

Let's use a free news api service called the News API to query for different news items from different sources. First, let's sign up for an API KEY at https://newsapi.org.

This will bring us to a screen with our API key from the News api site. Grab this key and hold on to it. We'll use this in a moment.

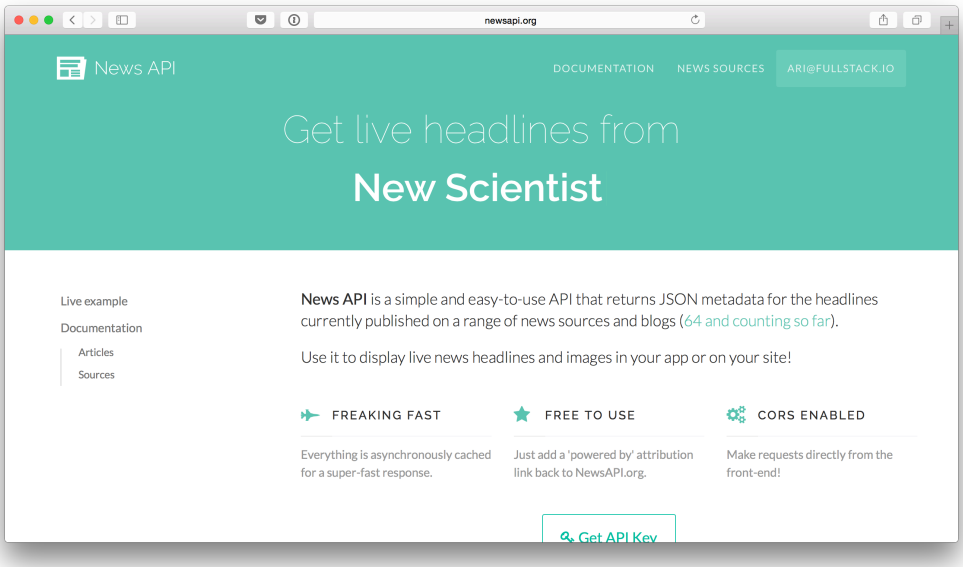

Figure 2:

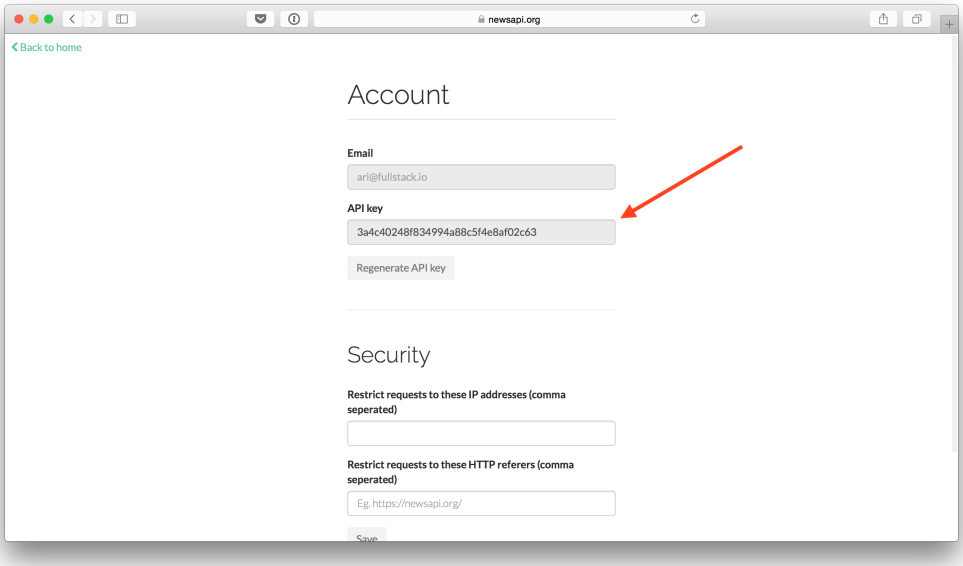

Figure 3:

In our application source, we'll see we have a directory created by the angular-cli at `src/environments`. This `src/environments` is a special folder that is merged and replaced at run-time by the cli. We can place different environment variables in the exported object from the `src/environments/environment.ts` file to use at run-time.

In the production environment, on the other hand we can place our variables for production applications in the `src/environments/environment.prod.ts` file to be used instead.

Let's place our `newsApiKey` we just gathered from the environment in this exported object. Personally, I like to store the domain we'll request as well in the environment file as we can define different servers we want later by defining them here.

Let's update the `src/environments/environment.ts` file to include these two values:

```
export const environment = {
  production: false,

  newsApiKey: 'YOUR KEY GOES HERE',
  newsDomain: 'https://newsapi.org'
};
```

When we modify the `environments` file, we will need to restart our server, so back in the terminal, let's kill the server and restart it.

6.1 Introduction to Observables

> The entire code library for this section can be found at the codebin available at: http://jsbin.com/xomoqa/28/edit?js,output

On our journey thus far, we've built our project API fetching using promises, but we haven't yet touched on the drawbacks of using them.

Taking a step out of our project to isolate using Observables, let's build an application to make requests to the github api to suggest github repositories based on a search term.

6.1.1 Promises

Promises address the issue of handling asynchronous actions in a logical, synchronous-looking way. We create a `Promise` object instance which wraps a function that gets executed when the asynchronous action has resolved or has been rejected.

In code, this process looks like this:

```
1  const fetchRepos = (term) => fetch(
2    `https://api.github.com/search/repositories?q=${term}`
3  );
4
5  const updateList = (items) => {
6    // ...
7  };
8
9  let promise = fetchRepos('fullstackio')
10 promise
11   .then(response => response.json())
12   .then(obj => obj.items)
13   .then(updateList);
```

The `fetchRepos()` method executes an asynchronous method using the `fetch()` API and returns back a `Promise` object that exposes a few methods publicly, which we'll use most often:

- `then()`
- `catch()`

In this previous example, we're creating a promise chain which represents multiple promises *linked* to each other. Once the first promise in the chain is resolved / rejected, the next function defined in `then()` will get executed, which will, in turn call the next one and so on and so forth.

One feature *and* drawback to depending upon promises entirely is that they are guaranteed to resolve once and only once. We *cannot* reuse `Promises` and instead are required to create entirely new `Promise` objects.

Although it's not difficult to create new `Promise` objects, they don't necessarily fulfill every situation. For instance, setting up multiple actions for a single data update. What about setting up multiple chains of updates to update a new value or set of values?

6.1.2 EventEmitter

One way we can handle dynamically updating events beyond promises are to use a process using the `EventEmitter`. The `EventEmitter` class provides a messaging bus we can both fire custom events and subscribe to the events to associate functionality when they are fired. The browser already emits events when the user is interacting with it, the `EventEmitter` is one way we can create our own custom events.

For instance, let's say that we want to have two functions run after our user types in the `<input />` element is typed, perhaps to make a suggested list of topics in another section of the page. Using the `EventEmitter`, we can emit a custom event from the page.

Elsewhere in our code, perhaps in a different component of the page, we can listen for this event and handle it however we see fit. Let's see this in code.

Let's first emit a custom event when our user types in the `<input />` element. This might look like:

```
// create our EventEmitter instance
const emitter = new EventEmitter();

// emit the event with the title of `input`
input.addEventListener('keyup', (evt) => {
  emitter.emit('input', evt);
});
```

Now we can set up an action on the `input` event. Let's say we want to log out in the console (as a trivial example) the character value of the `<input />` field:

```
emitter.on('input', (evt) => console.log(evt));
```

Somehwere else we might want to add an event that actually makes the search. We can add a second action to handle that elsewhere in our code:

```
emitter.on('input', (evt) => {
  const term = evt.target.value;
  fetchRepos(term)
    .then(resp => resp.json())
    .then(obj => obj.items)
    .then(items => {
      const sorted = items
        .sort((a, b) => b.stargazers_count - a.stargazers_count);
      return sorted;
    })
    .then(updateList);
})
```

Great! Now we've solved one issue where we can update other parts of our page from the same action which is completely unconnected to another part of the page. However, we are tied to know the specific name of the custom event. There must be another way, right?

However, what if we want to include other, more complex interaction with custom filtering or mapping along the data?

6.2 Observables

Observables offer a different way to approach dealing with custom asynchronous events. When we talk about Observables, we're talking about objects that have the abiity to track changes on themselves, but also to notify and react to changes in other parts of the system.

