



ng-book

The Complete Book on Angular 8



FULLSTACK.io



gistia

Nate Murray
Felipe Coury
Ari Lerner
Carlos Taborda

ng-book

The Complete Guide to Angular

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

© 2018 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.

Published in San Francisco, California by Fullstack.io.

We'd like to thank our contributors to this book including:

- *Frode Fikke for technical edits*
- *[Nic Raboy](#), and [Burke Holland](#) for their contribution to the NativeScript chapter*
- *Travas Nolte for his work on many chapters*
- *Daniel Rauf for his edits on RxJS, Redux, and Routing*

Contents

Book Revision	1
Bug Reports	1
Vote for New Content (new!)	1
Be notified of updates via Twitter	1
We'd love to hear from you!	1
How to Read This Book	2
Running Code Examples	2
Angular CLI	3
Code Blocks and Context	3
Code Block Numbering	4
A Word on Versioning	4
Getting Help	5
Emailing Us	6
Chapter Overview	6
Writing Your First Angular Web Application	1
Simple Reddit Clone	1
Getting started	4
Node.js and npm	4
TypeScript	4
Browser	5
Special instruction for Windows users	5
Angular CLI	5
Example Project	7
Writing Application Code	11
Running the application	11
Making a Component	13

CONTENTS

Importing Dependencies	15
Component Decorators	16
Adding a template with <code>templateUrl</code>	17
Adding a template	17
Adding CSS Styles with <code>styleUrls</code>	18
Loading Our Component	19
Adding Data to the Component	20
Working With Arrays	24
Using the User Item Component	28
Rendering the <code>UserItemComponent</code>	29
Accepting Inputs	30
Passing an Input value	31
Bootstrapping Crash Course	33
declarations	35
imports	35
providers	36
bootstrap	36
Expanding our Application	36
Adding CSS	38
The Application Component	39
Adding Interaction	41
Adding the Article Component	46
Rendering Multiple Rows	56
Creating an Article class	56
Storing Multiple Articles	61
Configuring the <code>ArticleComponent</code> with inputs	63
Rendering a List of Articles	65
Adding New Articles	67
Finishing Touches	68
Displaying the Article Domain	68
Re-sorting Based on Score	69
Deployment	70
Building Our App for Production	71
Uploading to a Server	72
Installing now	72
Full Code Listing	73

CONTENTS

Wrapping Up	73
Getting Help	73
TypeScript	74
Angular is built in TypeScript	74
What do we get with TypeScript?	76
Types	77
Trying it out with a REPL	79
Built-in types	79
Classes	82
Properties	82
Methods	83
Constructors	85
Inheritance	87
Utilities	89
Fat Arrow Functions	89
Template Strings	91
Wrapping up	92
How Angular Works	93
Application	93
The Navigation Component	95
The Breadcrumbs Component	95
The Product List Component	95
How to Use This Chapter	98
Product Model	99
Components	100
Component Decorator	103
Component selector	103
Component template	104
Adding A Product	104
Viewing the Product with Template Binding	107
Adding More Products	107
Selecting a Product	109
Listing products using <products-list>	109
The ProductsListComponent	113

CONTENTS

Configuring the ProductsListComponent @Component Options	113
Component inputs	114
Component outputs	116
Emitting Custom Events	118
Writing the ProductsListComponent Controller Class	120
Writing the ProductsListComponent View Template	121
The Full ProductsListComponent Component	123
The ProductRowComponent Component	125
ProductRowComponent Configuration	126
ProductRowComponent template	127
The ProductImageComponent Component	128
The PriceDisplayComponent Component	128
The ProductDepartmentComponent	129
NgModule and Booting the App	131
Booting the app	132
The Completed Project	133
Deploying the App	134
A Word on Data Architecture	135
Built-in Directives	137
Introduction	137
NgIf	137
NgSwitch	138
NgStyle	140
NgClass	143
NgFor	147
Getting an index	152
NgNonBindable	153
Conclusion	154
Forms in Angular	155
Forms are Crucial, Forms are Complex	155
FormControls and FormGroups	156
FormControl	156
FormGroup	157
Our First Form	158

CONTENTS

Loading the FormsModule	159
Reactive- vs. template-driven Forms	160
Simple SKU Form: @Component Decorator	160
Simple SKU Form: template	161
Simple SKU Form: Component Definition Class	165
Try it out!	166
Using FormBuilder	167
Reactive Forms with FormBuilder	168
Using FormBuilder	168
Using myForm in the view	170
Try it out!	171
Adding Validations	173
Explicitly setting the sku FormControl as an instance variable	174
Custom Validations	179
Watching For Changes	181
ngModel	183
Wrapping Up	185
Dependency Injection	186
Injections Example: PriceService	187
Dependency Injection Parts	192
Playing with an Injector	193
Providing Dependencies with NgModule	196
Providers are the Key	198
Providers	199
Using a Class	199
Using a Factory	204
Dependency Injection in Apps	207
More Resources	207
HTTP	208
Introduction	208
Using @angular/common/http	209
import from @angular/common/http	209
A Basic Request	211
Building the SimpleHttpComponent Component Definition	212

