ng-book 2

The Complete Book on AngularJS 2

Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

FULLSTACK.io
gistia

ng-book 2

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

Contents

Book Revision

Revision 1 - Covers up to Angular 2 (ba440a0, 2015-06-02)

Prerelease

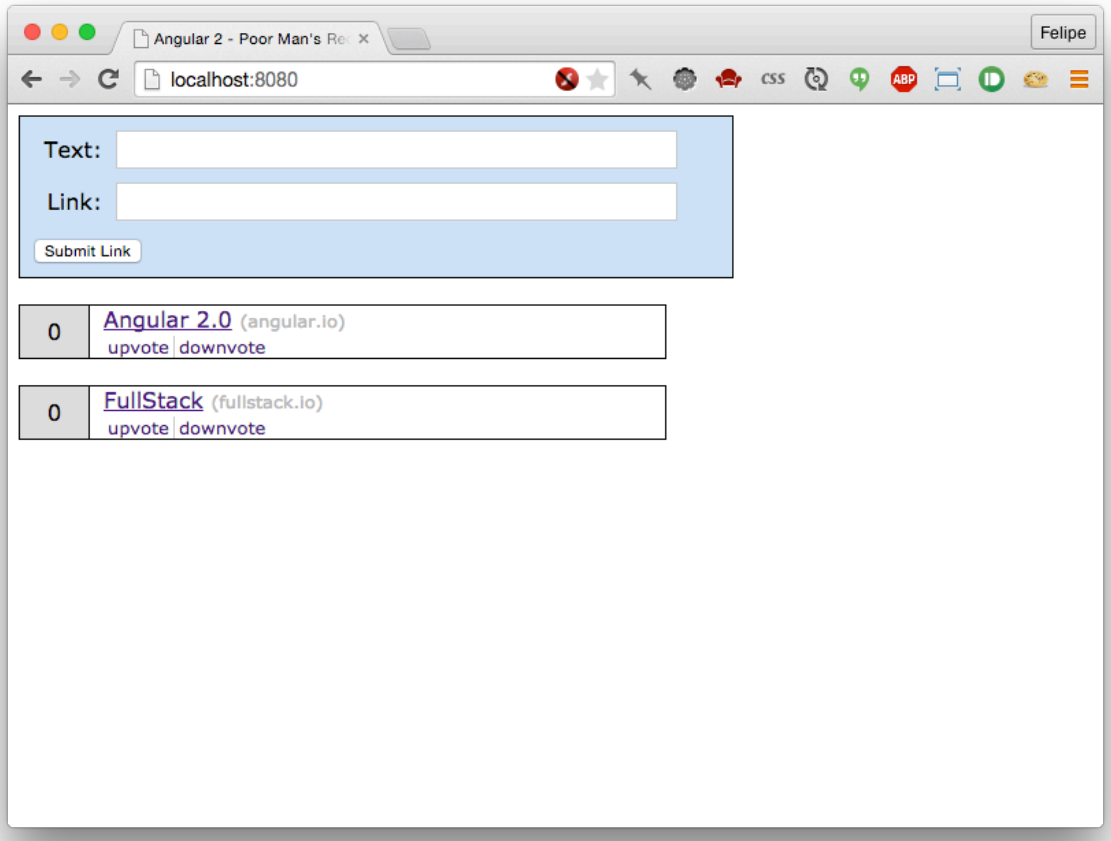
This book is a prerelease version and a work-in-progress.

Writing your First Angular2 Web Application

Poor Man's Reddit Clone

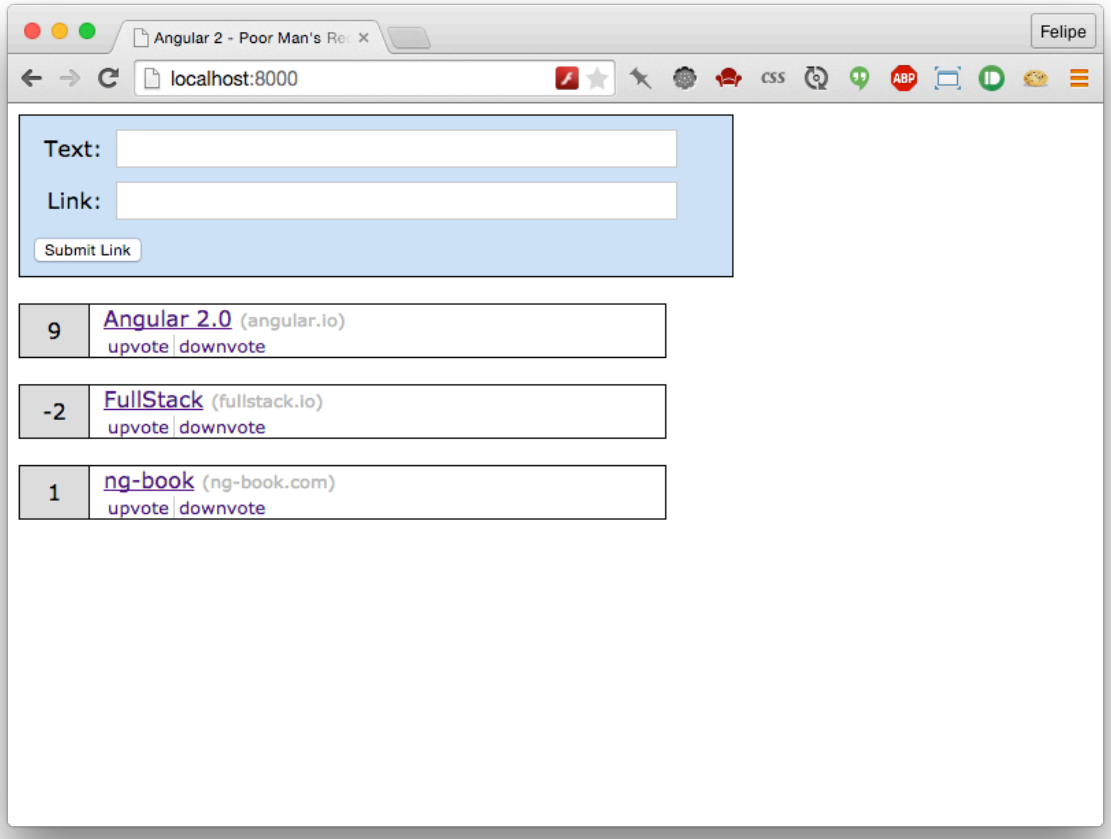
In this chapter we're going to write a toy application that allows the user to submit a new article with a title and an URL. You can think of it like a reddit-clone. By putting together a whole app we're going to touch on most of the parts of Angular 2.

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



Completed application

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



App with new article

For this example, we're going to use TypeScript. TypeScript is a superset of JavaScript ES6 that adds types. We're not going to talk about TypeScript in depth in this chapter, but if you're familiar with ES5/ES6 you should be able to follow along without any problems. We'll go over TypeScript more in depth in the next chapter.

Getting started

TypeScript

To get started with TypeScript, you will need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to the Node.js website for detailed informations:

https://nodejs.org/download/[1] .

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

Once you have Node.js setup, the next step is to install TypeScript. Make sure you install at least version 1.5 or greater. To install the 1.5 beta version, run the following `npm` command:

```
1 $ npm install -g 'typescript@^1.5.0-beta'
```

`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.

Example Project

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

Clone the git repository here[2]. Don't worry too much about its contents now.

```
1 $ git clone https://github.com/fcoury/angular2-reddit
```

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

```
1 $ npm install
```

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

[1] https://nodejs.org/download/
[2] https://github.com/fcoury/angular2-reddit

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

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

Traceur

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

To add Traceur to our app we add the following script tag:

```
    <script src="https://github.jspm.io/jmcriffey/bower-traceur-runtime@0.0.87/t\
raceur-runtime.js"></script>
```

For more information, check the Traceur GitHub repository[3].

SystemJS

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

To add SystemJS we need to add this:

```
    <script src="https://jspm.io/system@0.16.js"></script>
```

Angular2

And of course, we also need to add Angular itself. We have a version of Angular 2 in the git repo that we downloaded earlier, so to use it we can simply put this script tag:

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

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

Writing a hello world application

Here's how our code should look now:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Added libraries -->
    <script src="https://github.jspm.io/jmcriffey/bower-traceur-runtime@0.0.87/t\
raceur-runtime.js"></script>
    <script src="https://jspm.io/system@0.16.js"></script>
    <script src="bundle/angular2.dev.js"></script>
    <!-- Added libraries -->
  </head>
  <body>
  </body>
</html>
```

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

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Libraries -->
    <script src="https://github.jspm.io/jmcriffey/bower-traceur-runtime@0.0.87/t\
raceur-runtime.js"></script>
    <script src="https://jspm.io/system@0.16.js"></script>
    <script src="bundle/angular2.dev.js"></script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css" href="styles.css"> <!-- <- here -->
  </head>
  <body>
  </body>
</html>
```

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

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

```
1  /// <reference path="typings/angular2/angular2.d.ts" />
2
3  import {
4    Component,
5    View,
6    bootstrap,
7  } from "angular2/angular2";
8
9  @Component({
10   selector: 'hello-world'
11 })
12 @View({
13   template: `<div>Hello world</div>`
14 })
15 class HelloWorld {
16 }
17
18 bootstrap(HelloWorld);
```

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

The `import` statement defines the modules we want to use to write our code. Here we're importing three things: `Component`, `View`, and `bootstrap`. We're `importing` it `from "angular2/angular2"`. The `"angular2/angular2"` portion tells our program where to find the dependencies that we're looking for.