With RxJS (the Reactive EXtensions library we'll be using), we have *tons* of operators we can use to manipulate/filter/query/modify data as it changes in our system.

Not only can we create an object that we can keep track of changes

6.3 Back to requesting

Back in our `src/app/article.service.ts`, let's use these new variables in a new version of the `getArticles()`. Instead of returning an array of promises, let's return an observable sequence of articles. Let's also import the `Observable` object from `rxjs`:

```
// ...
import { Observable } from 'rxjs/Observable';

@Injectable()
export class ArticleService {
  // ...
  getArticles(): Observable<Article[]> {
    // ...
  }
}
```

This new `getArticles()` function will return an array of Articles. We'll need to grab the domain from the environment. In order to use the environment file, we'll need to import it. We can import this file just like any other typescript file in our app by requesting it using `import`:

```
import { environment } from '../environments/environment';
```

> This `environment` file will be *automatically* updated in production for us, so we don't need to deal with defining *which* environment we want… the angular cli manages this process for us.

Now we can grab the `newsDomain` value from the `environment` object:

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  // ...
}
```

We'll also need to update our request to return the `http` response. This requires us to *inject* the http response into our service, so Angular knows we want this as a dependency. Let's update the constructor to accept the `http` as a dependency:

```
import { Http } from '@angular/http';

@Injectable()
export class ArticleService {
  constructor(private http: Http) {}
  // ...
}
```

> In TypeScript, we can declare variables passed in the `constructor()` function as `private/public`, which unlike native JavaScript is akin to defining a variable on the instance of the class.
>
> For instance, the previous code above is equivalent to the following code:
>
> ```
> export class ArticleService {
>   private http: Http;
>   constructor(http:Http) {
>     this.http = http;
>   }
> }
> ```

Now we have access to the `http` variable from inside our `ArticleService` class, let's return the observable sequence returned by `http`.

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  return this.http.get(`${baseUrl}/v1/articles`);
}
```

If we run this request through in the browser, we'll see that it doesn't quite work out of the box. We need to send a few parameters to the server. We can send

params with an `http` request using the `URLSearchParams` object. The `URLSearchParams` object, exported by the `@angular/http` package is a key-value store that we can pass through in the second argument (a configuration object).

Let's create an instance of this object in the `getArticles()` method after importing it from the `@angular/http` package:

```
import { Http, URLSearchParams } from '@angular/http';

@Injectable()
export class ArticleService {
  getArticles(): Observable<Article[]> {
    let baseUrl: string = environment.newsDomain;
    let params = new URLSearchParams();

    return this.http.get(`${baseUrl}/v1/articles`, {
      search: params
    });
  }
}
```

Now we need to send a few parameters required by the News API:

- `apiKey` - our api key
- `source` - The *source* we want to request from (the News API allows us to request from multiple services).

Let's update our request to make a request to the `reddit-r-all` endpoint first (we'll come back and "un-hardcode" this later):

```
import { Http, URLSearchParams } from '@angular/http';

@Injectable()
export class ArticleService {
  getArticles(): Observable<Article[]> {
    let baseUrl: string = environment.newsDomain;
    let params = new URLSearchParams();
```

```
    params.set('apiKey', environment.newsApiKey);
    params.set('source', 'reddit-r-all');

    return this.http.get(`${baseUrl}/v1/articles`, {
      search: params
    });
  }
}
```

Running this in the browser, we'll see we have a big error.

There are a few errors contained in this error message. The first is that the `articles` list is no longer a promise, so we'll need to update this in our `ArticleListComponent`. Rather than unwrapping the promise, we can *listen* for changes… another way of saying this is we can `subscribe` to events on the sequence. When the response sequence changes, we can run a function on success or one on error.

Let's update the `ArticleListComponent.getArticles()` method to use `subscribe` to change the local articles listing:

```
export ArticleListComponent implements OnInit {
  articles: Article[];

  // ...
  getArticles(): void {
    this.articleService
      .getArticles()
      .subscribe(
        // success function
        articles => this.articles = articles,
        // error function
        error => console.error('Error fetching articles', error),
        // always executes regardless of result status
        () => console.log('Done getting Articles')
      );
  }
};
```

Now, the second error we have to deal with is that our response from the `this.http.get()` method returns back an Observable of `Response` object, instead of an Observable of `Articles` object.

Since the `ArticleService` is responsible for handling fetching Article objects, it makes sense for this object to be responsible for returning `Article` instances, rather than the component being responsible.

Luckily we can set up actions on the observable sequences to execute on the sequence of actions using the `.map()` function. To use the `map()` operator on a sequence, we'll need to import it from the `rxjs` package.

We'll use this `map` operator to first convert the response to `json` when it comes back from the raw `Request` sequence returned by the `Http` service:

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  let params = new URLSearchParams();
  params.set('apiKey', environment.newsApiKey);
  params.set('source', 'reddit-r-all');

  return this.http.get(`${baseUrl}/v1/articles`, {
    search: params
  })
    // convert the response to json
    .map(response => response.json());
}
```

The raw http request will be returned a JSON object with an array of articles. Since we're only really interested in the response's articles array, we can use the `map()` function again to return only the articles in the result:

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  let params = new URLSearchParams();
  params.set('apiKey', environment.newsApiKey);
  params.set('source', 'reddit-r-all');

  return this.http.get(`${baseUrl}/v1/articles`, {
```

```
    search: params
  })
    // convert the response to json
    .map(response => response.json())
    // return only the articles array
    .map(json => json.articles)
}
```

Finally, we'll want to return a result of Article objects rather than json objects. We can iterate over the articles returned by the sequence and create a new `Article` instance for the new details:

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  let params = new URLSearchParams();
  params.set('apiKey', environment.newsApiKey);
  params.set('source', 'reddit-r-all');

  return this.http.get(`${baseUrl}/v1/articles`, {
    search: params
  })
    // convert the response to json
    .map(response => response.json())
    // return only the articles array
    .map(json => json.articles)
    .map(articles => articles.map(article => {
      return new Article(article.title, article.url);
    }))
}
```

Our view will be updated and work again after this switch.

Before we move on from this example, it's somewhat annoying to have to pass in all of our values directly into the `Article` object. For instance, we now have a `publishedAt` and `description` keys we'll want to store in the `Article` object thanks to using results from the remote server.

Wouldn't it be nice to map our json keys directly to the `Article` object? We can *almost* make that a reality.

Let's dive into the `interface` idea from TypeScript. Basically, we can pass in an object that refers to an Article from the JSON and build an `Article` instance object from this interface.

Let's update our `Article` model object to use an interface to build an Article object from the JSON.

In the `src/app/article.ts` model file, let's add an interface we'll call `ArticleJSON`.

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

interface ArticleJSON {
  title: string;
  url: string;
  votes: number;
  publishedAt: string;
  description: string;
};
```

In this interface, we'll keep a list of the variables we want to track in the JSON object. We can then create an instance of the `ArticleJSON` object and map the values against the list in a function. Let's create a `static` function on the `Article` class we'll define as `fromJSON()`.

```
interface ArticleJSON {
  // ...
}

@Injectable()
export class Article {
  static fromJSON(json: ArticleJSON): Article {
    // ...
  }
}
```

The `fromJSON()` method will accept a `json` object and return an `Article` instance. Pretty sweet, eh? Let's actually make this thing work though. We will want to

create an instance of the `Article` class, but instead of using the `constructor()`, which expects our object properties to be mapped directly, we can use the native `Object.create()` method. This ensures we don't trip over the usage of the function in compilation:

```
interface ArticleJSON {
  // ...
}

@Injectable()
export class Article {
  static fromJSON(json: ArticleJSON): Article {
    // similar to new Article(), without the same constraints
    // imposed by TypeScript
    let article = Object.create(Article.prototype);
  }
}
```

The `article` object doesn't yet have the values mapped to it's keys from the `json` object. Using `Object.assign()`, we can map these keys directly to the object:

```
interface ArticleJSON {
  // ...
}

@Injectable()
export class Article {
  static fromJSON(json: ArticleJSON): Article {
    let article = Object.create(Article.prototype);
    return Object.assign(article, json, {
      votes: 0 // extra values not contained in the JSON
    });
  }
}
```

The `Object.assign()` method accepts a list of objects that it will merge into our `article` instance. We can use this last argument as a way to add other fields to our `Article` instance, such as dates that need conversion, for example.

