ng-book 2

The Complete Book on AngularJS 2

FULLSTACK.io
gistia

Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

ng-book 2

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

Contents

Book Revision

Revision 43 - Covers up to Angular 2 (2.2.0-rc.0, 2016-11-08)

Bug Reports

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

Chat With The Community!

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

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow @fullstackio[3]

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[4].

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

[5] mailto:us@fullstack.io

Writing your First Angular 2 Web Application

Simple Reddit Clone

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

You can think of this app as the beginnings of a site like Reddit[6] or Product Hunt[7].

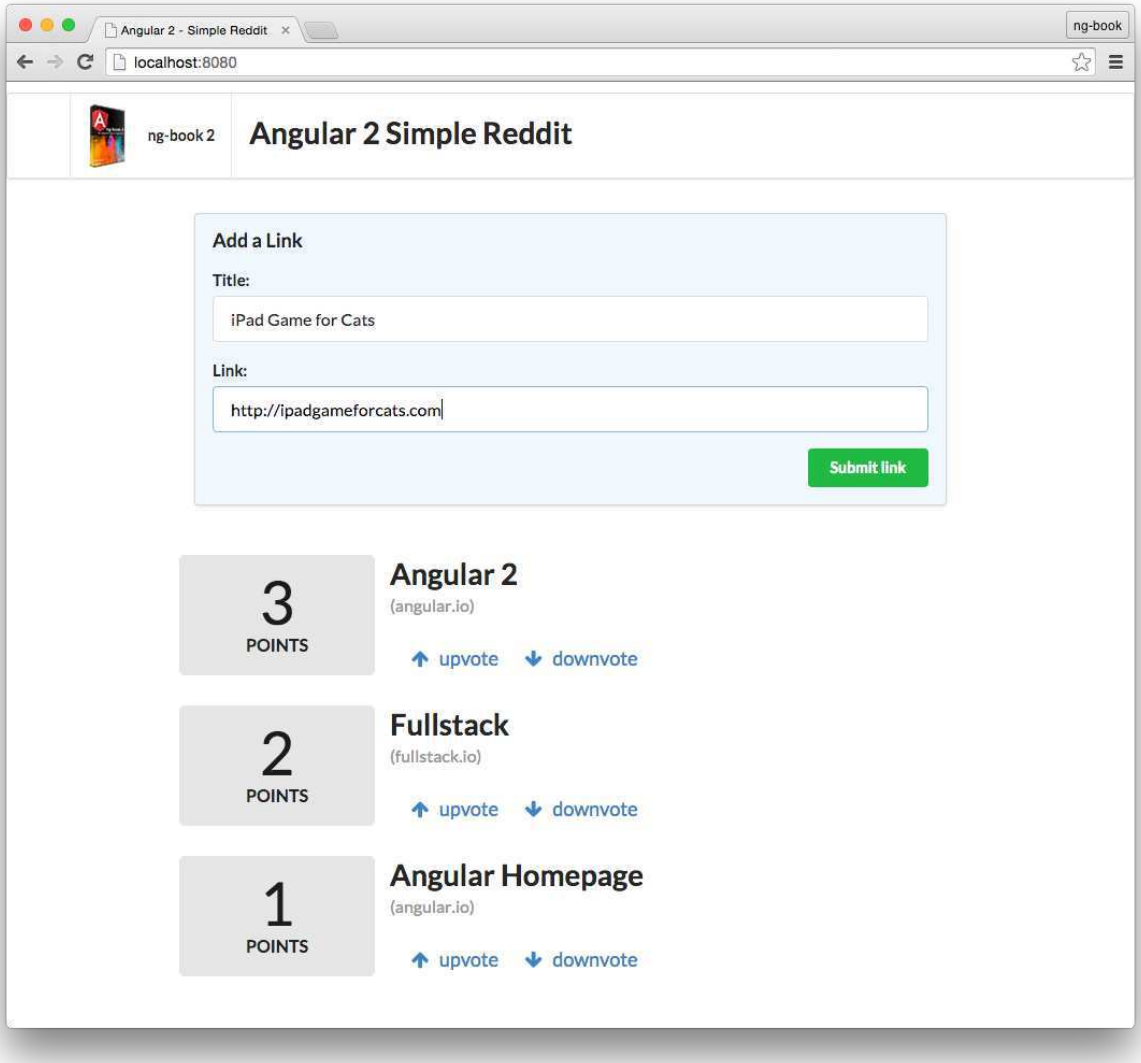
In this simple app we're going to cover most of the essentials of Angular 2 including:

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

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

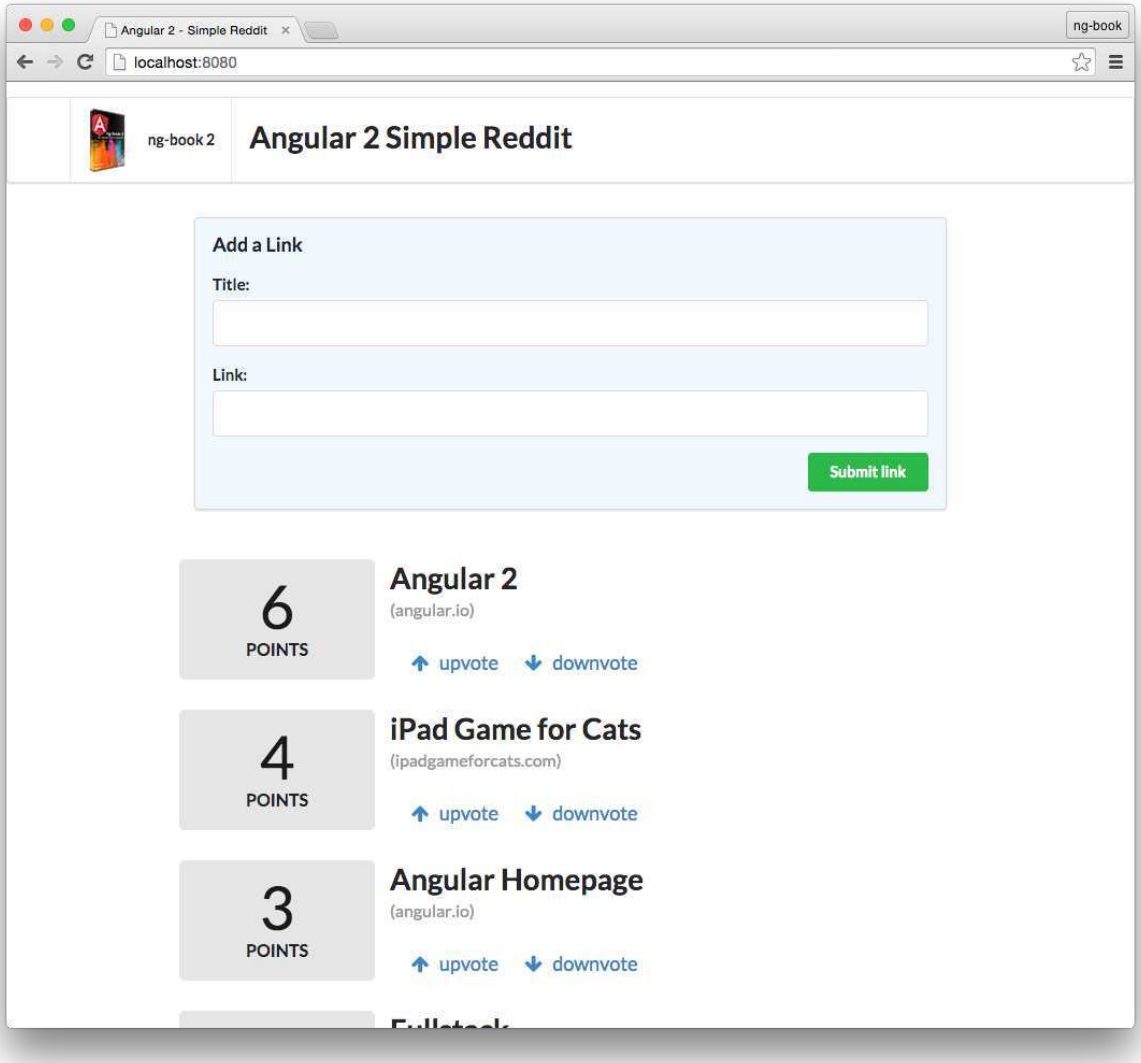
Here's a screenshot of what our app will look like when it's done:

[6] http://reddit.com
[7] http://producthunt.com



Completed application

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



App with new article

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

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

Getting started

TypeScript

To get started with TypeScript, you'll need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to the Node.js website[8] for detailed information.