Notice that the structure of this `import` is of the format `import { things } from wherever`. In the `{ things }` part what we are doing is called *destructuring*. Destructuring is a feature provided ES6 and we talk more about it in the next chapter. The idea with the `import` is a lot like `import` in Java or `require` in Ruby. We're just making these dependencies available to this file.

Making a `Component`

One of the big ideas behind Angular 2 is the idea of *components*. In our Angular apps we write HTML markup that becomes our interactive application. But the browser only knows so many tags. The built-ins like `<select>` or `<form>` or `<video>` all have functionality defined by our browser creator. But what if we want to teach the browser new tags? What if we wanted to have a `<weather>` tag that defines the weather? Or what if we wanted to have a `<login>` tag that defines where we should put the login?

That is the idea behind components.

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

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

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

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

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

Let's take these one at a time.

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

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

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

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

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

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

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

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

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

Making a `View`

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

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

Notice that we're defining our `template` string between backticks (` ... `). This is a new and fantastic feature of ES6 that allows us to do multiline strings. We could have written the markup above as:

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

Using backticks for multiline strings is **fantastic** and makes it way easier to put templates inside your code files.

Should I really be putting templates in my code files? The answer is, it depends. For a long time the commonly held belief was that you should keep your code and templates separate. While this might be easier for some teams, for some projects it just adds a lot of overhead. When you have to switch between a lot of files it adds overhead to your development. Personally, if my templates are smaller than a page I much prefer having the templates alongside the code. I can see both the logic and the view together and it's really easy to understand how they interact

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

Booting Our Application

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

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

Putting all the pieces together

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

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

Add the following to the `body` section:

```
<!doctype html>
<html>
  <head>
    <title>Angular 2 - Poor Man's Reddit</title>
    <!-- Libraries -->
    <script src="https://github.jspm.io/jmcriffey/bower-traceur-runtime@0.0.87/t\
raceur-runtime.js"></script>
    <script src="https://jspm.io/system@0.16.js"></script>
    <script src="bundle/angular2.dev.js"></script>
    <!-- Stylesheet -->
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <script>
      System.import('app');
    </script>

    <hello-world></hello-world>

  </body>
</html>
```

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

Compiling the code

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

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

```
$ tsc
$
```

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

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

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

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

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

We have one more step to test our application. We need to run a test web-server to serve our app from. If you did `npm install` earlier then you will have a web server you can use like so:

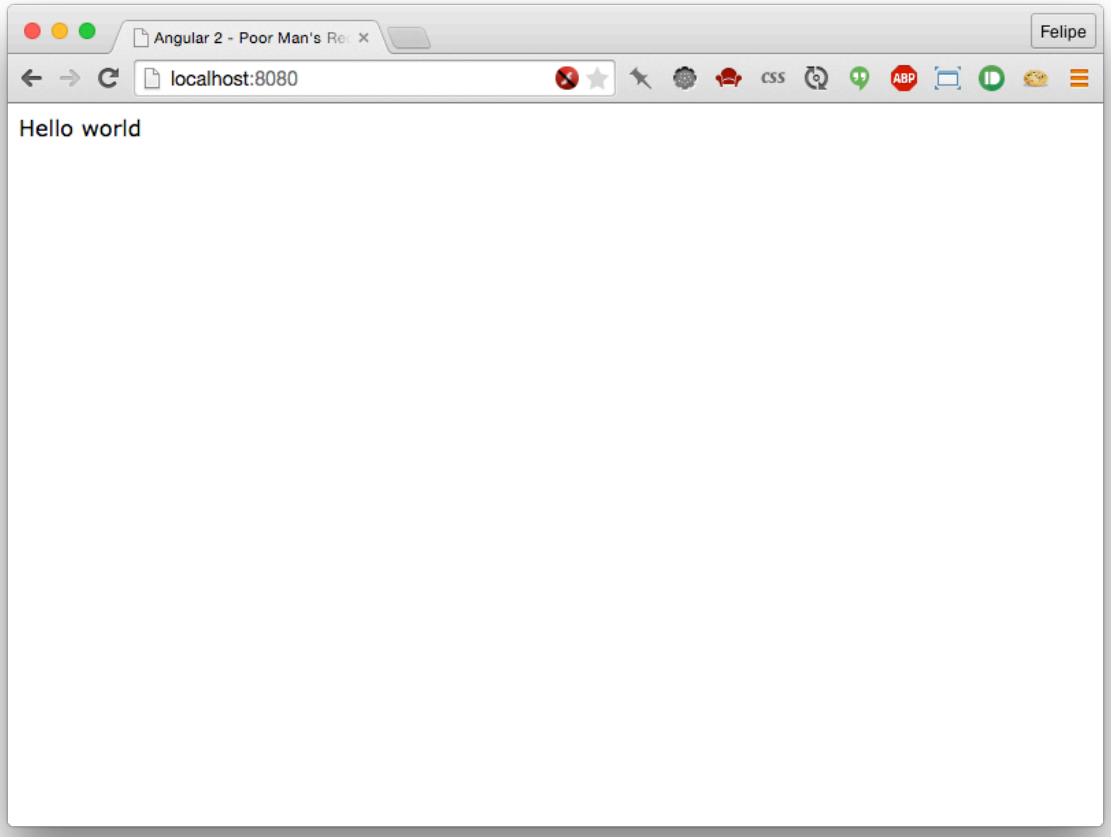
```
$ ./node_modules/http-server/bin/http-server
Starting up http-server, serving ./ on: http://0.0.0.0:8080
Hit CTRL-C to stop the server
```

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

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

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

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



Completed application

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

1. Make sure that your `app.js` file was created from the Typescript compiler `tsc`
2. Make sure that your webserver was started in the same directory as your `app.js` file
3. Make sure that your `index.html` file matches our code example above
4. Try opening the page in Chrome, right click, and pick "Inspect Element". Then click the "Console" tab and check for any errors.

Compiling on every change

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

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

Adding data to a component

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

Let's introduce `name` as a new property of our component. This way we can reuse the same component for different inputs.

Make the following changes:

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

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

Here we changed three things:

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

The typing is provided by TypeScript! This sets up a `name` property on *instances* of our `HelloWorld` class

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

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

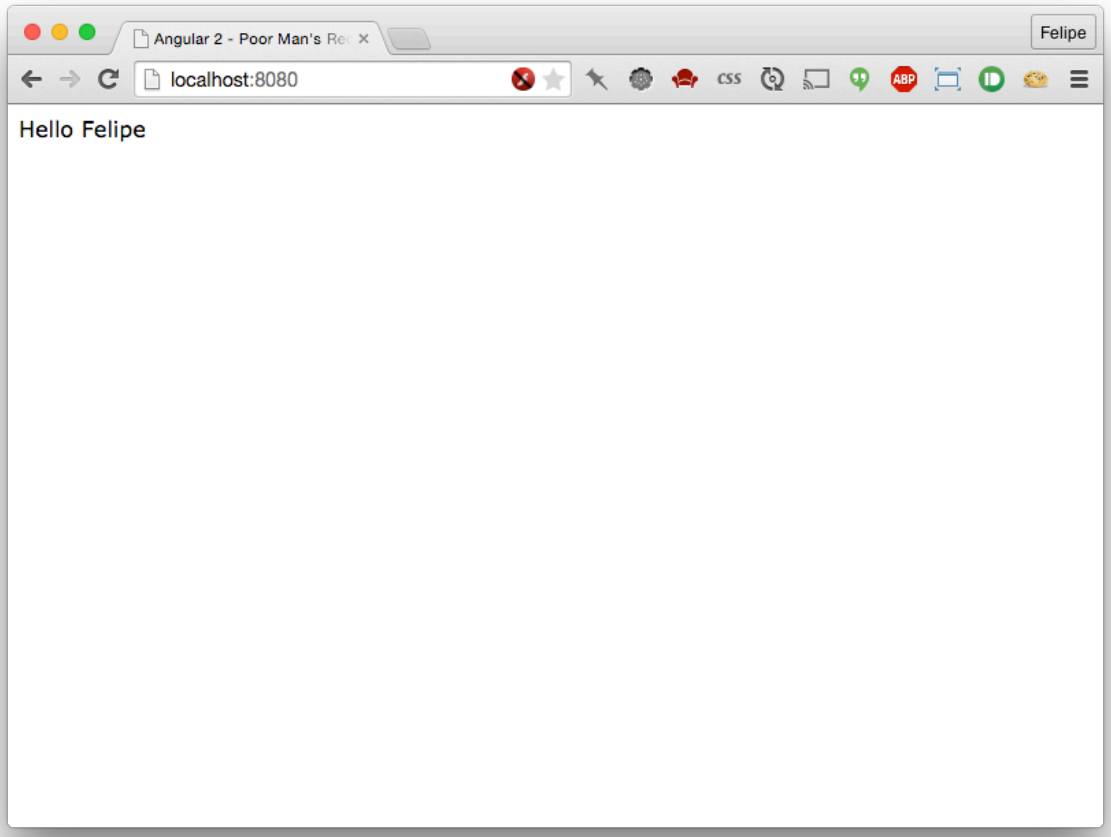
When we write:

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

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

3. Template Variable On the `View` notice that we added a new syntax: `{{ name }}`. The brackets are called "template-tags" (or "mustache tags"). Whatever is between the template tags will be expanded as a template. Here, because the View is *bound* to our Component, the `name` will expand to `this.name` e.g. `'Felipe'` in this case.

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



Application with Data

Working with arrays

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

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

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

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

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

@Component({
  selector: 'hello-world'
})
@View({
  directives: [For],
  template: `
  <ul>
    <li *for="#name of names">Hello {{ name }}</li>
  </ul>
  `
})
class HelloWorld {
  names: Array<string>;

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

bootstrap(HelloWorld);
```

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

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

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

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

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

The value states: `"#name of names"`. `names` is our array of names as specified on the `HelloWorld`

object. `#name` is called a *reference*. When we say `"#name of names"` we're saying loop over each element in `names` and assign each one to a variable called `name`.

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

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

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

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

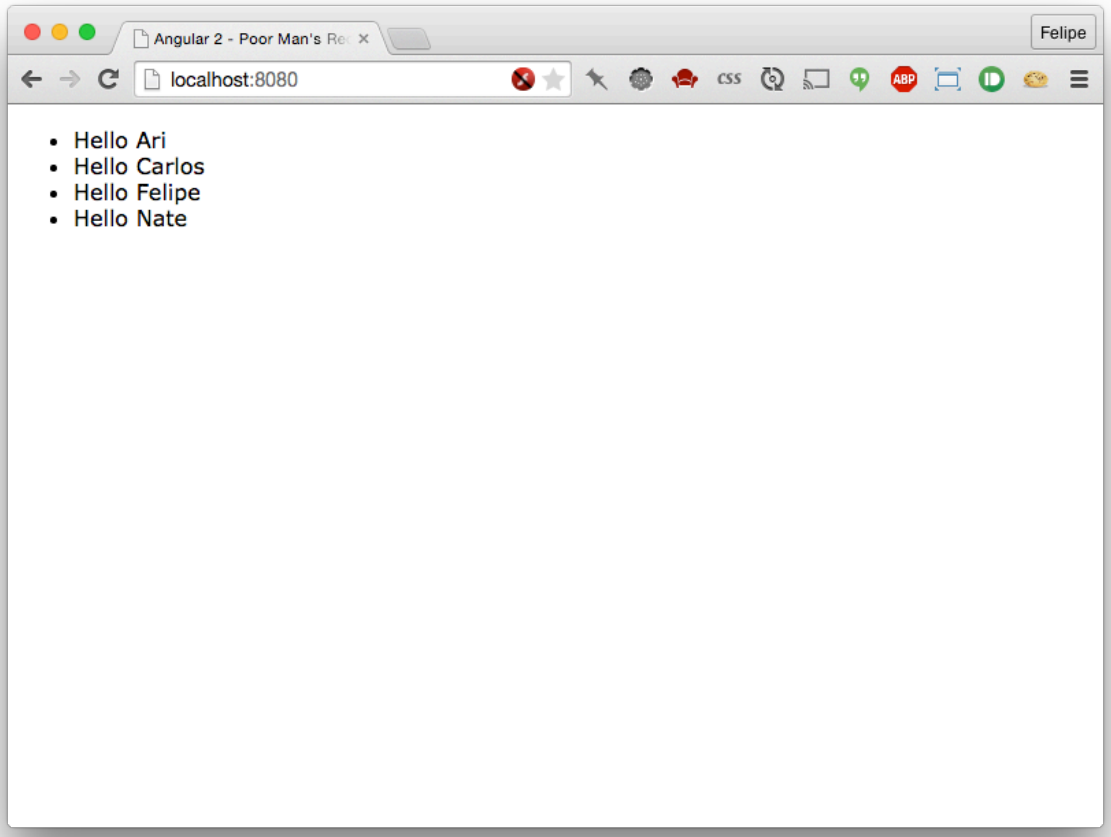
```
1   <li *for="#name of foobar">Hello {{ name }}</li>
```

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

If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the ng-for directive here[4]

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

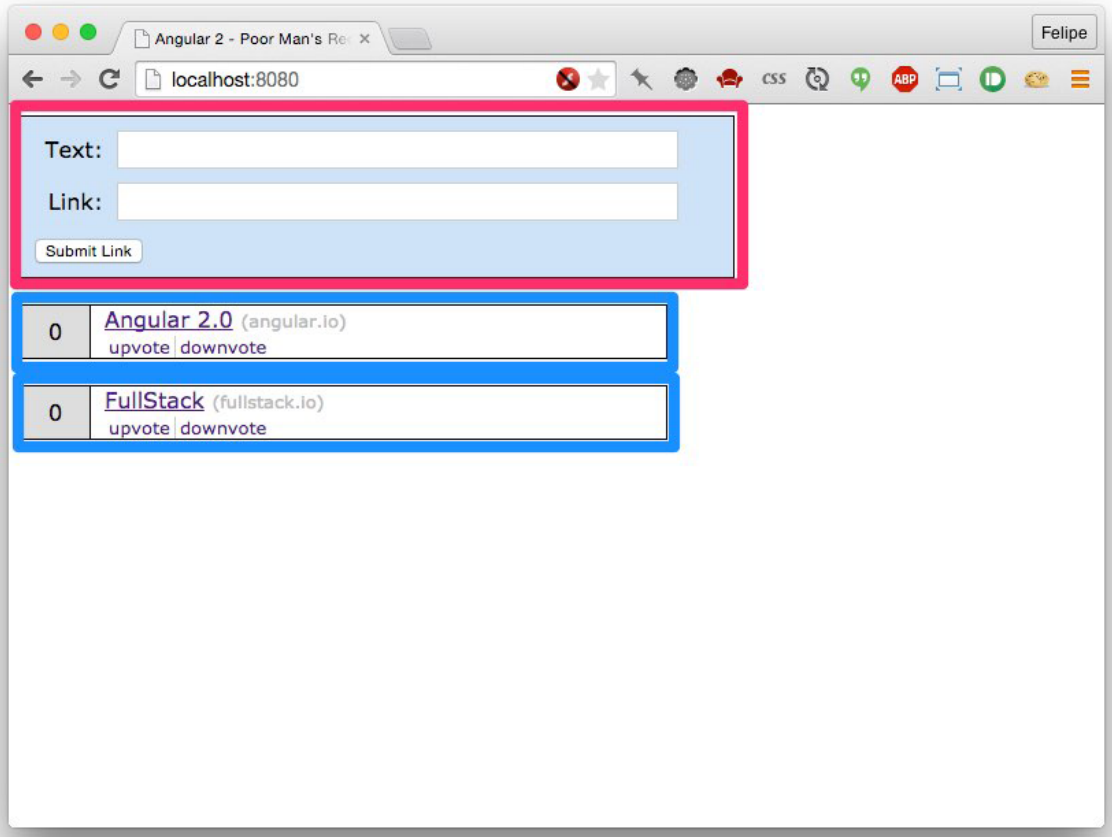
[4] https://github.com/angular/angular/blob/master/modules/angular2/src/directives/ng_for.ts



Application with Data

Expanding our Application

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



Application with Data

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

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

The form component

Let's start building the form component. For that, we'll change our `app.ts`. We're done with our `HelloWorld` component for now and instead we're going to build a component to represent our whole app: a `RedditApp` component:

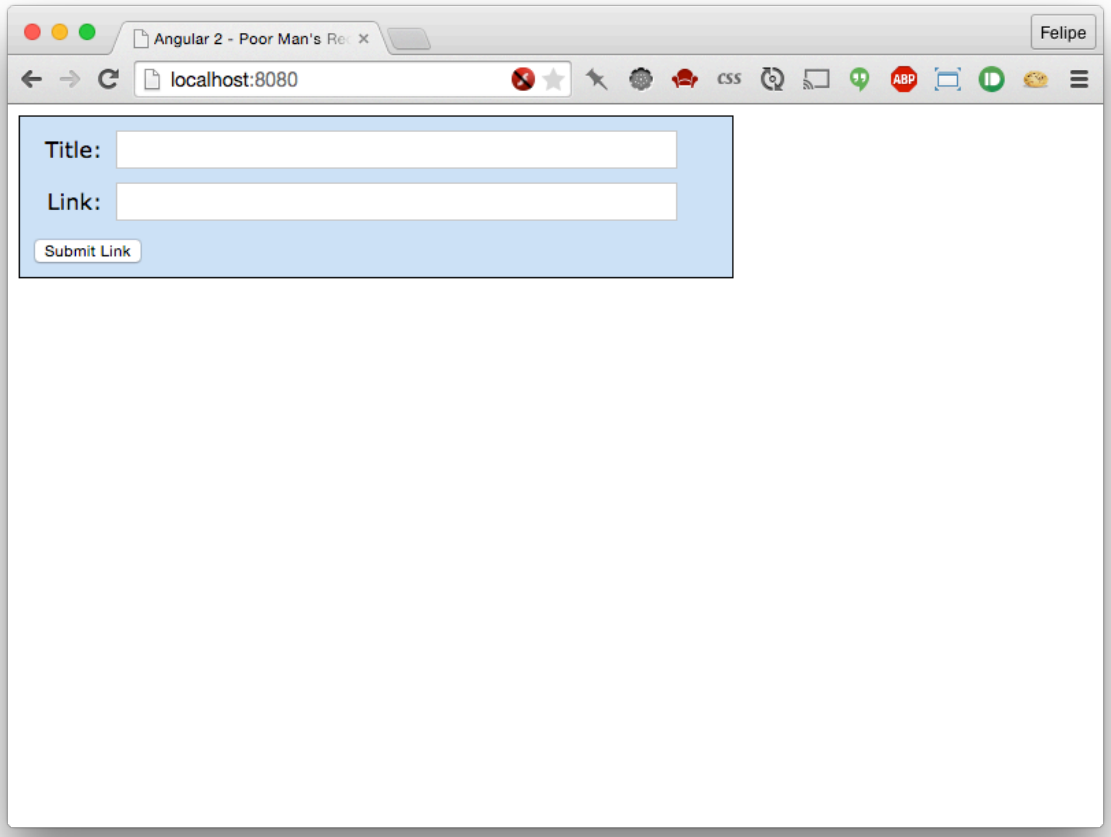
```
/// <reference path="typings/angular2/angular2.d.ts" />

