

Writing Your First Angular Web Application

Simple Reddit Clone

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

You can think of this app as the beginnings of a site like [Reddit](http://reddit.com)¹⁴ or [Product Hunt](http://producthunt.com)¹⁵.

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

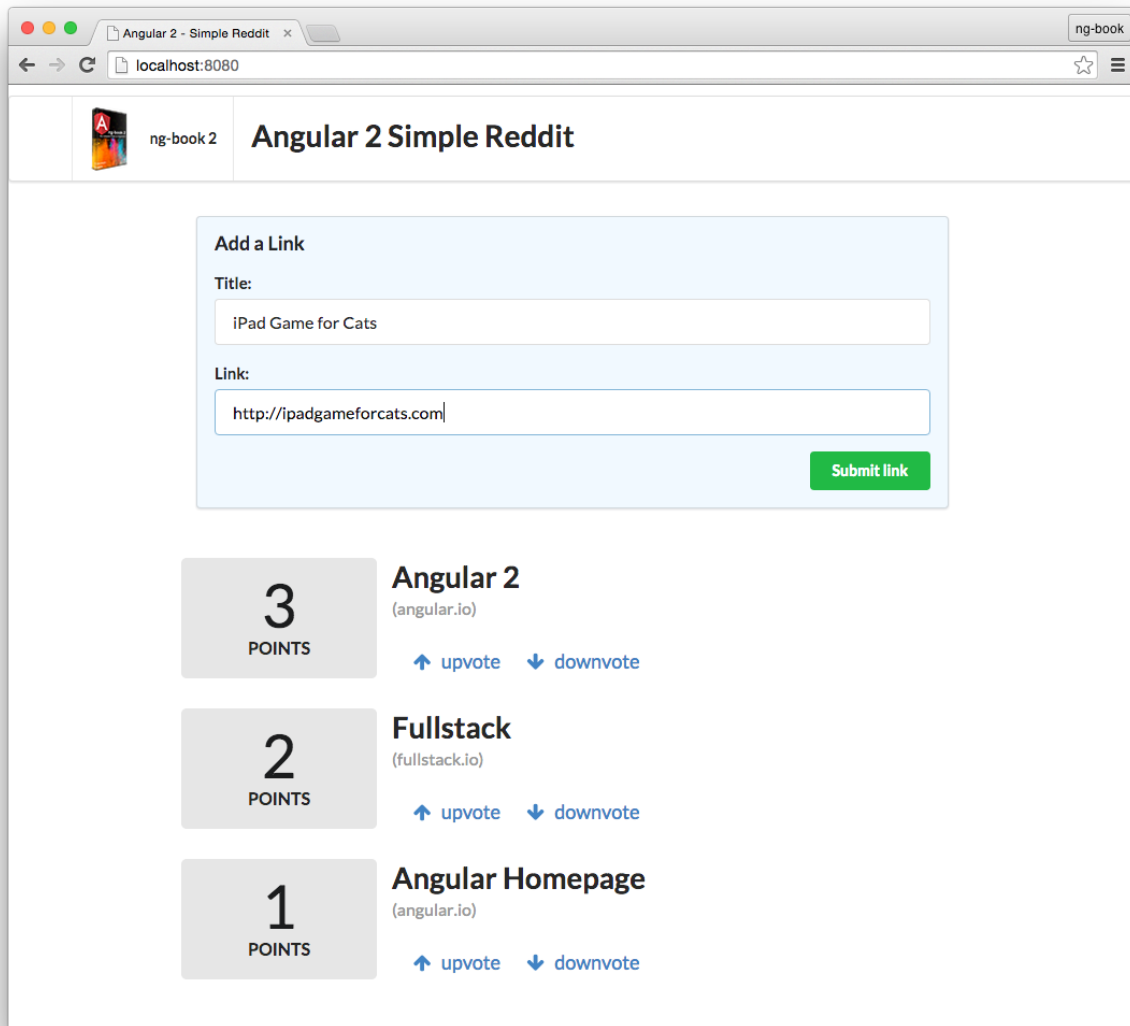
- Building custom components
- Accepting user input from forms
- Rendering lists of objects into views
- Intercepting user clicks and acting on them
- Deploying our app to a server

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

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

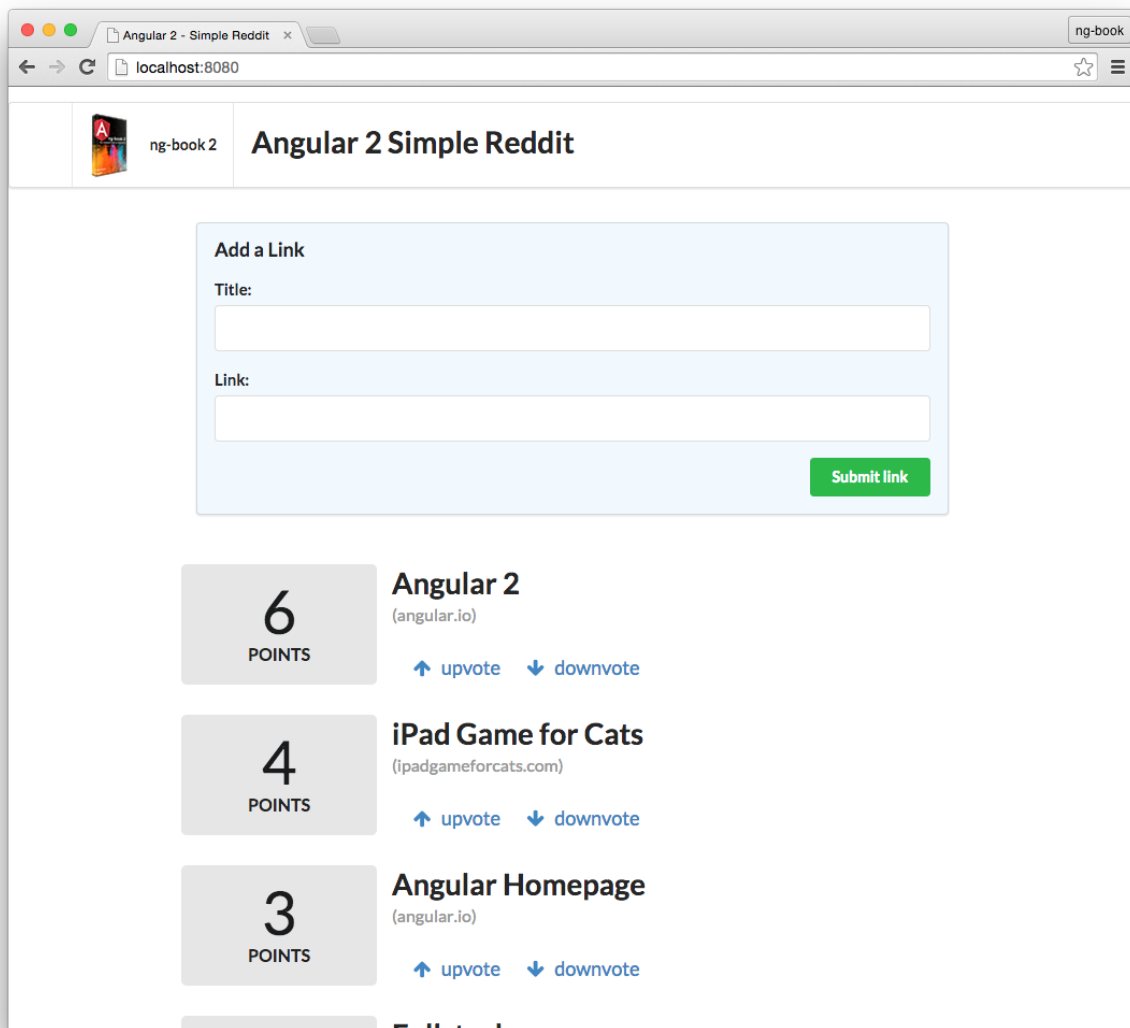
¹⁴<http://reddit.com>

¹⁵<http://producthunt.com>



Completed application

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



App with new article

In this project, and throughout the book, we're going to use TypeScript. TypeScript is a superset of JavaScript ES6 that adds types. We're not going to talk about TypeScript in depth in this chapter, but we'll go over TypeScript more in depth in [the next chapter](#).