CONTENTS

Building the SimpleHttpComponent template	212
Building the SimpleHttpComponent Controller	213
Full SimpleHttpComponent	214
Writing a YouTubeSearchComponent	215
Writing a SearchResult	217
Writing the YouTubeSearchService	218
Writing the SearchBoxComponent	224
Writing SearchResultComponent	232
Writing YouTubeSearchComponent	233
@angular/common/http API	237
Making a POST request	238
PUT / PATCH / DELETE / HEAD	238
Custom HTTP Headers	239
Summary	240
Routing	241
Why Do We Need Routing?	241
How client-side routing works	242
The beginning: using anchor tags	243
The evolution: HTML5 client-side routing	244
Writing our first routes	245
Components of Angular routing	245
Imports	245
Routes	246
Installing our Routes	248
RouterOutlet using <router-outlet>	248
RouterLink using [routerLink]	250
Putting it all together	251
Creating the Components	253
HomeController	253
AboutComponent	254
ContactComponent	255
Application Component	256
Configuring the Routes	257
Routing Strategies	259
Running the application	260

CONTENTS

Route Parameters	263
ActivatedRoute	264
Music Search App	265
First Steps	267
The SpotifyService	268
The SearchComponent	269
Trying the search	280
TrackComponent	282
Wrapping up music search	284
Router Hooks	285
AuthService	286
LoginComponent	288
ProtectedComponent and Route Guards	290
Nested Routes	297
Configuring Routes	297
ProductsModule	298
Summary	304
Data Architecture in Angular	305
An Overview of Data Architecture	305
Data Architecture in Angular	306
Data Architecture with Observables - Part 1: Services	308
Observables and RxJS	308
Note: Some RxJS Knowledge Required	308
Learning Reactive Programming and RxJS	309
Chat App Overview	310
Components	312
Models	313
Services	314
Summary	314
Implementing the Models	315
User	315
Thread	316
Message	316
Implementing UsersService	318

CONTENTS

currentUser stream	319
Setting a new user	319
UsersService.ts	321
The MessagesService	322
the newMessages stream	322
the messages stream	324
The Operation Stream Pattern	324
Sharing the Stream	326
Adding Messages to the messages Stream	327
Our completed MessagesService	332
Trying out MessagesService	335
The ThreadsService	337
A map of the current set of Threads (in threads)	337
A chronological list of Threads, newest-first (in orderedthreads)	342
The currently selected Thread (in currentThread)	343
The list of Messages for the currently selected Thread (in current- ThreadMessages)	345
Our Completed ThreadsService	348
Data Model Summary	350
Data Architecture with Observables - Part 2: View Components	351
Building Our Views: The AppComponent Top-Level Component	351
The ChatThreadsComponent	354
ChatThreadsComponent template	355
The Single ChatThreadComponent	356
ChatThreadComponent Controller and ngOnInit	357
ChatThreadComponent template	358
The ChatWindowComponent	358
The ChatMessageComponent	370
The ChatMessageComponent template	372
The ChatNavBarComponent	373
The ChatNavBarComponent @Component	373
The ChatNavBarComponent template	375
Summary	376
Introduction to Redux with TypeScript	378

CONTENTS

Redux	379
Redux: Key Ideas	380
Core Redux Ideas	381
What’s a <i>reducer</i> ?	381
Defining Action and Reducer Interfaces	382
Creating Our First Reducer	383
Running Our First Reducer	384
Adjusting the Counter With <i>actions</i>	385
Reducer switch	387
Action “Arguments”	388
Storing Our State	389
Using the Store	391
Being Notified with <i>subscribe</i>	391
The Core of Redux	396
A Messaging App	397
Messaging App state	397
Messaging App actions	398
Messaging App reducer	399
Trying Out Our Actions	403
Action Creators	404
Using Real Redux	406
Using Redux in Angular	407
Planning Our App	408
Setting Up Redux	409
Defining the Application State	409
Defining the Reducers	409
Defining Action Creators	410
Creating the Store	411
Providing the Store	413
Bootstrapping the App	415
The AppComponent	416
<i>imports</i>	416
The template	417
The constructor	418
Putting It All Together	420
What’s Next	420

CONTENTS

References	421
Intermediate Redux in Angular	422
Context For This Chapter	423
Chat App Overview	423
Components	424
Models	424
Reducers	425
Summary	425
Implementing the Models	426
User	426
Thread	427
Message	427
App State	428
A Word on Code Layout	428
The Root Reducer	429
The UsersState	430
The ThreadsState	430
Visualizing Our AppState	431
Building the Reducers (and Action Creators)	433
Set Current User Action Creators	433
UsersReducer - Set Current User	434
Thread and Messages Overview	435
Adding a New Thread Action Creators	436
Adding a New Thread Reducer	436
Adding New Messages Action Creators	438
Adding A New Message Reducer	439
Selecting A Thread Action Creators	441
Selecting A Thread Reducer	442
Reducers Summary	443
Building the Angular Chat App	444
The top-level AppComponent	445
The ChatPage	447
Container vs. Presentational Components	448
Building the ChatNavBarComponent	449
Redux Selectors	451