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

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

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

bootstrap(RedditApp);
```

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

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

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

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



Form

Adding Interaction

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

Let's add some interaction:

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

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

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

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

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

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

Binding `input`s to values

Notice in our first input tag we have the following:

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

The new syntax here is the *reference* to `#newtitle`. This markup tells angular to bind the **value** of this input to the variable `newtitle`. The effect is that this makes the variable `newtitle` available to the actions within this view. Similarly we add `#newlink` to the other input tag.

Binding actions to events

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

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

All together:

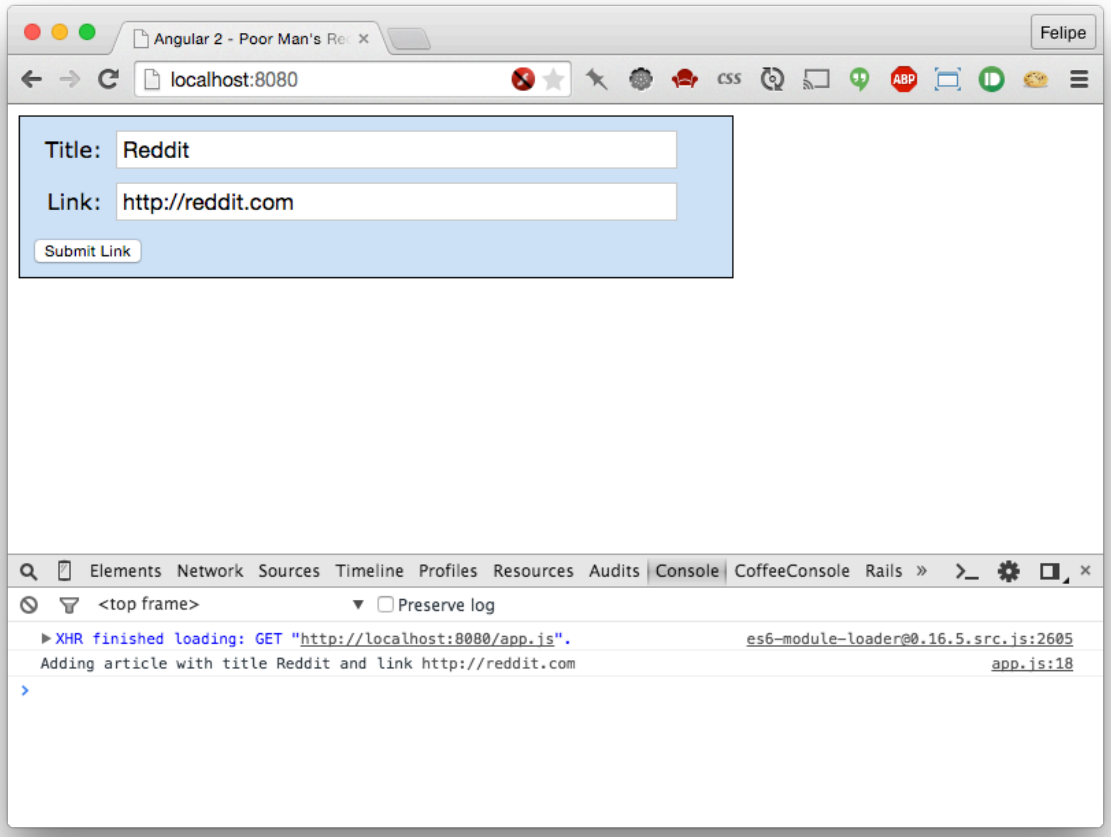
```
1  <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
```

Defining the Action Logic

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

Try it out!

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



Clicking the Button

Adding the article component

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

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

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

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

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

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

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

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

1. Describing the Component properties by annotating the class with `@Component`

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

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

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

1. We're showing `votes` and the `title` with the template expansion strings `{{ votes }}` and `{{ title }}`. The values come from the value of `votes` and `title` property of the `RedditArticle` class.
2. We can also use template strings in property values like we do in the a `href="{{ link }}"`. The value of the `href` will be dynamically populated with the value of the `href` from the component class
3. On our upvote/downvote links we have an action. We use `(click)` to bind `voteUp()`/`voteDown()` to their respective buttons. When the upvote button is pressed, the `voteUp()` function will be called on the `RedditArticle` class (and downvote respectively).

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

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

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

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

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

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

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

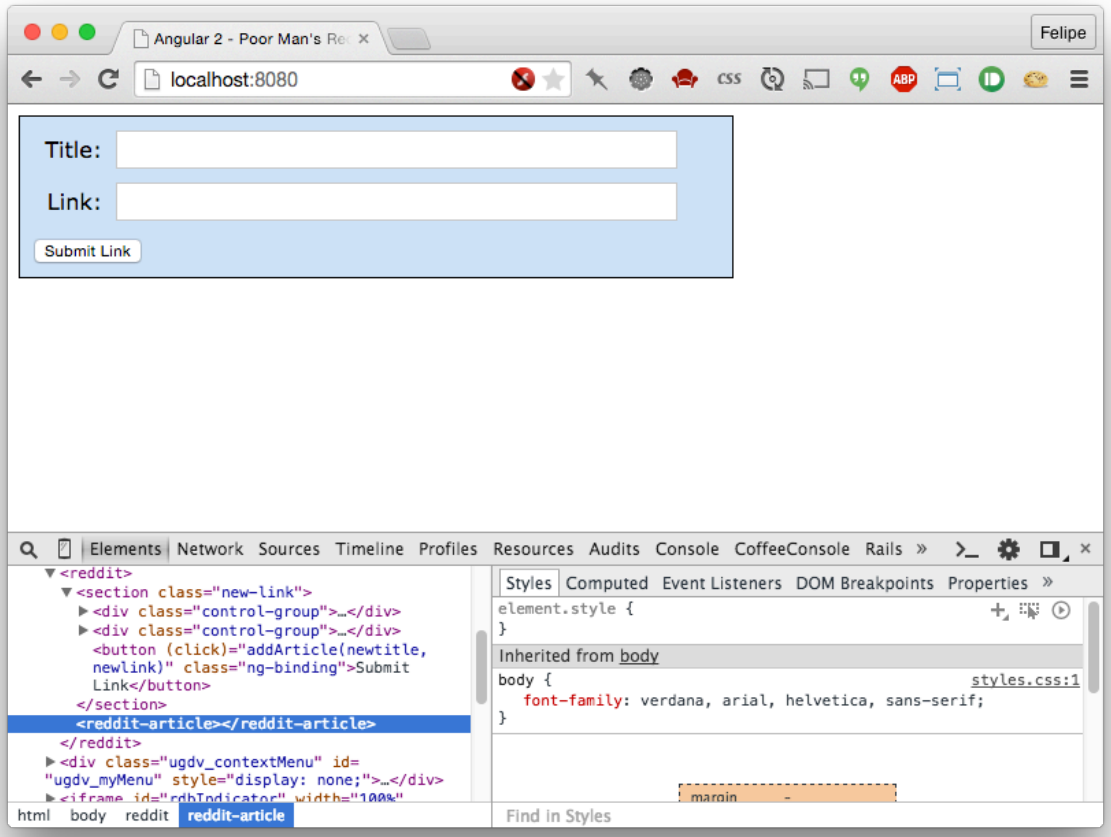
In `voteUp` we increment `this.votes` by one. Similarly we decrement for `voteDown`.

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

In this case, we want the `RedditApp` component to render this new component, so let's change that component code. First we will add the `<reddit-article>` tag to the `RedditApp`'s template:

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

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


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

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



Unexpanded tag when inspecting the DOM

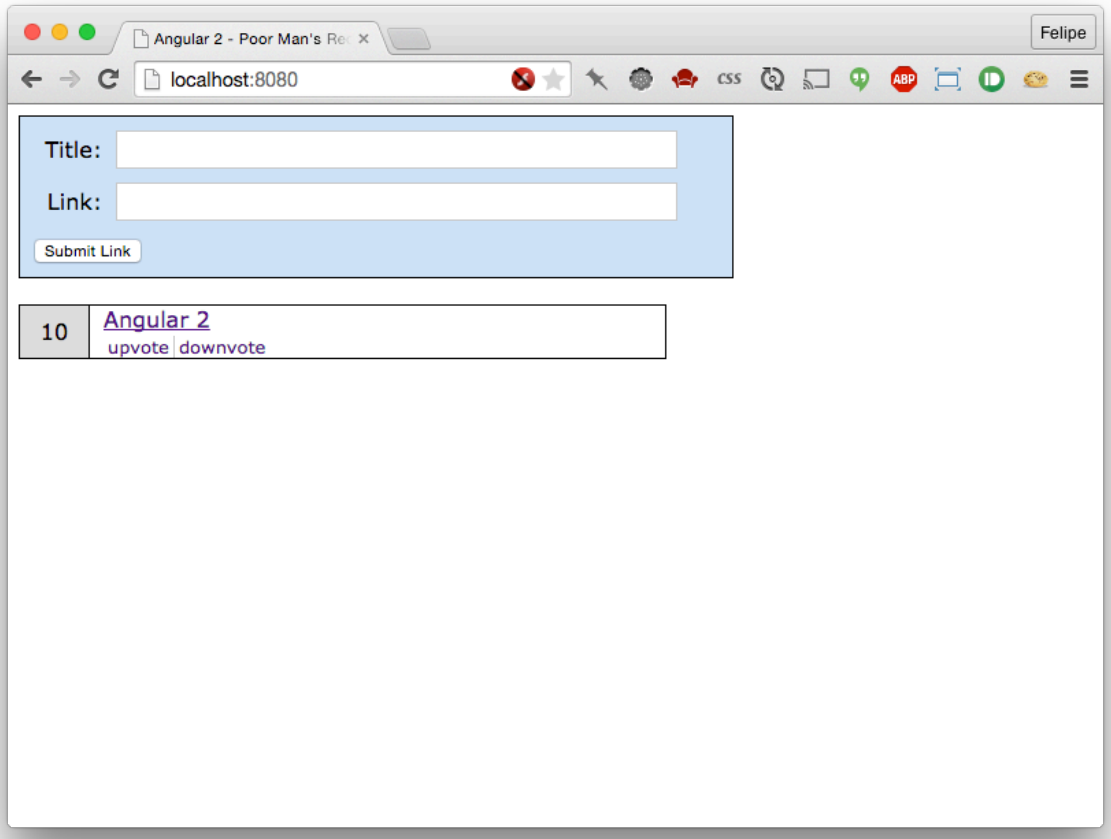
That happens because the `RedditApp` component doesn't know about the `RedditArticle` component yet!

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

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

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

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



Rendered RedditArticle component

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

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

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

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

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

Rendering multiple components

Creating an Article class

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

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

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

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

Each article has a `title`, a `link` and a total for the `votes`. When creating a new article we need the `title` and the `link`, and we also assume the recently submitted article has zero votes.

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

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

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

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

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

