



ng-book 2

The Complete Book on AngularJS 2



FULLSTACK.io



gistia

Ari Lerner
Felipe Coury
Nate Murray
Carlos Taborda

ng-book 2

The Complete Guide to Angular

Written by Nate Murray, Felipe Coury, Ari Lerner, and Carlos Taborda

© 2017 Fullstack.io

All rights reserved. No portion of the book manuscript may be reproduced, stored in a retrieval system, or transmitted in any form or by any means beyond the number of purchased copies, except for a single backup or archival copy. The code may be used freely in your projects, commercial or otherwise.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Typeset using Leanpub.

Published in San Francisco, California by Fullstack.io.



We'd like to thank:

- Our technical editor Frode Fikke
- Nic Raboy, and Burke Holland for contributing the NativeScript chapter

Contents

Book Revision	1
Bug Reports	1
Chat With The Community!	1
Be notified of updates via Twitter	1
We'd love to hear from you!	1
Writing your First Angular 2 Web Application	1
Simple Reddit Clone	1
Getting started	4
TypeScript	4
angular-cli	4
Example Project	5
Writing Application Code	9
Running the application	9
Making a Component	10
Importing Dependencies	12
Component Annotations	12
Adding a template with templateUrl	13
Adding a template	13
Adding CSS Styles with styleUrls	14
Loading Our Component	15
Adding Data to the Component	16
Working With Arrays	19
Using the User Item Component	22
Rendering the User ItemComponent	23
Accepting Inputs	24
Passing an Input value	25
Bootstrapping Crash Course	26
Expanding our Application	28
Adding CSS	30
The Application Component	30
Adding Interaction	32
Adding the Article Component	36
Rendering Multiple Rows	45

CONTENTS

Creating an Article class	45
Storing Multiple Articles	50
Configuring the ArticleComponent with inputs	51
Rendering a List of Articles	52
Adding New Articles	54
Finishing Touches	55
Displaying the Article Domain	55
Re-sorting Based on Score	56
Full Code Listing	57
Wrapping Up	57
Getting Help	57
TypeScript	58
Angular 2 is built in TypeScript	58
What do we get with TypeScript?	59
Types	60
Trying it out with a REPL	61
Built-in types	62
Classes	64
Properties	64
Methods	65
Constructors	67
Inheritance	68
Utilities	70
Fat Arrow Functions	70
Template Strings	72
Wrapping up	73
How Angular Works	74
Application	74
The Navigation Component	75
The Breadcrumbs Component	75
The Product List Component	76
Product Model	78
Components	79
Component Annotation	81
Component selector	81
Component template	82
Adding A Product	82
Viewing the Product with Template Binding	84
Adding More Products	85
Selecting a Product	86
Listing products using <products-list>	86

CONTENTS

The ProductsList Component	89
Configuring the ProductsList @Component Options	90
Component inputs	90
Component outputs	93
Emitting Custom Events	95
Writing the ProductsList Controller Class	96
Writing the ProductsList View Template	97
The Full ProductsList Component	99
The ProductRow Component	101
ProductRow Component Configuration	101
ProductRow Component Definition Class	102
ProductRow template	103
ProductRow Full Listing	103
The ProductImage Component	104
The PriceDisplay Component	105
The ProductDepartment Component	105
NgModule and Booting the App	106
Booting the app	108
The Completed Project	108
A Word on Data Architecture	109
Built-in Directives	110
Introduction	110
NgIf	110
NgSwitch	110
NgStyle	113
NgClass	115
NgFor	118
Getting an index	123
NgNonBindable	124
Conclusion	125
Forms in Angular 2	126
Forms are Crucial, Forms are Complex	126
FormControls and FormGroups	126
FormControl	126
FormGroup	127
Our First Form	128
Loading the FormsModule	129
Simple SKU Form: @Component Annotation	130
Simple SKU Form: template	130
Simple SKU Form: Component Definition Class	134
Try it out!	134

CONTENTS

Using FormBuilder	136
Reactive Forms with FormBuilder	137
Using FormBuilder	137
Using myForm in the view	138
Try it out!	139
Adding Validations	140
Explicitly setting the sku FormControl as an instance variable	142
Custom Validations	147
Watching For Changes	148
ngModel	150
Wrapping Up	152
HTTP	153
Introduction	153
Using @angular/http	154
import from @angular/http	154
A Basic Request	155
Building the SimpleHTTPComponent @Component	156
Building the SimpleHTTPComponent template	156
Building the SimpleHTTPComponent Controller	157
Full SimpleHTTPComponent	159
Writing a YouTubeSearchComponent	160
Writing a SearchResult	162
Writing the YouTubeService	162
Writing the SearchBox	171
Writing SearchResultComponent	178
Writing YouTubeSearchComponent	179
@angular/http API	183
Making a POST request	183
PUT / PATCH / DELETE / HEAD	183
RequestOptions	184
Summary	185
Routing	186
Why Do We Need Routing?	186
How client-side routing works	187
The beginning: using anchor tags	188
The evolution: HTML5 client-side routing	188
Writing our first routes	189
Components of Angular 2 routing	189
Imports	189
Routes	190
Installing our Routes	191

CONTENTS

RouterOutlet using <router-outlet>	192
RouterLink using [routerLink]	193
Putting it all together	194
Creating the Components	196
HomeComponent	196
AboutComponent	196
ContactComponent	196
Application Component	197
Configuring the Routes	199
Routing Strategies	200
Path location strategy	201
Running the application	202
Route Parameters	204
ActivatedRoute	205
Music Search App	206
First Steps	208
The SpotifyService	209
The SearchComponent	210
Trying the search	220
TrackComponent	222
Wrapping up music search	224
Router Hooks	224
AuthService	225
LoginComponent	226
ProtectedComponent and Route Guards	228
Nested Routes	234
Configuring Routes	235
ProductsComponent	235
Summary	240
Dependency Injection	241
Injections Example: PriceService	242
“Don’t Call Us...”	244
Dependency Injection Parts	246
Playing with an Injector	247
Providing Dependencies with NgModule	249
Providers	250
Using a Class	250
Using a Factory	251
Using a Value	253
Using an alias	253
Dependency Injection in Apps	253
Working with Injectors	255

CONTENTS

Substituting values	262
NgModule	266
NgModule vs. JavaScript Modules	266
The Compiler and Components	267
Dependency Injection and Providers	267
Component Visibility	268
Specifying Providers	270
Conclusion	271
Data Architecture in Angular 2	272
An Overview of Data Architecture	272
Data Architecture in Angular 2	273
Data Architecture with Observables - Part 1: Services	274
Observables and RxJS	274
Note: Some RxJS Knowledge Required	274
Learning Reactive Programming and RxJS	274
Chat App Overview	276
Components	277
Models	278
Services	279
Summary	279
Implementing the Models	280
User	280
Thread	280
Message	281
Implementing UserService	281
currentUser stream	282
Setting a new user	283
UserService.ts	284
The MessagesService	285
the newMessages stream	285
the messages stream	287
The Operation Stream Pattern	287
Sharing the Stream	289
Adding Messages to the messages Stream	290
Our completed MessagesService	293
Trying out MessagesService	296
The ThreadsService	298
A map of the current set of Threads (in threads)	298
A chronological list of Threads, newest-first (in orderedthreads)	303
The currently selected Thread (in currentThread)	303
The list of Messages for the currently selected Thread (in currentThreadMessages)	305

CONTENTS

Our Completed ThreadsService	308
Data Model Summary	310
Data Architecture with Observables - Part 2: View Components	311
Building Our Views: The ChatApp Top-Level Component	311
The ChatThreads Component	314
ChatThreads Controller	314
ChatThreads template	315
The Single ChatThread Component	316
ChatThread Controller and ngOnInit	317
ChatThread template	318
ChatThread Complete Code	319
The ChatWindow Component	320
The ChatMessage Component	330
Setting incoming	331
The ChatMessage template	332
The Complete ChatMessage Code Listing	333
The ChatNavBar Component	338
The ChatNavBar @Component	338
The ChatNavBar Controller	338
The ChatNavBar template	340
The Completed ChatNavBar	340
Summary	342
Next Steps	343
Introduction to Redux with TypeScript	344
Redux	345
Redux: Key Ideas	345
Core Redux Ideas	346
What's a <i>reducer</i> ?	346
Defining Action and Reducer Interfaces	347
Creating Our First Reducer	348
Running Our First Reducer	348
Adjusting the Counter With <i>actions</i>	349
Reducer switch	350
Action "Arguments"	352
Storing Our State	353
Using the Store	354
Being Notified with subscribe	354
The Core of Redux	358
A Messaging App	359
Messaging App state	359
Messaging App actions	360

CONTENTS

Messaging App reducer	361
Trying Out Our Actions	364
Action Creators	365
Using Real Redux	367
Using Redux in Angular	369
Planning Our App	370
Setting Up Redux	370
Defining the Application State	371
Defining the Reducers	371
Defining Action Creators	372
Creating the Store	372
CounterApp Component	374
Providing the Store	375
Bootstrapping the App	376
The CounterComponent	377
imports	377
The template	377
The constructor	379
Putting It All Together	380
What's Next	382
References	383
Intermediate Redux in Angular	384
Context For This Chapter	384
Chat App Overview	385
Components	386
Models	386
Reducers	387
Summary	387
Implementing the Models	388
User	388
Thread	388
Message	389
App State	389
A Word on Code Layout	390
The Root Reducer	390
The UsersState	391
The ThreadsState	391
Visualizing Our AppState	393
Building the Reducers (and Action Creators)	394
Set Current User Action Creators	394
UsersReducer - Set Current User	395
Thread and Messages Overview	396