CONTENTS

Threads Selectors	453
Unread Messages Count Selector	454
Building the ChatThreadsComponent	455
ChatThreadsComponent Controller	456
ChatThreadsComponent template	457
The Single ChatThreadComponent	458
ChatThreadComponent template	460
Building the ChatWindowComponent	461
The ChatMessageComponent	469
Setting incoming	470
The ChatMessageComponent template	470
Summary	472
Advanced Components	473
Styling	474
View (Style) Encapsulation	476
Shadow DOM Encapsulation	481
No Encapsulation	482
Creating a Popup - Referencing and Modifying Host Elements	485
Popup Structure	486
Using ElementRef	488
Binding to the host	490
Adding a Button using exportAs	493
Creating a Message Pane with Content Projection	495
Changing the Host's CSS	497
Using ng-content	497
Querying Neighbor Directives - Writing Tabs	499
ContentTabComponent	500
ContentTabsetComponent Component	501
Using the ContentTabsetComponent	504
Lifecycle Hooks	505
OnInit and OnDestroy	506
OnChanges	511
DoCheck	517
AfterContentInit, AfterViewInit, AfterContentChecked and After- ViewChecked	531

CONTENTS

Advanced Templates	538
Rewriting ngIf - ngBookIf	539
Rewriting ngFor - NgBookFor	542
Change Detection	548
Customizing Change Detection	553
Zones	561
Observables and OnPush	562
Summary	567
Testing	568
Test driven?	568
End-to-end vs. Unit Testing	569
Testing Tools	569
Jasmine	569
Karma	570
Writing Unit Tests	571
Angular Unit testing framework	571
Setting Up Testing	572
Testing Services and HTTP	573
HTTP Considerations	574
Stubs	575
Mocks	576
HttpClient HttpTestingController	577
TestBed.configureTestingModule and Providers	578
Testing getTrack	578
Testing Routing to Components	585
Creating a Router for Testing	586
Mocking dependencies	589
Spies	590
Back to Testing Code	594
fakeAsync and advance	596
inject	597
Testing ArtistComponent's Initialization	598
Testing ArtistComponent Methods	599
Testing ArtistComponent DOM Template Values	600
Testing Forms	603

CONTENTS

Creating a ConsoleSpy	606
Installing the ConsoleSpy	607
Configuring the Testing Module	608
Testing The Form	609
Refactoring Our Form Test	611
Testing HTTP requests	615
Testing a POST	615
Testing DELETE	618
Testing HTTP Headers	619
Testing YouTubeSearchService	621
Conclusion	628
Converting an AngularJS 1.x App to Angular	629
Peripheral Concepts	629
What We're Building	630
Mapping AngularJS 1 to Angular	632
Requirements for Interoperability	634
The AngularJS 1 App	635
The ng1-app HTML	636
Code Overview	638
ng1: PinsService	638
ng1: Configuring Routes	640
ng1: HomeController	641
ng1: / HomeController template	641
ng1: pin Directive	642
ng1: pin Directive template	643
ng1: AddController	644
ng1: AddController template	647
ng1: Summary	649
Building A Hybrid	649
Hybrid Project Structure	650
Bootstrapping our Hybrid App	653
What We'll Upgrade	655
A Minor Detour: Typing Files	658
Writing ng2 PinControlsComponent	662
Using ng2 PinControlsComponent	664

CONTENTS

Downgrading ng2 PinControlsComponent to ng1	665
Adding Pins with ng2	668
Upgrading ng1 PinsService and \$state to ng2	669
Writing ng2 AddPinComponent	670
Using AddPinComponent	677
Exposing an ng2 service to ng1	678
Writing the AnalyticsService	678
Downgrade ng2 AnalyticsService to ng1	679
Using AnalyticsService in ng1	680
Summary	681
References	682
NativeScript: Mobile Applications for the Angular Developer	683
What is NativeScript?	683
Where NativeScript Differs from Other Popular Frameworks	684
What are the System and Development Requirements for NativeScript?	685
Creating your First Mobile Application with NativeScript and Angular	688
Adding Build Platforms for Cross Platform Deployment	688
Building and Testing for Android and iOS	689
Installing JavaScript, Android, and iOS Plugins and Packages	690
Understanding the Web to NativeScript UI and UX Differences	690
Planning the NativeScript Page Layout	691
Adding UI Components to the Page	692
Styling Components with CSS	694
Developing a Geolocation Based Photo Application	696
Creating a Fresh NativeScript Project	697
Creating a Multiple Page Master-Detail Interface	697
Creating a Flickr Service for Obtaining Photos and Data	701
Creating a Service for Calculating Device Location and Distance	707
Including Mapbox Functionality in the NativeScript Application	710
Implementing the First Page of the Geolocation Application	712
Implementing the Second Page of the Geolocation Application	718
Try it out!	719
NativeScript for Angular Developers	720
Changelog	721