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

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

```
1 $ npm install -g typescript
```

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

Windows Users: We'll be using Linux/Mac-style commands on the command line throughout this book. We'd highly recommend you install Cygwin[9] as it will let you run commands just as we have them written out in this book.

angular-cli

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

To install `angular-cli`, just run the following command:

```
1 $ npm install -g angular-cli@1.0.0-beta.18
```

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

[8] https://nodejs.org/download/
[9] https://www.cygwin.com/

```
$ ng
Could not start watchman; falling back to NodeWatcher for file system events.
Visit http://ember-cli.com/user-guide/#watchman for more info.
Usage: ng <command (Default: help)>
```

The reason we got that huge output is because when we run `ng` with no arguments, it runs the default `help` command, which explains how to use the tool.

If you're running OSX or Linux, you probably received this line among the output:

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

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

```
$ brew install watchman
```

If you're on OSX and got an error when running brew, it means that you probably don't have Homebrew installed. Please refer to the page http://brew.sh/ to learn how to install it and try again.

If you're on Linux, you may refer to the page https://ember-cli.com/user-guide/#watchman for more information about how to install watchman.

If you're on Windows instead, you don't need to install anything and `angular-cli` will use the native Node.js watcher.

And with that we have `angular-cli` and its dependencies installed. Throughout this chapter we're going to use this tool to create our first application.

Example Project

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

Let's open up the terminal and run the `ng new` command to create a new project from scratch:

```
$ ng new angular2_hello_world
```

Once you run it, you'll see the following output:

```
installing ng2
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/app/index.ts
  create src/app/shared/index.ts
  create src/assets/.gitkeep
  create src/assets/.npmignore
  create src/environments/environment.dev.ts
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create src/typings.d.ts
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
 Installing packages for tooling via npm
```

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

```
Installed packages for tooling via npm.
```

There are a lot of files generated! Don't worry too much about all of them yet. We'll walk through what each one means and is used for throughout the book. For now, let's focus on getting started with Angular code.

Let's go inside the `angular2_hello_world` directory, which the `ng` command created for us and see what has been created:

```
$ cd angular2_hello_world
$ tree -F -L 1
.
├── README.md              // an useful README
├── angular-cli.json       // angular-cli configuration file
├── e2e/                   // end to end tests
├── karma.conf.js          // unit test configuration
├── node_modules/          // installed dependencies
├── package.json           // npm configuration
├── protractor.conf.js     // e2e test configuration
├── src/                   // application source
└── tslint.json            // linter config file

3 directories, 6 files
```

For now, the folder we're interested on is `src`, where our application lives. Let's take a look at what was created there:

```
$ cd src
$ tree -F
.
|-- app/
|   |-- app.component.css
|   |-- app.component.html
|   |-- app.component.spec.ts
|   |-- app.component.ts
|   |-- app.module.ts
|   |-- index.ts
|   `-- shared/
|       `-- index.ts
|-- assets/
|-- environments/
|   |-- environment.dev.ts
|   |-- environment.prod.ts
|   `-- environment.ts
|-- favicon.ico
|-- index.html
|-- main.ts
|-- polyfills.ts
```

```
|-- styles.css
|-- test.ts
|-- tsconfig.json
`-- typings.d.ts

4 directories, 18 files
```

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

code/first_app/angular2_hello_world/src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular2HelloWorld</title>
  <base href="/">

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

Let's break it down a bit:

code/first_app/angular2_hello_world/src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular2HelloWorld</title>
  <base href="/">
```

If you're familiar with writing HTML file, this first part should be trivial, we're declaring the page charset, title and base href.

code/first_app/angular2_hello_world/src/index.html

```
8   <meta name="viewport" content="width=device-width, initial-scale=1">
```

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

code/first_app/angular2_hello_world/src/index.html

```
12  <app-root>Loading...</app-root>
13 </body>
14 </html>
```

The `app-root` tag is where our application will be rendered. We'll see this later when we inspect other parts of the source code. The text **Loading**... is a placeholder that will be displayed before our app code loads. We can use this technique to inform the user the application is still loading by using either a message like we're doing here, or a spinner or other kind of progress notification we see fit.

Writing Application Code

Running the application

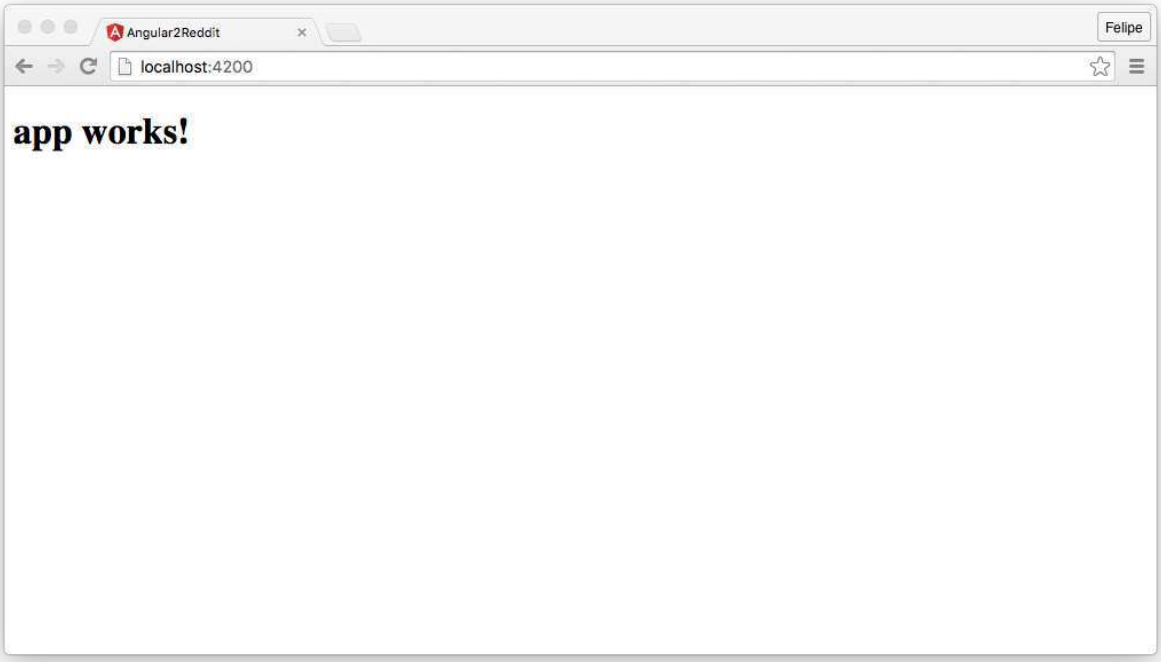
Before making any changes, let's load our app from the generated application into the browser. `angular-cli` has a built in HTTP server that we can use to start our app. Back in the terminal, at the root of our application (for the previously generated application, this will be in the directory the generated created `./angular2_hello_world`) and run:

```
1 $ ng serve
2 ** NG Live Development Server is running on http://localhost:4200. **
3 // a bunch of debug messages
4
5 Build successful - 1342ms.
```

Our application is now running on localhost port 4200. Let's open the browser and visit http://localhost:4200[10]:

Note that if for some reason port `4200` is taken it may start on another port number. Be sure to read the messages on your machine to find your exact development URL

[10] http://localhost:4200



Running application

Alright, now that we the setup for the application in place, and we know how to run it, it's time to start writing some code.

Making a `Component`

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

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

What if we want to teach the browser new tags? What if we wanted to have a `<weather>` tag that shows the weather? Or what if we wanted to have a `<login>` tag that creates a login panel?

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

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

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

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

To create a new component using `angular-cli`, we'll use the **generate** command.

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

```
$ ng generate component hello-world
installing component
  create src/app/hello-world/hello-world.component.css
  create src/app/hello-world/hello-world.component.html
  create src/app/hello-world/hello-world.component.spec.ts
  create src/app/hello-world/hello-world.component.ts
```

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

1. A `Component` annotation
2. A component definition class

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

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

Notice that we suffix our TypeScript file with `.ts` instead of `.js` The problem is our browser doesn't know how to interpret TypeScript files. To solve this gap, the `ng serve` command live-compiles our `.ts` to a `.js` file automatically.

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

Importing Dependencies

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

We `import Component` from the module `"@angular/core"`. The `"@angular/core"` portion tells our program **where to find the dependencies** that we're looking for. In this case, we're telling the compiler that `"@angular/core"` defines and exports two JavaScript/TypeScript objects called `Component` and `OnInit`.

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

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

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

Component Annotations

After importing our dependencies, we are declaring the component:

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.ts