CONTENTS

Adding a New Thread Action Creators	396
Adding a New Thread Reducer	397
Adding New Messages Action Creators	398
Adding A New Message Reducer	399
Selecting A Thread Action Creators	401
Selecting A Thread Reducer	402
Reducers Summary	403
Building the Angular Chat App	403
The top-level ChatApp	405
The ChatPage	406
Container vs. Presentational Components	407
Building the ChatNavBar	408
Redux Selectors	410
Threads Selectors	411
Unread Messages Count Selector	412
Building the ChatThreads Component	413
ChatThreads Controller	414
ChatThreads template	415
The Single ChatThread Component	416
ChatThread @Component and template	417
Building the ChatWindow Component	418
The ChatMessage Component	426
Setting incoming	426
The ChatMessage template	427
Summary	428
Advanced Components	430
Styling	430
View (Style) Encapsulation	432
Shadow DOM Encapsulation	436
No Encapsulation	438
Creating a Popup - Referencing and Modifying Host Elements	441
Popup Structure	441
Using ElementRef	443
Binding to the host	445
Adding a Button using exportAs	448
Creating a Message Pane with Content Projection	450
Changing the host CSS	451
Using ng-content	451
Querying Neighbor Directives - Writing Tabs	453
Tab Component	454
Tabset Component	454
Using the Tabset	456

CONTENTS

Lifecycle Hooks	458
OnInit and OnDestroy	459
OnChanges	463
DoCheck	469
AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked	481
Advanced Templates	489
Rewriting ngIf - ngBookIf	490
Rewriting ngFor - ngBookRepeat	492
Change Detection	498
Customizing Change Detection	502
Zones	509
Observables and OnPush	510
Summary	514
Testing	515
Test driven?	515
End-to-end vs. Unit Testing	515
Testing Tools	516
Jasmine	516
Karma	517
Writing Unit Tests	517
Angular Unit testing framework	517
Setting Up Testing	518
Testing Services and HTTP	521
HTTP Considerations	522
Stubs	522
Mocks	523
Http MockBackend	524
TestBed.configureTestingModule and Providers	524
Testing getTrack	525
Testing Routing to Components	531
Creating a Router for Testing	532
Mocking dependencies	535
Spies	535
Back to Testing Code	538
fakeAsync and advance	540
inject	541
Testing ArtistComponent's Initialization	541
Testing ArtistComponent Methods	543
Testing ArtistComponent DOM Template Values	544
Testing Forms	547
Creating a ConsoleSpy	549
Installing the ConsoleSpy	550

CONTENTS

Configuring the Testing Module	551
Testing The Form	551
Refactoring Our Form Test	554
Testing HTTP requests	557
Testing a POST	558
Testing DELETE	559
Testing HTTP Headers	560
Testing YouTubeService	562
Conclusion	568
Converting an Angular 1 App to Angular 2	569
Peripheral Concepts	569
What We're Building	570
Mapping Angular 1 to Angular 2	571
Requirements for Interoperability	573
The Angular 1 App	573
The ng1-app HTML	575
Code Overview	576
ng1: PinsService	576
ng1: Configuring Routes	578
ng1: HomeController	579
ng1: / HomeController template	579
ng1: pin Directive	580
ng1: pin Directive template	580
ng1: AddController	582
ng1: AddController template	584
ng1: Summary	587
Building A Hybrid	587
Hybrid Project Structure	587
Bootstrapping our Hybrid App	592
What We'll Upgrade	593
A Minor Detour: Typing Files	596
Writing ng2 PinControlsComponent	602
Using ng2 PinControlsComponent	603
Downgrading ng2 PinControlsComponent to ng1	604
Adding Pins with ng2	606
Upgrading ng1 PinsService and \$state to ng2	608
Writing ng2 AddPinComponent	609
Using AddPinComponent	615
Exposing an ng2 service to ng1	615
Writing the AnalyticsService	616
Downgrade ng2 AnalyticsService to ng1	616
Using AnalyticsService in ng1	617

CONTENTS

Summary	618
References	619
NativeScript: Mobile Applications for the Angular Developer	620
What is NativeScript?	620
Where NativeScript Differs from Other Popular Frameworks	621
What are the System and Development Requirements for NativeScript?	622
Creating your First Mobile Application with NativeScript and Angular	624
Adding Build Platforms for Cross Platform Deployment	624
Building and Testing for Android and iOS	624
Installing JavaScript, Android, and iOS Plugins and Packages	625
Understanding the Web to NativeScript UI and UX Differences	626
Planning the NativeScript Page Layout	626
Adding UI Components to the Page	628
Styling Components with CSS	629
Developing a Geolocation Based Photo Application	630
Creating a Fresh NativeScript Project	631
Creating a Multiple Page Master-Detail Interface	632
Creating a Flickr Service for Obtaining Photos and Data	636
Creating a Service for Calculating Device Location and Distance	641
Including Mapbox Functionality in the NativeScript Application	645
Implementing the First Page of the Geolocation Application	646
Implementing the Second Page of the Geolocation Application	652
Try it out!	653
NativeScript for Angular Developers	654
Changelog	655
Revision 51 - 2017-02-14	655
Revision 50 - 2017-02-10	655
Revision 49 - 2017-01-18	655
Revision 48 - 2017-01-13	655
Revision 47 - 2017-01-06	655
Revision 46 - 2017-01-03	655
Revision 45 - 2016-12-05	655
Revision 44 - 2016-11-17	656
Revision 43 - 2016-11-08	656
Revision 42 - 2016-10-14	656
Revision 41 - 2016-09-28	656
Revision 40 - 2016-09-20	656
Revision 39 - 2016-09-03	657
Revision 38 - 2016-08-29	657
Revision 37 - 2016-08-02	657
Revision 36 - 2016-07-20	657

CONTENTS

Revision 35 - 2016-06-30	657
Revision 34 - 2016-06-15	657
Revision 33 - 2016-05-11	658
Revision 32 - 2016-05-06	658
Revision 31 - 2016-04-28	658
Revision 30 - 2016-04-20	659
Revision 29 - 2016-04-08	659
Revision 28 - 2016-04-01	659
Revision 27 - 2016-03-25	659
Revision 26 - 2016-03-24	659
Revision 25 - 2016-03-21	659
Revision 24 - 2016-03-10	659
Revision 23 - 2016-03-04	659
Revision 22 - 2016-02-24	660
Revision 21 - 2016-02-20	660
Revision 20 - 2016-02-11	660
Revision 19 - 2016-02-04	661
Revision 18 - 2016-01-29	661
Revision 17 - 2016-01-28	661
Revision 16 - 2016-01-14	661
Revision 15 - 2016-01-07	661
Revision 14 - 2015-12-23	661
Revision 13 - 2015-12-17	662
Revision 12 - 2015-11-16	662
Revision 11 - 2015-11-09	663
Revision 10 - 2015-10-30	663
Revision 9 - 2015-10-15	663
Revision 8 - 2015-10-08	664
Revision 7 - 2015-09-23	664
Revision 6 - 2015-08-28	664
Revision 5	664
Revision 4	664
Revision 3	665
Revision 2	665
Revision 1	665

Book Revision

Revision 51 - Covers up to Angular 2 (2.4.7, 2017-02-14)

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: us@fullstack.io¹.

Chat With The Community!

We're experimenting with a community chat room for this book using Gitter. If you'd like to hang out with other people learning Angular 2, come [join us on Gitter](https://gitter.im/ng-book/ng-book)²!

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, [follow @fullstackio](https://twitter.com/fullstackio)³

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io⁴.

¹<mailto:us@fullstack.io?Subject=ng-book%20%20feedback>

²<https://gitter.im/ng-book/ng-book>

³<https://twitter.com/fullstackio>

⁴<mailto:us@fullstack.io?Subject=ng-book%20%20testimonial>



Considering Angular.JS for your next project?

- Using Angular for your Internal Tools on top of an existing API?
- Building a modern single page app, cross-device mobile app?
- Having issues implementing Angular on Rails?
- Need a project that needs to be rescued?

You don't have to do it alone, we - **the authors of ng-book can help.**

We can help you:

- Build your next MVP
- Testing and Code Coverage
- Implement Build and Deployment automation
- We support Rails, Backbone.JS, React.JS and others.

We offer a free 30min discussion where we can help you identify if you might need our assistance either doing consulting, engineering or training your team in Angular.JS.

Request Free Consultation

Get in touch with carlos@ng-book.com or <http://ng-book.com/consulting>

⁵<mailto:us@fullstack.io>

Writing your First Angular 2 Web Application

Simple Reddit Clone

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

You can think of this app as the beginnings of a site like [Reddit](http://reddit.com)⁶ or [Product Hunt](http://producthunt.com)⁷.