```
35    }
36  }
```

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

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

`voteUp` and `voteDown` current break the Law of Demeter[5] which says that a given object should assume as little as possible about the structure or properties any other objects. One way to detect this is to be suspicious when you see long method/property chains like `foo.bar.baz.bam`. This pattern of long-method chaining is also affectionately referred to as a "train-wreck".

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

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

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

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

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

Checkout our `RedditArticle` component definition now: it's so short! We've moved a lot of logic **out** of our component and into our models. An analogous MVC guideline would be Fat Models, Skinny Controllers[6]. The idea is that we want to move most of our domain logic to our models so that our components do the minimum work possible.

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

Storing multiple `Articles`

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

Start by changing `RedditApp` to have a collection of articles:

```
1  class RedditApp {
2    articles: Array<Article>;
3
4    constructor() {
5      this.articles = [
6        new Article('Angular 2', 'http://angular.io'),
7        new Article('Fullstack', 'http://fullstack.io')
8      ];
9    }
10
11   addArticle(title, link) {
12     console.log("Adding article with title", title.value, "and link", link.value\
13 );
14   }
15 }
```

Notice that our `RedditApp` has the line:

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

The `Array<Article>` might look a little funny if you're not used to typing your javascript. The word for this pattern is *generics*. It's a concept seen in Java, among others, and the idea is that your collection (the `Array`) is typed. That is, the `Array` is a collection that will only hold objects of type `Article`.

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

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

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

Configuring a `RedditArticle` Component with properties

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

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

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

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

```
class RedditArticle {
  article: Article;

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

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

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

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

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

It looks like this:

```
@Component({
  selector: 'reddit-article',
  properties: {
    'article': 'article'
  }
})
@View({
  // same ...
})
class RedditArticle {
}
```

Notice that `properties` is an `Object {}`. There are two parts to each property you want to define:

1. The key defines the **input** attribute in the **parent markup**
2. The value defines the **reference** variable in the **view**

Here we're using `'article'` in both places because it's easier to keep track of, but it's important to know that you don't have to do it that way.

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

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

```
21        `
22  })
23  class RedditArticle {
24  }
```

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

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

Rendering a list of Articles

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

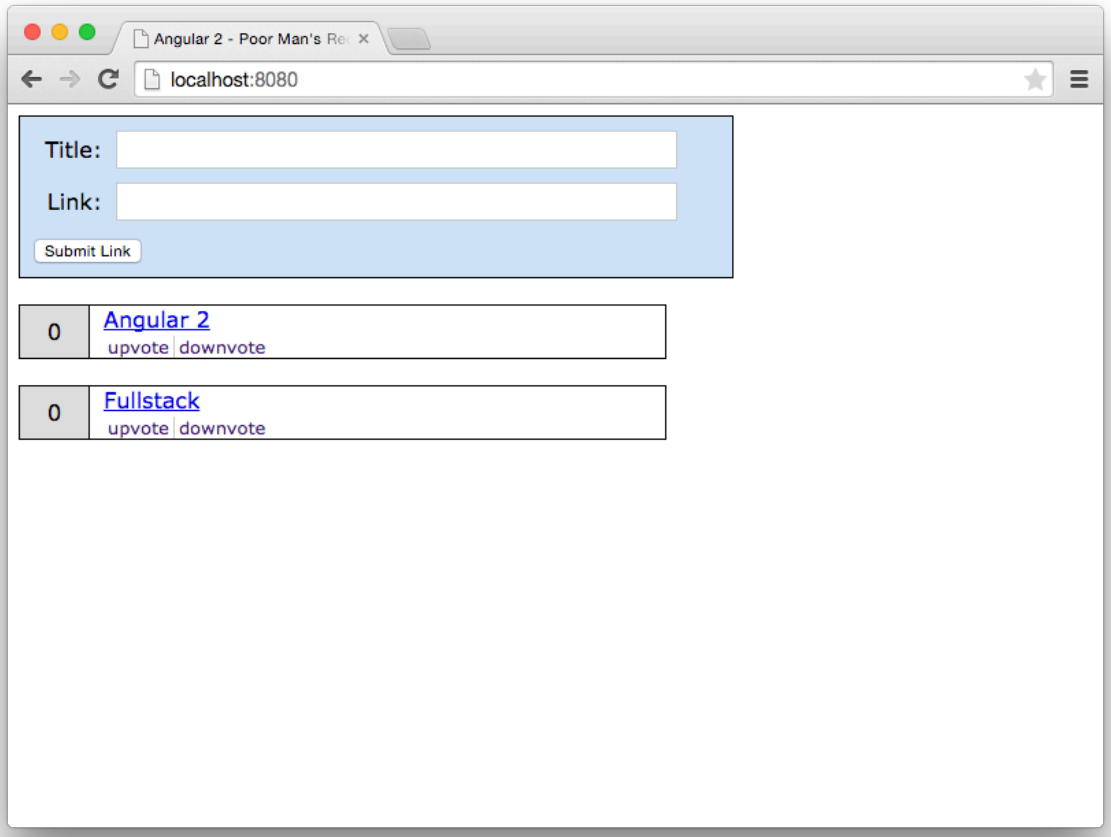
```
1   @Component({
2     selector: 'reddit'
3   })
4   @View({
5     template: `
6       <section class="new-link">
7         <div class="control-group">
8           <div><label for="title">Title:</label></div>
9           <div><input name="title" #newtitle></div>
10        </div>
11        <div class="control-group">
12          <div><label for="link">Link:</label></div>
13          <div><input name="link" #newlink></div>
14        </div>
15
16        <button (click)="addArticle(newtitle, newlink)">Submit Link</button>
17      </section>
18
19      <reddit-article
20         *for="#article of articles"
21         [article]="article">
22      </reddit-article>
23    `,
24
25    directives: [RedditArticle, For]
```

```
26 })
27 class RedditApp {
28   addArticle(title, link) {
29     console.log("Adding article with title", title.value, "and link", link.value\
30 );
31   }
32 }
```