Let's store the `publishedAt` date provided by the News Api. As we're not using the constructor in creating our `Article`, we'll need to set this parameter on the object itself.

```
interface ArticleJSON {
  // ...
}

@Injectable()
export class Article {
  public publishedAt: Date;

  static fromJSON(json: ArticleJSON): Article {
    let article = Object.create(Article.prototype);
    return Object.assign(article, json, {
      votes: 0
    });
  }
}
```

Since we want the `publishedAt` attribute to be a Date object instead of a string, we can parse it in the last argument of our `Object.assign()` call:

```
interface ArticleJSON {
  // ...
}

@Injectable()
export class Article {
  public publishedAt: Date;

  static fromJSON(json: ArticleJSON): Article {
    let article = Object.create(Article.prototype);
    return Object.assign(article, json, {
      votes: 0 ,
      publishedAt: json.publishedAt ?
                    new Date(json.publishedAt) :
```

```
                    new Date()
    });
  }
}
```

Now we can update our `ArticleService` to use this `fromJSON()` method instead of passing the specific details we want to track. In the `src/app/article.service.ts` file, let's update our `getArticles()` function to use the interface:

```
getArticles(): Observable<Article[]> {
  let baseUrl: string = environment.newsDomain;
  let params = new URLSearchParams();
  params.set('apiKey', environment.newsApiKey);
  params.set('source', 'reddit-r-all');

  return this.http.get(`${baseUrl}/v1/articles`, {
    search: params
  })
    // convert the response to json
    .map(response => response.json())
    // return only the articles array
    .map(json => json.articles)
    .map(articles => articles.map(article => Article.fromJSON(article)));
}
```

Our view still works and we've decoupled the constructor interface from the `Article` model, making it more flexible in the future.

In any case, back in the view… notice that we're returning the observable this time instead of an array. We can't quite sort the articles in the same way as we previously did with static variables.

There are multiple methods for dealing with sorting, but we'll use one powerful way for handling sorting. Let's flex our Observable muscles a bit further and provide sorting through observable subscriptions.

In order to do this, we'll need to change up our observable structure again. Rather than just pass an observable and resolve it immediately into an array of `Article`

objects in the component, we can expose an observable from our service to the outside world.

In order to do this, we'll need to use the `BehaviorSubject` object exported by rxjs. The `BehaviorSubject` object inherits from the `Observable` and `Observer` objects in rxjs, which basicaly means we can subscribe to an observable and do some things with the result. We can *also* subscribe to the `BehaviorSubject` object to set up a chain of observables.

The `BehaviorSubject` itself is actually a layer atop the `Subject` object from rxjs that provides access to the last value to subscribers. Because stream messages are pubished immediately, a new subscriber to a stream might miss values published in the stream. Using the `BehaviorSubject`, we can avoid this race condition.

Sounds intimidating, but in usage it's a lot easier to grapple with. Let's move away from the `getArticles()` function and instead set up a pipeline of observables.

First, let's import the `BehaviorSubject` object from `rxjs` in our `src/article.service.ts` file:

```
// src/article.service.ts
import { Observable, BehaviorSubject } from 'rxjs';
```

We'll expose a member property on the `ArticleService` of `articles` to be a list of articles. We'll set these articles as a `BehaviorSubject` in the `ArticleService`:

```
@Injectable()
export class ArticleService {
  public articles: BehaviorSubject<Article[]> =
    new BehaviorSubject([]);
}
```

At this point, let's actually look at one issue that you're likely to encounter if we keep the implementation as it is. Right now, we're exposing the `BehaviorSubject` directly to the app. This is all well and fine for smaller apps, but can become a big headache when our apps get bigger and we have more subscribers subscribing to these Subject instances.

If any object in our app manipulates the `articles` BehaviorSubject, it can be hard to track down where the change occurs. Instead, we want to provide an API for manipulating the streams through the `ArticleService` so any member of our application that wants to update the values in the stream will need to call through this api.

Instead of providing the `articles` as a public interface, let's move the `BehaviorSubject` to be a private variable and expose an `Observable` publicly.

```
@Injectable()
export class ArticleService {
  private _articles: BehaviorSubject<any> =
    new BehaviorSubject<any>([]);

  public articles: Observable<Article[]> =
    this._articles.asObservable();
}
```

With this `articles` property on the `ArticleService`, we can subscribe to changes outside of the `ArticleService` to this observable. Even though it's not hooked up yet, let's modify the consuming of this observable in the `src/app/articles/article-list/article-list.component.ts` so when we do hook it up, we'll visually see the result.

Let's add a parameter we'll call `articleSubscription` of the `Observable` type:

```
export class ArticleListComponent implements OnInit {
  // ...
  articleSubscription: Observable<Article[]>;

  constructor(
    private articleService: ArticleService
  ) => {
    this.articleSubscription =
      this.articleService.articles;
  }
}
```

The template of the `ArticleListComponent` won't pull from the `articles` object any more, instead we'll iterate over this observable. Let's modify the template to reflect this new behavior:

```
<div class="ui main text container">
  <div class="ui grid posts">
    <app-article
      *ngFor="let article of articlesSubscription"
      [article]="article"></app-article>
  </div>
</div>
```

Since the `articlesSubscription` is actually an asynchronous object, not a JavaScript object and Angular won't know what to do with the object, we'll need to tell Angular that the result it is expecting to render is asynchronous. We can do this with a `pipe`.

6.3.1 Pipes

If you're familiar with Angular 1, `pipes` are like filters. They allow us to modify the result of an expression inside the template. There are all sorts of pipes available from Angular and even a pretty easy way to create them.

The `AsyncPipe` will tell Angular that it should expect an asynchronous value rather than a resolved one. They are pretty easy to use. Adding a pipe character | and then the name of the pipe we want to use. In this case, the name we'll use is `async`, like so:

```
<div class="ui main text container">
  <div class="ui grid posts">
    <app-article
      *ngFor="let article of articlesSubscription | async"
      [article]="article"></app-article>
  </div>
</div>
```

Back to the `ArticleService`, we'll want to modify the `articlesSubscription` when we get new articles from our web api. Let's modify our `getArticles()` function to update the article list in the `ArticleService`.

For simplicity, let's move the fetching from the API into it's own function and we'll use the result of the http request as an internal `Observable`:

```
@Injectable()
export class ArticleService {
  public articles: Observable<Article[]> =
    this._articles.asObservable();

  // ...
  getArticles() {
    // ...
  }

  // The http request
  _fetchArticles(): Observable<any> {
    let baseUrl: string = environment.newsDomain;
    let params = new URLSearchParams();
    params.set('apiKey', environment.newsApiKey);
    params.set('source', 'reddit-r-all');

    return this.http.get(`${baseUrl}/v1/articles`, {
      search: params
    })
    .map(response => response.json())
    .map(json => json.articles);
  }
}
```

Now in our `getArticles()` function we can subscribe to the changes from the `_fetchArticles()` function as an observable. Notice that this is no longer returning an array of articles, but an `Observable` object instead. Let's update the `getArticles()` function to subscribe to the result of the function and *push* a new value onto the `articles` property:

```
@Injectable()
export class ArticleService {
  public articles: Observable<Article[]> =
    this._articles.asObservable();

  // ...
  getArticles() {
    this._fetchArticles()
        .subscribe((result) => {
          // the result is a list of articles from the
          // http request as an observable stream
          const articles =
              res.map(article => Article.fromJSON(article));
          // Next "push" the new articles onto the
          // `_articles` Observable
          this._articles.next(articles);
        });
  }
}
```

Believe it or not, that's it. When we call the `getArticles()` function, it will create a new subscribable object, subscribe to it's results, and then push them onto the `articles` subject property as the 'next' results.

In our browser, the article stream loads again.

Okay, but why all this work to get something we had working in the first place? Why make this more complex?

It turns out that using this single `Observable` abstraction makes more advanced patterns pretty easy to build. In addition, it simplifies the mental model of our application state as well as provides performance improvements. But enough hand-wavy stuff; let's use this new model to add functionality to our `ArticleService`.

We started this journey down `Observable` functionality looking to order our application data. Let's implement sort in our `ArticleService` for our articles. We can expose another `Observable` property that we can call `orderedArticles`.

```
@Injectable()
export class ArticleService {
```

```
  public orderedArticles: Observable<Article[]>;
  public articles: Observable<Article[]> =
    this._articles.asObservable();
}
```

The `orderedArticles` Observable object can listen (subscribe) on the `_articles` observable for any changes. When changes do occur on the `_articles` property, we can sort the articles how we would like.