CONTENTS

Revision 74 - 2019-05-30	721
Revision 73 - 2019-01-08	721
Revision 72 - 2018-12-12	721
Revision 71 - 2018-10-23	721
Revision 70 - 2018-09-13	721
Revision 69 - 2018-09-08	722
Revision 68 - 2018-05-08	722
Revision 67 - 2018-01-17	722
Revision 66 - 2017-11-14	722
Revision 65 - 2017-11-01	723
Revision 64 - 2017-09-15	723
Revision 63 - 2017-08-02	723
Revision 62 - 2017-06-23	723
Revision 61 - 2017-05-24	723
Revision 60 - 2017-04-27	724
Revision 59 - 2017-04-07	724
Revision 58 - 2017-03-24	724
Revision 57 - 2017-03-23	724
Revision 56 - 2017-03-22	725
Revision 55 - 2017-03-17	725
Revision 54 - 2017-03-10	725
Revision 53 - 2017-03-01	726
Revision 52 - 2017-02-22	726
Revision 51 - 2017-02-14	726
Revision 50 - 2017-02-10	726
Revision 49 - 2017-01-18	727
Revision 48 - 2017-01-13	727
Revision 47 - 2017-01-06	727
Revision 46 - 2017-01-03	727
Revision 45 - 2016-12-05	727
Revision 44 - 2016-11-17	727
Revision 43 - 2016-11-08	728
Revision 42 - 2016-10-14	728
Revision 41 - 2016-09-28	728
Revision 40 - 2016-09-20	729
Revision 39 - 2016-09-03	729

CONTENTS

Revision 38 - 2016-08-29	729
Revision 37 - 2016-08-02	729
Revision 36 - 2016-07-20	729
Revision 35 - 2016-06-30	729
Revision 34 - 2016-06-15	730
Revision 33 - 2016-05-11	730
Revision 32 - 2016-05-06	730
Revision 31 - 2016-04-28	731
Revision 30 - 2016-04-20	731
Revision 29 - 2016-04-08	731
Revision 28 - 2016-04-01	731
Revision 27 - 2016-03-25	731
Revision 26 - 2016-03-24	731
Revision 25 - 2016-03-21	732
Revision 24 - 2016-03-10	732
Revision 23 - 2016-03-04	732
Revision 22 - 2016-02-24	733
Revision 21 - 2016-02-20	733
Revision 20 - 2016-02-11	733
Revision 19 - 2016-02-04	733
Revision 18 - 2016-01-29	734
Revision 17 - 2016-01-28	734
Revision 16 - 2016-01-14	734
Revision 15 - 2016-01-07	734
Revision 14 - 2015-12-23	735
Revision 13 - 2015-12-17	735
Revision 12 - 2015-11-16	736
Revision 11 - 2015-11-09	736
Revision 10 - 2015-10-30	737
Revision 9 - 2015-10-15	737
Revision 8 - 2015-10-08	738
Revision 7 - 2015-09-23	738
Revision 6 - 2015-08-28	738
Revision 5 - 2015-08-01	738
Revision 4 - 2015-07-30	738
Revision 3 - 2015-07-21	739

CONTENTS

Revision 2 - 2015-07-15	739
Revision 1 - 2015-07-01	739

Book Revision

Revision 74 - Covers up to Angular 8 (8.0.0, 2019-05-31)

Bug Reports

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

Vote for New Content (new!)

We're constantly updating the book, writing new blog posts, and producing new material. You can now [cast your vote for new content here](#)².

Be notified of updates via Twitter

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

We'd love to hear from you!

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

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

²<https://fullstackio.canny.io/ng-book>

³<https://twitter.com/fullstackio>

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

How to Read This Book

This book aims to be the single most useful resource on learning Angular. By the time you're done reading this book, you (and your team) will have everything you need to build reliable, powerful Angular apps.

Angular is a rich and feature-filled framework, but that also means it can be tricky to understand all of its parts. In this book, we'll walk through everything from installing the tools, writing components, using forms, routing between pages, and calling APIs.

But before we dig in, there are a few guidelines I want to give you **in order to get the most out of this book**. Briefly, I want to tell you:

- how to approach **the code examples** and
- **how to get help** if something goes wrong

Running Code Examples

This book comes with a library of runnable code examples. The code is available to download from the same place where you downloaded this book.

We use the program `npm`⁵ to run **every example** in this book. This means you can type the following commands to run any example:

```
npm install
npm start
```



If you're unfamiliar with `npm`, we cover how to get it installed in the [Getting Started](#) section in the first chapter.

⁵<https://www.npmjs.com/>

After running `npm start`, you will see some output on your screen that will tell you what URL to open to view your app.

If you're ever unclear on how to run a particular sample app, check out the `README.md` in that project's directory. Every sample project contains a `README.md` that will give you the instructions you need to run each app.

Angular CLI

With a couple of minor exceptions, every project in this book was built on [Angular CLI](#)⁶. Unless specified otherwise, you can use the `ng` commands in each project.

For instance, to run an example you can run `ng serve` (this is, generally, what is run when you type `npm start`). For most projects you can compile them to JavaScript with `ng build` (we'll talk about this more in the first chapter). And you can run end-to-end tests with `ng e2e`, etc.