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

Getting started

Node.js and npm

To get started with Angular, you'll need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to [the Node.js website](#)¹⁶ for detailed information.

Make sure you install Node 6.9.0 or higher.



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

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

```
$ npm -v
```

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

Your npm version should be 3.0.0 or higher.

TypeScript

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

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



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

Browser

We highly recommend using the [Google Chrome Web Browser](#)¹⁷ to develop Angular apps. We'll use the Chrome developer toolkit throughout this book. To follow along with our development and debugging we recommend downloading it now.

¹⁶<https://nodejs.org/download/>

¹⁷<https://www.google.com/chrome/>

Special instruction for Windows users

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

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



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

Ensure IIS is installed (Windows)

If you're on a Windows machine and have yet to do any web development on it, you may need to install IIS (Internet Information Services) in order to run web servers locally.

See [this tutorial](#)¹⁸ for installing IIS.

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
```

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:

```
1 $ ng --version
```

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

¹⁸<http://www.howtogeek.com/112455/how-to-install-iis-8-on-windows-8/>



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

```
1 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:

```
1 $ brew install watchman
```



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

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

If you're on Windows instead, you don't need to install anything and Angular CLI will use the native Node.js watcher.

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

```
1 $ ng --help
```

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

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

Example Project

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

```
1 $ ng new --ng4 angular-hello-world
```



If you're using a version of Angular CLI that is *newer* than `@angular/cli@1.0.0-rc.4`, you may omit the `--ng4` option as Angular 4 will be the default.

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

```
1 installing ng2
2   create .editorconfig
3   create README.md
4   create src/app/app.component.css
5   create src/app/app.component.html
6   create src/app/app.component.spec.ts
7   create src/app/app.component.ts
8   create src/app/app.module.ts
9   create src/assets/.gitkeep
10  create src/environments/environment.prod.ts
11  create src/environments/environment.ts
12  create src/favicon.ico
13  create src/index.html
14  create src/main.ts
15  create src/polyfills.ts
16  create src/styles.css
17  create src/test.ts
18  create src/tsconfig.json
19  create .angular-cli.json
20  create e2e/app.e2e-spec.ts
21  create e2e/app.po.ts
22  create e2e/tsconfig.json
23  create .gitignore
24  create karma.conf.js
25  create package.json
26  create protractor.conf.js
27  create tslint.json
28 Successfully initialized git.
29 Installing packages for tooling via npm.
30 Installed packages for tooling via npm.
```



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

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

```
1 Project 'angular-hello-world' successfully created.
```

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

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

```
1 $ cd angular-hello-world
2 $ tree -F -L 1
3 .
4 ├── README.md           // an useful README
5 ├── .angular-cli.json   // angular-cli configuration file
6 ├── e2e/                // end to end tests
7 ├── karma.conf.js       // unit test configuration
8 ├── node_modules/       // installed dependencies
9 ├── package.json        // npm configuration
10 ├── protractor.conf.js  // e2e test configuration
11 ├── src/                // application source
12 └── tslint.json         // linter config file
```



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

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

```
1 $ cd src
2 $ tree -F
3 .
4 |-- app/
5 |   |-- app.component.css
6 |   |-- app.component.html
7 |   |-- app.component.spec.ts
8 |   |-- app.component.ts
9 |   `-- app.module.ts
10 |-- assets/
11 |-- environments/
12 |   |-- environment.prod.ts
13 |   `-- environment.ts
14 |-- favicon.ico
15 |-- index.html
16 |-- main.ts
17 |-- polyfills.ts
18 |-- styles.css
19 |-- test.ts
20 `-- tsconfig.json
```


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

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

```
1  <!doctype html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title>AngularHelloWorld</title>
6    <base href="/">
7
8    <meta name="viewport" content="width=device-width, initial-scale=1">
9    <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root>Loading...</app-root>
13 </body>
14 </html>
```

Let's break it down a bit:

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

```
1  <!doctype html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title>AngularHelloWorld</title>
6    <base href="/">
7
8    <meta name="viewport" content="width=device-width, initial-scale=1">
9    <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
```

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

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

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

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

The `app-root` tag is **where our application will be rendered**. The text `Loading...` is a placeholder that will be displayed *before our app code loads*. For instance, we could put a loading “spinner” `img` tag here and the user would see this as our JavaScript and Angular app is loading.

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

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

Writing Application Code

Running the application

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

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

```
1 $ cd angular-hello-world
2 $ ng serve
3 ** NG Live Development Server is running on http://localhost:4200. **
4 // ...
5 // a bunch of other messages
6 // ...
7 Compiled successfully.
```

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

<http://localhost:4200>¹⁹

¹⁹<http://localhost:4200>



Note that if you get the message:

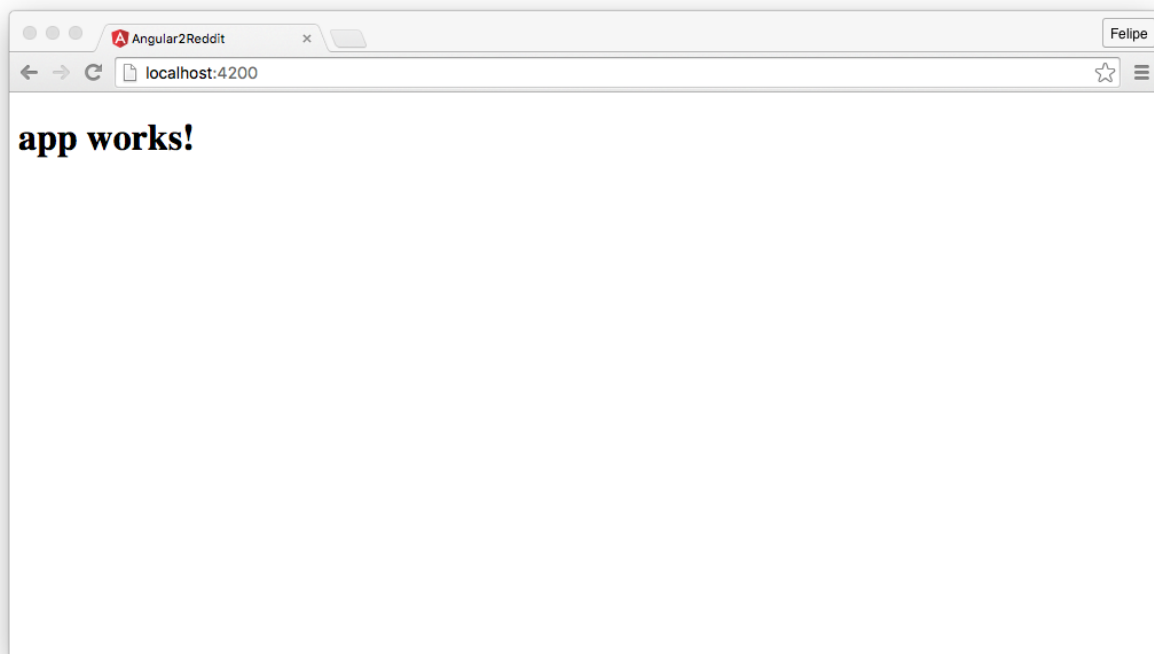
```
1 Port 4200 is already in use. Use '--port' to specify a different port
```

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

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

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

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



Running application

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

Making a Component

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

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

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

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



If you have a background in AngularJS 1.X, you can think of **components as the new version of directives**.