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

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

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



Multiple articles being rendered

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

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

This will:

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

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

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

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

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

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

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

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

Wrapping Up

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

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

1. Split your app into components

2. Create the views
3. Define your models
4. Display your models
5. Add interaction

Onward!

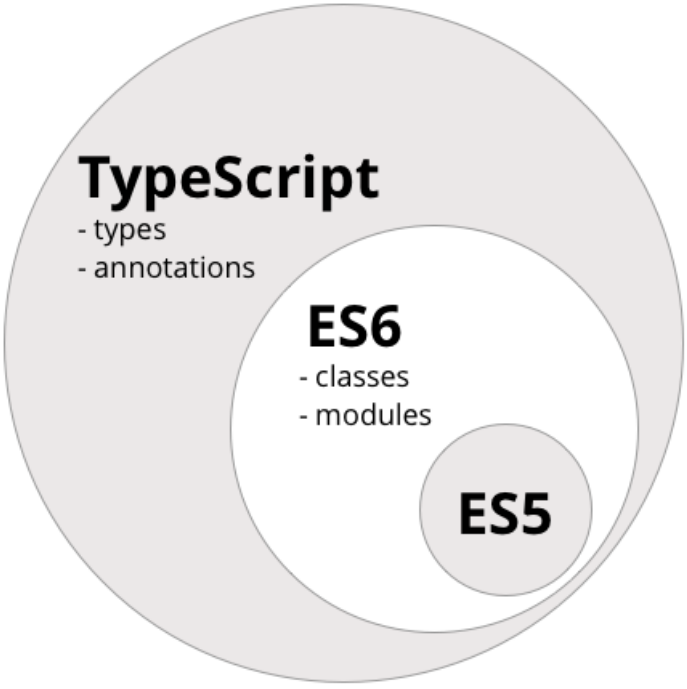
TypeScript

Angular 2 is built in TypeScript

Angular 2 is built in a Javascript-like language called TypeScript[7].

You might be skeptical of using a new language just for Angular, but it turns out, there are a lot of great reasons to use TypeScript instead of plain Javascript.

TypeScript isn't a completely new language, it's a superset of ES6. If we write ES6 code, it's perfectly valid and compilable TypeScript code. Here's a diagram that shows the relationship between the languages:



ES5, ES6, and TypeScript

What is ES5? What is ES6? ES5 is short for "ECMAScript 5", otherwise known as "regular Javascript". ES5 is the normal Javascript we all know an love. It runs in more-or-less every browser. ES6 is the next version of Javascript, which we talk more about below.

[7] http://www.typescriptlang.org/

At the publishing of this book, very few browsers will run ES6 out of the box, much less TypeScript. To solve this issue we have *transpilers* (or sometimes called *transcompiler*). The TypeScript transpiler takes our TypeScript code as input and outputs ES5 code that nearly all browsers understand.

For converting TypeScript to ES5 there is a single transpiler written by the core TypeScript team. However if we wanted to convert *ES6* code (not TypeScript) to *ES5* there are two major ES6-to-ES5 transpilers: **traceur**[8] by Google and **babel**[9] created by the JavaScript community. We're not going to be using either directly for this book, but they're both great projects that are worth knowing about.

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

```
npm install -g 'typescript@^1.5.0-beta'
```

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

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

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

What do we get with TypeScript?

There are five big improvements that TypeScript bring over ES5:

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

Let's deal with these one at a time.

[8] https://github.com/google/traceur-compiler
[9] https://babeljs.io/

Types

The major improvement of TypeScript over ES6, that gives the language its name, is the typing system.

For some people the lack of type checking is considered one of the benefits of using a language like JavaScript. You might be a little skeptical of type checking but I'd encourage you to give it a chance. One of the great things about type checking is that

1. it helps when *writing* code because it can prevent bugs at compile time and
2. it helps when *reading* code because if gives a clarifies your intentions

It's also worth noting that types are optional in TypeScript. If we want to write some quick code or prototype a feature, we can omit types and gradually add them as the code becomes more mature.

TypeScript's basic types are the same ones we've been using implicitly when we write "normal" JavaScript code: strings, numbers, booleans, etc.

Up until ES5, we would define variables with the `var` keyword, like `var name;`.

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

```
var name: string;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Trying it out with a REPL

To play with the examples on this chapter, let's install a nice little utility called **TSUN**[10] (TypeScript Upgraded Node):

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

Now start tsun:

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

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

>
```

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

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

Built-in types

String

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

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

Number

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

```
var age: integer = 36;
```

Boolean

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

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

Array

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

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

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

Or similarly with a `number`:

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

Enums

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

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

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

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

In a way that `Employee` is `3`, `Manager` is `4` and Admin is `5`, and we can even set individual values:

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

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

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

Any

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

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

Void

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

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

Classes

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

A number of good practices have been adopted by the JavaScript community to compensate the lack of classes. A good summary of those good practices can be found in Mozilla Developer Network's JavaScript Guide[11], and you can find a good overview on the Introduction to Object-Oriented Javascript[12] page.

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

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

```
class Vehicle {
}
```

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

Properties

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

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

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

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

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

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

Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// set initial age
p.age = 6;

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

// -> 18
```

Constructors

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

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

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

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

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

Is the same as:

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

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

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

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

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

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

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

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

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

It makes our previous example a little easier to write:

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

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

Inheritance

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

If you want to have a deeper understanding of how inheritance used to work in ES5, take a look at the Mozilla Developer Network article about it: Inheritance and the prototype chain[13].

TypeScript fully supports inheritance and, unlike ES5, it's built into the core language. Inheritance is achieved through the `extends` keyword.

To illustrate, let's say we've created a `Report` class:

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

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

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

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

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

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

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

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

Running this should show:

```
First line
Second line
```

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

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

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

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

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

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

Utilities

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

- fat arrow function syntax
- template strings

Fat Arrow Functions

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

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

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

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

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

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

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

Or as a statement:

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

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

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

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

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

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

Template Strings

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

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

Variables in strings

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

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

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

console.log(greeting);
```