```
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
```

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

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

What is going on here? If you have a Java background it may look familiar to you. These are annotations.

> Angular 1's dependency injection used the annotation concept behind the scenes. Even if you're not familiar with them, annotations are a way to add functionality to code using the compiler.

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

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

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

Similar to CSS selectors, XPath, or JQuery selectors, there are lots of ways to configure a selector. Angular Components adds their own special sauce to the selector mix, and we'll cover that later on. For now, keep in mind that we're **defining a new HTML markup tag**.

The `selector` property here indicates which DOM element this component is going to use. This way any `<app-hello-world></app-hello-world>` tags that appear within a template will be compiled using this `Component` class and all of it's definitions within it.

Adding a template with `templateUrl`

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

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.html

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

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

Adding a `template`

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

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

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

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

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

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

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

Adding CSS Styles with `styleUrls`

Notice the key `styleUrls`:

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

This code says that we want to use the CSS in the file `hello-world.component.css` as the styles for this component. Angular 2 uses a concept called "style-encapsulation" which means that styles specified for a particular component *only apply to that component*. We talk more about this in-depth later on in the book in the Styling section of Advanced Components.

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

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

Loading Our Component

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

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

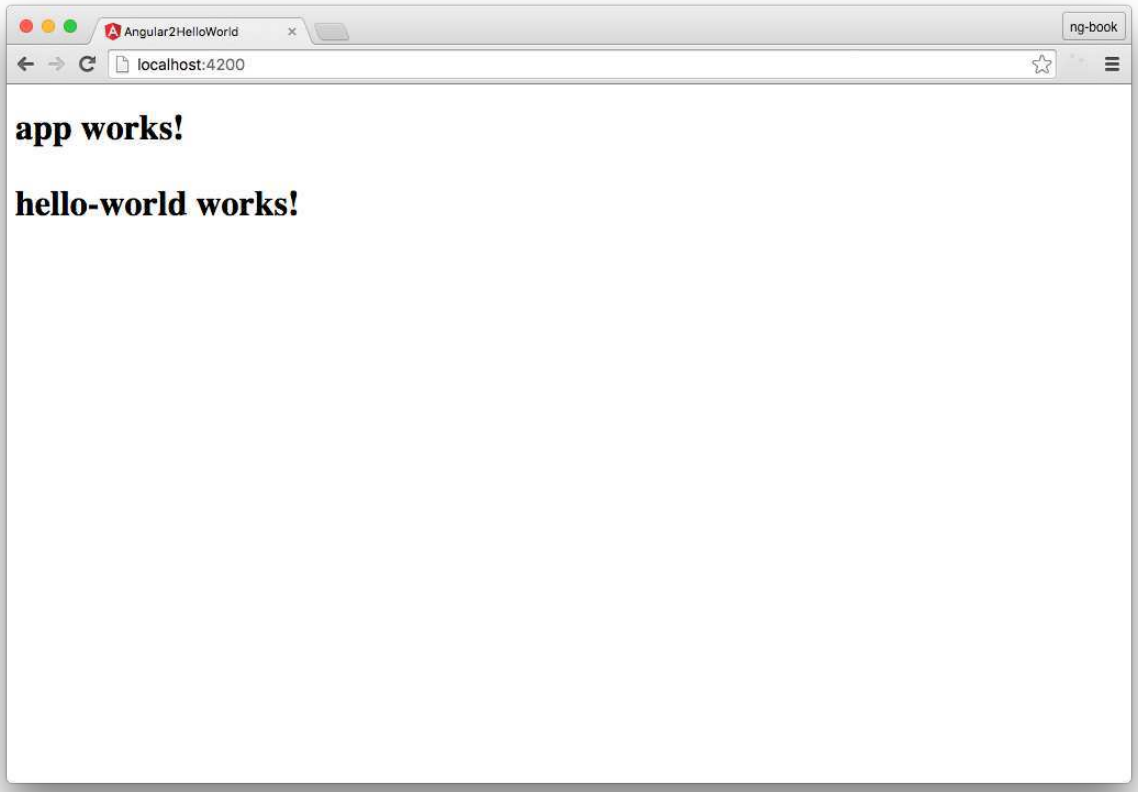
In order to change that, we need to add our component tag to a template that is already being rendered. Open up the file: `first_app/angular2_hello_world/src/app/app.component.html`

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

code/first_app/angular2_hello_world/src/app/app.component.html

```
1 <h1>
2   {{title}}
3
4   <app-hello-world></app-hello-world>
5 </h1>
```

Now refresh the page and take a look:



Hello world works

It works!

Adding Data to the Component

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

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

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

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

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

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

code/first_app/angular2_hello_world/src/app/app.component.html

```
<h1>
  {{title}}

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

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

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

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

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

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

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

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

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

  ngOnInit() {
  }

}
```

Notice that we've changed two things:

1. `name` Property

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

The typing of the name is a feature of using TypeScript and gives some assurances of the value that it will be a `string`. This sets up a `name` property on *instances* of our `UserItemComponent` class and the compiler ensures that `name` is a `string`.

2. A Constructor

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

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

When we write:

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

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

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

Rendering The Template

With the value filled out, we can use the templating syntax (which is two squiggly brackets `{{ }}`) to display the value of the variable in our template. For instance:

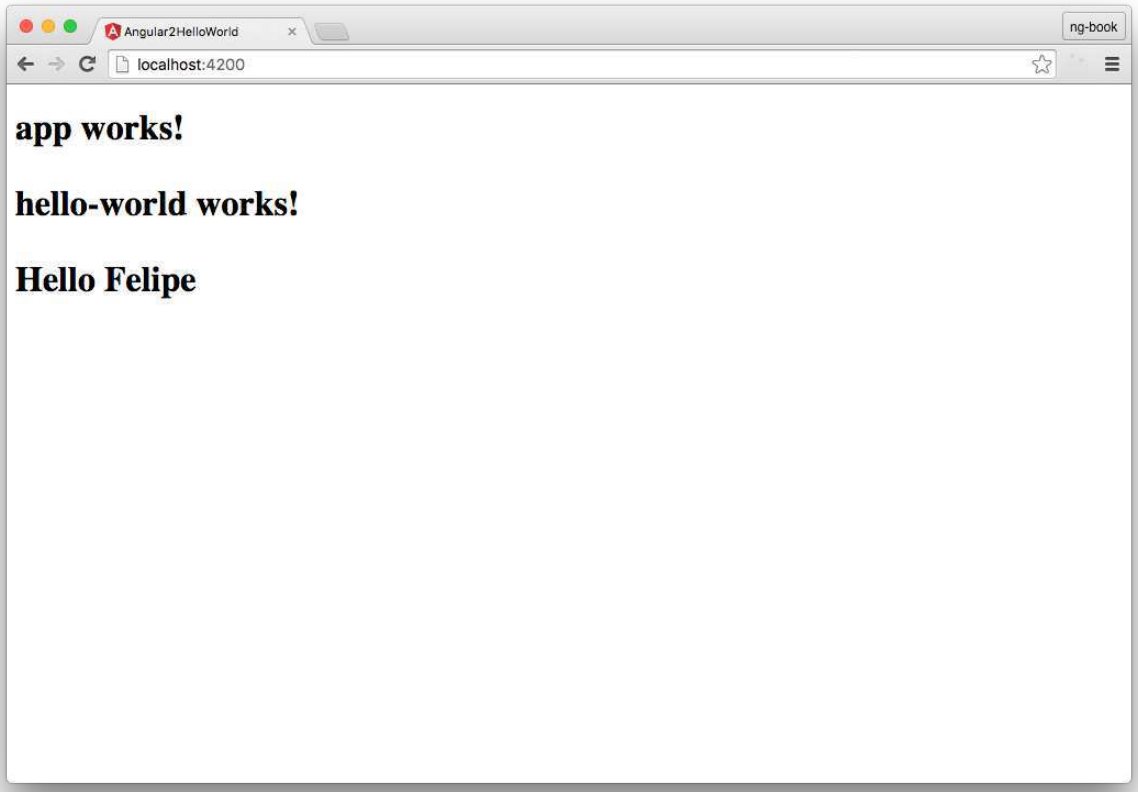
code/first_app/angular2_hello_world/src/app/user-item/user-item.component.html