Let's create our very first component. When we have this component written, we will be able to use it in our HTML document using the `app-hello-world` tag:

```
1 <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:

```
1 $ ng generate component hello-world
2 installing component
3   create src/app/hello-world/hello-world.component.css
4   create src/app/hello-world/hello-world.component.html
5   create src/app/hello-world/hello-world.component.spec.ts
6   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 decorator
2. A component definition class

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

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

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-hello-world',
5   templateUrl: './hello-world.component.html',
6   styleUrls: ['./hello-world.component.css']
7 })
8 export class HelloWorldComponent implements OnInit {
9
10  constructor() { }
11
12  ngOnInit() {
13  }
14
15 }
```

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



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

Importing Dependencies

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

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

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

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

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

Component Decorators

After importing our dependencies, we are declaring the component:

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

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

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

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

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

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

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

```
1 @Component({
2   selector: 'app-hello-world'
3   // ... more here
4 })
```

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

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

Adding a template with `templateUrl`

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

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

```
1 <p>
2   hello-world works!
3 </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:

```
1 @Component({
2   selector: 'app-hello-world',
3   template: `
4     <p>
5       hello-world works inline!
6     </p>
7   `
8 })
```

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



Should 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 to 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`:

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

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

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



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

Loading Our Component

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

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

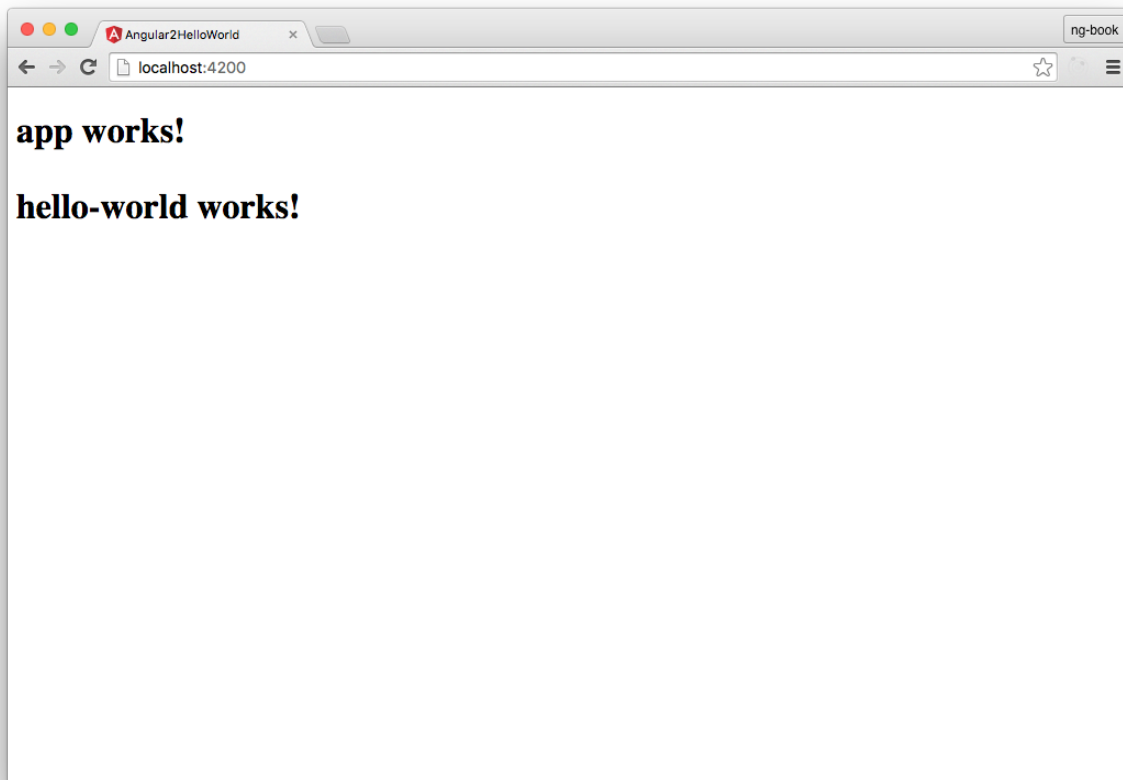
In order to change that, we need to add our component tag to a template that is already being rendered. Open up the file: `first_app/angular-hello-world/src/app/app.component.html`

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

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

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

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

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

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

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

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

```
8 export class UserItemComponent implements OnInit {
9   name: string; // <-- added name property
10
11   constructor() {
12     this.name = 'Felipe'; // set the name
13   }
14
15   ngOnInit() {
16   }
17
18 }
```

Notice that we've changed two things:

1. name Property

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

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

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

2. A Constructor

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

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

When we write:

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

```
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

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

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

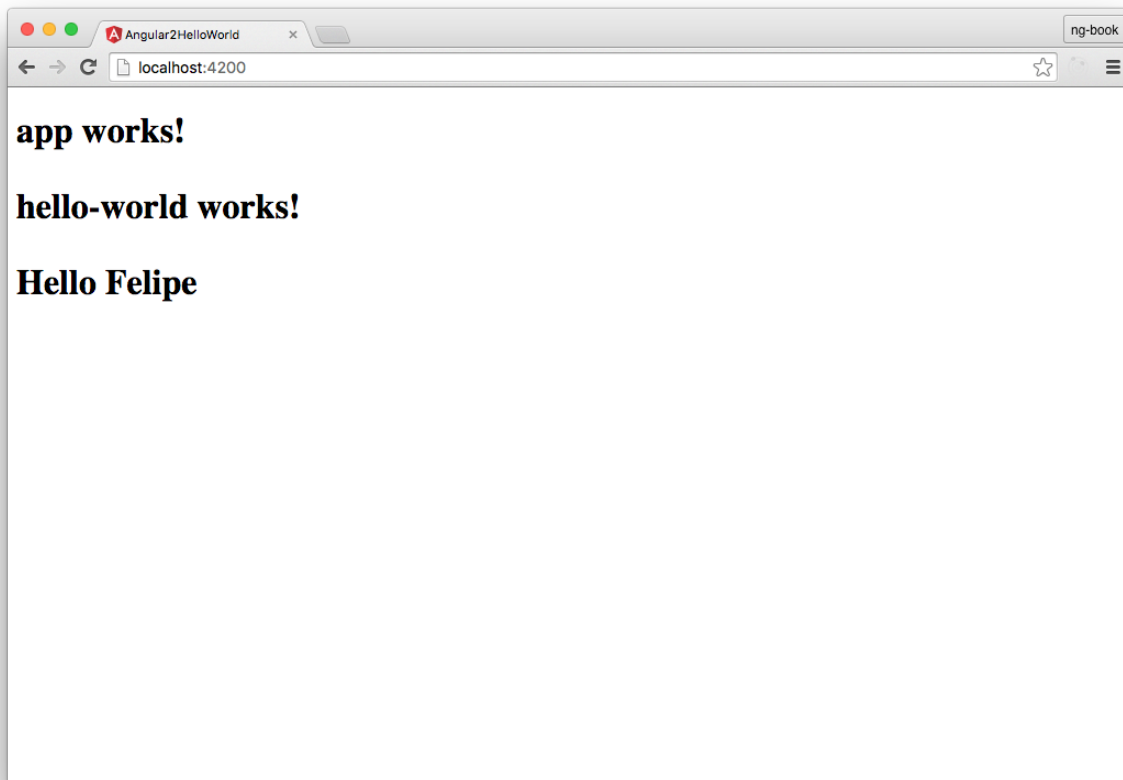
```
1 <p>  
2   Hello {{ name }}  
3 </p>
```

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

Whatever is between the template tags will be expanded as an *expression*. Here, because the template is *bound* to our Component, the `name` will expand to the value of `this.name` i.e. 'Felipe'.

Try It Out

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



Application with Data

Working With Arrays

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

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



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

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

```
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/angular-hello-world/src/app/app.component.html`

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

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

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

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

```
8 export class UserListComponent implements OnInit {
9   names: string[];
10
11   constructor() {
12     this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
13   }
14
15   ngOnInit() {
16   }
17
18 }
```

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

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

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

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

Here's what our new template will look like:

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

```

1 <ul>
2   <li *ngFor="let name of names">Hello {{ name }}</li>
3 </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 `UserList-Component` 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:

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

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

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

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



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

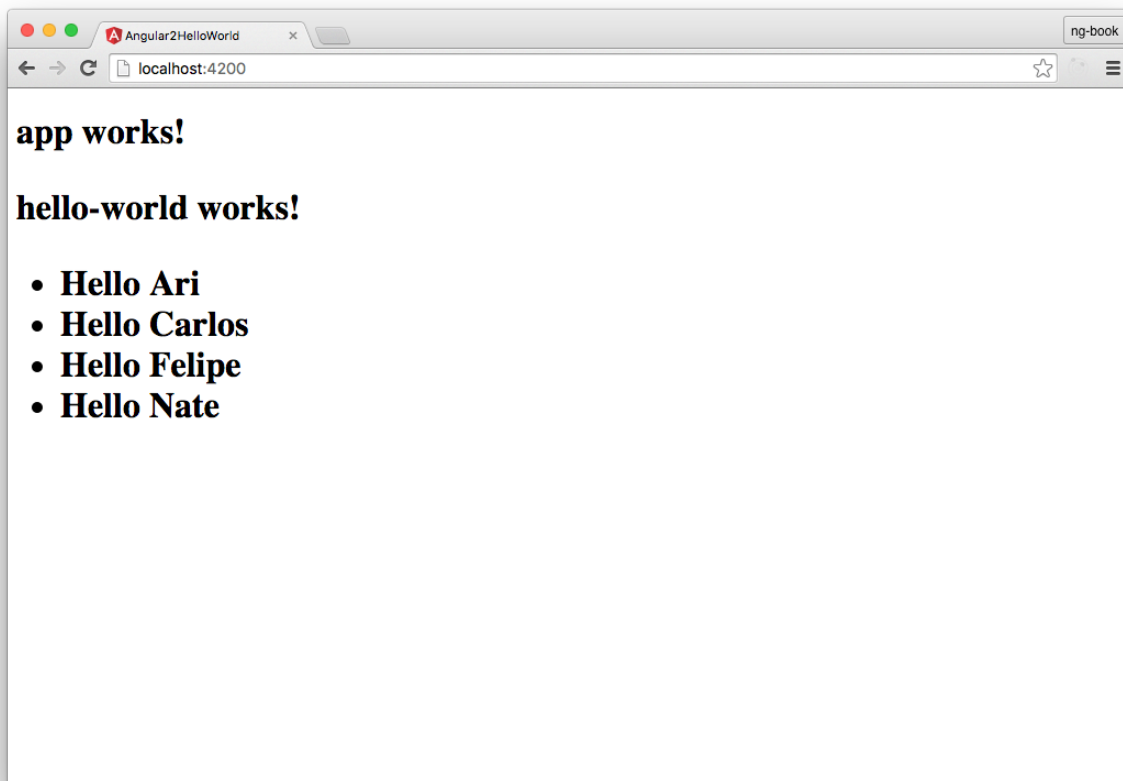
Note that the capitalization here isn't a typo: `NgFor` is the capitalization of the *class* that implements the logic and `ngFor` is the "selector" for the attribute we want to use.



If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the [NgFor directive here](https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.ts)²⁰.

²⁰https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.ts

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



Application with Data

Using the User Item Component

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

To do this, we need to do three things:

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

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

Rendering the `UserItemComponent`

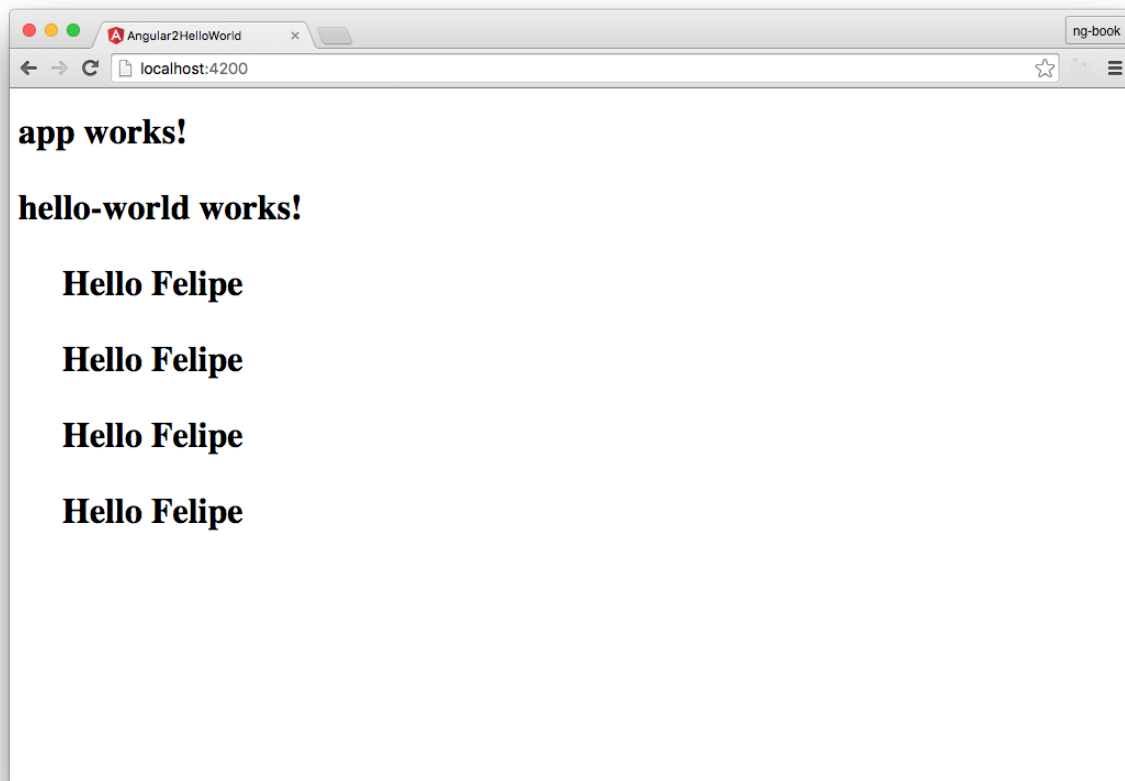
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template:

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

```
1 <ul>
2   <li *ngFor="let name of names">
3     <app-user-item></app-user-item>
4   </li>
5 </ul>
```

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

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



Application with Data

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

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

Accepting Inputs

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

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

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

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

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

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

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

Passing an Input value

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

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

```
1 <ul>
2   <li *ngFor="let name of names">
3     <app-user-item [name]="name"></app-user-item>
4   </li>
5 </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:

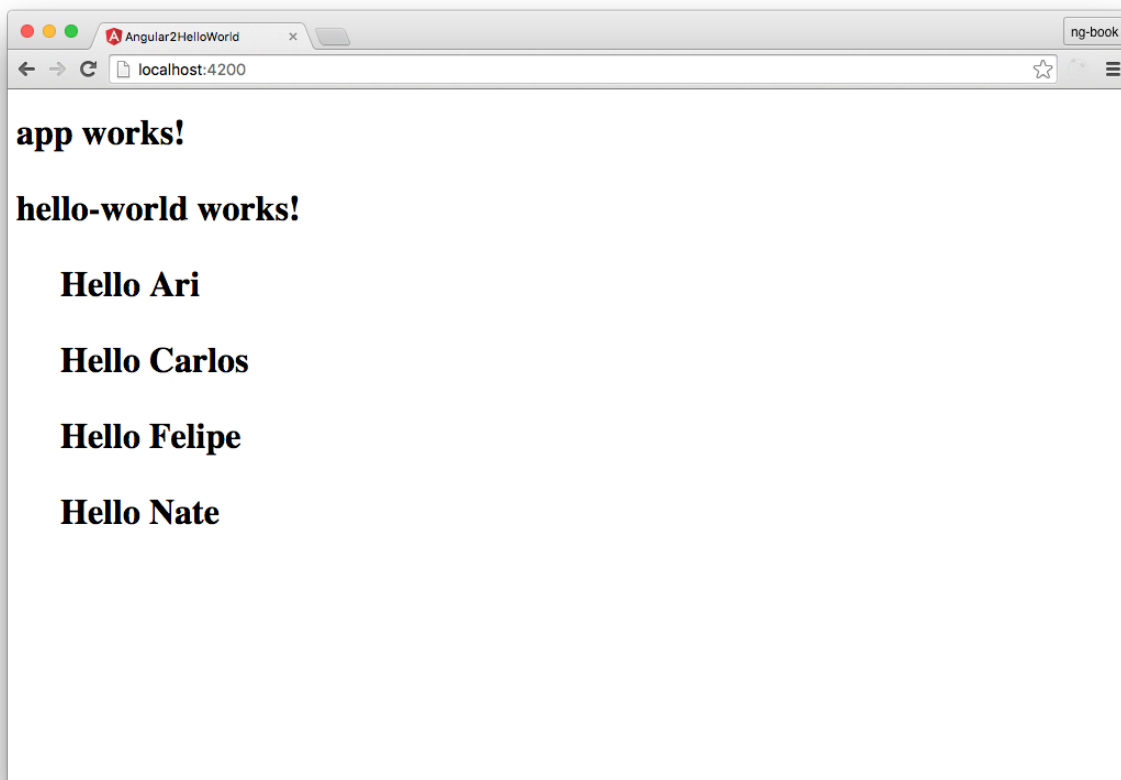
```
1 <li *ngFor="let individualUserName of names">
2   <app-user-item [name]="individualUserName"></app-user-item>
3 </li>
```

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

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

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

Now our list of names works!