Without getting too far into the details, Angular CLI is based on Webpack, a tool which helps process and bundle our various TypeScript, JavaScript, CSS, HTML, and image files. **Angular CLI is not a requirement** for using Angular. It's simply a wrapper around Webpack (and some other tooling) that makes it easy to get started.

Code Blocks and Context

Nearly every code block in this book is pulled from a **runnable code example**, which you can find in the sample code. For example, here is a code block pulled from the first chapter:

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

```
8 export class AppComponent {  
9   title = 'app';  
10 }
```

Notice that the header of this code block states the path to the file which contains this code: `code/first-app/angular-hello-world/src/app/app.component.ts`.

⁶<https://github.com/angular/angular-cli>

If you ever feel like you're missing the context for a code example, open up the full code file using your favorite text editor. **This book is written with the expectation that you'll also be looking at the example code alongside the manuscript.**

For example, we often need to `import` libraries to get our code to run. In the early chapters of the book we show these `import` statements, because it's not clear where the libraries are coming from otherwise. However, the later chapters of the book are more advanced and they focus on *key concepts* instead of repeating boilerplate code that was covered earlier in the book. **If at any point you're not clear on the context, open up the code example on disk.**

Code Block Numbering

In this book, we sometimes build up a larger example in steps. If you see a file being loaded that has a numeric suffix, that generally means we're building up to something bigger.

For instance, in the Dependency Injection chapter you may see a code block with the filename: `price.service.1.ts`. When you see the `.N.ts` syntax that means we're building up to the ultimate file, which will **not** have a number. So, in this case, the final version would be: `price.service.ts`. We do it this way so that a) we can unit test the intermediate code and b) you can see the whole file in context at a particular stage.

A Word on Versioning

As you may know, the Angular covered in this book is a descendant of an earlier framework called "AngularJS". This can sometimes be confusing, particularly when reading supplementary blogs or documentation.

The official branding guidelines state that "*AngularJS*" is a term reserved for AngularJS 1.x, that is, the early versions of "Angular".

Because the new version of Angular used TypeScript (instead of JavaScript) as the primary language, the 'JS' was dropped, leaving us with just *Angular*. For a long time the only consistent way to distinguish the two was folks referred to the *new* Angular as *Angular 2*.

However, the Angular team in 2017 switched to *semantic versioning* with a new major-release upgrade slated for every 6 months. Instead of calling the next versions *Angular 4*, *Angular 5*, and so on, the number is also dropped and it's just *Angular*.

In this book, when we're referring to *Angular* we'll just say *Angular* or sometimes *Angular X*, just to avoid confusion. When we're talking about "the old-style JavaScript Angular" we'll use the term *AngularJS* or *AngularJS 1.x*.

Getting Help

While we've made every effort to be clear, precise, and accurate you may find that when you're writing your code you run into a problem.

Generally, there are three types of problems:

- A "bug" in the book (e.g. how we describe something is wrong)
- A "bug" in our code
- A "bug" in your code

If you find an inaccuracy in how we describe something, or you feel a concept isn't clear, [email us!](#) We want to make sure that the book is both accurate and clear.

Similarly, if you've found a bug in our *code* we definitely [want to hear about it](#).

If you're having trouble getting your own app working (and it isn't *our* example code), this case is a bit harder for us to handle.

Your first line of defense, when getting help with your custom app, should be our [unofficial community chat room](#)⁷. We (the authors) are there from time-to-time, but there are hundreds of other readers there who may be able to help you faster than we can.

If you're still stuck, we'd still love to hear from you, and here are some tips for getting a clear, timely response.

⁷<https://gitter.im/ng-book/ng-book>

Emailing Us

If you're emailing us asking for technical help, here's what we'd like to know:

- What [revision of the book](#) are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.8, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- [What have you tried](#)⁸ already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

The **absolute best way to get technical support** is to send us a short, self-contained example of the problem.

But in any case email us at us@fullstack.io⁹. We look forward to hearing from you.

Chapter Overview

Before we dive in, I want to give you a feel for the rest of the book and what you can expect inside.

The first few chapters provide the **foundation** you need to get up and running with Angular. You'll create your **first apps**, use **the built-in components**, and start **creating your components**.

Next we'll move into intermediate concepts such as using **forms**, using **APIs**, **routing** to different pages, and using *Dependency Injection* to organize our code.

After that, we'll move into more **advanced concepts**. We spend a good part of the book talking about *data architectures*. Managing state in client/server applications is hard and we dive deep into two popular approaches: using **RxJS Observables** and using **Redux**. In these chapters, we'll show how to build the same app, two different

⁸<http://mattgummell.com/what-have-you-tried/>

⁹<mailto:us@fullstack.io>

ways, so you can compare and contrast and evaluate what's best for you and your team.

After that, we'll discuss how to write complex, **advanced components** using Angular's most powerful features. Then we talk about how to write **tests** for our app and how we can **upgrade our Angular 1 apps** to Angular. Finally, we close with a chapter on writing **native mobile apps** with Angular using **NativeScript**.