```
1  <p>
2    Hello {{ name }}
3  </p>
```

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

Try it out

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



Application with Data

Working With Arrays

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

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

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

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

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

code/first_app/angular2_hello_world/src/app/app.component.html

```
<h1>
  {{title}}

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

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

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

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

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.ts

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

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

  ngOnInit() {
  }

}
```

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

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

Now we can update our template to render this list of names. To do this, we will use `*ngFor`, which will iterate over a list of items and generate a new tag for each one. Here's what our new template will look like:

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

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

We updated the template with one `ul` and one `li` with a new `*ngFor="let name of names"` attribute. The `*` character and `let` syntax can be a little overwhelming at first, so let's break it down:

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

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

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

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

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

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

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

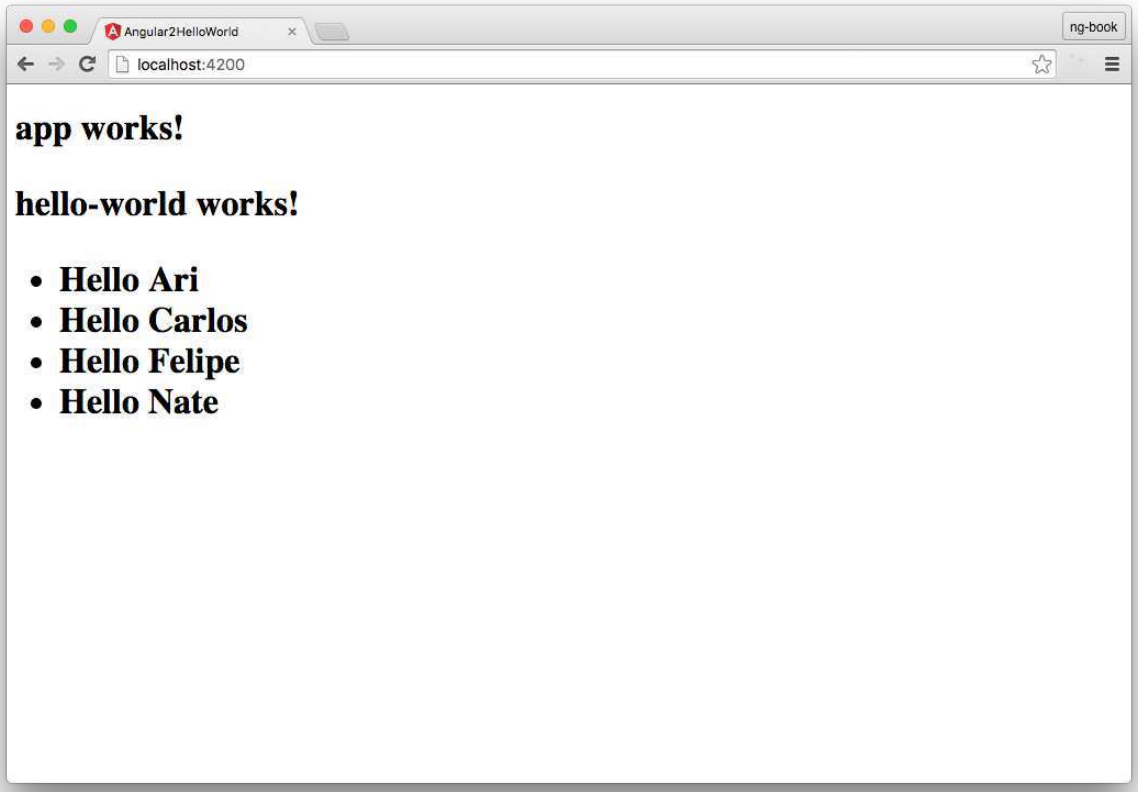
We'd get an error because `foobar` isn't a property on the component.

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

If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the NgFor directive here[11]

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

[11] https://github.com/angular/angular/blob/master/modules/%40angular/common/src/directives/ng_for.ts



Application with Data

Using the User Item Component

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

To do this, we need to do three things:

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

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

Rendering the `UserItemComponent`

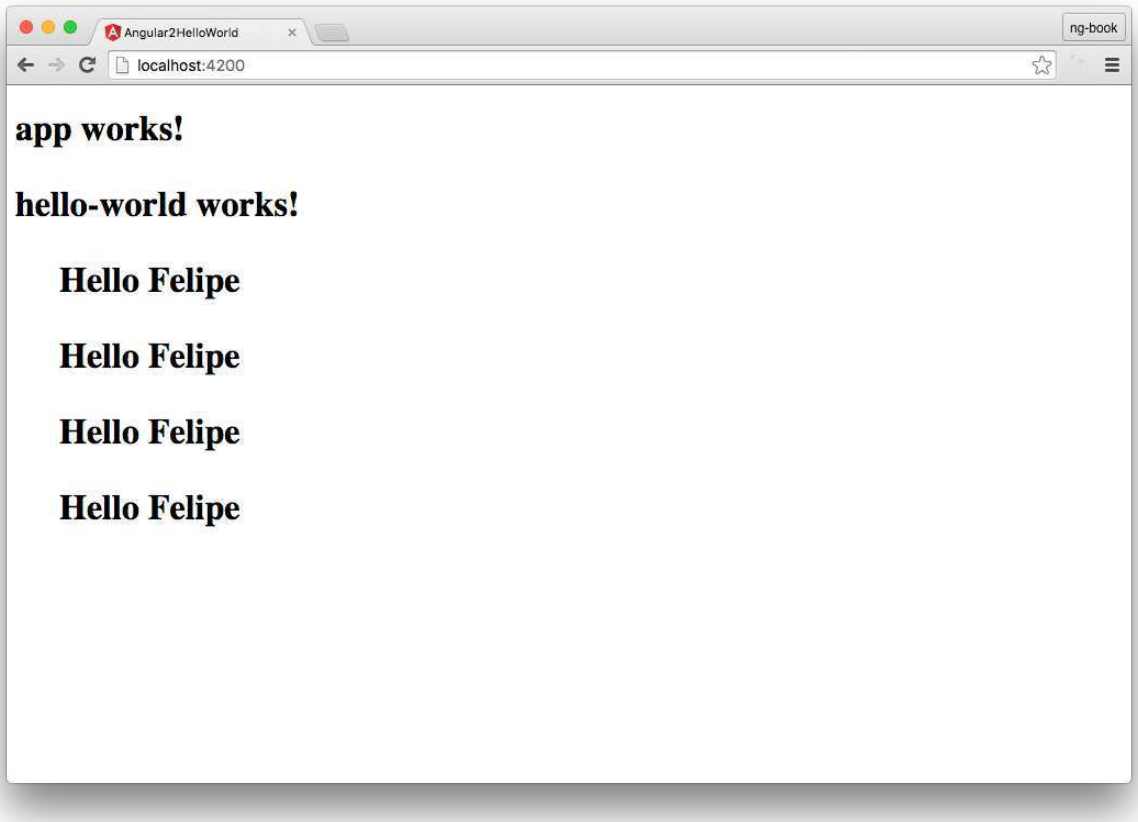
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template. What we're going to do is replace the `<li>` tag with the `app-user-item` tag:

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

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

Notice that while we swapped out the `li` tag for `app-user-item` we left in the `ngFor` attribute because we *still want to loop* over the list of names.

Notice that we also removed the inner content of this template because *the component* has it's own template. If we reload our browser, this is what we will see:



Application with Data

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

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

Accepting Inputs

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

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

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