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

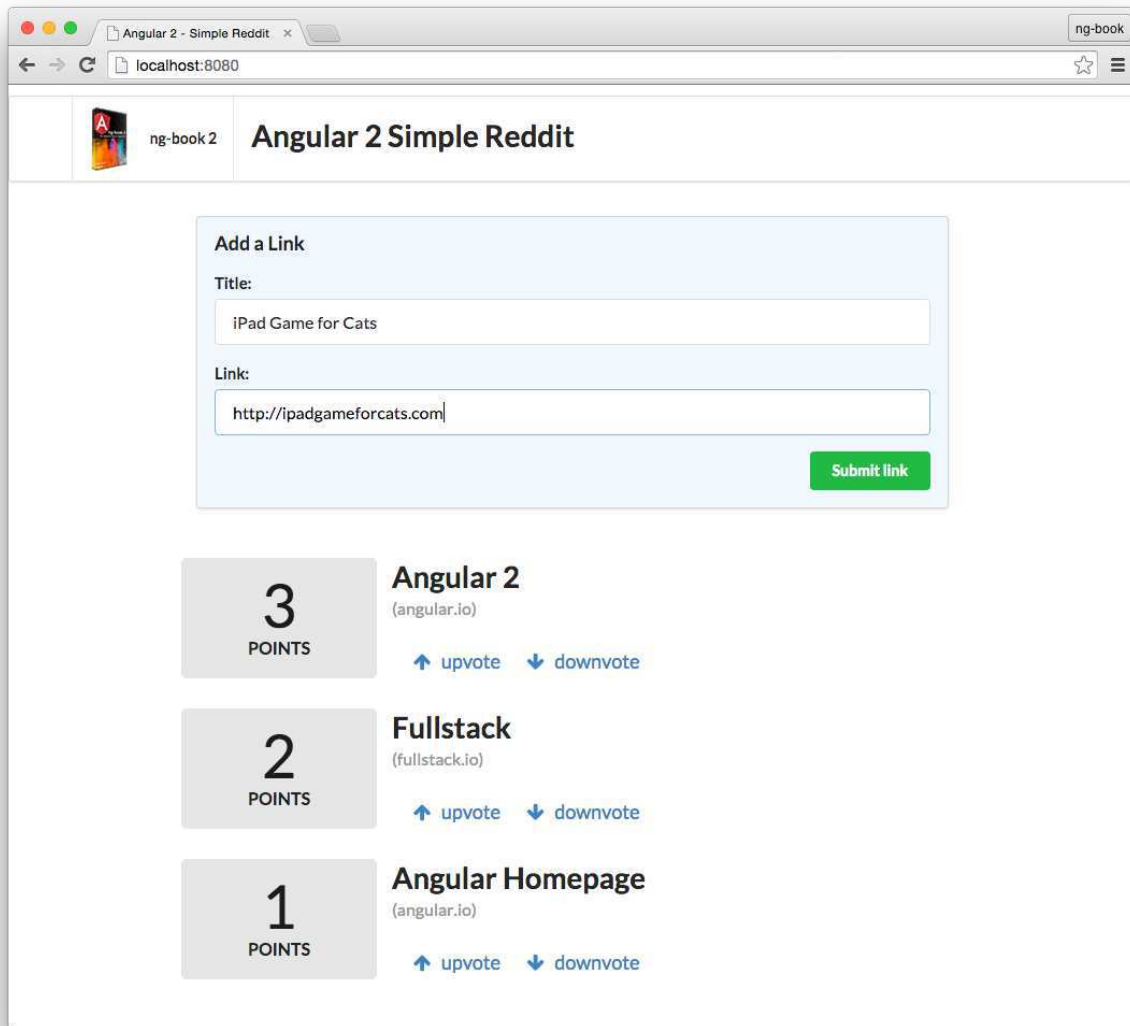
- Building custom components
- Accepting user input from forms
- Rendering lists of objects into views
- Intercepting user clicks and acting on them

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

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

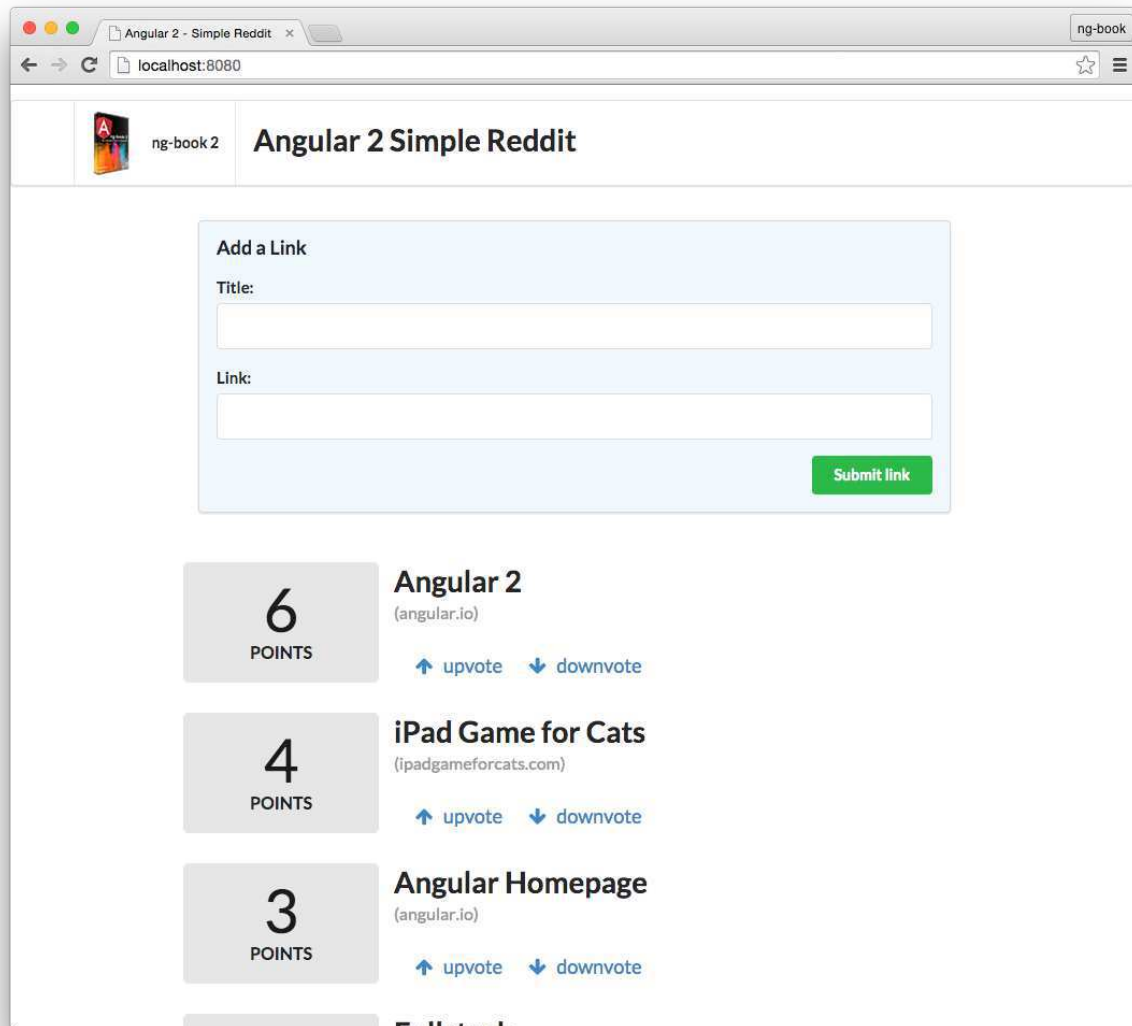
⁶<http://reddit.com>

⁷<http://producthunt.com>



Completed application

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



App with new article

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

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

Getting started

TypeScript

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



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

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

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



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



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

angular-cli

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

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

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

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

⁸<https://nodejs.org/download/>

⁹<https://www.cygwin.com/>


```
1 $ ng
2 Could not start watchman; falling back to NodeWatcher for file system events.
3 Visit http://ember-cli.com/user-guide/#watchman for more info.
4 Usage: ng <command> (Default: help)>
```

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

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

```
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.

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

Example Project

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

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

```
1 $ ng new angular2_hello_world
```

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/app/index.ts
10  create src/app/shared/index.ts
11  create src/assets/.gitkeep
12  create src/assets/.npmignore
13  create src/environments/environment.dev.ts
14  create src/environments/environment.prod.ts
15  create src/environments/environment.ts
16  create src/favicon.ico
17  create src/index.html
18  create src/main.ts
19  create src/polyfills.ts
20  create src/styles.css
21  create src/test.ts
22  create src/tsconfig.json
23  create src/typings.d.ts
24  create angular-cli.json
25  create e2e/app.e2e-spec.ts
26  create e2e/app.po.ts
27  create e2e/tsconfig.json
28  create .gitignore
29  create karma.conf.js
30  create package.json
31  create protractor.conf.js
32  create tslint.json
33 Successfully initialized git.
34 □ Installing packages for tooling via npm
```

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

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

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

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

```
1 $ cd angular2_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
13
14 3 directories, 6 files
```

For now, the folder we're interested in is `src`, where our application lives. 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 |   |-- index.ts
11 |   |-- shared/
12 |       |-- index.ts
13 |-- assets/
14 |-- environments/
15 |   |-- environment.dev.ts
16 |   |-- environment.prod.ts
17 |   |-- environment.ts
18 |-- favicon.ico
19 |-- index.html
20 |-- main.ts
21 |-- polyfills.ts
```