Application with Names Working

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

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

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

Bootstrapping Crash Course

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

```
1 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.

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

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

Take a look at this code:

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

```
11 @NgModule({
12   declarations: [
13     AppComponent,
14     HelloWorldComponent,
15     UserItemComponent,
16     UserListComponent
17   ],
18   imports: [
19     BrowserModule,
20     FormsModule,
21     HttpClientModule
22   ],
23   providers: [],
24   bootstrap: [AppComponent]
25 })
26 export class AppModule { }
```

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

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

declarations

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

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

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

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

imports

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

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



`import` vs. `imports`?

You might be asking the question, “What’s the difference between importing a class at the top of the file and putting a module in `imports`?”

The short answer is that you put something in your `NgModule`’s `imports` if you’re going to be using it in your templates or with *dependency injection*. We haven’t talked about *dependency injection*, but rest assured, we will.

providers

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



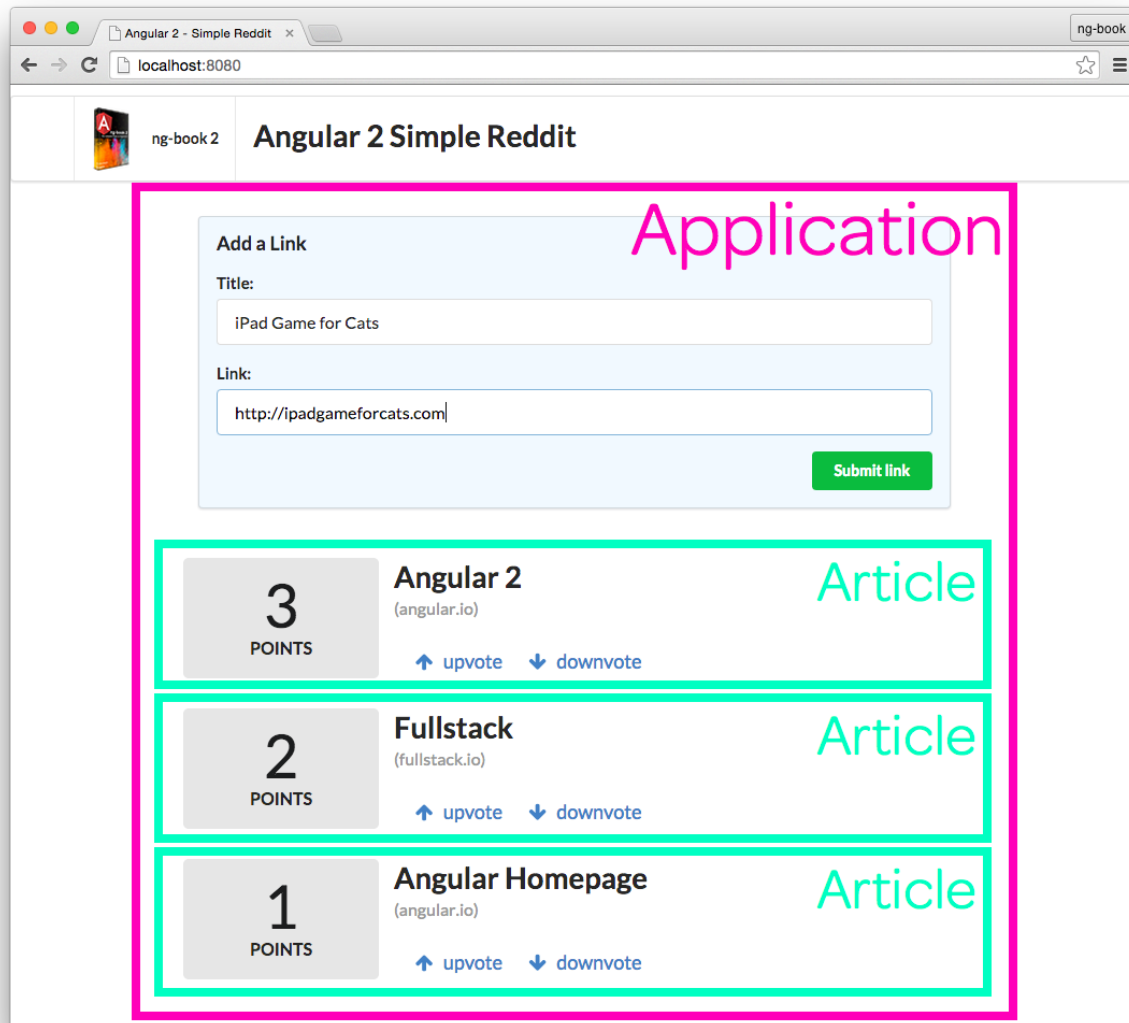
Learn more about this in [the section on Dependency Injection](#).

bootstrap

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

Expanding our Application

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



Application with Data

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

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



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

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

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

```
1 ng new angular-reddit
```



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

Adding CSS

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



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

Copy:

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

into your application's folder.

For this project we're going to be using [Semantic-UI](http://semantic-ui.com/)²¹ to help with the styling. Semantic-UI is a CSS framework, similar to [Zurb Foundation](http://foundation.zurb.com)²² or [Twitter Bootstrap](http://getbootstrap.com)²³. We've included it in the sample code download so all you need to do is copy over the files specified above.

²¹<http://semantic-ui.com/>

²²<http://foundation.zurb.com>

²³<http://getbootstrap.com>

The Application Component

Let's now build a new component which will:

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

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

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

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'app works!';
10 }
```



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

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

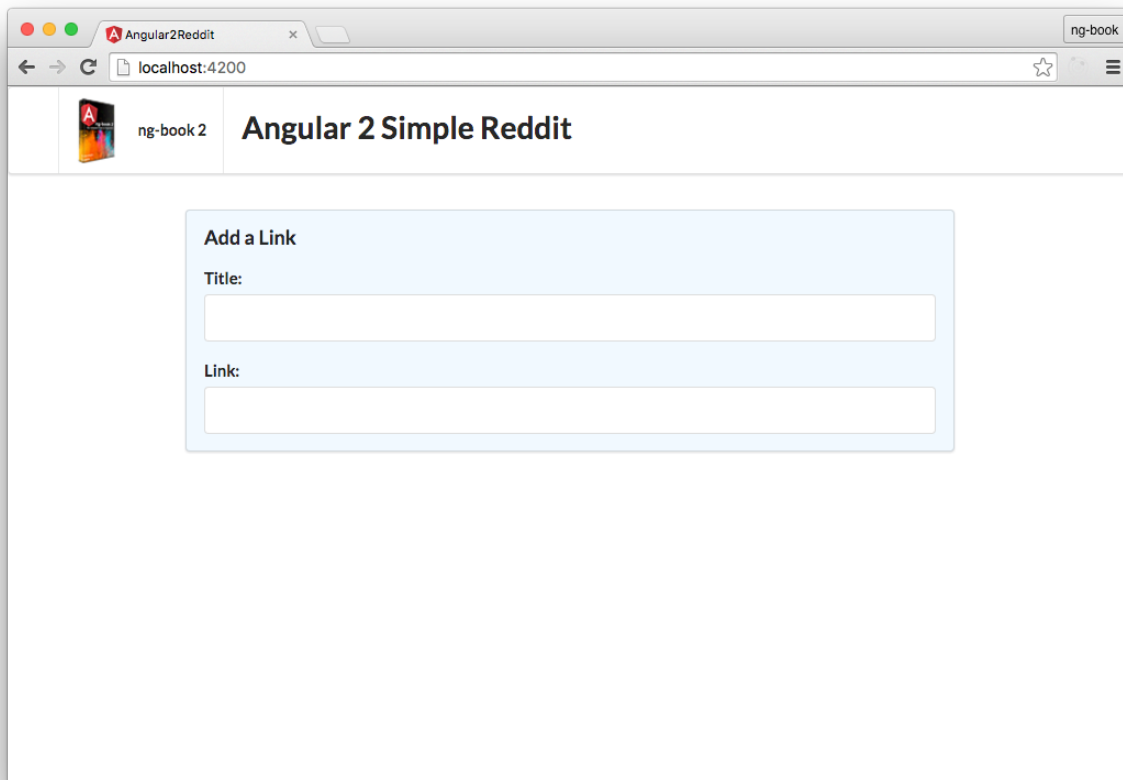
Let's change the template a bit to include a form for adding links. We'll use a bit of styling from the `semantic-ui` package to make the form look a bit nicer:

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

```
1 <form class="ui large form segment">
2   <h3 class="ui header">Add a Link</h3>
3
4   <div class="field">
5     <label for="title">Title:</label>
6     <input name="title">
7   </div>
8   <div class="field">
9     <label for="link">Link:</label>
10    <input name="link">
11  </div>
12 </form>
```

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

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



Form

Adding Interaction

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

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

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

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

```
8 export class AppComponent {  
9   addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {  
10     console.log(`Adding article title: ${title.value} and link: ${link.value}`);  
11     return false;  
12   }  
13 }
```

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

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

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

```
1 <form class="ui large form segment">  
2   <h3 class="ui header">Add a Link</h3>  
3  
4   <div class="field">  
5     <label for="title">Title:</label>  
6     <input name="title" #newtitle> <!-- changed -->  
7   </div>  
8   <div class="field">  
9     <label for="link">Link:</label>  
10    <input name="link" #newlink> <!-- changed -->  
11  </div>  
12  
13  <!-- added this button -->  
14  <button (click)="addArticle(newtitle, newlink)"  
15    class="ui positive right floated button">  
16    Submit link  
17  </button>  
18  
19 </form>
```

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

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

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

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

Binding inputs to values

Notice in our first input tag we have the following:

```
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:

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



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

Defining the Action Logic

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

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

```
9   addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
10     console.log(`Adding article title: ${title.value} and link: ${link.value}`);
11     return false;
12   }
```

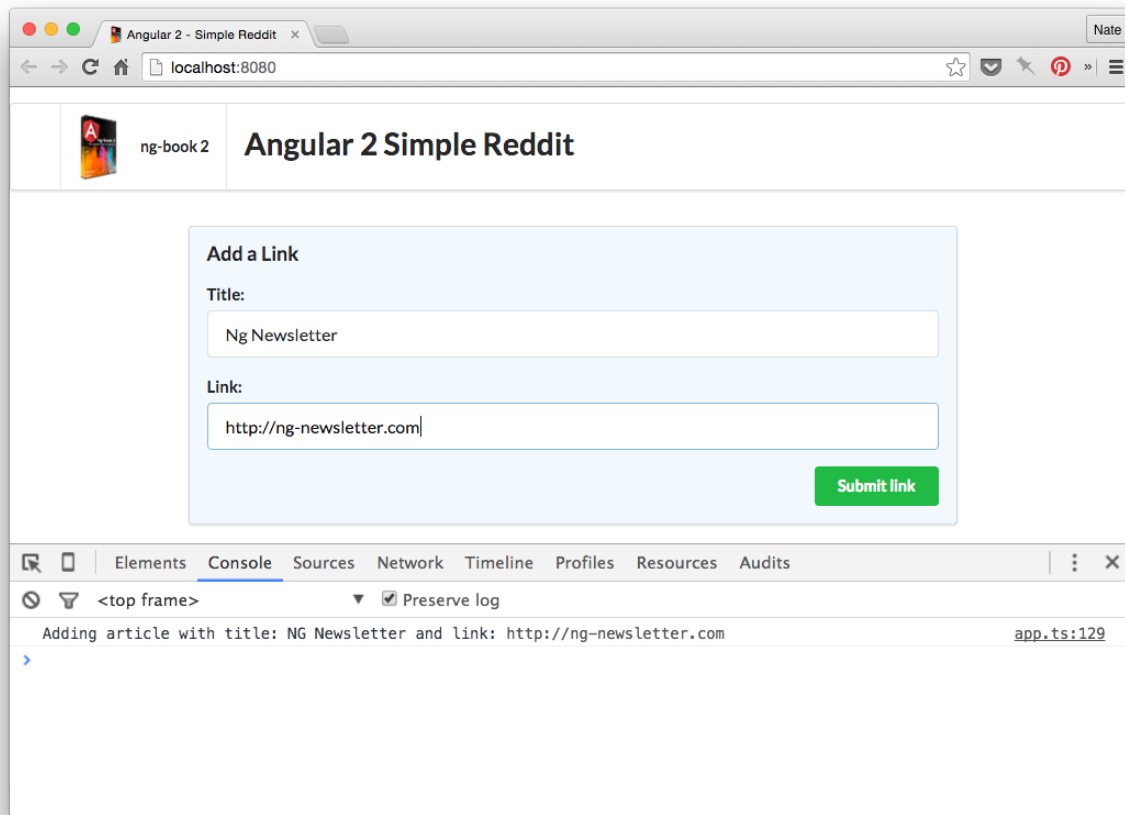


Notice that we're using backtick strings again. This is a really handy feature of ES6: backtick strings will expand template variables!

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:

1 ng generate component article

We have three parts to defining this new component:

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

Let's talk through each part in detail:

Creating the ArticleComponent template

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

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

```
1 <div class="four wide column center aligned votes">
2   <div class="ui statistic">
3     <div class="value">
4       {{ votes }}
5     </div>
6     <div class="label">
7       Points
8     </div>
9   </div>
10 </div>
11 <div class="twelve wide column">
12   <a class="ui large header" href="{{ link }}">
13     {{ title }}
14   </a>
15   <ul class="ui big horizontal list voters">
16     <li class="item">
17       <a href (click)="voteUp()">
18         <i class="arrow up icon"></i>
19         upvote
20       </a>
21     </li>
22     <li class="item">
23       <a href (click)="voteDown()">
24         <i class="arrow down icon"></i>
25         downvote
26       </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/angular-reddit/src/app/article/article.component.ts

```
7 @Component({
8   selector: 'app-article',
9   templateUrl: './article.component.html',
10  styleUrls: ['./article.component.css'],
11 })
```

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.

Creating the ArticleComponent Definition Class

Finally, we create the ArticleComponent definition class:

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

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

```
31
32   ngOnInit() {
33   }
34
35 }
```

Here we create four properties on `ArticleComponent`:

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

We want each app-article to be on its own row. We’re using Semantic UI, and Semantic provides a [CSS class for rows](http://semantic-ui.com/collections/grid.html)²⁴ called `row`.

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



We import `HostBinding` from the package `@angular/core`. For instance we can add `HostBinding` like this:

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

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



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

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

²⁴<http://semantic-ui.com/collections/grid.html>

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

```
18 constructor() {  
19     this.title = 'Angular 2';  
20     this.link = 'http://angular.io';  
21     this.votes = 10;  
22 }
```

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

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

```
24 voteUp() {  
25     this.votes += 1;  
26 }  
27  
28 voteDown() {  
29     this.votes -= 1;  
30 }
```