```
1  import {
2    Component,
3    OnInit,
4    Input      // <--- added this
5  } from '@angular/core';
6
7  @Component({
8    selector: 'app-user-item',
9    templateUrl: './user-item.component.html',
10   styleUrls: ['./user-item.component.css']
11 })
12 export class UserItemComponent implements OnInit {
13   @Input() name: string; // <-- added Input annotation
14
15   constructor() {
16     // removed setting name
17   }
18
19   ngOnInit() {
20   }
21
22 }
```

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

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

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

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

Passing an Input value

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

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

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

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

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

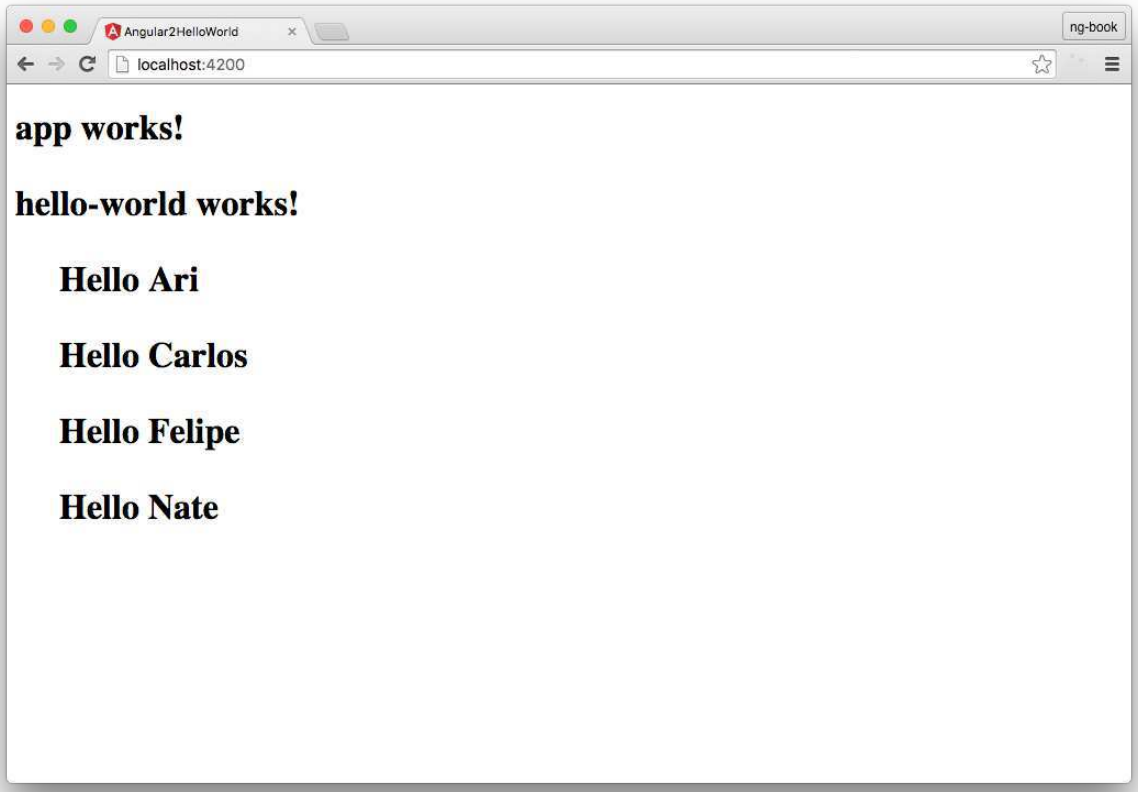
```
  <app-user-item
    *ngFor="let individualUserName of names"
    [name]="individualUserName">
  </app-user-item>
```

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

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

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

Now rendering our list of names is working!



Application with Names Working

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

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

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

Bootstrapping Crash Course

Every app has a main entry point. This application was built using `angular-cli` which is built on a tool called webpack. You don't have to understand webpack to use Angular, but it is helpful to understand the flow of how your application boots.

We run this app by calling the command:

```
ng serve
```

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

At a high level, it looks like this:

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

We'll talk about this process more later in the book, but for now the thing I want to focus on is the Angular module system: `NgModules`.

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

Let's a look at the code:

code/first_app/angular2_hello_world/src/app/app.module.ts

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

The first thing we see is an `@NgModule` annotation. Like all annotations, this `@NgModule( ... )` code adds metadata to the class immediately following (`AppModule`).

Our `@NgModule` annotation has three keys: `declarations`, `imports`, and `bootstrap`.

`declarations` specifies the components that are **defined in this module**. You may have noticed that when we used `ng generate` it automatically added our components to this list! This is an important idea in Angular:

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

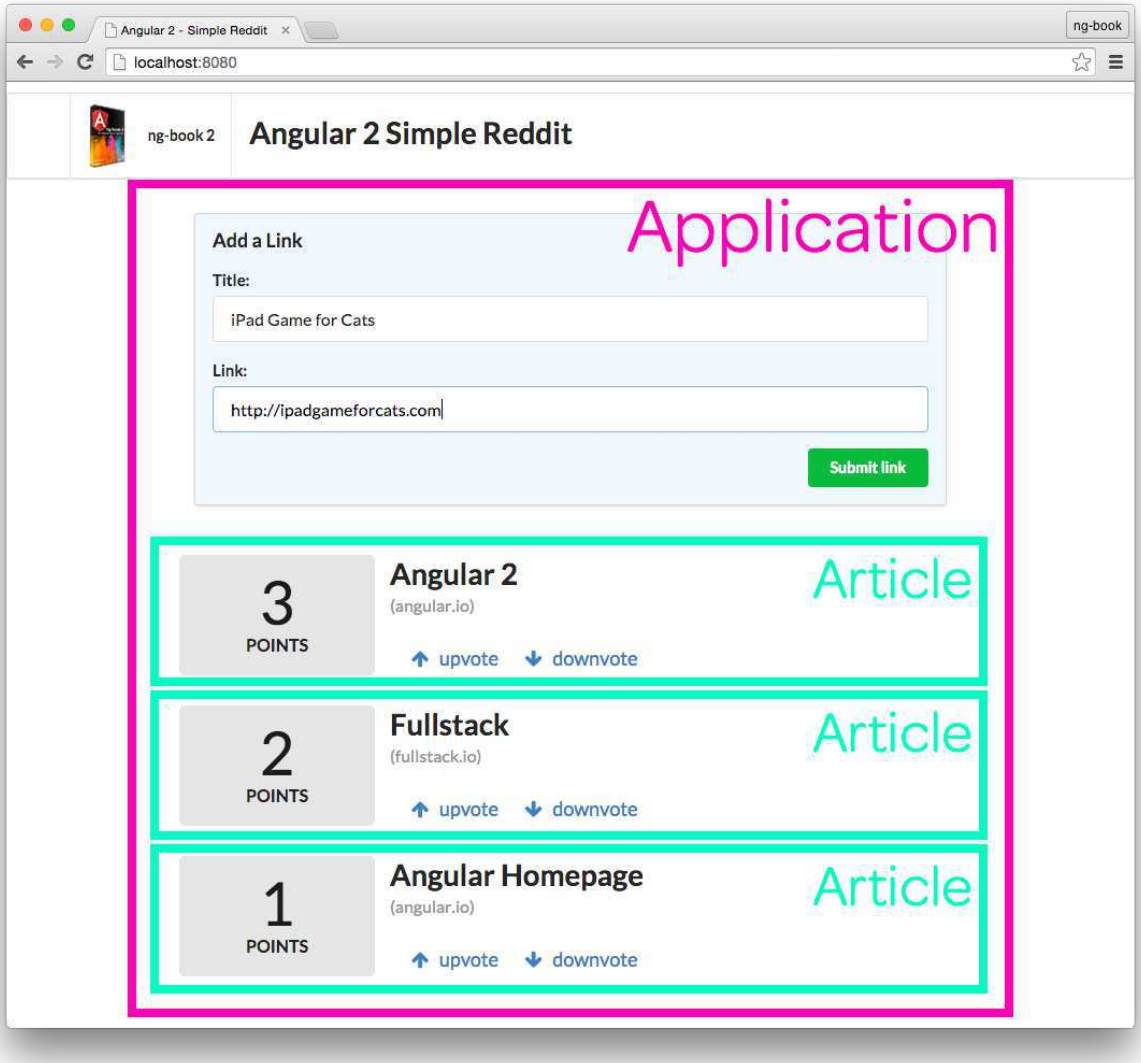
`imports` describes which *dependencies* this module has. We're creating a browser app, so we want to import the `BrowserModule`.

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

We talk more about `NgModules` in the section on `NgModules`

Expanding our Application

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



Application with Data

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

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

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

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

But first thing's first, let's generate a new application by running the same **ng new** command we ran before to create a new application passing it the name of the app we want to create (here, we'll create an application called `angular2_reddit`):