Let's implement sorting on the `publishedAt` date of our articles using this strategy:

```
@Injectable()
export class ArticleService {
  public orderedArticles: Observable<Article[]>;

  constructor(private http:Http) {
    // We can connect our subscription in the
    // constructor
    this.orderedArticles = this._articles
        .map((articles: Article[]) => {
          return articles
            .sort((a: Article, b: Article) => {
              return b.publishedAt.getTime() -
                     a.publishedAt.getTime();
            });
        });
  }
}
```

Just like that, we now have ordering with our `orderedArticles` subscription. By changing the subscription we listen on in our `ArticleListComponent`, we can get sorting/ordering directly in our app.

```
export class ArticleListComponent implements OnInit {
  // ...
  articleSubscription: Observable<Article[]>;
```

```
4
5   constructor(
6     private articleService: ArticleService
7   ) => {
8     this.articleSubscription =
9       this.articleService.orderedArticles;
10  }
11 }
```

At this point, however we don't have a way to fetch new articles. If we want to *refresh* the app's list of articles, we have to update the articles observable in the `ArticleService`. We can use another `Observable` (don't worry, they are lightweight) to hold on to operations in this Observable.

Let's create a `Subject` in the `ArticleService` we'll call `_refreshSubject`. This `_refreshSubject` property will be a way for us to request a fetch from the API. Anytime this value is updated, we'll run the API request to get new data.

```
1  export class ArticleService {
2    // ...
3    private _refreshSubject: BehaviorSubject<string> = new
4      BehaviorSubject<string>('reddit-r-all');
5  }
```

We can use this `_refreshSubject` variable to populate the article subscription.

In order to tell the `ArticleService` we want to make a new fetch to articles, we'll need:

1. Bind our `_refreshSubject` subject to call the `getArticles()` method everytime it changes.
2. Add a method to update the `_refreshSubject`.

Since we can subscribe on the `Subject` instance variable, step one can be accomplished by calling the `getArticles()` method within a subscription. We'll set this up in the `constructor()` method of the `ArticleService`:

```
export class ArticleService {
  // ...
  constructor(
    private http: Http
  ) {
    this._refreshSubject
        .subscribe(this.getArticles.bind(this));
  }
}
```

6.3.2 Why bind?

We're using the `bind()` method within the subscription to make sure that the `getArticles()` method is called on the instance of the `ArticleService` class.

The `bind()` method ensures the `this` referenced in the method is the instance of the `ArticleService` class.

Second, let's add an `updateArticles()` method on the `ArticleService` class to push a new `sourceKey` on the `getArticles` method:

```
export class ArticleService {
  // ...
  public updateArticles(sourceKey:string): void {
    this._refreshSubject.next(sourceKey);
  }
}
```

When a string gets pushed on the `_refreshSubject`, the function will be called and the `articles` will be updated.

We want to make sure that we provide a declarative way for the user to fetch new Articles. Let's create a new function we'll call `updateArticles()` to push operations into it.

```
@Injectable()
export class ArticleService {
```

```
3   // add a new string to the _refreshSubject
4   updateArticles(source) {
5     this._refreshSubject.next(source);
6   }
7 }
```

When our `ArticleListComponent` calls `updateArticles()`, the `getArticles()` function will be called and the `_articles` map will be automatically updated through the power of streams.

Let's update the `ArticleListComponent` to call the `updateArticles()` function. Let's pass it a string of `reddit-r-all`, which is one of the news sources the `newsApi` knows about.

```
1 export class ArticleListComponent implements OnInit {
2   // ...
3   ngOnInit(): void {
4     this.articleService
5         .updateArticles('reddit-r-all');
6   }
7 }
```

Awesome! Now our app still works in the browser and we're using Observables all the way down!

The way we have our app set up now, we can support multiple news sources. The newsApi endpoint supports multiple sources and has a list of all available sources at an endpoint of `/v1/sources`. Now that we know how to make an async request for asynchronous data from our `ArticleService` object, let's add another one to fetch the whole list.

Since we'll be using a similar method for making API requests to our backend, we let's update the `_fetchArticles()` method we previously wrote to use a new method we'll create called `_makeHttpRequest()`. Abstracting the `http` part, we can update our `ArticleService` to look like:

```
1 @Injectable()
2 export class ArticleService {
3   // ...
```

```
4
5     _fetchArticles(): Observable<any> {
6       return this._makeHttpRequest('/v1/articles', 'reddit-r-all')
7                   .map(json => json.articles);
8     }
9
10    // moved the http responsibility to a common method that accepts
11    // a path for request
12    _makeHttpRequest(path: string, sourceKey?: string): Observable<any> {
13      let baseUrl: string = environment.newsDomain;
14      let params = new URLSearchParams();
15      params.set('apiKey', environment.newsApiKey);
16      if (sourceKey && sourceKey !== '') {
17        params.set('source', sourceKey);
18      }
19
20      return this.http.get(`${baseUrl}${path}`, {
21        search: params
22      }).map(response => response.json());
23    }
24  }
```

> sourceKey?
>
> Not to worry, we'll talk about what that sourceKey thing is doing there shortly. For the time being, pretend it doesn't exist.

Switching our code to work in this way allows us to not require duplicate code to make the request to fetch sources. Let's add the `_fetchSources()` method now to fetch sources:

```
1   @Injectable()
2   export class ArticleService {
3     // ...
4     _fetchSources(): Observable<any> {
5       return this._makeHttpRequest('/v1/sources')
6                   .map(json => json.sources);
7     }
8   }
```

As we did previously, let's create a private `_sources` BehaviorSubject variable to hold on to the latest value of the fetched sources and create a public sources Observable object to use from the outside:

```
@Injectable()
export class ArticleService {
  private _sources: BehaviorSubject<any> =
    new BehaviorSubject<any>([]);

  public sources: Observable<any> = this._sources.asObservable();
  // ...
}
```

Let's expose an API function on the `ArticleService` object to fetch the sources from the backend we'll call `getSources()`. We'll implement this function in a near-exact method as the `getArticles()` function. The difference being we won't create a custom class for this source, but instead pass the list directly as a json object.

Since we'll only want to use a list with values, we'll also filter out any responses that have zero sources. We'll never expect the list to be returned empty, but this is one usage for the `filter()` function on an observable, as an example.

```
@Injectable()
export class ArticleService {
  // ...

  getSources(): void {
    this._fetchSources()
        .filter(list => list.length > 0)
        .subscribe(this._sources);
  }
}
```

Now let's show the list of possible sources in our view. Let's add another observable in our `ArticleListComponent` to subscribe to from our `ArticleService`. Let's call it `sourcesSubscription`:

```
export class ArticleListComponent implements OnInit {
  // ...
  private sourcesSubscription: Observable<string>;

  constructor(
    private articleService: ArticleService
  ) => {
    // ...
    this.sourcesSubscription = this.articleService.sources;
  }
}
```

Finally, let's call the `getSources()` function when the `ArticleListComponent` component boots up, so we fetch all the available sources immediately:

```
export class ArticleListComponent implements OnInit {
  // ...
  ngOnInit(): void {
    this.articleService.getSources();
  }
}
```

In our view, we can iterate over the sources just as though we had done it previously, using the `async` pipe. We'll also place this in a sidebar component that we'll create soon, so for the time being, let's get this listing up and running. In our view of `src/app/articles/article-listing/article-listing.component.html`, let's update the html to iterate over our list. This should look familiar as it will be using the same method we used to list the articles.

```
<ul class="ui horizontal list">
  <li
    *ngFor="let source of sourcesSubscription | async"
    class="item">
      <a class="ui small button">{{ source['name'] }}</a>
  </li>
</ul>
```

Now we have a list of links of all the different sources available via the newsApi. However, this is pretty boring, showing a bunch of links that don't do very much. Let's make these links pull from the different sources. In order to do that, however… we'll need to talk about routing.

7 Routing

It's pretty rare that we'll ever have a single-page application with only one view. Additionally, we'll usually want to provide the ability for our users to copy+paste links from the browser url bar content and share the same content with others. Client-side routing allows us to enable these features and Angular 2 comes with a router implementation that we can take advantage of out of the box.

From a high-level, the way client-side routing works is by using the app path/state to change the content for one part of a Component. For the *main* module, this is the location on the page where we want Angular to switch the content dynamically.

We won't go into great detail about all the different routing options available to us, as the materials that come with this guide are far more complete that we can cover, but let's get started adding routes to our app.