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

Using the `app-article` Component

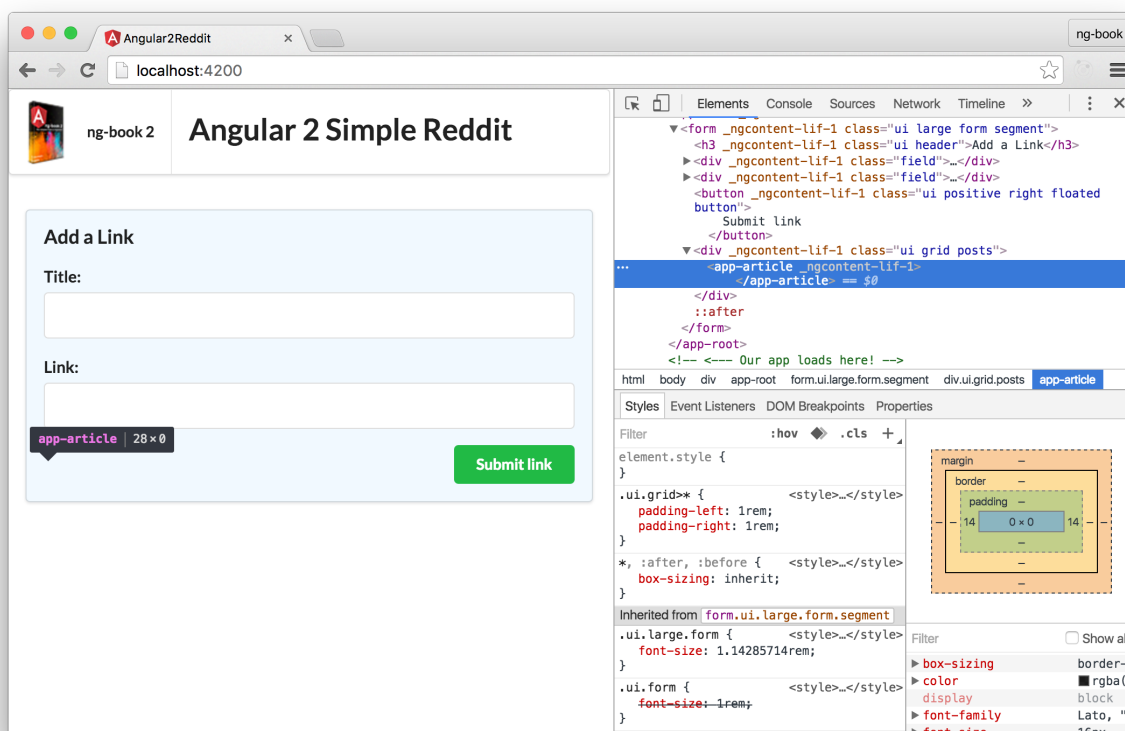
In order to use this component and make the data visible, we have to add a `<app-article></app-article>` tag somewhere in our markup.

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:

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

If we generated the `ArticleComponent` using Angular CLI (via `ng generate component`), by default it should have “told” Angular about our `app-article` tag (more on that below). However, if we created this component “by hand” and we reload the browser now, we might see that the `<app-article>` tag wasn’t compiled. Oh no!

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



Unexpanded tag when inspecting the DOM

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



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

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

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



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

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

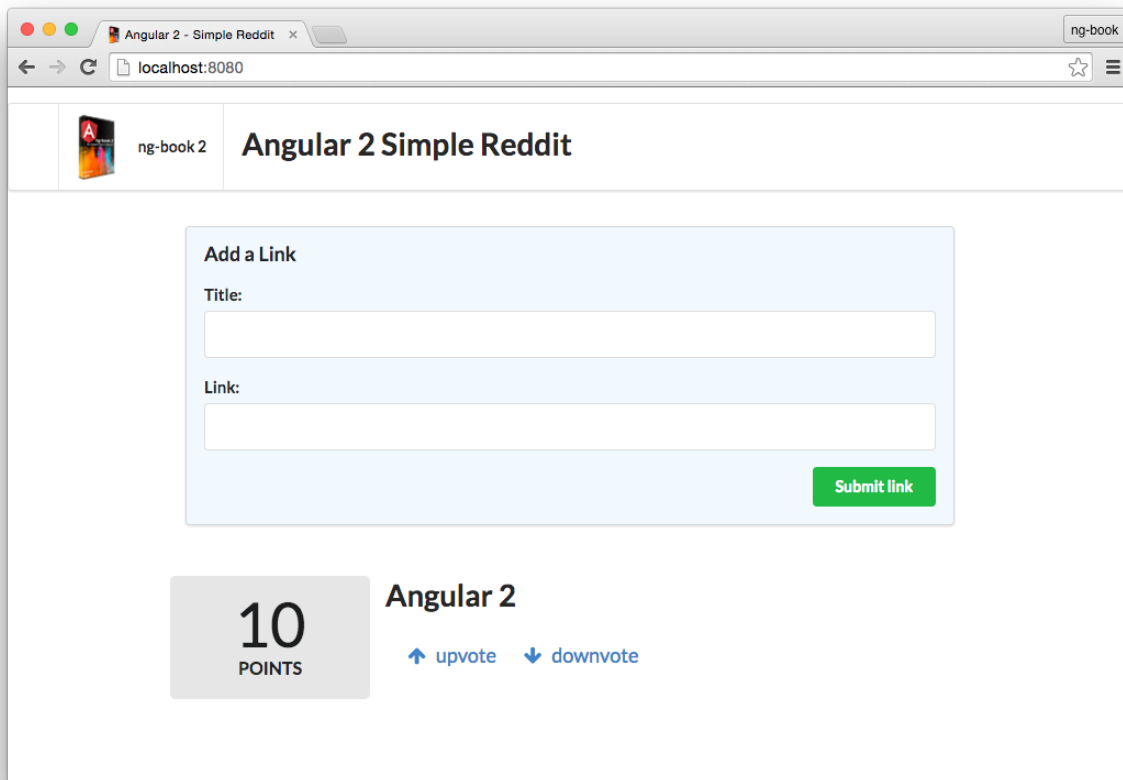
code/first-app/angular-reddit/src/app/app.module.ts

```
6 import { AppComponent } from './app.component';
7 import { ArticleComponent } from './article/article.component';
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/angular-reddit/src/app/article/article.model.ts

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

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

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

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

First let's import the class:

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

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

Then let's use it:

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

```
13 export class ArticleComponent implements OnInit {
14   @HostBinding('attr.class') cssClass = 'row';
15   article: Article;
16
17   constructor() {
18     this.article = new Article(
19       'Angular 2',
20       'http://angular.io',
21       10);
22   }
23
24   voteUp(): boolean {
25     this.article.votes += 1;
26     return false;
27   }
28
29   voteDown(): boolean {
30     this.article.votes -= 1;
31     return false;
32   }
33
34   ngOnInit() {
35   }
36
37 }
```

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

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

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

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

```

1 <div class="four wide column center aligned votes">
2   <div class="ui statistic">
3     <div class="value">
4       {{ article.votes }}
5     </div>
6     <div class="label">
7       Points
8     </div>
9   </div>
10 </div>
11 <div class="twelve wide column">
12   <a class="ui large header" href="{{ article.link }}">
13     {{ article.title }}
14   </a>
15   <ul class="ui big horizontal list voters">
16     <li class="item">
17       <a href (click)="voteUp()">
18         <i class="arrow up icon"></i>
19         upvote
20       </a>
21     </li>
22     <li class="item">
23       <a href (click)="voteDown()">
24         <i class="arrow down icon"></i>
25         downvote
26       </a>
27     </li>
28   </ul>
29 </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](http://en.wikipedia.org/wiki/Law_of_Demeter)²⁵ 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

²⁵http://en.wikipedia.org/wiki/Law_of_Demeter

internals. To fix that, let's add `voteUp` and `voteDown` methods on the `Article` class (we'll also add a domain function, which we'll talk about in a moment):

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

```
1 export class Article {
2   title: string;
3   link: string;
4   votes: number;
5
6   constructor(title: string, link: string, votes?: number) {
7     this.title = title;
8     this.link = link;
9     this.votes = votes || 0;
10  }
11
12  voteUp(): void {
13    this.votes += 1;
14  }
15
16  voteDown(): void {
17    this.votes -= 1;
18  }
19
20  // domain() is a utility function that extracts
21  // the domain from a URL, which we'll explain shortly
22  domain(): string {
23    try {
24      // e.g. http://foo.com/path/to/bar
25      const domainAndPath: string = this.link.split('///')[1];
26      // e.g. foo.com/path/to/bar
27      return domainAndPath.split('/')[0];
28    } catch (err) {
29      return null;
30    }
31  }
32 }
```

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

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

```
13 export class ArticleComponent implements OnInit {
14     @HostBinding('attr.class') cssClass = 'row';
15     article: Article;
16
17     constructor() {
18         this.article = new Article(
19             'Angular 2',
20             'http://angular.io',
21             10);
22     }
23
24     voteUp(): boolean {
25         this.article.voteUp();
26         return false;
27     }
28
29     voteDown(): boolean {
30         this.article.voteDown();
31         return false;
32     }
33
34     ngOnInit() {
35     }
36
37 }
```



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 know 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](#)²⁶. The idea is that we want to move most of our logic to our models so that our components do the minimum work possible.