By using this book, **you're going to learn how to build real Angular apps** faster than spending hours parsing out-dated blog posts.

So hold on tight - you're about to become an Angular expert, and have a lot of fun along the way. Let's dig in!

- Nate (@eigenjoy¹⁰)

¹⁰<https://twitter.com/eigenjoy>

Writing Your First Angular Web Application

Simple Reddit Clone

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

You can think of this app as the beginnings of a site like [Reddit](http://reddit.com)¹¹ or [Product Hunt](http://producthunt.com)¹².

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

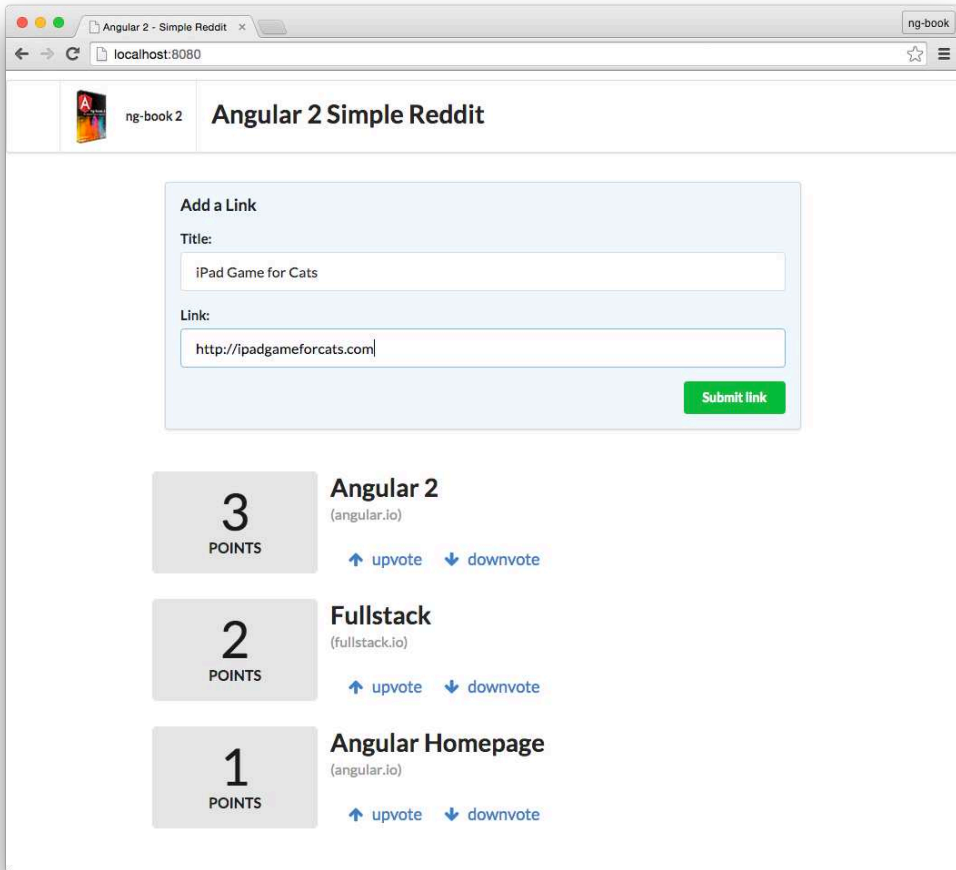
- Building custom components
- Accepting user input from forms
- Rendering lists of objects into views
- Intercepting user clicks and acting on them
- Deploying our app to a server