In order to add routes to our application, we'll need to create a routing object. A routing object lists all the available routes for a particular root component in our DOM tree. Let's create the route object in a file we'll call `src/app/app.routes.ts`. Let's create this file:

```
1 touch src/app/app.routes.ts
```

In order to create these routes for our app component, we'll need to import the `RouterModule` from the `@angular/router` package. We'll also pull in the `Routes` type from the package as well.

```
1 import { Routes, RouterModule } from '@angular/router;
```

We'll create routes in an object defined as `Routes` which is essentially a list of route description objects. Without going into too much detail about the available

properties on these route description objects, they describe the path a particular route will be active, which component should show in the view, and any actions that they should take (if any).

To start off with, let's create a route for a page we haven't created yet, the about page. In order to create a route to the about page, we'll need to have an about component. Let's use the cli to create one:

```
mkdir src/app/about
ng generate component about/about
```

> The generator can create components in a directory if given in the command. Keeping our functional components together on a per-view basis, we created the `AboutComponent` in the nested directory of `src/about` with the previous command.

We can add a route to the `/about` path by creating a route object and passing it the `AboutComponent`. In the `src/app/app.routes.ts` file, let's create the routes object:

```
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  {
    path: 'about', component: AboutComponent
  }
];

export const appRoutes = RouterModule.forRoot(routes);
```

We use the `RouterModule.forRoot()` function is a special function we'll use to pass a routing object to the `AppModule` which registers the Routing dependencies as well as the routes for the root of the application.

Let's import these `routes` in our `AppModule` and set them as an import for the `@NgModule` (in `src/app/app.module.ts`):

```
import { BrowserModule } from '@angular/platform-browser';
// ...
import { appRoutes } from './app.routes';
// ...

@NgModule({
  declarations: [
    // ...
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    appRoutes
  ],
  // ...
})
export class AppModule { }
```

In the browser, we'll see that we encounter an error.

The error `Cannot find primary outlet to load` happens because we haven't told Angular where we want to display the routed content. We'll need to add a reference to the `RouterOutlet` directive provided by Angular. Let's open up the `src/app/app.component.ts` file and add the `<router-outlet>` component:

```
<div id="wrapper" class="ui">
  <app-header></app-header>
  <router-outlet></router-outlet>
</div>
```

Back in the browser, let's navigate to the `about` path in the url and we'll see the about page content shows up.

Not very exciting yet…

Let's add a route for our article listing component. Since we'll want to know which source we are going to match on, we'll include a variable in the path.

Let's update the `src/app/app.routes.ts` file to include a second route:

```
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { ArticleListComponent } from './articles/article-list/article-list.component';

const routes: Routes = [
  {
    path: 'about', component: AboutComponent
  },
  {
    path: 'news/:sourceKey',
    component: ArticleListComponent
  }
];

export const appRoutes = RouterModule.forRoot(routes);
```

Back in the browser, let's navigate to the page and we'll see we end up with another error.

Since our browser is at the root path, Angular doesn't know how to load a component for the root url. However, if we visit the path '/news/hacker-news', we'll see our `ArticleListComponent` renders.

Let's make the default route redirect our user to a default news listing path. We can do this by adding a command of `redirectTo` in our route description object at the path of ''.

```
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { ArticleListComponent } from './articles/article-list/article-list.component';

const routes: Routes = [
  {
    path: '',
    redirectTo: 'news/reddit-r-all',
    pathMatch: 'full'
  },
  {
    path: 'about', component: AboutComponent
```

```
  },
  {
    path: 'news/:sourceKey',
    component: ArticleListComponent
  }
];

export const appRoutes = RouterModule.forRoot(routes);
```

Notice that we used a parameter we haven't talked about yet, the `pathMatch` option. Again, without going into too much detail about the `pathMatch` option, it's required when we're using a `redirectTo` option. It tells the router how to match the URL to the route path. Since we want to redirect the user whenever we end up at the `''` route, we need to set this to `full`.

We can use `prefix` if we had a *prefix* route, which we don't have here. For more details about this option, check the accompanying materials.

Back in our browser, we'll see when we go to the root url, our browser will redirect us back to the `ArticleListComponent` at `news/reddit-r-all`.

With our routing defined, let's use the routing parameter we defined (called `:sourceKey`). In order to get access to the route params, we'll need to *inject* the `ActivatedRoute` service into the `ArticleListComponent`.

```
// ...
// Let's import the ActivatedRoute service
import { ActivatedRoute } from '@angular/router';

export class ArticleListComponent implements OnInit {
  constructor(
    private articleService: ArticleService,
    private activeRoute: ActivatedRoute
  ) => {
    // ...
  }
}
```

The `activeRoute` provides all sorts of information about the current route, including the url, the data of a route, parameters, query parameters, the routing

configuration, and more.

Since we're using a routing parameter, we'll work with the `params` object provided by the `ActivatedRoute` service. The `params` provided by the `ActivatedRoute` service is exposed to us as an Observable, so we can subscribe to changes in the route _just like we did with the article and source listings.

Let's set up this subscription in the `ngOnInit()` function (better to set it up in here, rather than the `constructor()` function as it makes testing easier).

```
// ...
export class ArticleListComponent implements OnInit {
  // ...
  ngOnInit(): void {
    this.activeRoute.params.subscribe(params => {
      // the route params have changed
    });
  }
}
```

The route parameter we're interested in is the `sourceKey`, which is available on the `params` object at `params['sourceKey']`. When we retrieve this parameter, we can shift the request for articles to be fetched with the source at `sourceKey`:

```
// ...
export class ArticleListComponent implements OnInit {
  // ...
  ngOnInit(): void {
    this.activeRoute.params.subscribe(params => {
      const sourceKey = params['sourceKey'];
      // Update the articles here
      // we'll implement this in the final section
      // of this course
    });
  }
}
```

We'll need to change our `ArticleService.getArticles()` method to accept a string, which will be the source key.

```
@Injectable()
export class ArticleService {
  // ...

  _fetchArticles(sourceKey: string): Observable<any> {
    return this
             ._makeHttpRequest('/v1/articles', sourceKey)
             .map(json => json.articles);
  }

  _makeHttpRequest(path: string, sourceKey?: string): Observable<any> {
    // ...
  }
}
```

Awesome. When we load the page at `news/reddit-r-all`, we'll see the request to the newsApi goes out with the dynamic source of the `:sourceKey` ('reddit-r-all', in this case).

Let's hook up that source listing now, so we can switch which news source we pull from dynamically.

Technically, we want to route our user to a new location in our app. Back with the markup we added to list the sources, let's add a routing path to the `news/:sourceKey` route of our app for each one.

To add a route that the Router understands, we'll use the directive parameter of `routerLink`:

```
<ul class="ui horizontal list">
  <li
    *ngFor="let source of sourcesSubscription | async"
    class="item">
      <a
        routerLink="/news/{{ source['id'] }}"
        class="ui small button">{{ source['name'] }}</a>
  </li>
</ul>
```

In the browser, we can see that the browser is flooded with options for picking a news source to fetch from.

When we click between sources (and scroll down), we'll see the article listing *automatically* repopulates with new article data.

Yay! Observables!

Wouldn't it be great if we could tell which of the sources was active visually instead of having to look at the url? Well… we can visually indicate it by using another angular directive: the `routerLinkActive` directive, which adds a CSS class when the `routerLink` is the activated one.

Let's update our list to include this `routerLinkActive` class to our links:

```
<ul class="ui horizontal list">
  <li
    *ngFor="let source of sourcesSubscription | async"
    class="item">
      <a
        routerLink="/news/{{ source['id'] }}"
        routerLinkActive="blue"
        class="ui small button">{{ source['name'] }}</a>
  </li>
</ul>
```

One more time, back in the browser, now we'll see as we click through different sources, the current source button is colored blue.

Great job! We've created a real news reader, complete with real news, and voting (ephemeral storage). We have a handful of more topics to cover before we can call our app complete.

8 Sorting

Let's change the sorting of our article listing. As it stands right now, our ordering is fixed (by time, descending). Let's make this dynamic.