```
1 ng new angular2_reddit
```

We provide a completed version of our `angular2_reddit` in the example code download

Adding CSS

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

If you're building your app from scratch, you'll want to copy over a few files from our completed example in the `first_app/angular2_reddit` folder.

Copy:

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

into your application's folder.

For this project we're going to be using Semantic-UI[12] to help with the styling. Semantic-UI is a CSS framework, similar to Zurb Foundation[13] or Twitter Bootstrap[14]. We've included it in the sample code download so all you need to do is copy over the files specified above.

The Application Component

Let's now build a new component which will:

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

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

[12] http://semantic-ui.com/
[13] http://foundation.zurb.com
[14] http://getbootstrap.com

code/first_app/angular2_reddit/src/app/app.component.ts

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

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

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

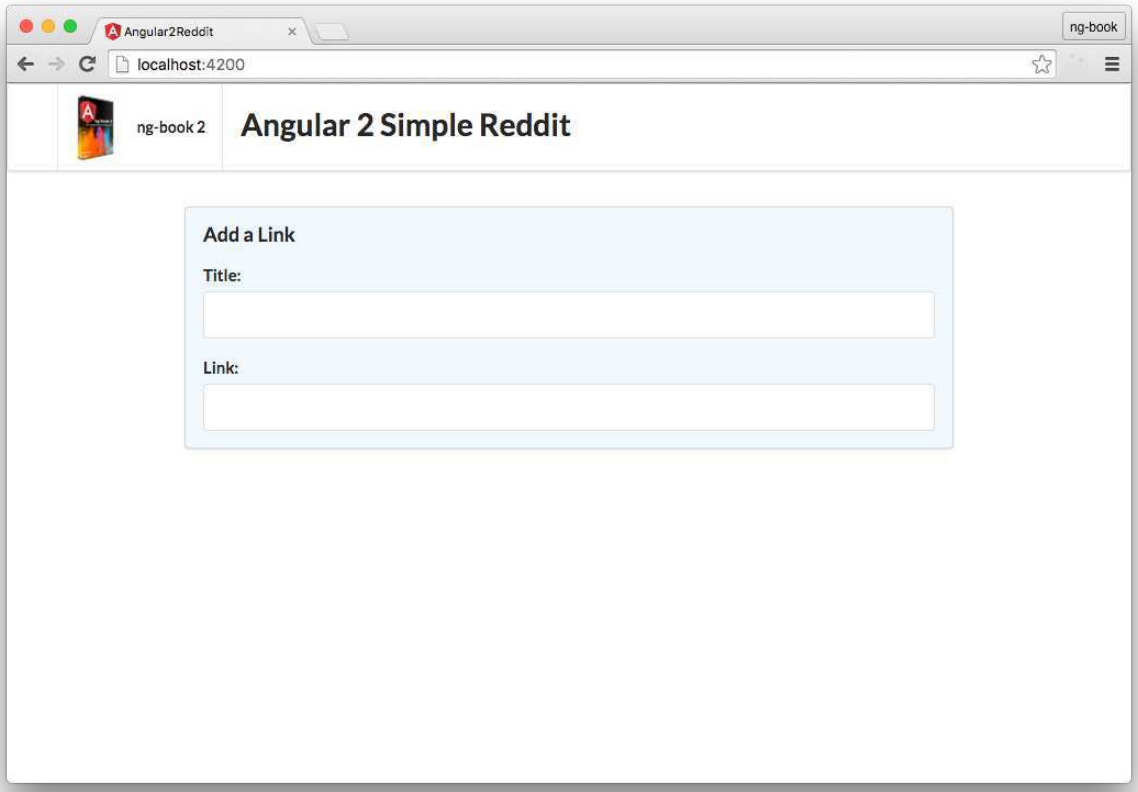
code/first_app/angular2_reddit/src/app/app.component.html

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

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

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

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



Form

Adding Interaction

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

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

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

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

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

code/first_app/angular2_reddit/src/app/app.component.ts

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

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

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

code/first_app/angular2_reddit/src/app/app.component.html

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

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

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

</form>
```

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

To recap what we've done, we've made **four** changes:

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

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

Binding `input`s to values

Notice in our first input tag we have the following:

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

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

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

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

Binding actions to events

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

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

All together:

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

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

Defining the Action Logic

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

code/first_app/angular2_reddit/src/app/app.component.ts

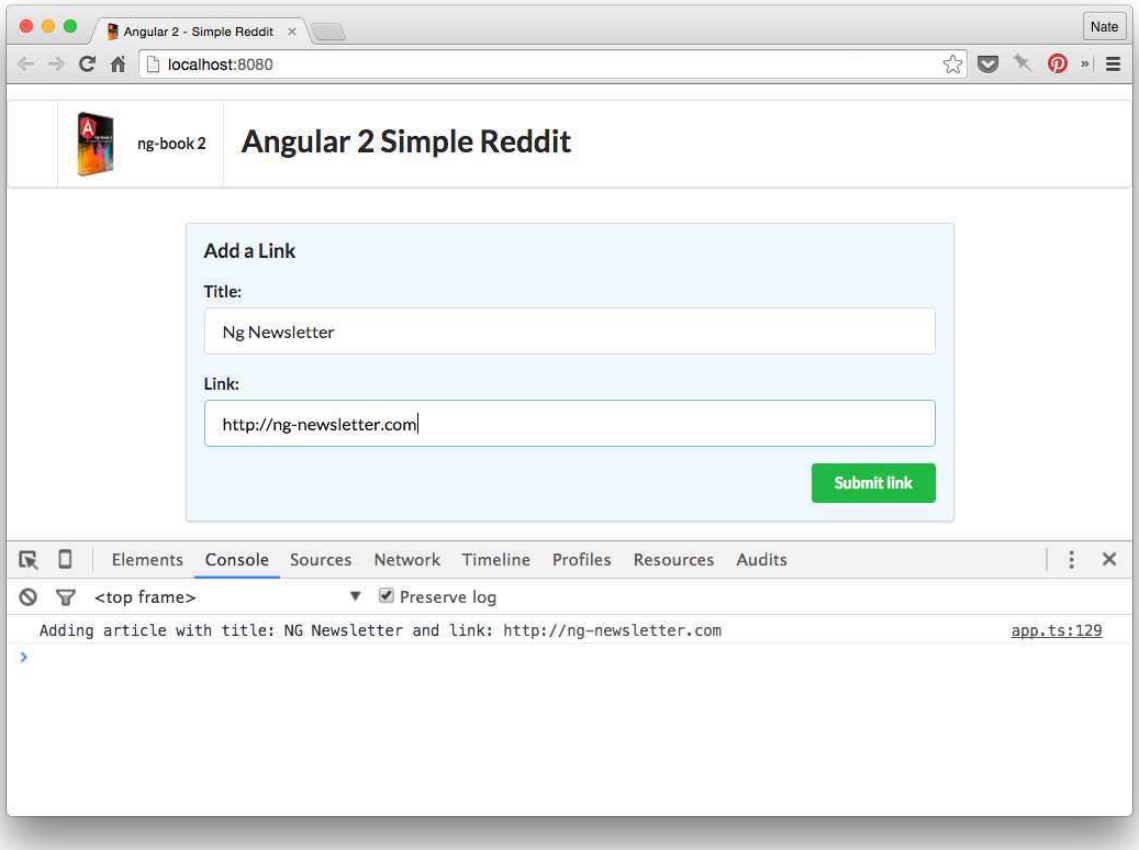
```
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
```

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

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

Try it out!

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



Clicking the Button

Adding the Article Component

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

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

A reddit-article

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

```
ng generate component article
```

We have three parts to defining this new component:

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

Let's talk through each part in detail:

Creating the `ArticleComponent` template

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

code/first_app/angular2_reddit/src/app/article/article.component.html

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

```
27      </li>
28    </ul>
29  </div>
```

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

A Single reddit-article Row

We have two columns:

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

We specify these columns with the CSS classes `four wide column` and `twelve wide column` respectively (remember that these come from SemanticUI's CSS).

We're showing `votes` and the `title` with the template expansion strings `{{ votes }}` and `{{ title }}`. The values come from the value of `votes` and `title` property of the `ArticleComponent` class, which we'll define in a minute.