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

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

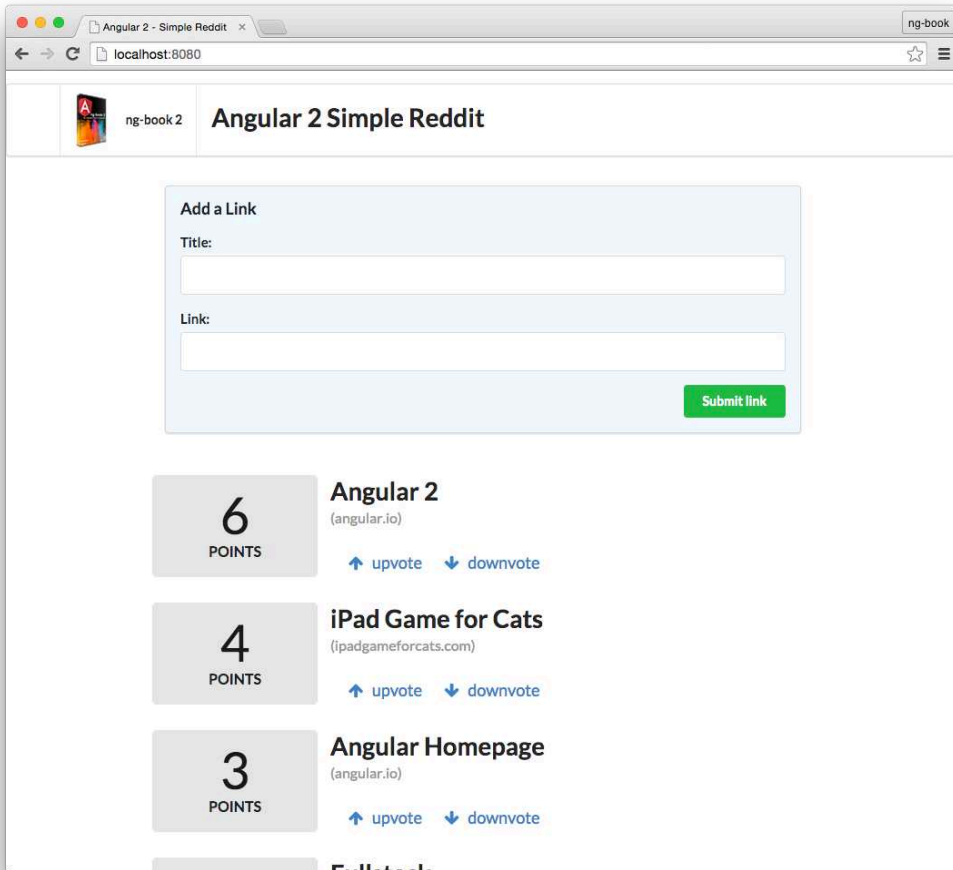
¹¹<http://reddit.com>

¹²<http://producthunt.com>



Completed application

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



App with new article

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

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

Getting started

Node.js and npm

To get started with Angular, you'll need to have Node.js installed. There are a couple of different ways you can install Node.js, so please refer to [the Node.js website](https://nodejs.org/download/)¹³ for detailed information.

Make sure you install Node 12.2.0 or higher.



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

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

```
$ npm -v
```

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

Your npm version should be 6.9.0 or higher.

TypeScript

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

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

¹³<https://nodejs.org/download/>



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

Browser

We highly recommend using the [Google Chrome Web Browser](#)¹⁴ to develop Angular apps. We'll use the Chrome developer toolkit throughout this book. To follow along with our development and debugging we recommend downloading it now.

Special instruction for Windows users

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

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



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

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, run the following command:

¹⁴<https://www.google.com/chrome/>


```
1 $ npm install -g @angular/cli
```

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

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

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



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

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

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

```
1 $ brew install watchman
```



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

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

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

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

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

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

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

Example Project

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

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

Once you run it, you may be asked a few questions about your configuration preferences.



For example, if asked if you want to add Angular routing, in this example say **No**, but you'll probably want to use it for a bigger project

When asked about what CSS framework you want to use, we'll just use CSS for now.

After you answer the questions, you'll see (roughly) following output:

Once you run it, you'll see (roughly) following output:

```
1 CREATE angular-hello-world/README.md (1034 bytes)
2 CREATE angular-hello-world/angular.json (3504 bytes)
3 CREATE angular-hello-world/package.json (1323 bytes)
4 CREATE angular-hello-world/tsconfig.json (384 bytes)
5 CREATE angular-hello-world/tslint.json (2805 bytes)
6 CREATE angular-hello-world/.editorconfig (245 bytes)
7 CREATE angular-hello-world/.gitignore (503 bytes)
8 CREATE angular-hello-world/src/environments/environment.prod.ts (51 bytes)
9 CREATE angular-hello-world/src/environments/environment.ts (631 bytes)
10 CREATE angular-hello-world/src/favicon.ico (5430 bytes)
11 CREATE angular-hello-world/src/index.html (304 bytes)
12 CREATE angular-hello-world/src/main.ts (370 bytes)
13 CREATE angular-hello-world/src/polyfills.ts (3194 bytes)
14 CREATE angular-hello-world/src/test.ts (642 bytes)
15 CREATE angular-hello-world/src/assets/.gitkeep (0 bytes)
16 CREATE angular-hello-world/src/styles.css (80 bytes)
17 CREATE angular-hello-world/src/browserslist (375 bytes)
18 CREATE angular-hello-world/src/karma.conf.js (964 bytes)
19 CREATE angular-hello-world/src/tsconfig.app.json (194 bytes)
20 CREATE angular-hello-world/src/tsconfig.spec.json (282 bytes)
21 CREATE angular-hello-world/src/tslint.json (314 bytes)
22 CREATE angular-hello-world/src/app/app.module.ts (314 bytes)
23 CREATE angular-hello-world/src/app/app.component.css (0 bytes)
24 CREATE angular-hello-world/src/app/app.component.html (1141 bytes)
25 CREATE angular-hello-world/src/app/app.component.spec.ts (986 bytes)
26 CREATE angular-hello-world/src/app/app.component.ts (207 bytes)
27 CREATE angular-hello-world/e2e/protractor.conf.js (752 bytes)
28 CREATE angular-hello-world/e2e/src/app.e2e-spec.ts (299 bytes)
29 CREATE angular-hello-world/e2e/src/app.po.ts (208 bytes)
30 CREATE angular-hello-world/e2e/tsconfig.e2e.json (213 bytes)
31 ...
32 added 1146 packages in 105.319s
33     Successfully initialized git.
```

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



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

There are a lot of files generated! Don't worry about understanding all of them yet.