Let's add some controls in our view to allow the user to change sorting by time and votes. We'll set this up in reverse. In our `ArticleListComponent`,

let's add a button group for sorting by time and votes in the `article-list.component.html`:

```
<!-- ... -->
  <div class="ui buttons">
    <button
      class="ui icon mini green button"
      (click)="changeDirection()">
      <i [ngClass]="{'up': sortDirection > 0,
                     'down': sortDirection < 0}"
        class="arrow up icon"></i>
    </button>
    <button
      class="ui mini red button"
      (click)="changeSort('Time')">
        Sort by time
      </button>
    <button
      class="ui mini blue button"
      (click)="changeSort('Votes')">
        Sort by votes
    </button>
  </div>
<!-- ... -->
```

Let's also go implement the `changeSort()` and `changeDirection()` functions in the `ArticleListComponent`. Since sorting requires a direction, we'll keep two values as values on our component:

- the current filter (in our case, 'Time' and 'Votes')
- the direction (1 for ascending, -1 for descending)

Let's add these variables to our `ArticleListComponent`, and set a few defaults:

```
// ...
export class ArticleListComponent implements OnInit {
```

```
  // ...
  private currentFilter: string = 'Time';
  private sortDirection: number = 1;

  // ...
  changeDirection(): void {}
  changeSort(filter: string): void {}
}
```

In the UI, we want to be able to sort the list by clicking on one of the sort buttons *or* by clicking on the direction button. If our user clicks on the sort option button a second time, we'll also want to update the direction.

Since we'll want to update the sort after running through the actions above, let's create a private helper function to actually call the sorting after we call the `changeSort()` or `changeDirection()` functions:

```
// ...
export class ArticleListComponent implements OnInit {
  // ...
  changeDirection(): void {
    this._updateSort();
  }
  changeSort(filter: string): void {
    this._updateSort();
  }
  _updateSort() {
    // actually update the sort
  }
}
```

Implementing the `changeDirection()` function is pretty straight-forward. Since the value is either a positive 1 or a negative one, we can multiple the current direction number by `-1` to toggle between the two.

```
// ...
export class ArticleListComponent implements OnInit {
```

```
  // ...
  changeDirection(): void {
    this.sortDirection = this.sortDirection * -1;
    this._updateSort();
  }
}
```

Updating the current filter isn't complicated either. We'll run a check to see if the filter hasn't changed. If not, we'll change the direction, save the new filter and update the sort:

```
// ...
export class ArticleListComponent implements OnInit {
  // ...
  changeSort(filter: string): void {
    if (filter === this.currentFilter) {
      this.changeDirection();
    } else {
      this.currentFilter = filter;
      this._updateSort();
    }
  }
}
```

Now that our UI has been updated, let's get it actually sorting our articles. In order to do that, we'll need to create a new method in the `ArticleService`, where we currently control the ordering of our article list.

We'll create an API method we'll call `sortBy()`, where we pass the filter and the direction. Before we implement the functionality, let's update our `_updateSort()` method to call out to the `ArticleService`:

```
// ...
export class ArticleListComponent implements OnInit {
  // ...
  _updateSort() {
    this.articleService
```

```
      .sortBy(this.currentFilter, this.sortDirection);
  }
}
```

In order to keep both the sorting and the direction in our `ArticleService`, we'll want to keep two more subjects. We'll deal with observables as our article listing ordering is kept in an Observable. We will create a combined stream of our observables to update the `orderedArticles` observable whenever *any of the combined observable* data is updated.

Let's create the new subjects in our `ArticleService`:

```
// ...
export class ArticleService {
  // ...

  // sortBy subject
  private _sortBySubject: BehaviorSubject<ArticleSortOrderFun> =
    new BehaviorSubject<ArticleSortOrderFun>(sorts['Time']);

  // direction subject
  private _sortDirectionSubject: BehaviorSubject<number> =
    new BehaviorSubject<number>(1);
}
```

Notice that we are initializing our `_sortDirectionSubject` with a number (1 – to sort in a descending order).

We're also initializing the sortBySubject with a function type called `ArticleSortOrderFun`. The `ArticleSortOrderFun` is an interface we can create in TypeScript. We'll want to create each sorting function as a function called with the direction which should return a function to sort a list of articles. It's a good idea to create a type interface to enforce the argument type and return values.

```
interface ArticleSortFun {
  (a: Article, b: Article): number;
}
interface ArticleSortOrderFun {
```

```
  (direction: number): ArticleSortFun;
};
export class ArticleService {
  // ...
}
```

Let's actually create a sort function with the `ArticleSortOrderFun` type. This sorting function will be similar to the way we implemented the first attempt, except we'll include the direction.

```
interface ArticleSortFun {
  (a: Article, b: Article): number;
}
interface ArticleSortOrderFun {
  (direction: number): ArticleSortFun;
};

// sorting
const sortByTime: ArticleSortOrderFun =
  (direction: number) => (a: Article, b: Article) => {
    return direction *
      (b.publishedAt.getTime() = a.publishedAt.getTime());
  };

export class ArticleService {
  // ...
}
```

Let's write the second sorting function, we'll call `sortByVotes` using the same approach:

```
interface ArticleSortFun {
  (a: Article, b: Article): number;
}
interface ArticleSortOrderFun {
  (direction: number): ArticleSortFun;
};
```

```
7
8  // ...
9  const sortByVotes: ArticleSortOrderFun =
10   (direction: number) => (a: Article, b: Article) => {
11     return direction * (b.votes - a.votes);
12   };
13
14 export class ArticleService {
15   // ...
16 }
```

With both of our sort functions defined (for `Time` and `Votes`), we can create a sorting object to allow our API function to use the sorting function by string:

```
1  const sorts = {
2    'Time': sortByTime,
3    'Votes': sortByVotes
4  };
5  export class ArticleService {
6    // ...
7  }
```

Finally, with our sorting functions written, we can implement our API and integrate sorting.

Strategically, we want to resort our `orderedArticles` object whenever *any of the sorting, direction, or articles* list are updated. That is, we can't just watch the `this.articles` as we are currently doing.

If we were to just watch the current `_articles` Subject, our article list won't resort if we just change the filter or direction. What we need is a combined Observable that combines all 3 sources.

RxJS supports this functionality out of the box using a method called `combinedLatest()`.

Let's create this combined observable in our `ArticleService`:

```
1  // ...
2  export class ArticleService {
```

```
3   constructor(private http:Http) {
4     // ...
5
6     // Create the merged Observable
7     this.orderedArticles =
8       Observable.combineLatest(
9         this._articles,
10        this._sortBySubject,
11        this._sortDirectionSubject
12      )
13  }
14 }
```

With our combined observable, we'll actually want to run the sorting with any of the updated variables. We can do this by adding a `map()` call and running the sorting over the list of articles:

```
1  // ...
2  export class ArticleService {
3    constructor(private http:Http) {
4      // ...
5
6      // Create the merged Observable
7      this.orderedArticles =
8        Observable.combineLatest(
9          this._articles,
10         this._sortBySubject,
11         this._sortDirectionSubject
12       )
13       .map([
14         articles, sorter, direction
15       ]) => articles.sort(sorter(direction));
16   }
17 }
```

The `combinedLatest()` observable exposes the streams as arguments of an array passed through map. We're calling the `sorter()` function, which accepts a num-

ber (direction) and returns back the actual sort function, which we pass to the native JavaScript sort function.

Last step we need to do before we call this section complete is define the `sortBy()` API function. All we need to do is pass a new variable into the new sorting / direction subjects and the ordering is already taken care of.

```
export class ArticleService {
  // ...
  sortBy(filter, direction): void {
    this._sortDirectionSubject.next(direction);
    this._sortBySubject.next(sorts[filter]);
  }
}
```

In our view, when we click on any of the sorting buttons, our list of articles will resort.

9 Voting

Let's hook up our voting on our articles. We currently have our `voteUp()` and `voteDown()` functions implemented in our `ArticleListComponent`, so when the voting functions are called, the article object itself will be updated, but the stream of articles aren't updated.

What we'll need to do is pass through a change event on the article from the `ArticleComponent` to the `ArticleListComponent` (child to parent communication). We'll pass through an output parameter on the `ArticleComponent`, so the child component knows how to communicate up to the parent component.

The steps we'll take to implement voting are:

1. The user clicks the vote button
2. The `article` object updates it's own state
3. The `ArticleComponent` emits a change event, which calls the parent `onVote()` method.
4. The parent receives the event and calls to the `ArticleService`

5. The `ArticleService` updates the global listing of articles with the latest articles.

Let's update the article listing in `src/app/articles/article-listing/article-listing.component.html` to include an `onVote()` function to the `ArticleComponent`. The `$event` definition is the event that Angular will pass up when our child component emits a change event.

```
<div class="ui grid posts">
  <app-article
    *ngFor="let article of articlesSubscription | async"
    [article]="article"
    (onVote)="onVoteArticle($event)"></app-article>
</div>
```

9.0.1 Child to parent communication

We'll implement our communication between our child and parent components using event listeners. This is *one* way to implement this heirarchical communication between components. One nice way of handling our communication using event emitters is that it still keeps our one-way binding.