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

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

Creating the `ArticleComponent`

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
3  @Component({
4    selector: 'app-article',
5    templateUrl: './article.component.html',
6    styleUrls: ['./article.component.css'],
7    host: {
8      class: 'row'
9    }
10 })
```

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

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

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

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

We want each `app-article` to be on its own row. We're using Semantic UI, and Semantic provides a CSS class for rows[15] called `row`.

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

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

Creating the `ArticleComponent` Definition Class

Finally, we create the `ArticleComponent` definition class:

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
11 export class ArticleComponent implements OnInit {
12   votes: number;
13   title: string;
14   link: string;
15
16   constructor() {
17     this.title = 'Angular 2';
18     this.link = 'http://angular.io';
19     this.votes = 10;
20   }
21
22   voteUp() {
23     this.votes += 1;
24   }
25
26   voteDown() {
27     this.votes -= 1;
28   }
29
30   ngOnInit() {
31   }
32
33 }
```

Here we create three properties on `ArticleComponent`:

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

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
16    constructor() {
17      this.title = 'Angular 2';
18      this.link = 'http://angular.io';
19      this.votes = 10;
20    }
```

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
22    voteUp() {
23      this.votes += 1;
24    }
25
26    voteDown() {
27      this.votes -= 1;
28    }
```

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

Using the `app-article` Component

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

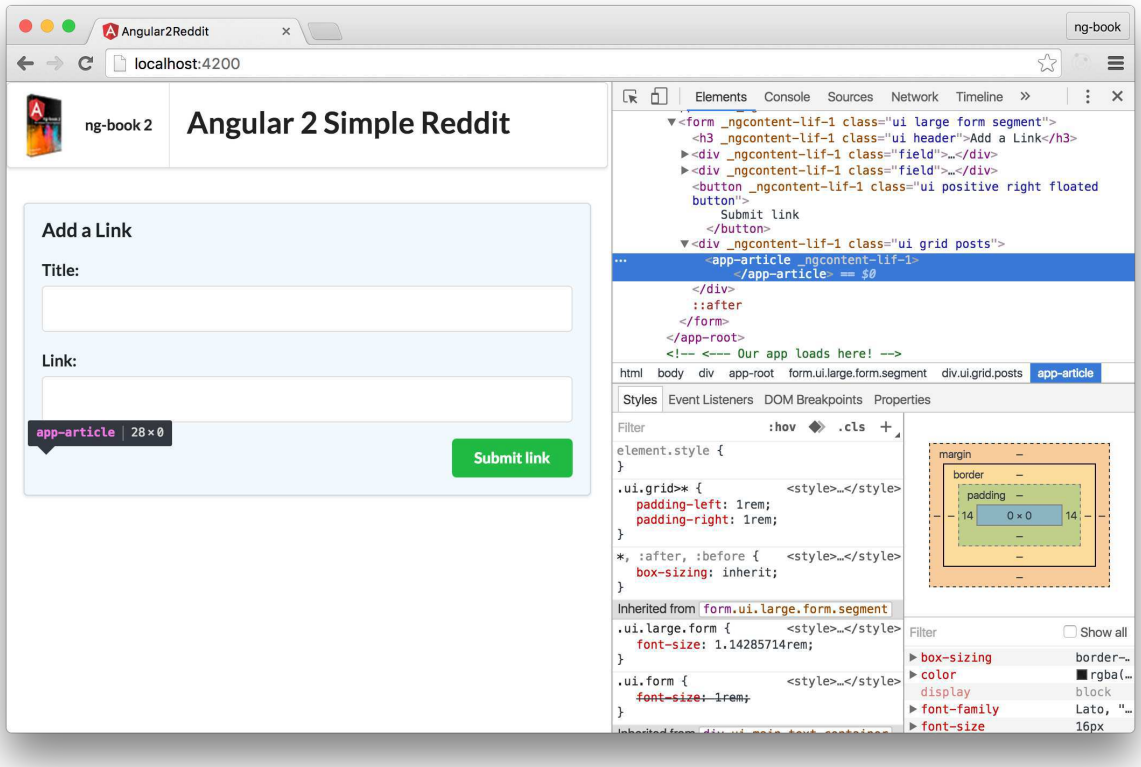
In this case, we want the `AppComponent` to render this new component, so let's update the code in that component. Add the `<app-article>` tag to the `AppComponent`'s template right after the closing `</form>` tag:

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

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

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

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



Unexpanded tag when inspecting the DOM

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

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

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

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

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

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

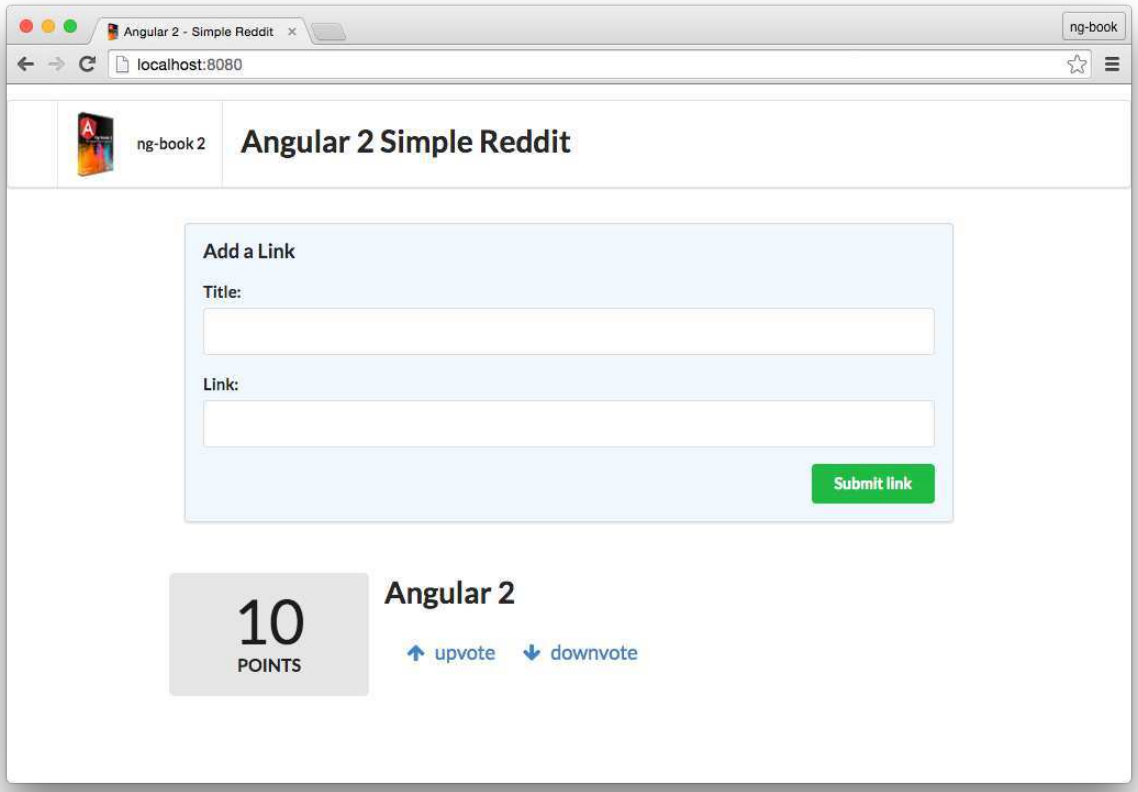
code/first_app/angular2_reddit/src/app/app.module.ts

```
6  import { AppComponent } from './app.component';
7  import { ArticleComponent } from './article/article.component.ts';
8
9  @NgModule({
10   declarations: [
11     AppComponent,
12     ArticleComponent // <-- added this
13   ],
```

See here that we are:

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

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



Rendered ArticleComponent component

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

JavaScript, by default, **propagates the `click` event to all the parent components**. Because the `click` event is propagated to parents, our browser is trying to follow the empty link, which tells the browser to reload.

To fix that, we need to make the click event handler to return `false`. This will ensure the browser won't try to refresh the page. Let's update our code so that each of the functions `voteUp()` and `voteDown()` return a boolean value of `false` (tells the browser *not* to propagate the event upwards):

```
1  voteDown(): boolean {
2    this.votes -= 1;
3    return false;
4  }
5  // and similarly with `voteUp()`
```

Now when we click the links we'll see that the votes increase and decrease properly without a page refresh.

Rendering Multiple Rows

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

Creating an Article class

A good practice when writing Angular code is to try to isolate the data structures we are using from the component code. To do this, let's create a data structure that represents a single article. Let's add a new file `article.model.ts` to define an `Article` class that we can use.

code/first_app/angular2_reddit/src/app/article/article.model.ts

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

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

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

Each article has a `title`, a `link`, and a total for the `votes`. When creating a new article we need the `title` and the `link`. The `votes` parameter is optional (denoted by the `?` at the end of the name) and defaults to zero.