```
22 |-- styles.css
23 |-- test.ts
24 |-- tsconfig.json
25 `-- typings.d.ts
26
27 4 directories, 18 files
```

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

`code/first_app/angular2_hello_world/src/index.html`

```
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>Angular2HelloWorld</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/angular2_hello_world/src/index.html`

```
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>Angular2HelloWorld</title>
6   <base href="/">
```

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

code/first_app/angular2_hello_world/src/index.html

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

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

code/first_app/angular2_hello_world/src/index.html

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

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

Writing Application Code

Running the application

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

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

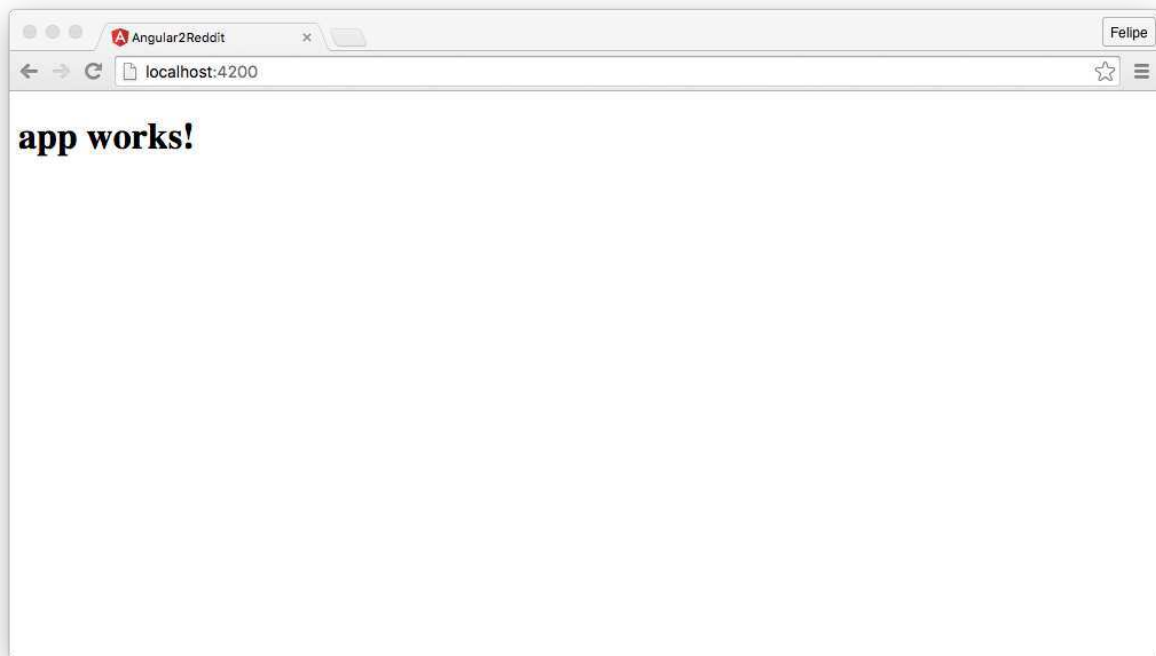
Our application is now running on localhost port 4200. Let's open the browser and visit:

<http://localhost:4200>¹⁰



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

¹⁰<http://localhost:4200>



Running application

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

Making a Component

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

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

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

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



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

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

```
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 annotation
2. A component definition class

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

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

```
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 }
```



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

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

Importing Dependencies

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

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

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

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

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

Component Annotations

After importing our dependencies, we are declaring the component:

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

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

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

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

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

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

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

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

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

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

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

Adding a template with `templateUrl`

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

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

```
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 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 2 uses a concept called “style-encapsulation” which means that styles specified for a particular component *only apply to that component*. We talk more about this in-depth later on in the book in the [Styling section of Advanced Components](#).

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



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

Loading Our Component

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

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

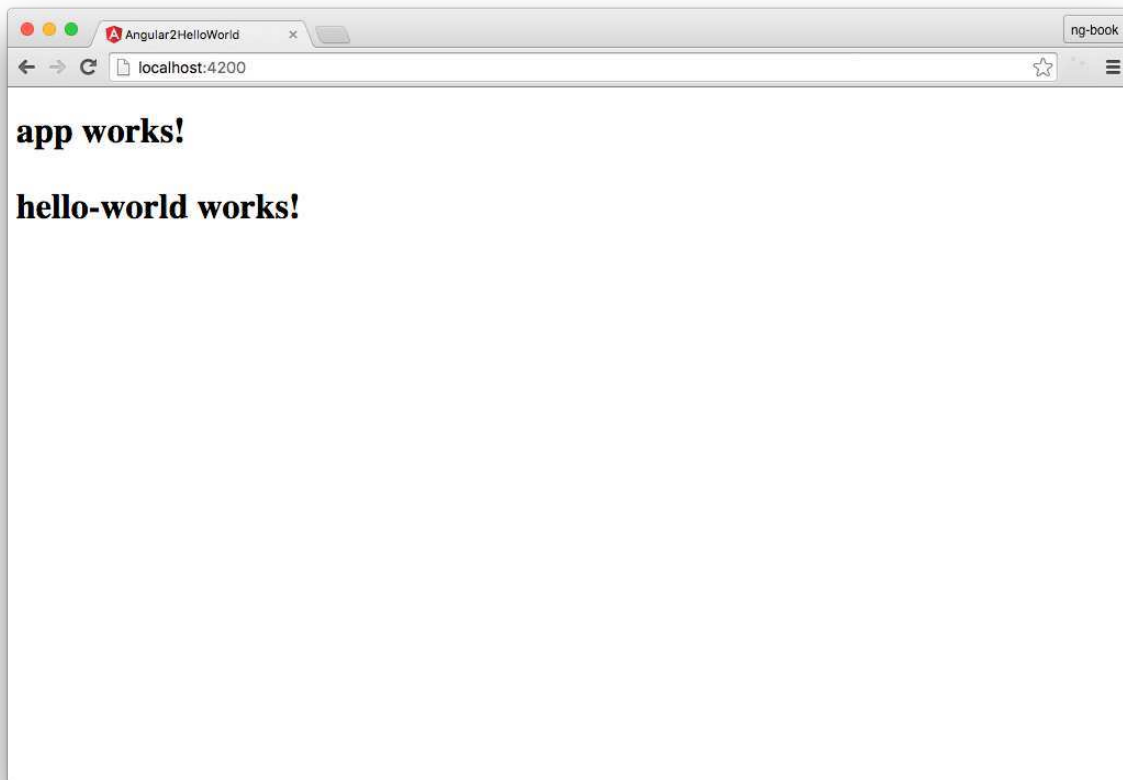
In order to change that, we need to add our component tag to a template that is already being rendered. Open up the file: `first_app/angular2_hello_world/src/app/app.component.html`

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

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

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

Now refresh the page and take a look:



Hello world works

It works!

Adding Data to the Component

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

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

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

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

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

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

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

```
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/angular2_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 `name` is the name of the attribute we want to set and `string` is the *type*.

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

2. A Constructor

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

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

When we write:

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

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

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

Rendering The Template

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

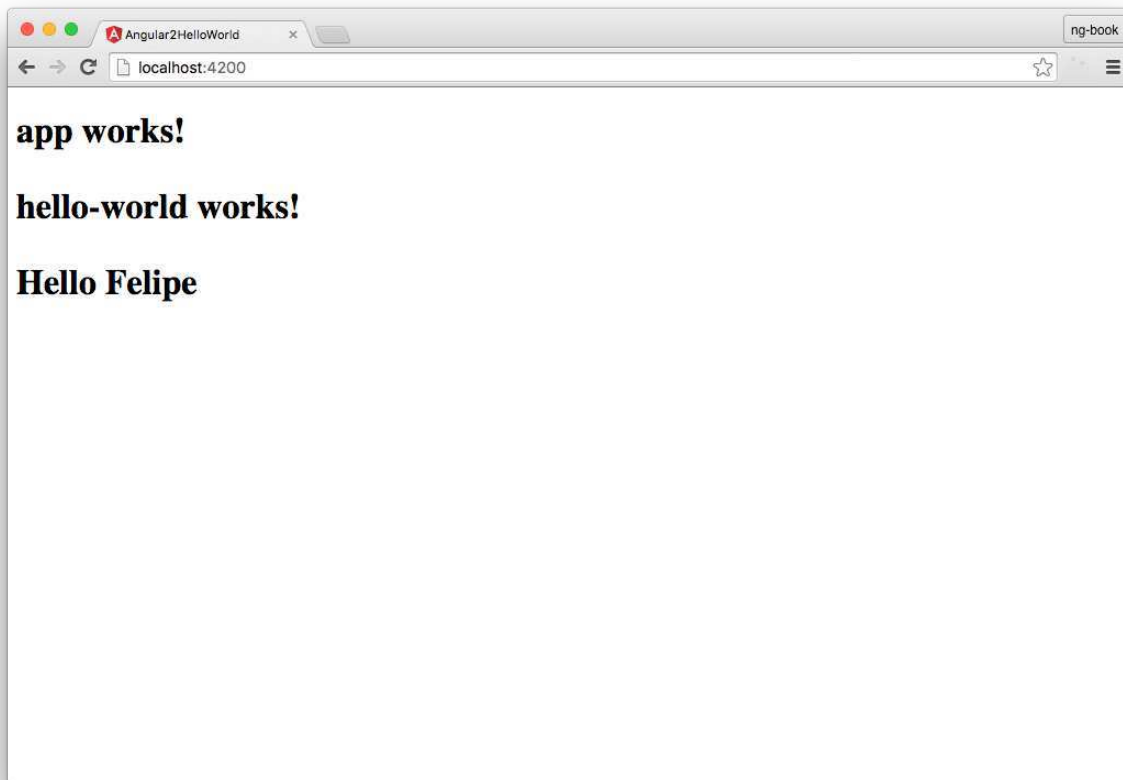
`code/first_app/angular2_hello_world/src/app/user-item/user-item.component.html`

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

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

Try it out

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



Application with Data

Working With Arrays

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

If you’ve worked with Angular 1 before, you’ve probably used the `ng-repeat` directive. In Angular 2, the analogous directive is called `NgFor` (we use it in the markup as `*ngFor`, which we’ll talk about soon). Its syntax is slightly different but they have the same purpose: **repeat the same markup for a collection of objects**.

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

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

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

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

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

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

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

```
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/angular2_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 `HelloWorld` object. `let name` is called a *reference*. When we say `"let name of names"` we're saying loop over each element in `names` and assign each one to a *local* variable called `name`.