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

Multiline strings

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

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

// do something with `template`
```

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

Wrapping up

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

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

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

Let's get back to Angular!

How Angular Works

In this chapter, we're going to talk about the high-level concepts in Angular 2. The idea is that by taking a step back we can see how all the pieces fit together.

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

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

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

However, ng2 Components have some significant advantages over ng1 directives and we'll talk about that below. First, let's start at the top: the Application.

Application

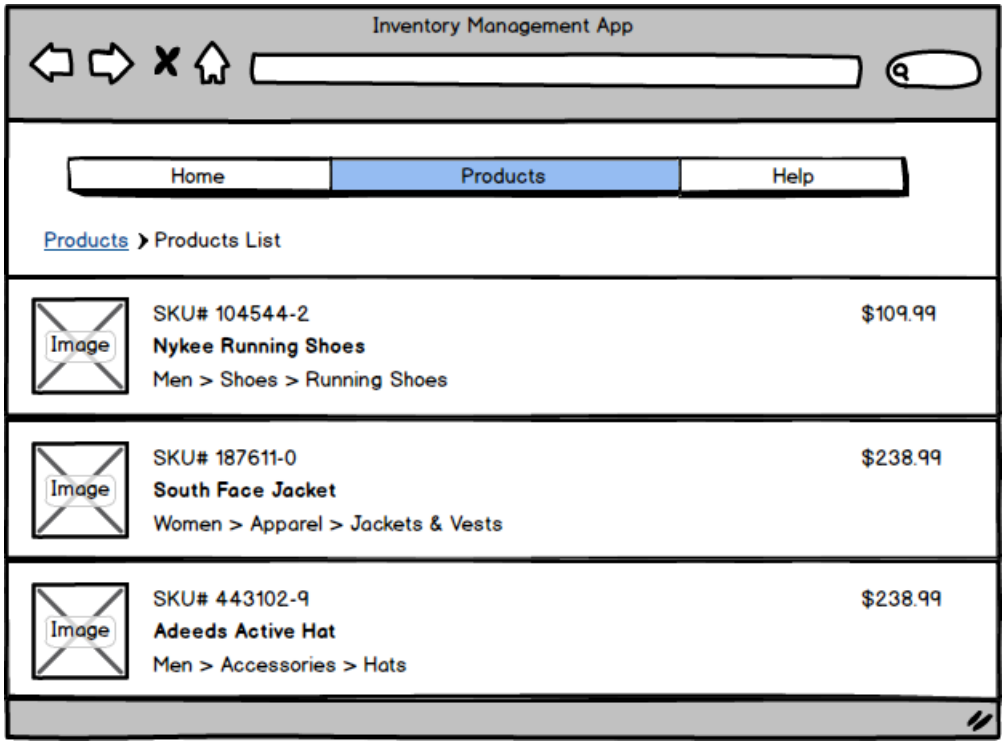
An ng2 Application is nothing more than a tree of Components.

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

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

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

For example, let's talk about an imaginary inventory management application that is represented by the following page mockup:



Inventory Management App

Given this mockup, to write this application the first thing we'd to do is split it up into individual components (organized into a tree).

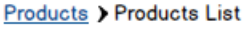
In this example, we could group the page into three high level components

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

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

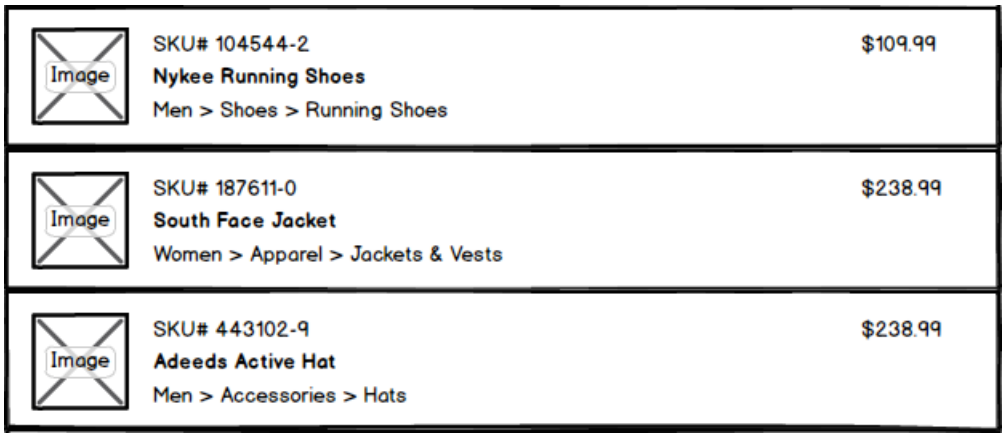
Navigation Component

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



Breadcrumbs Component

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



Product List Component

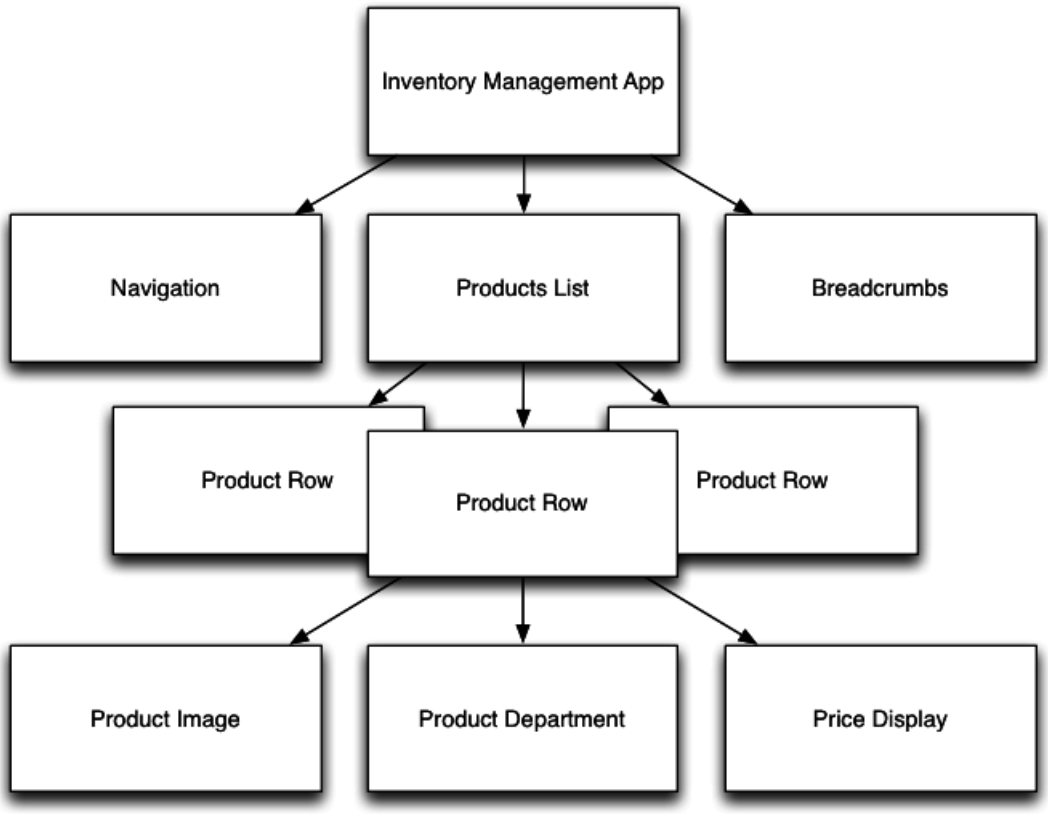
Breaking this component down into the next level of smaller components, we could say that the Product List is componse of multiple Product Rows.

Product Row Component

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

- the **Product Image** component, that would be responsible to render a product image, given its image name (imagine that the component knows how to get the image using a proper Amazon S3 URL, for instance);
- the **Product Department** would have the responsability of, given a department id, render the whole department tree, like *Men > Shoes > Running Shoes*;
- the **Price Display** would be a more generic component, that could be reused accross the application, to properly render a price. Imagine that our implementation customizes the pricing if the user is logged in to include system-wide tier discounts or include shipping for instance. We could implement all this behavior into this component.

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



App Tree Diagram

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

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

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

Components

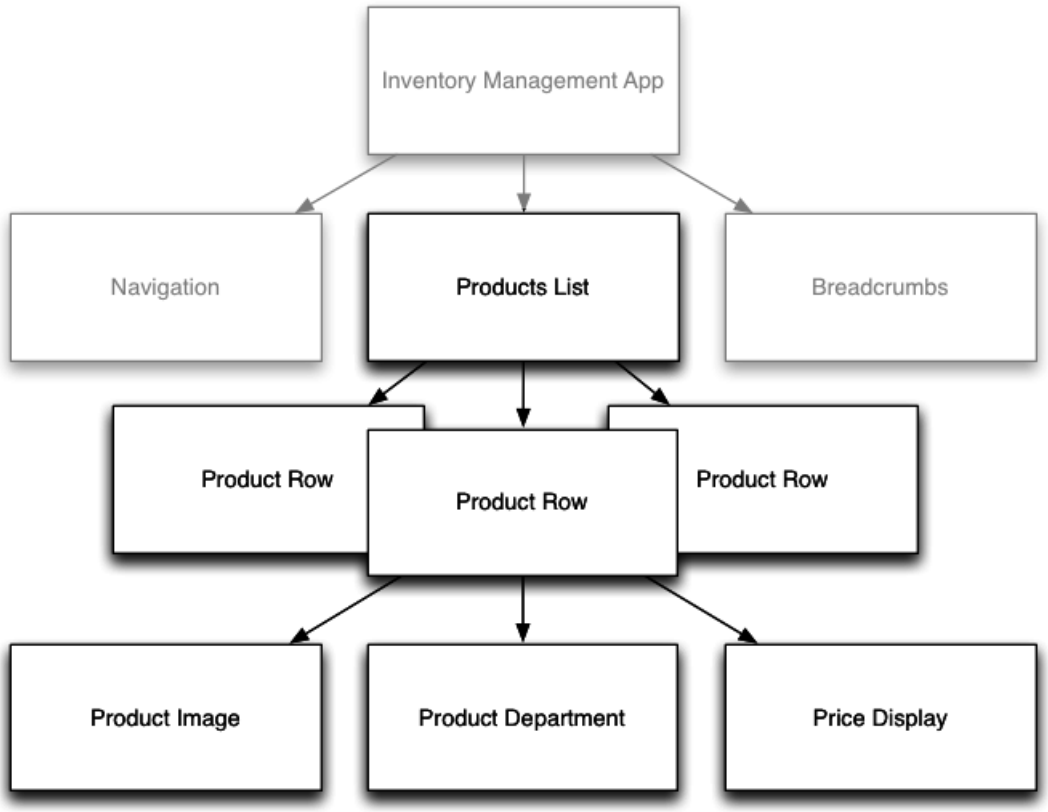
As you can see, Components are the fundamental building block of Angular 2 applications. We break our application into more granular child components.

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

Each components is composed of three parts:

- Component *Decorator*
- A View
- A Controller

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



Products List Component

An implementation of the Products List component could be:

```
// The "component decorator"
@Component({
  selector: 'products-list',
  properties: {
    products: 'products'
  }
})

// The view annotation
@View({
  directives: [ProductRow],
  template: `
  <div class="products-list">
```

```
    <div product-row *for="#product in products" [product]="product">
    </div>
  </div>
  `
})

// The "controller" class
class ProductsList {
  products: Array<Product>;
}
```

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

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

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

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

Component Decorator

The component decorator is where you declare how the outside world will interact with your component.

There are lots of options available to configure a component, which we cover in the Components chapter. Here we're just going to touch on some of the basics.

Component `selector`

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

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

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

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

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

Component `properties`

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

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

`properties` takes a key-value object which should specify all "inputs" to this component. For instance we could have a different component that looks like this:

```
@Component({
  //...
  properties: {
    name: 'name',
    age: 'age',
    enabled: 'isEnabled'
  }
  //...
})
```

In the `properties` object, the key and the value have a specific meaning:

The **key** of the object (`name`, `age` and `enabled`) configures how the property is **visible to the outside world**.

The **values** (`name`, `age` and `isEnabled`) represent how that incoming property will be **visible ("bound") in the controller**.

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

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

Controller

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

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

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

Views

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

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

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

- `directives`: This specifies the other components we want to be able to use in this view. This option takes an `Array` of classes. Unlike Angular 1, where all directives are essentially globals, in Angular 2 you must specifically say which directives you're going to be using. Here we say we're going to use the `ProductRow` directive.
- `template`: This specifies the HTML template that we want to use for rendering this component. Notice that we're using TypeScript's backtick "` multi-line string syntax. Also you've probably noticed there is a lot of syntax in that view. We'll go over each part in detail.

Properties and Events

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

Data flows *in* to your component via property bindings and *out* of your component through event bindings.

That previous sentence is so important, it's worth repeating:

Data flows **in** to your component via **property bindings** and **out** of your component through **event bindings**.

Think of the set of property bindings + event bindings as defining the **public API** of your component.

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

For "writing" data You application isn't going to be read-only. There are also going to be inputs by the user. To handle an input from a user we use **events**.

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

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

Data binding

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

Private Properties

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

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

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

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

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

This previous example is an instance of a *controller-only property*. You can think of it as a private property for your component. A "private", or controller-only, property is only visible within the component itself. Their commonly use for properties that are only used internally.

Public Properties

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

Contrasting with controller-only properties, the other type of property is a component properties. Component properties are declared **on the component decorator** and therefore made visible outside the component. You can think of these as "public" properties.

```
1  @Component({
2    selector: 'some-component',
3    properties: {
4      number: 'number'
5    }
6  })
7
8  @View({
9    template: `
10   The next number is {{ number + 1 }}
11   `
12 })
13 class SomeComponent {
14   number: number;
15 }
```

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

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

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

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

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

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

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

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

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

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

Event binding

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

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

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

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

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

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

  constructor() {
    this.value = 1;
  }

  increase() {
    this.value++;
  }

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

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

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

In this example, the event is internal to the component. When creating our own components we can also expose "public" events that allow the component to talk to the outside world. We'll do that in the next example.

Putting it all together

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

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

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

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

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

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

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

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

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

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

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

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

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

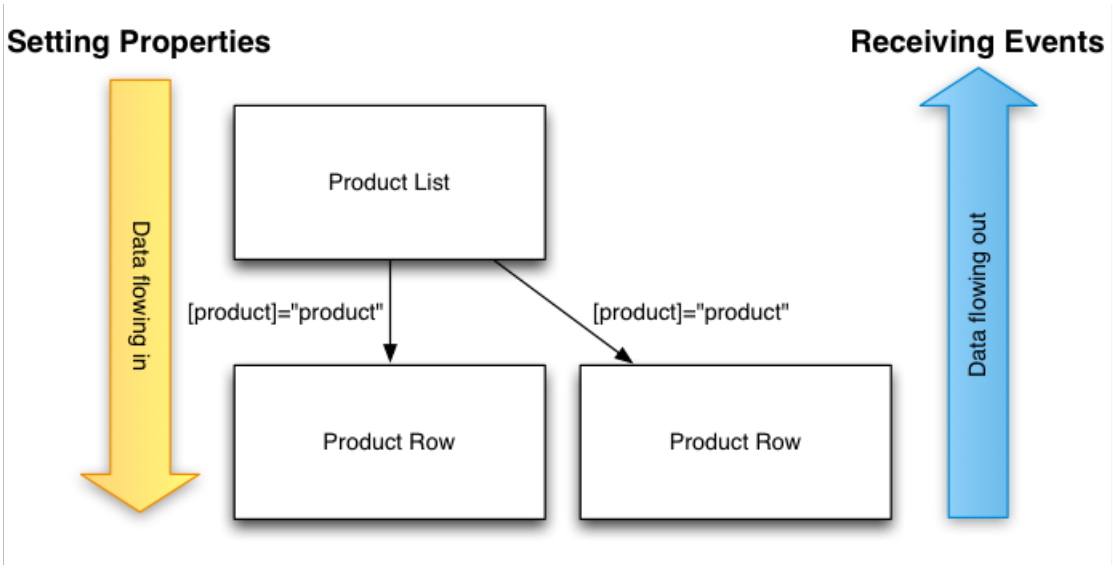
You can see that we are calling the `next` method of the `EventEmitter`, and passing the product this `ProductRow` holds. What we are doing is forwarding the event on to any one who is listening. This way, if we had another component that used ours, we could write:

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

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

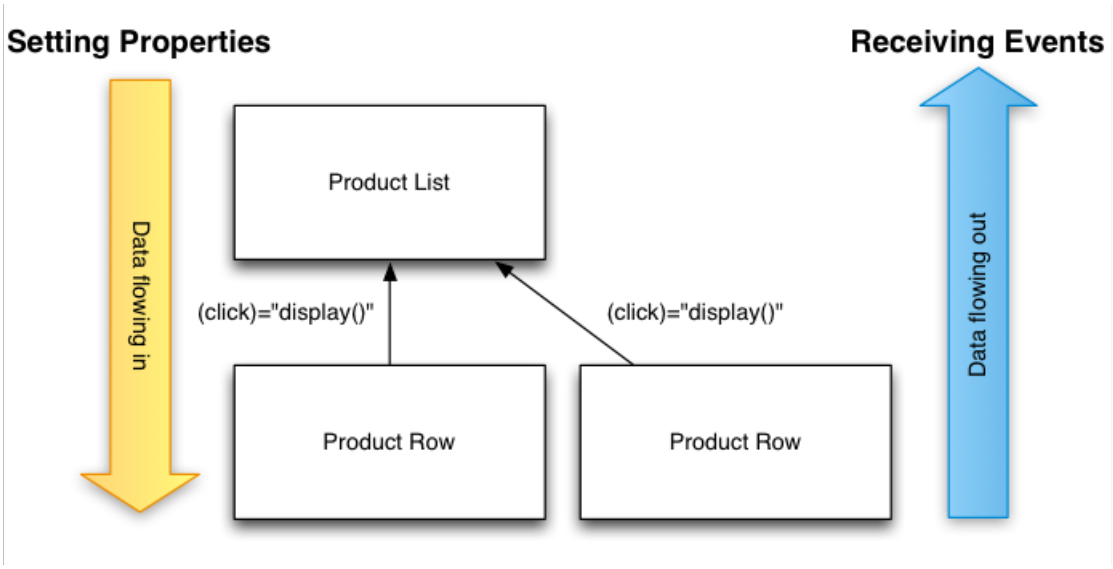
Data Flow

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



Data Flowing Into ProductRow

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



Data Flowing Out from ProductRow

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

Summary

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

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

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

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

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