Various other options are available through a data service, such as using redux, adding bi-directional variable bindings, through an Angular 2 service, etc.

We set the `(onVote)` action binding to the `ArticleComponent` to call the `onVoteArticle()` function on the `ArticleListComponent`.

Let's implement the `onVoteArticle()` function in the `ArticleListComponent` first. This method is simple and will only pass on an updated event value to the `ArticleService`:

```
export class ArticleListComponent implements OnInit {
  // ...
  onVoteArticle(event: any) {
    this.articleService.updateArticleVotes(event);
```

```
  }
}
```

In the child `ArticleComponent` component, we'll establish an `Output()` event emitter bound to the `onVote` function. We'll use this `onVote` emitter to call any changes from the child to the parent.

```
import { EventEmitter,
  Component, OnInit, HostBinding,
  Input, Output } from '@angular/core';
// ...

@Component({
  // ...
})
export class ArticleComponent implements OnInit {
  @Input() article: Article;
  @Output() onVote = new EventEmitter();
}
```

The only actions we're interested in with the `article` are the changes to the vote count, so we'll trigger this `onVote()` event in the `voteUp()` and `voteDown()` actions with the article which was voted upon.

Let's update the `ArticleComponent` adding these updates on the voting functions:

```
@Component({
  // ...
})
export class ArticleComponent implements OnInit {
  @Input() article: Article;
  @Output() onVote = new EventEmitter();

  voteUp(): boolean {
    this.article.voteUp();
    this.onVote.emit(this.article);
```

```
    return false;
  }

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

Last piece of this puzzle is to implement the `ArticleService.updateArticleVotes()` function.

Conceptually, we're going to iterate over the list of articles in our service and find the article the user voted on and return the *new* `Article` instance, instead of the existing one. The rest, we'll pass directly through to the `_articles` BehaviorSubject as the `next` value. Everything will act as it normally does and we'll be set.

```
export class ArticleService {
  // ...
  updateArticleVotes(article: Article) {
    const articles = this._articles.getValue()
      .map(a => (a.url === article.url) ? article : a);
    this._articles.next(articles);
  }
}
```

In our browser, let's click on the `voteUp` of one article. We'll see that a vote is actually added this time.

10 The sidebar

The final piece we'll talk about is getting the sidebar to show up in the appropriate location, listing all of the possible sources in a tucked away sidebar, rather than all over the screen.

In order to get the sidebar component mounted in our view, we'll need to update a few of our styles first. Let's head to the `src/app/app.component.html` file and update the style to include the semantic-ui updates to bind the sidebar to the top and bottom:

```
<div class="ui bottom attached segment">
  <app-sidebar></app-sidebar>
  <div class="pusher">
    <router-outlet></router-outlet>
  </div>
</div>
```

Our sidebar currently doesn't look that spectacular yet… let's update the sidebar template so that we get it placed somewhere on the page looking better than it looks now.

Let's open up the `src/app/sidebar/sidebar.component.html` file and update it to include some semantic-ui styles:

```
<div class="sidebar-container">
  <div
    class="ui container visible fixed inverted left vertical sidebar menu">
    <div class='item'>
      <div class='header'>News sources</div>
    </div> <!-- item -->
    <div class='item'>
      <div class='menu'>
        <a
          class='item news-item'
          *ngFor='let source of sources | async'
          routerLink='/news/{{ source["id"] }}'
          routerLinkActive='active'>
          <span class='side-news-item'>
            <span class='side-news-item'>
              {{ source['name'] }}
            </span>
          </span>
        </a>
```

```
      </div> <!-- menu -->
    </div> <!-- item -->
  </div> <!-- sidebar menu container -->
</div> <!-- sidebar-container -->
```

The sidebar styling looks pretty decent out of the box, but we're not actually displaying any data yet, so we'll just have a black box on the side of the view.

Let's update our application so that we pull news sources from the newsApi.

We'll take the same approach we did with selecting articles, but we'll use a different route endpoint to make the request.

Earlier, we broke out our http request method into a helper we called `_makeHttpRequest()` in our `ArticleService`:

```
export class ArticleService {
  // ...

  private _makeHttpRequest(
    path: string,
    sourceKey?: string
  ): Observable<any> {
    let params = new URLSearchParams();
    params.set('apiKey', environment.newsApiKey);
    if (sourceKey && sourceKey !== '') {
      params.set('source', sourceKey);
    }

    return this.http
            .get(`${environment.baseUrl}${path}`, {
              search: params
            }).map(resp => resp.json());
  }
}
```

We broke this out ahead of time so that we could use it again to make a different request to our backend. Let's use this `_makeHttpRequest()` method from the `ArticleService` to make a request to the `/sources` api:

```
export class ArticleService {
  // ...
  public getSources(): void {
    this._makeHttpRequest('/v1/sources')
  }
}
```

Right here, we haven't actually made a request that we'll capture. In the same way that we requested the articles and updated a `Subject`. Let's create our `BehaviorSubject` on the `ArticleService` and expose an observable object the same way we did with the `articles`.

Let's update the `src/app/article.service.ts` file:

```
export class ArticleService {
  // ...
  private _sources: BehaviorSubject<any> =
    new BehaviorSubject<any>([]);

  public sources: Observable<any> = this._sources.asObservable();
}
```

Now we can update this `_sources` BehaviorSubject when the sources object returns by subscribing to the result of the observable http response object:

```
export class ArticleService {
  // ...
  public getSources(): void {
    this._makeHttpRequest('/v1/sources')
      .map(json => json.sources)
      .subscribe(this._sources);
  }
}
```

That's it. We've hooked up a new service method to fetch sources from the newsApi server.

Just like we did previously with the `articles`, we can subscribe to any changes on the `sources` observable inside the `Sidebar` component.

Currently, our sidebar component looks pretty bare:

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

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

  ngOnInit() {}

}
```

The steps we need to take to hook up our sources are:

1. Add a local variable to the `SidebarComponent` to hold on to the sources observable.
2. Connect the variable to the `articleSources` variable.
3. Fetch the sources when our sidebar mounts.

Since we've covered each of these steps previously, let's implement this in one fell swoop:

```
import { Component, OnInit } from '@angular/core';
import { ArticleService } from '../article.service';
import { Observable } from 'rxjs';

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

```
})
export class SidebarComponent implements OnInit {
  private sources: Observable<any>;

  constructor(
    private articleService: ArticleService
  ) {
    this.sources = this.articleService.sources;
  }

  ngOnInit() {
    this.articleService.getSources();
  }

}
```

Reloading the page, our sources come up automatically and our list populates. The `sourceKey` parameter updates in our address bar, which will automatically call the `updateArticles()` method from within the `ArticleListComponent` and our page will automatically update.

11 Deployment

When we are ready to deploy our application, we'll want to ship our application with the proper environment configuration, minified, obsfucated, and compressed into a production-ready version of our application.

The `angular-cli` has a `build` command which does exactly that.

Within our project root, let's call the `build` command and see what it generates:

```
ng build
```

Great. Now our application is compressed, minified, and built in the `dist/` directory.

Only one issue with this is that we built our application using the development environment instead of the production environment.

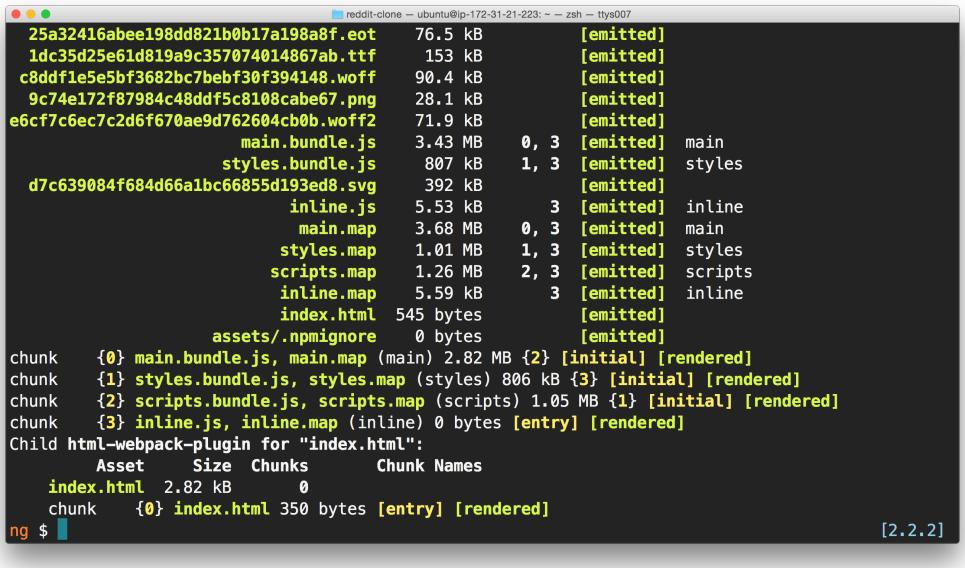

Figure 4:

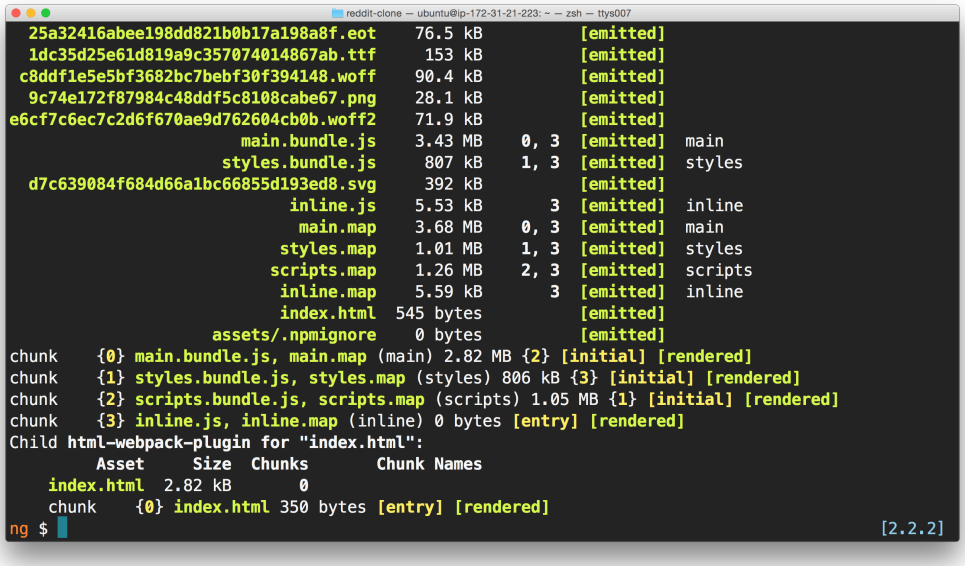

Figure 5:

We can pass an argument to the `build` command to tell the cli we want to bundle our application with the production environment values.

```
1 ng build --env prod
```

Note that the `prod` environment variable matches the extension of the `environment.prod.ts` file we created previously.

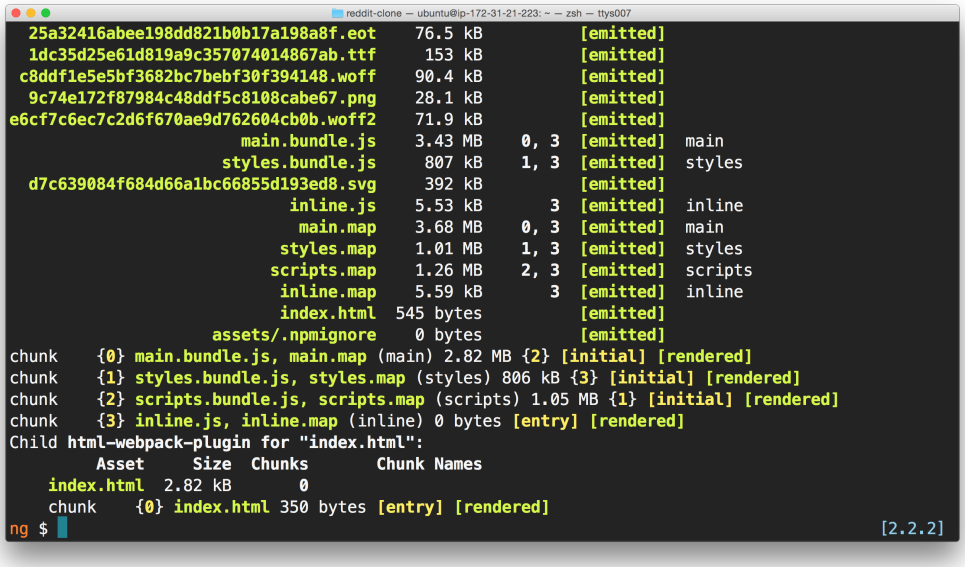

Figure 6:

Now that we've built our application in production, we have a single directory that we can deploy to a hosted location. For instance, we can deploy to S3 (using their static hosting), Dropbox, any ole' server running Apache or nginx, and more.

11.0.1 Github Pages

As an example, let's deploy our application to Github Pages. We'll want to deploy our application using a github repository.

In order to deploy our application, we'll need to create a repository on Github.

We'll need to make sure we add the remote origin in our local repository:

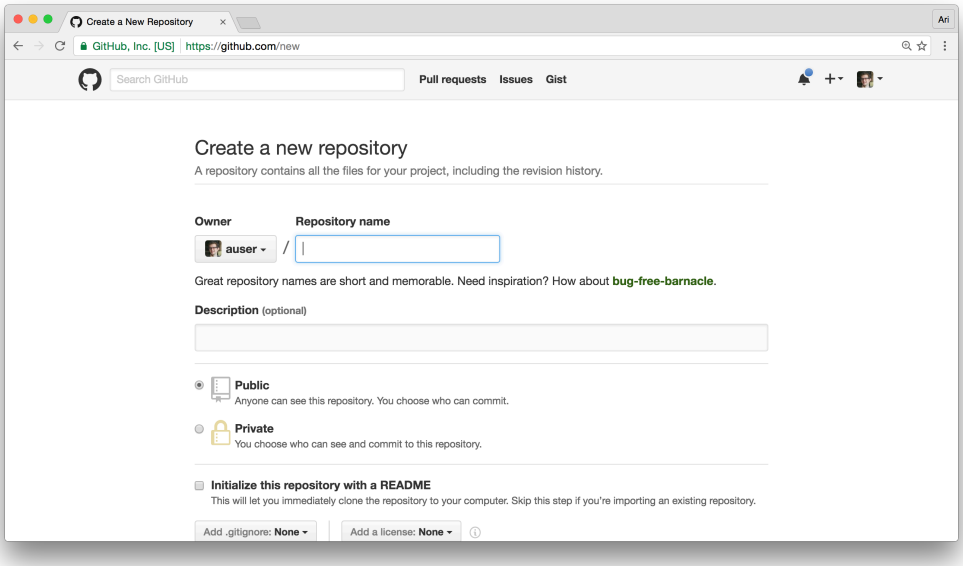

Figure 7:

```
git remote add origin git@github.com:fullstackio/reddit-clone.git
git push -u origin master
```

Now, since the google git repo has been added, we can push our site to our github repository. However, if we push the master branch to our github repo, our site won't be hosted in production.

Instead we want to work on the `gh-pages` branch as Github Pages uses this branch to host our site.

We'll need to update the `.gitignore` file to remove the `dist` entry. Find the `.gitignore` file and remove the line with `dist/`.

Additionally, we really want is to push *only* the `dist/` directory contents. Luckily, we can do this by using the `subtree` git command.

```
git add -A dist
git commit -am "Removed dist/ gitignore"
git subtree push --prefix dist origin gh-pages
```

Once we do that, we can go to our site and see it on github.

12 Just the very beginning

We made it to the end. We've built a complex application, taking each step from the very beginning.

We've covered a lot of topics in this course.

- Building Angular 2 components
- Heirarchical structure
- Using TypeScript models
- Creating custom TypeScript types
- Styling Angular 2 components
- Using the `angular-cli` to generate components
- Integrating Observables in our application using RxJS
- Making API calls
- Driving our components through data
- Routing and complex page applications
- Adding interactivity in our application
- Deployment
- And more…

As the landscape of Angular 2 changes, so will the code-base.

For any questions, comments, or concerns about this content, feel free to reach out to us and engage with the community. ## Contact info

Feel free to reach out to us on Slack, via email, or twitter.