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



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

```
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>
```

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



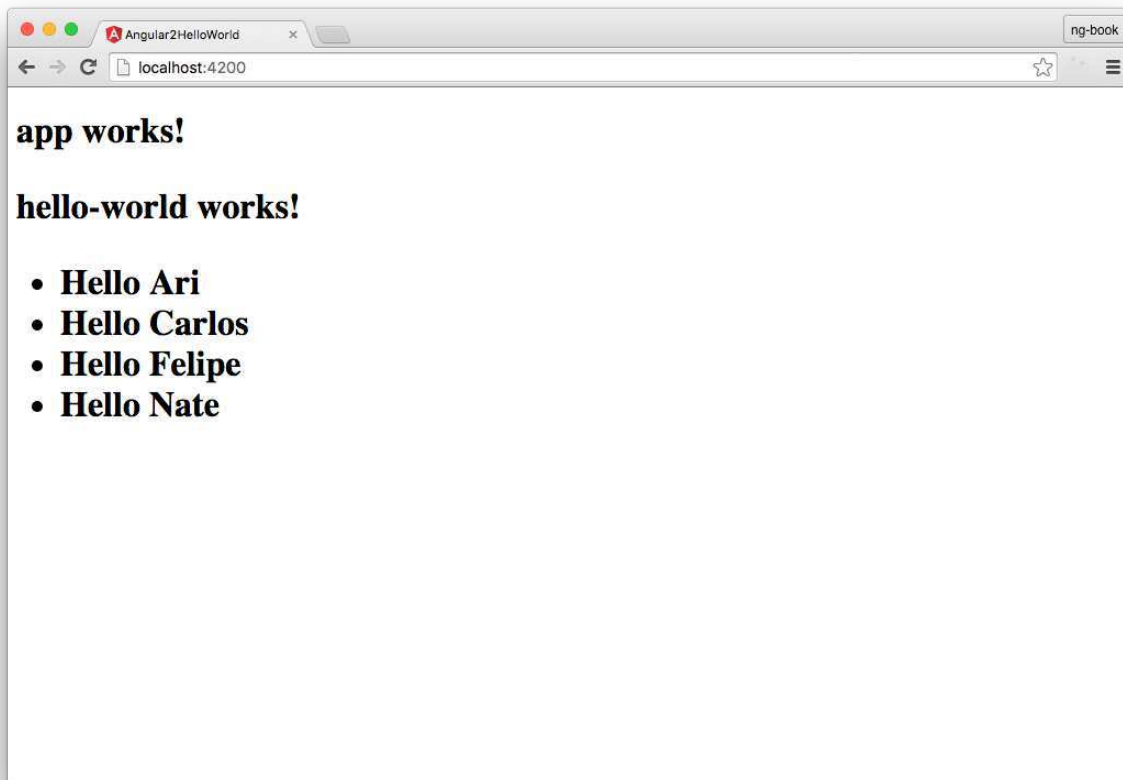
`NgFor` repeats the element that the `ngFor` is called. That is, we put it on the `li` tag and **not** the `ul` tag because we want to repeat the list element (`li`) and not the list itself (`ul`).



If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the [NgFor directive here](https://github.com/angular/angular/blob/master/modules/%40angular/common/src/directives/ng_for.ts)¹¹

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

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



Application with Data

Using the User Item Component

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

To do this, we need to do three things:

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

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

Rendering the `UserItemComponent`

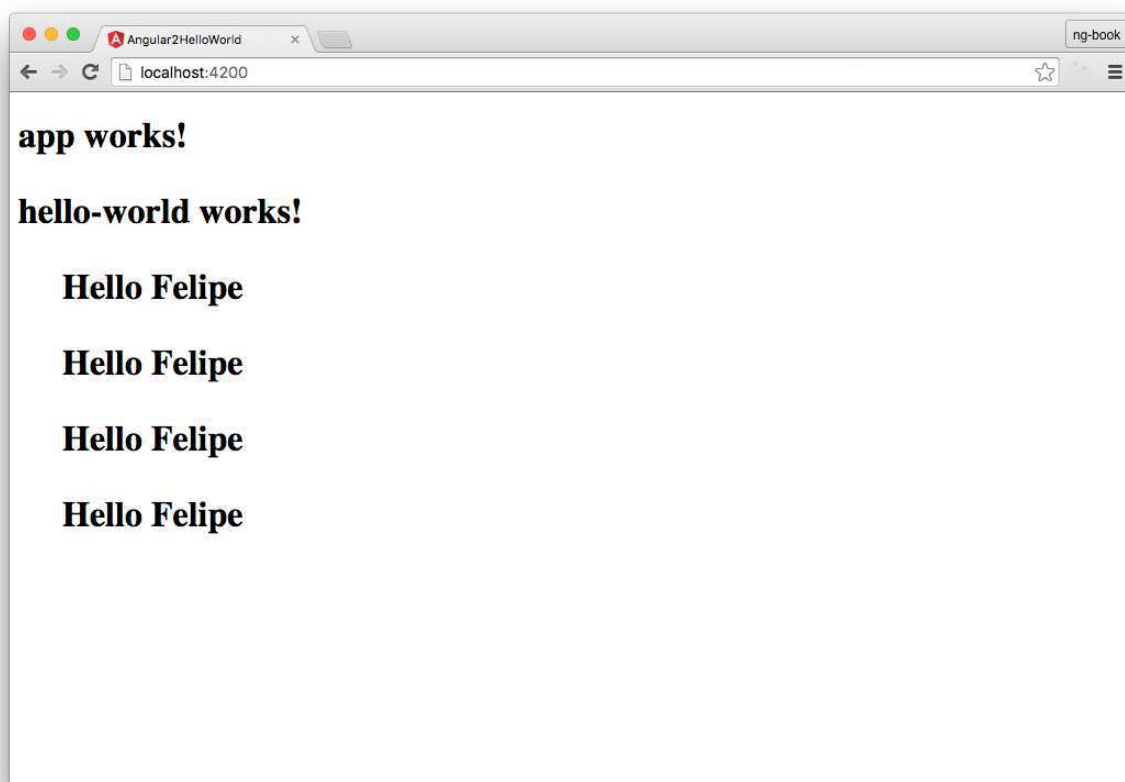
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template. What we're going to do is replace the `` tag with the `app-user-item` tag:

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

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

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

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



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

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

Accepting Inputs

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

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

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

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

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

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

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

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

Passing an Input value

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

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

```
1 <ul>
2   <app-user-item
3     *ngFor="let name of names"
4     [name]="name">
5   </app-user-item>
6 </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:

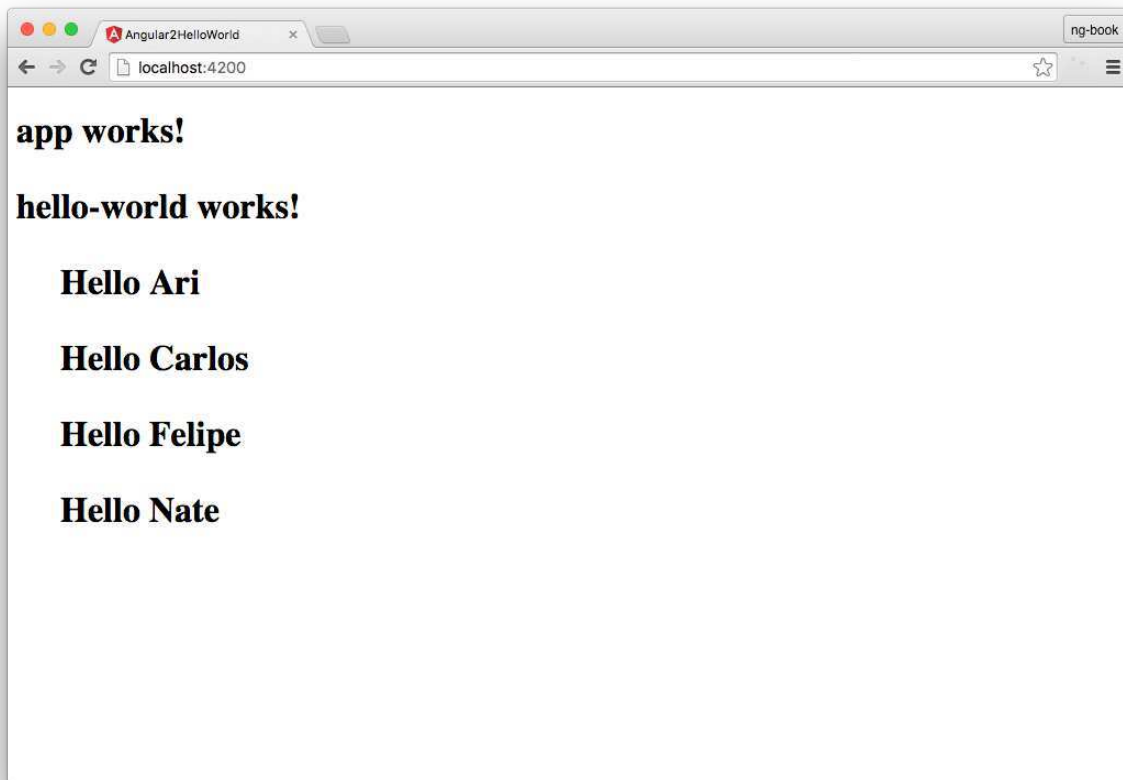
```
1 <app-user-item
2   *ngFor="let individualUserName of names"
3   [name]="individualUserName">
4 </app-user-item>
```

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

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

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

Now rendering our list of names is working!



Application with Names Working

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

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

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

Bootstrapping Crash Course

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

We run this app by calling the command:


```
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.

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

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

Let's a look at the code:

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

```
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` annotation. Like all annotations, this `@NgModule(...)` code adds metadata to the class immediately following (`AppModule`).

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

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

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

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

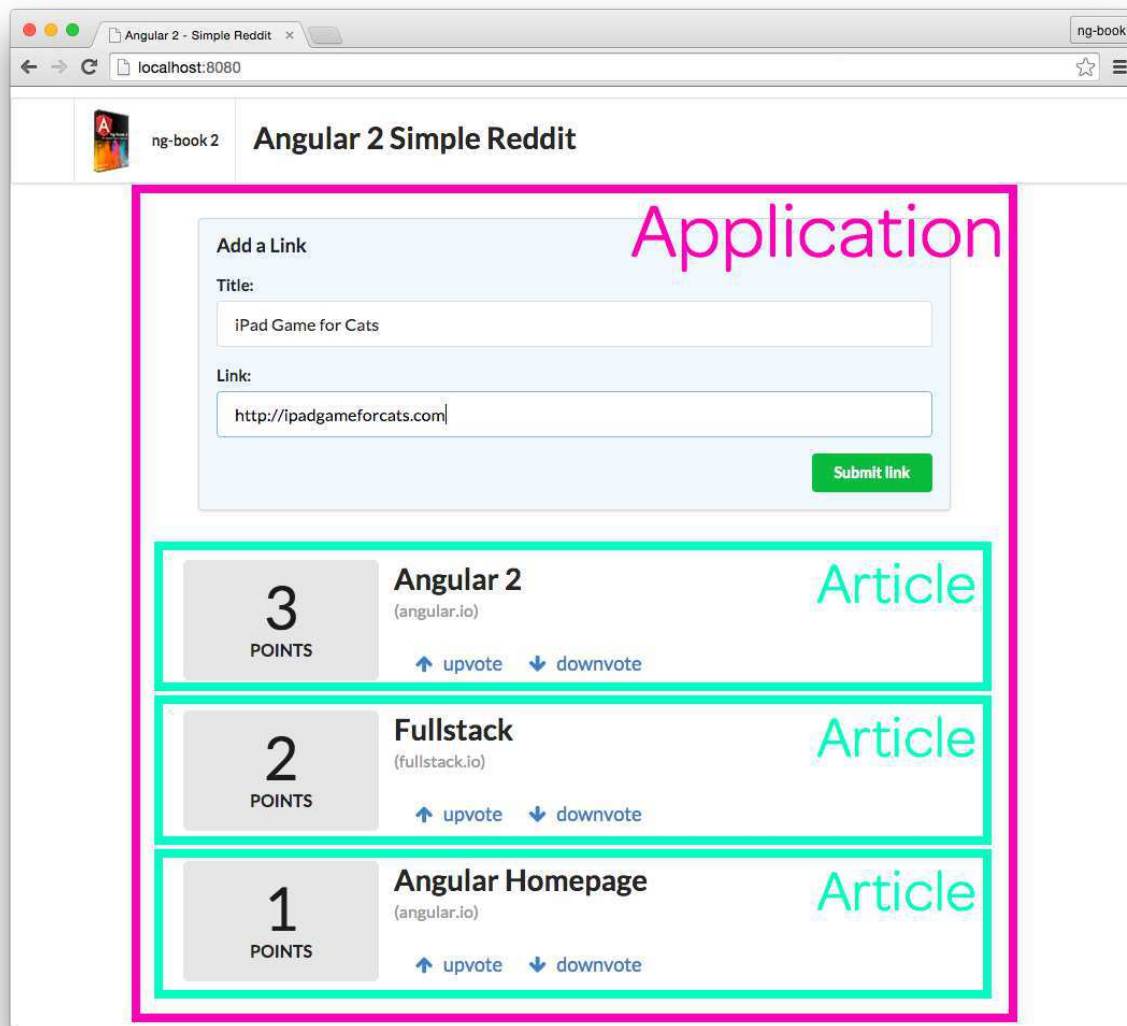
`bootstrap` tells Angular that when this module is used to bootstrap an app, we need to load the `AppComponent` component as the top-level component.



We talk more about `NgModules` in [the section on `NgModules`](#)

Expanding our Application

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



Application with Data

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

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



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

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

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

```
1 ng new angular2_reddit
```



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

Adding CSS

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



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

Copy:

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

into your application's folder.

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

The Application Component

Let's now build a new component which will:

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

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

¹²<http://semantic-ui.com/>

¹³<http://foundation.zurb.com>

¹⁴<http://getbootstrap.com>

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

```
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 }
```

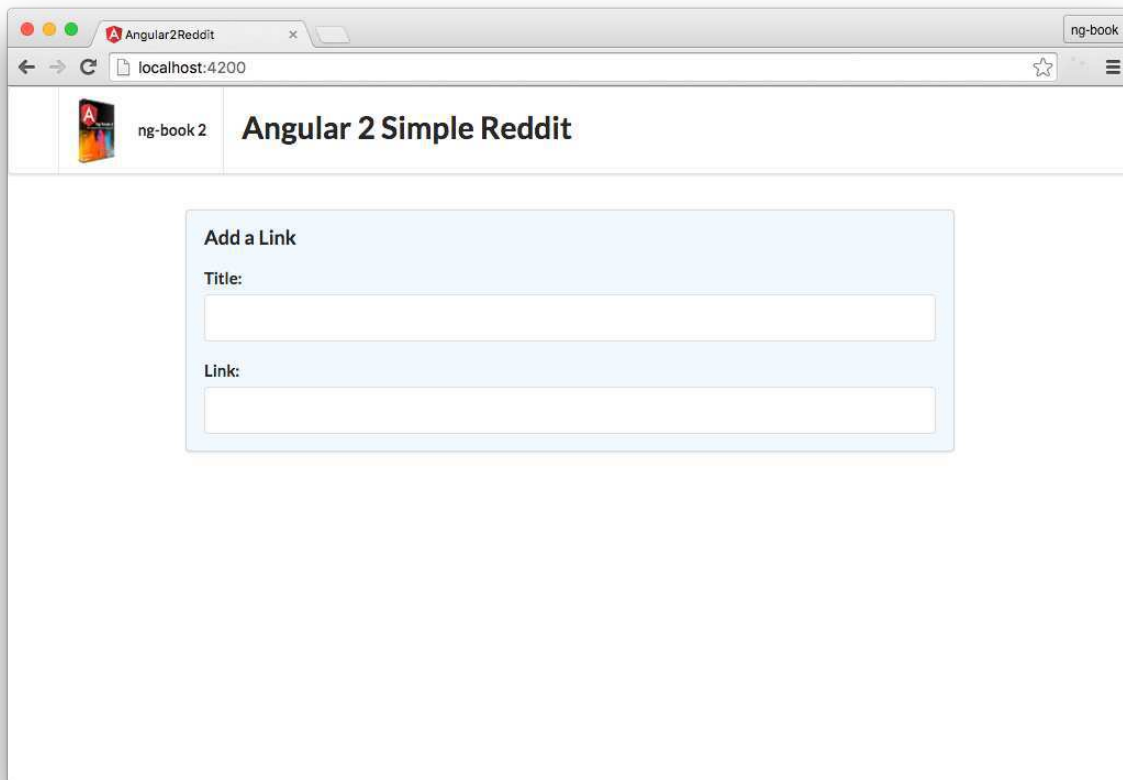
Let's change the template a bit to include a form for adding links. We'll use a bit of styling from the semantic-ui package to make the form look a bit nicer:

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

```
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 parenthesis `()`. For instance, to add a function call to the `<button />` `onClick` event, we can pass it through like so:

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

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

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

```

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/angular2_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 `#title` and `#link` to the appropriate `<input />` elements, we can **pass them as variables** into the `addArticle()` function on the button!

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

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

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

Binding 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/angular2_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   }
```

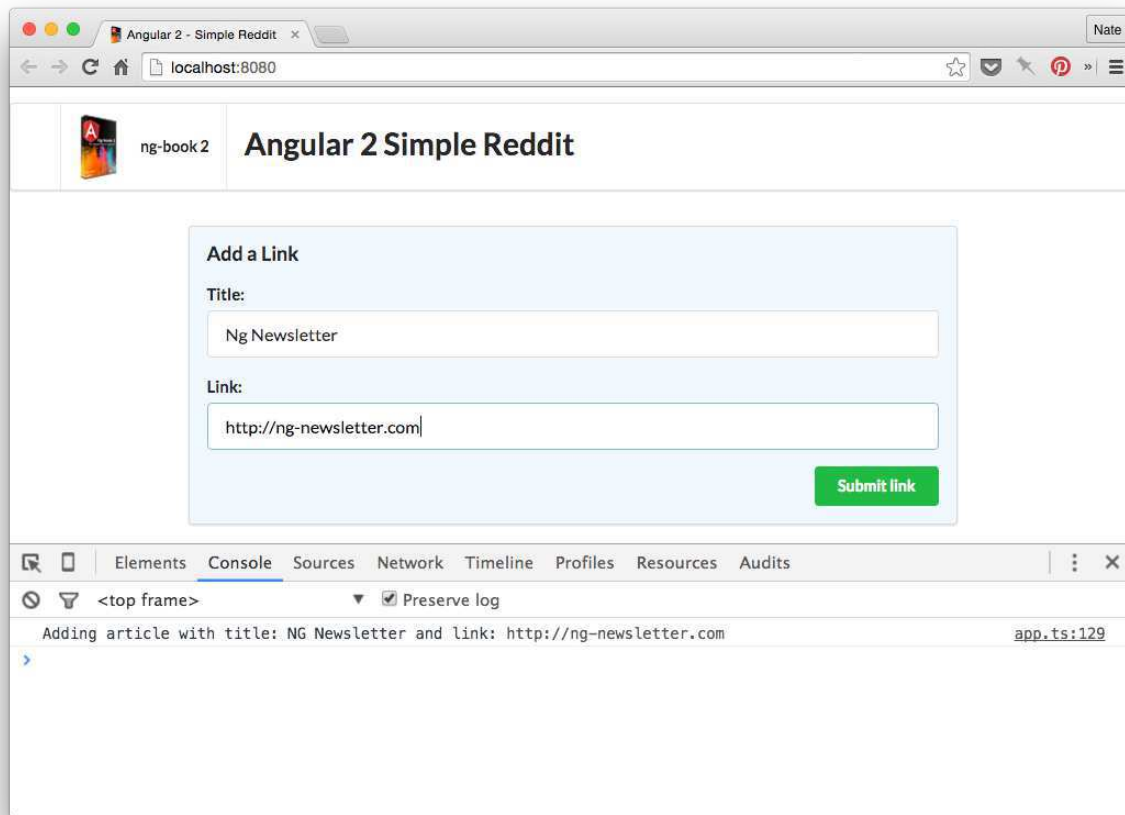


Notice that we're using backtick strings again. This is a really handy feature of ES6: backtick strings will expand template variables!

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/angular2_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/angular2_reddit/src/app/article/article.component.ts

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

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

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

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

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

We want each `app-article` to be on its own row. We're using Semantic UI, and Semantic provides a [CSS class for rows¹⁵](http://semantic-ui.com/collections/grid.html) called `row`.

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



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

Creating the `ArticleComponent` Definition Class

Finally, we create the `ArticleComponent` definition class:

¹⁵<http://semantic-ui.com/collections/grid.html>

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

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

Here we create three properties on ArticleComponent:

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

In the constructor() we set some default attributes:

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

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

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

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

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

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

Using the `app-article` Component

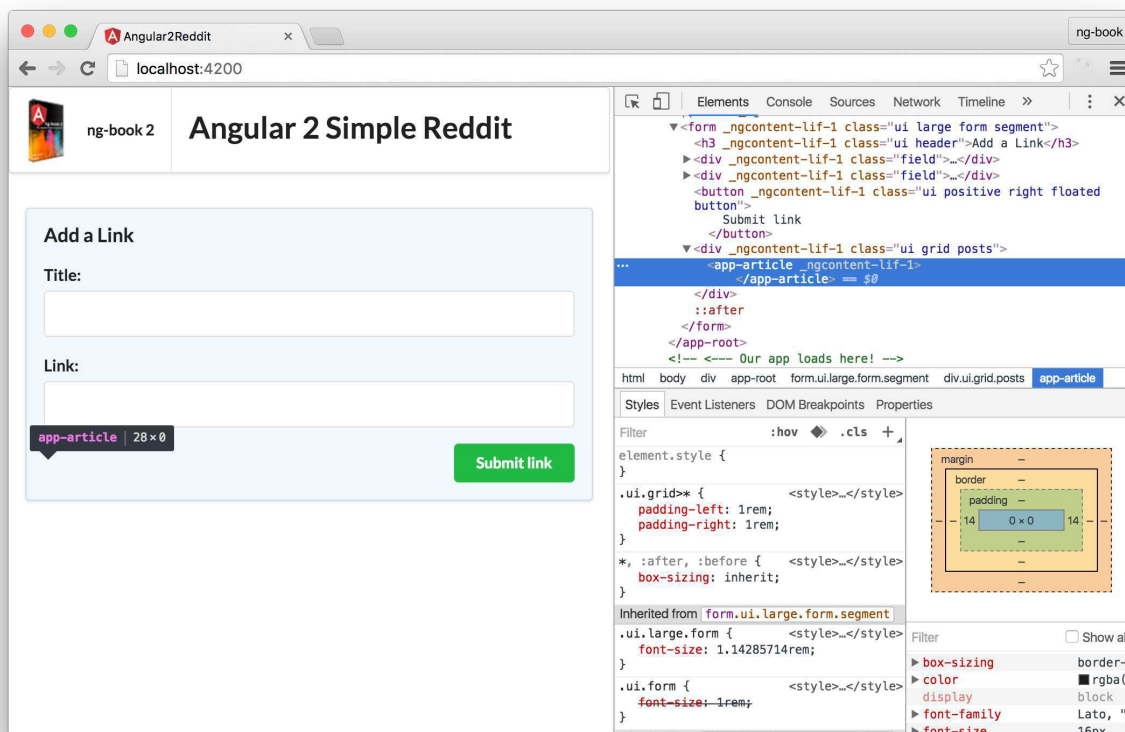
In order to use this component and make the data visible, we have to add a `<app-article></app-article>` tag somewhere in our markup.

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

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

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



Unexpanded tag when inspecting the DOM

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



Angular 1 Note: If you've used Angular 1 it might be surprising that our app doesn't know about our new `app-article` component. This is because in Angular 1, directives match globally. However, in Angular you need 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 (`RedditAppModule`). However, if `ArticleComponent` were part of a *different* module, then we might import it with imports.

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

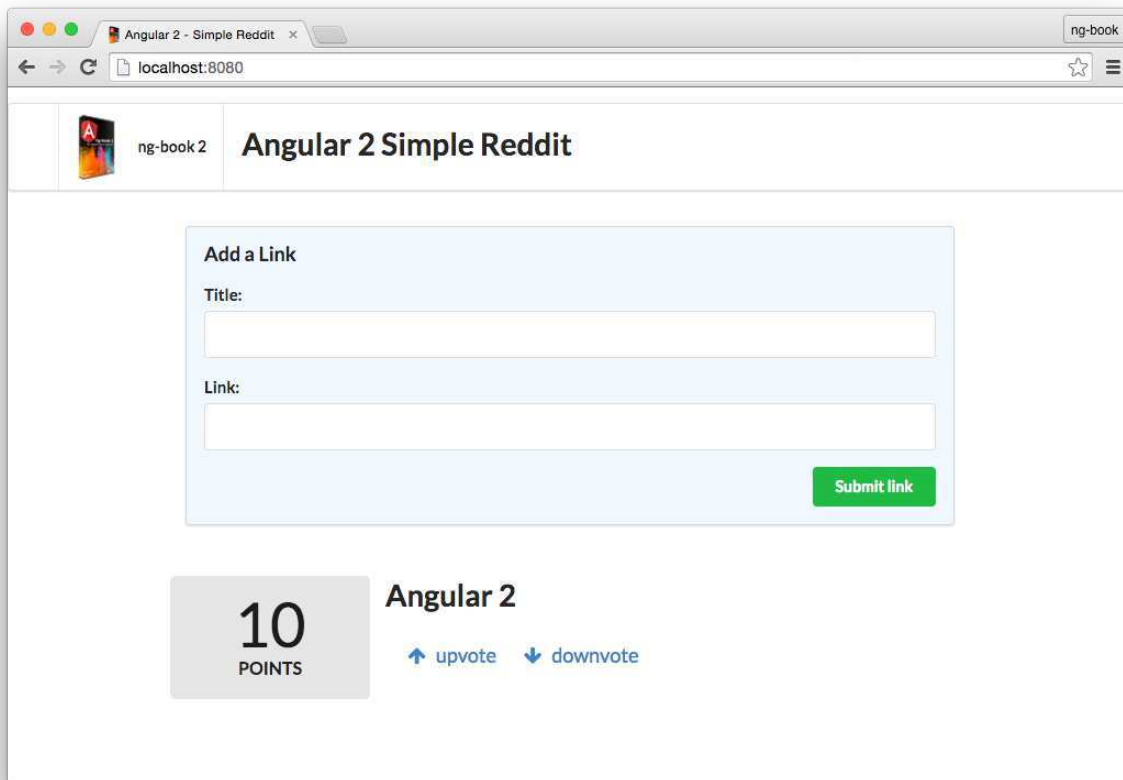
code/first_app/angular2_reddit/src/app/app.module.ts