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
12 export class ArticleComponent implements OnInit {
13   article: Article;
14
15   constructor() {
16     this.article = new Article(
17       'Angular 2',
18       'http://angular.io',
19       10);
20   }
21
22   voteUp(): boolean {
23     this.article.votes += 1;
24     return false;
25   }
26
27   voteDown(): boolean {
28     this.article.votes -= 1;
29     return false;
30   }
31
32   ngOnInit() {
33   }
34
35 }
```

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

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

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

code/first_app/angular2_reddit/src/app/article/article.component.html

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

Reload the browser and everything still works.

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

`voteUp` and `voteDown` currently break the Law of Demeter[16] which says that a given object should assume as little as possible about the structure or properties of other objects.

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

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

code/first_app/angular2_reddit/src/app/article/article.model.ts

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

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

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

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

  domain(): string {
    try {
      const link: string = this.link.split('//')[1];
      return link.split('/')[0];
    } catch (err) {
      return null;
    }
  }
}
```

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular 2',
      'http://angular.io',
      10);
```

```
20    }
21
22    voteUp(): boolean {
23      this.article.voteUp();
24      return false;
25    }
26
27    voteDown(): boolean {
28      this.article.voteDown();
29      return false;
30    }
31
32    ngOnInit() {
33    }
34
35  }
```

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

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

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

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

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

Checkout our `ArticleComponent` component definition now: it's so short! We've moved a lot of logic **out** of our component and into our models. The corresponding MVC guideline here might be Fat Models, Skinny Controllers[17]. The idea is that we want to move most of our domain logic to our models so that our components do the minimum work possible.

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

Storing Multiple `Article`s

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

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

code/first_app/angular2_reddit/src/app/app.component.ts

```
9  export class AppComponent {
10   articles: Article[];
11
12   constructor() {
13     this.articles = [
14       new Article('Angular 2', 'http://angular.io', 3),
15       new Article('Fullstack', 'http://fullstack.io', 2),
16       new Article('Angular Homepage', 'http://angular.io', 1),
17     ];
18   }
19
20   addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
21     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
22     this.articles.push(new Article(title.value, link.value, 0));
23     title.value = '';
24     link.value = '';
25     return false;
26   }
27 }
```

Notice that our `AppComponent` has the line:

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

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

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

code/first_app/angular2_reddit/src/app/app.component.ts

```
12   constructor() {
13     this.articles = [
14       new Article('Angular 2', 'http://angular.io', 3),
15       new Article('Fullstack', 'http://fullstack.io', 2),
16       new Article('Angular Homepage', 'http://angular.io', 1),
17     ];
18   }
```

Configuring the `ArticleComponent` with `inputs`

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

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
12 export class ArticleComponent implements OnInit {
13   article: Article;
14
15   constructor() {
16     this.article = new Article(
17       'Angular 2',
18       'http://angular.io',
19       10);
20   }
```

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

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

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

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

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

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

```
<app-article [article]="myArticle"></app-article>
```

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

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

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

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @Input() article: Article;

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

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

  ngOnInit() {
  }

}
```

Rendering a List of Articles

Earlier we configured our `AppComponent` to store an array of `articles`. Now let's configure `AppComponent` to *render* all the `articles`. To do so, instead of having the `<app-article>` tag alone,

we are going to use the NgFor directive to iterate over the list of articles and render a app-article for each one:

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

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

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

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

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

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

We are using the variable article many times in that previous code snippet, it's (potentially) clearer if we rename the temporary variable created by NgFor to foobar:

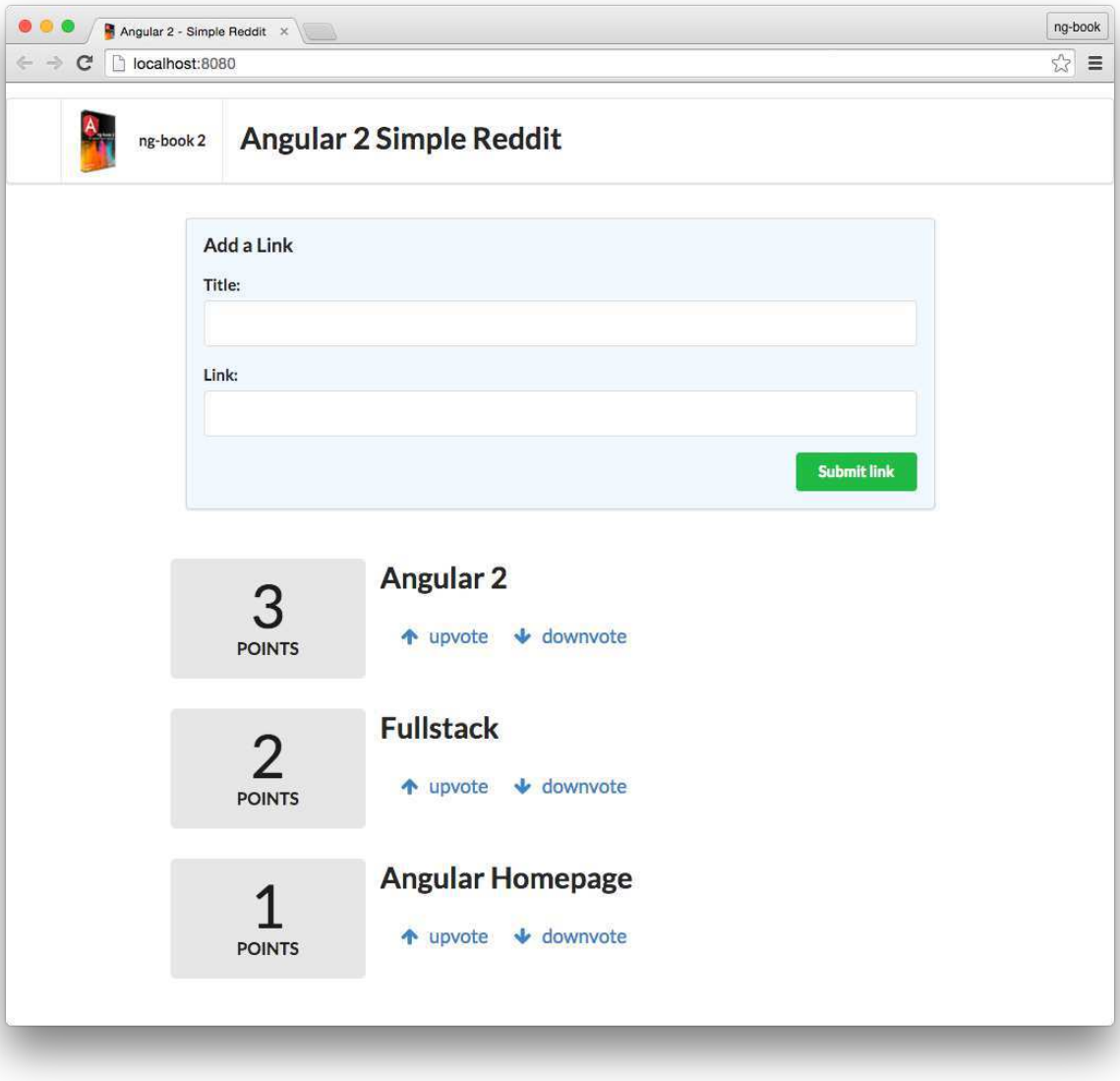
```
  <app-article
    *ngFor="let foobar of articles"
    [article]="foobar">
  </app-article>
```

So here we have three variables:

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

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

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



Multiple articles being rendered

Adding New `Article`s

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

code/first_app/angular2_reddit/src/app/app.component.ts

```
20   addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
21     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
22     this.articles.push(new Article(title.value, link.value, 0));
23     title.value = '';
24     link.value = '';
25     return false;
26   }
```

This will:

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

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

After adding a new article in our input fields and clicking the **Submit Link** we will see the new article added!

Finishing Touches

Displaying the Article Domain

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

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

code/first_app/angular2_reddit/src/app/article/article.model.ts

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

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

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

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

Re-sorting Based on Score

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

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

code/first_app/angular2_reddit/src/app/app.component.ts

```
28   sortedArticles(): Article[] {
29     return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
30   }
```

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

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

Full Code Listing

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

Wrapping Up

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

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

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

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

Getting Help

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

- Come join our (free!) community and chat with us on Gitter[18]
- Email us directly at us@fullstack.io[19]

Onward!

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

This is the end of the preview!

Head over to ng-book.com/2 to download the full package.

The full package includes the book, the source code, and a video screencast!