Storing Multiple Articles

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

```
1 import { Component } from '@angular/core';
2 import { Article } from './article/article.model'; // <-- import this
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.css']
8 })
9 export class AppComponent {
10   articles: Article[]; // <-- component property
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   }
```

Notice that our `AppComponent` has the line:

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

²⁶<http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model>

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

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

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

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

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

```
13  export class ArticleComponent implements OnInit {
14    @HostBinding('attr.class') cssClass = 'row';
15    article: Article;
16
17    constructor() {
18      this.article = new Article(
19        'Angular 2',
20        'http://angular.io',
21        10);
22    }
```

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 decorator on a property of a Component:

```
1 class ArticleComponent {
2   @Input() article: Article;
3   // ...
```

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:

```
1 <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/angular-reddit/src/app/article/article.component.ts

```
1 import {
2   Component,
3   OnInit,
4   Input,      // <-- added,
5   HostBinding
6 } from '@angular/core';
7 import { Article } from './article.model'; // <-- added
8
9 @Component({
10   selector: 'app-article',
11   templateUrl: './article.component.html',
12   styleUrls: ['./article.component.css']
13 })
14 export class ArticleComponent implements OnInit {
15   @HostBinding('attr.class') cssClass = 'row';
16   @Input() article: Article;
17 }
```

```
18 constructor() {
19     // article is populated by the Input now,
20     // so we don't need anything here
21 }
22
23 voteUp(): boolean {
24     this.article.voteUp();
25     return false;
26 }
27
28 voteDown(): boolean {
29     this.article.voteDown();
30     return false;
31 }
32
33 ngOnInit() {
34 }
35
36 }
```

**Don't forget to import!**

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

Rendering a List of Articles

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

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

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

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

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

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

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



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

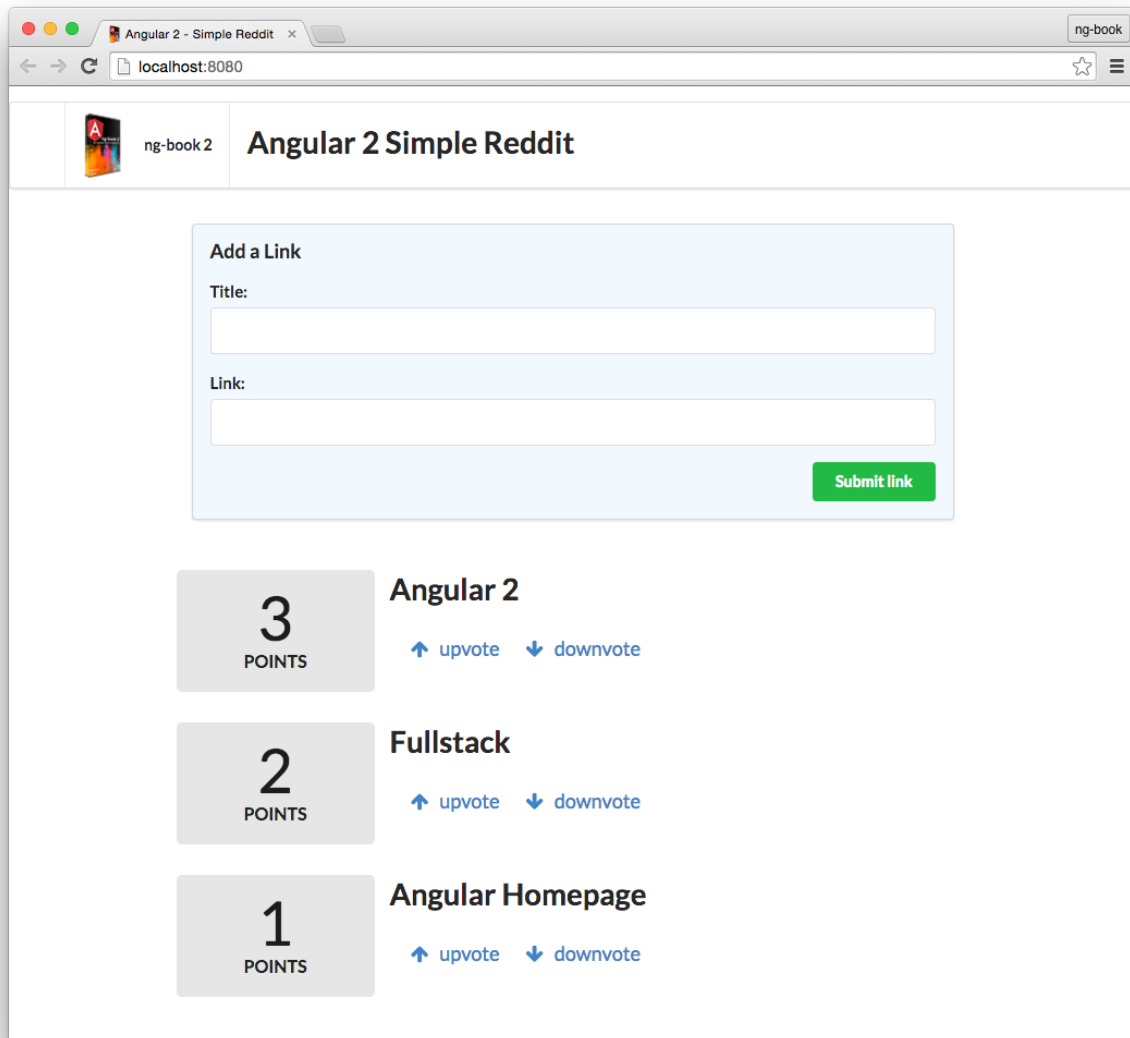
```
1  <app-article
2    *ngFor="let foobar of articles"
3    [article]="foobar">
4  </app-article>
```

So here we have three variables:

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

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

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



Multiple articles being rendered

Adding New Articles

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

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

```
22 domain(): string {  
23   try {  
24     // e.g. http://foo.com/path/to/bar  
25     const domainAndPath: string = this.link.split('//')[1];  
26     // e.g. foo.com/path/to/bar  
27     return domainAndPath.split('/')[0];  
28   } catch (err) {  
29     return null;  
30   }  
31 }
```

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 (note: URL must include *http://*).

Re-sorting Based on Score

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

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

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

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



ES6 Arrow Function

The above code snippet uses “arrow” (`=>`) functions from ES6. You can [read more about arrow functions here](#)²⁷

`sort()` We’re also calling the `sort()` function, which is a built-in which you can [read about here](#)²⁸

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

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

Deployment

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



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

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

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

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

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

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

²⁷https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

²⁸https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

Building Our App for Production

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

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

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

This command tells the `ng` tool to **build** our application for a production environment. We also set the `--base-href` to a single slash `/`.

The `base-href` describes what the ‘root’ URL of our application will be. So, for instance, if you wanted to deploy your app to a subfolder on your server under `/ng-book-demo/`, you could base `--base-href` `'/ng-book-demo/'`

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

```
1 $ ls dist/
2 136B  assets/
3 5.3K  favicon.ico
4 27K   flags.9c74e172f87984c48ddf.png
5 306K  icons.2980083682e94d33a66e.svg
6 119K  icons.706450d7bba6374ca02f.ttf
7 55K   icons.97493d3f11c0a3bd5cbd.woff2
8 70K   icons.d9ee23d59d0e0e727b51.woff
9 59K   icons.f7c2b4b747b1a225eb8d.eot
10 1.1K  index.html
11 1.4K  inline.44deb5fed75ee6385e18.bundle.js
12 17K   main.c683e6eda100e8873d71.bundle.js
13 82K   polyfills.b81504c68200c7bfeb16.bundle.js
14 503K  styles.7f23e351d688b00e8a5b.bundle.css
15 440K  vendor.cc4297c08c0803bddc87.bundle.js
```

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

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

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

Uploading to a Server

There are **lots** of ways to host your HTML and JavaScript. For this demo, we're going to use the easiest way possible: [now](#)²⁹.



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

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

Installing `now`

We can install `now` using `npm`:

```
1 npm install -g now
```

To deploy a site with `now` is very easy:

```
1 cd dist # change into the dist folder
2 now
```

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

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

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

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

Full Code Listing

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

²⁹<https://zeit.co/now>

Wrapping Up

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

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

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

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

Getting Help

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

- Come join our community and [chat with us on Gitter](https://gitter.im/ng-book/ng-book)³⁰
- Email us directly at us@fullstack.io³¹

Onward!

³⁰<https://gitter.im/ng-book/ng-book>

³¹<mailto:us@fullstack.io>