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

See here that we are:

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

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



Rendered ArticleComponent component

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

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

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

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

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

Rendering Multiple Rows

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

Creating an Article class

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

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

```
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/angular2_reddit/src/app/article/article.component.ts

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

Then let's use it:

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

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

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

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

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

code/first_app/angular2_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)¹⁶ which says that a given object should assume as little as possible about the structure or properties of other objects.

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

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

code/first_app/angular2_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(): string {
21     try {
22       const link: string = this.link.split('///')[1];
23       return link.split('/')[0];
24     } catch (err) {
25       return null;
26     }
27   }
28 }
```

We can then change ArticleComponent to call these methods:

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

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

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



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

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

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

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

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



Checkout our `ArticleComponent` component definition now: it’s so short! We’ve moved a lot of logic **out** of our component and into our models. The corresponding MVC guideline here might be [Fat Models, Skinny Controllers](http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model)¹⁷. The idea is that we want to move most of our logic to our models so that our components do the minimum work possible.

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

Storing Multiple 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/angular2_reddit/src/app/app.component.ts

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

Notice that our AppComponent has the line:

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

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

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

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

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

Configuring the ArticleComponent with inputs

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

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

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

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

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

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

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

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

```
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/angular2_reddit/src/app/article/article.component.ts

```
16 export class ArticleComponent implements OnInit {  
17   @Input() article: Article;  
18  
19   voteUp(): boolean {  
20     this.article.voteUp();  
21     return false;  
22   }  
23  
24   voteDown(): boolean {  
25     this.article.voteDown();  
26     return false;  
27   }  
28  
29   ngOnInit() {  
30   }  
31  
32 }
```

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:

```
1      Submit link
2    </button>
3  </form>
4
5  <!-- start adding here -->
6  <div class="ui grid posts">
7    <app-article
8      *ngFor="let article of articles"
9      [article]="article">
10   </app-article>
11 </div>
12 <!-- 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`:

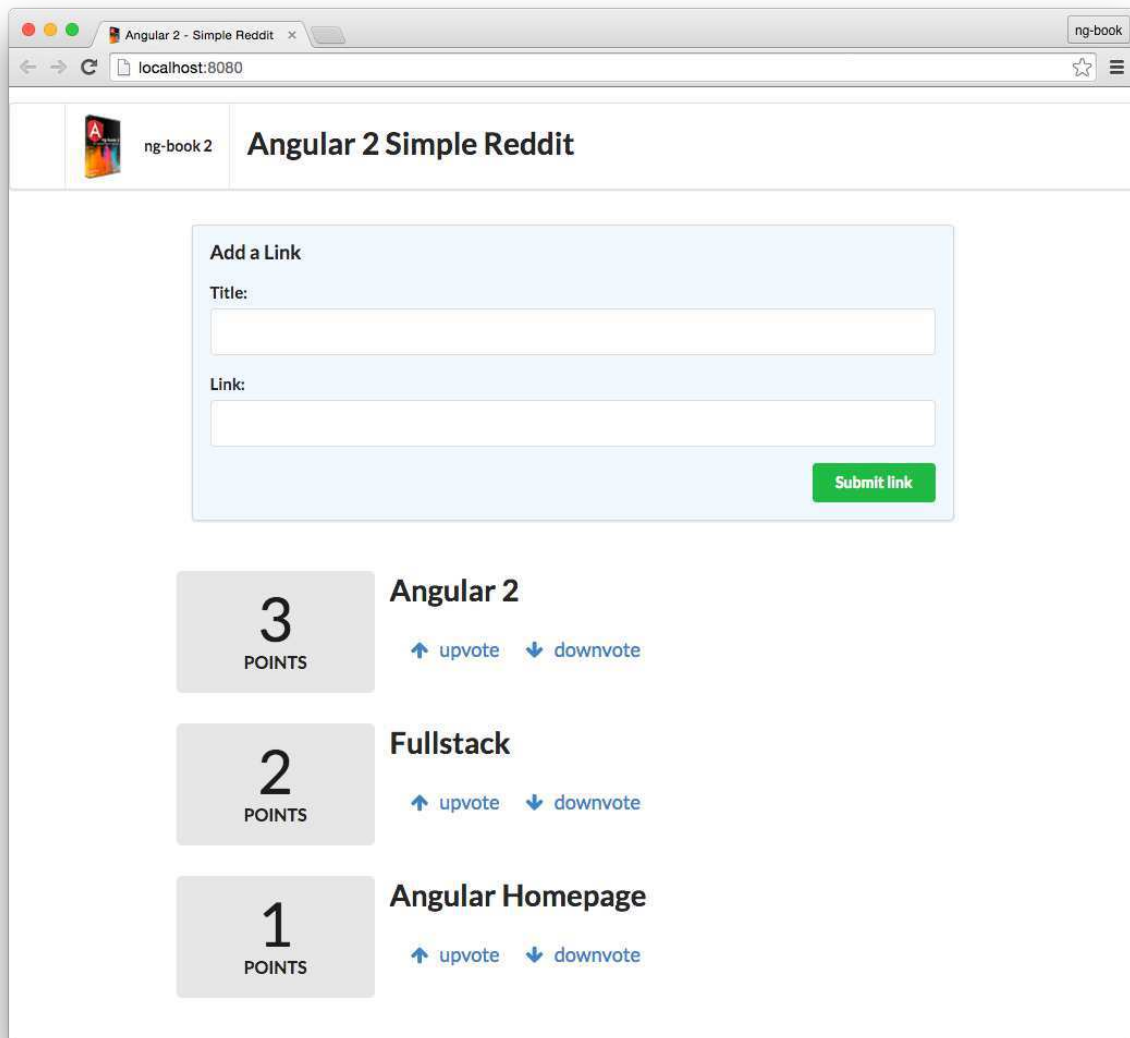
```
1  <app-article
2    *ngFor="let foobar of articles"
3    [article]="foobar">
4  </app-article>
```

So here we have three variables:

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

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

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



Multiple articles being rendered

Adding New 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/angular2_reddit/src/app/app.component.ts

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

This will:

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



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

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

Finishing Touches

Displaying the Article Domain

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

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

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

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

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

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

And now when we reload the browser, we will see the domain name of each URL (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/angular2_reddit/src/app/app.component.ts

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

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

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

Full Code Listing

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

Wrapping Up

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

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

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

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

Getting Help

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

- Come join our (free!) community and [chat with us on Gitter](https://gitter.im/ng-book/ng-book)¹⁸
- Email us directly at us@fullstack.io¹⁹

Onward!

¹⁸<https://gitter.im/ng-book/ng-book>

¹⁹<mailto:us@fullstack.io>

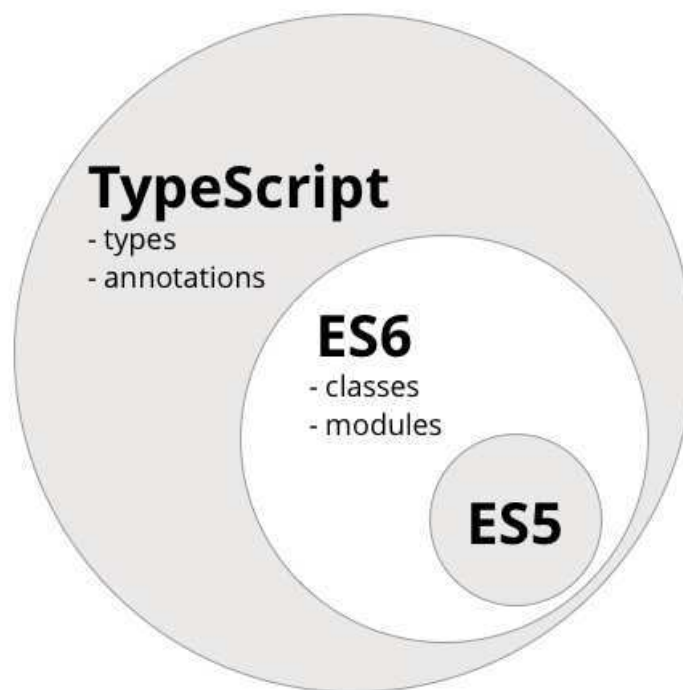
TypeScript

Angular 2 is built in TypeScript

Angular 2 is built in a Javascript-like language called [TypeScript](http://www.typescriptlang.org/)²⁰.

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

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



ES5, ES6, and TypeScript



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

²⁰<http://www.typescriptlang.org/>

This is the end of the preview!

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

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