Throughout the book we'll walk through what each one means and what it's used for.

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

```
1 $ cd angular-hello-world
2 $ tree -F -L 1
3 .
4 |-- README.md           // a useful README
5 |-- angular.json       // angular-cli configuration file
6 |-- e2e/               // end-to-end tests
7 |-- node_modules/     // installed dependencies
8 |-- package-lock.json  // npm dependencies lockfile
9 |-- package.json       // npm configuration
10 |-- src/              // our application's code
11 |-- tsconfig.json     // typescript config
12 `-- tslint.json       // linting config
13
14 3 directories, 6 files
```



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

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

```
1 $ cd src
2 $ tree -F
3 .
4 |-- app/
5 |   |-- app.component.css
6 |   |-- app.component.html
7 |   |-- app.component.spec.ts
8 |   |-- app.component.ts
9 |   `-- app.module.ts
10 |-- assets/
11 |-- browserslist
12 |-- environments/
13 |   |-- environment.prod.ts
```

```
14 |   |-- environment.ts
15 |   |-- favicon.ico
16 |   |-- index.html
17 |   |-- karma.conf.js
18 |   |-- main.ts
19 |   |-- polyfills.ts
20 |   |-- styles.css
21 |   |-- test.ts
22 |   |-- tsconfig.app.json
23 |   |-- tsconfig.spec.json
24 |   |-- tslint.json
25 |
26 | 3 directories, 18 files
```

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

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

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

Let's break it down a bit:

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

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

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

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

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

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

The `app-root` tag is **where our application will be rendered**.

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

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

Writing Application Code

Running the application

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

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

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

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

<http://localhost:4200>¹⁵



Note that if you get the message:

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

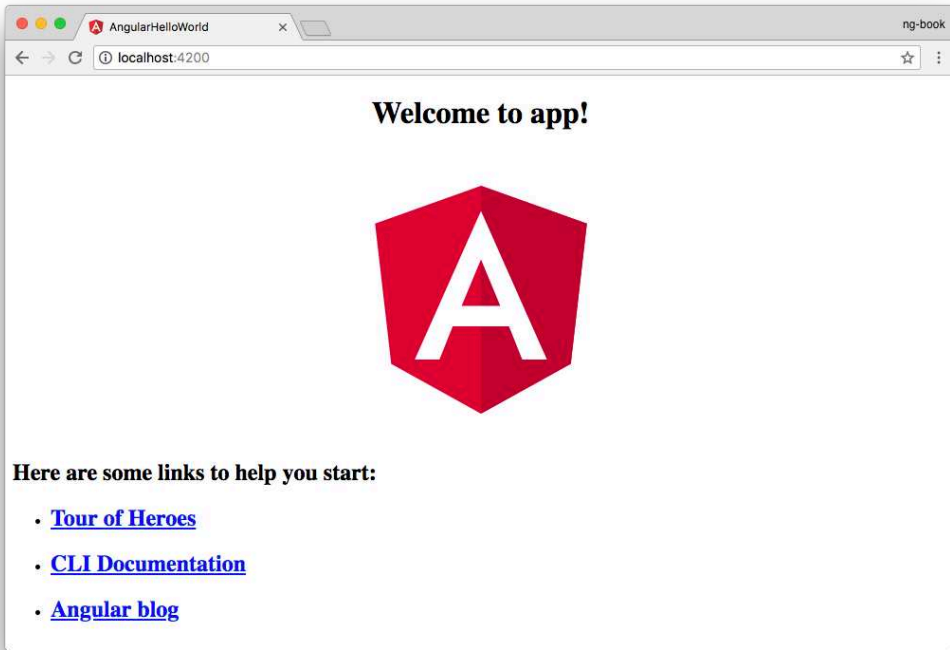
This means that you already have another service running on port 4200. If this is the case you can either 1. shut down the other service or 2. use the `--port` flag when running `ng serve` like this:

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

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

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

¹⁵<http://localhost:4200>



Running application

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

Making a Component

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

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

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

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



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

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

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

To create a new component using Angular CLI, we'll use the **generate** command.

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

```
1 $ ng generate component hello-world
2 CREATE src/app/hello-world/hello-world.component.css (0 bytes)
3 CREATE src/app/hello-world/hello-world.component.html (30 bytes)
4 CREATE src/app/hello-world/hello-world.component.spec.ts (657 bytes)
5 CREATE src/app/hello-world/hello-world.component.ts (288 bytes)
6 UPDATE src/app/app.module.ts (414 bytes)
```

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

1. A Component decorator
2. A component definition class

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

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

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

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



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

Importing Dependencies

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

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

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

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

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

Component Decorators

After importing our dependencies, we are declaring the component:

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

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

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

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

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

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

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

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

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

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

Adding a template with `templateUrl`

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

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

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

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

Adding a template

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

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

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

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



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

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

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

Adding CSS Styles with `styleUrls`

Notice the key `styleUrls`:

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

This code says that we want to use the CSS in the file `hello-world.component.css` as the styles for this component. Angular 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 its argument. This is because we can load multiple stylesheets for a single component.

Loading Our Component

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

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

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

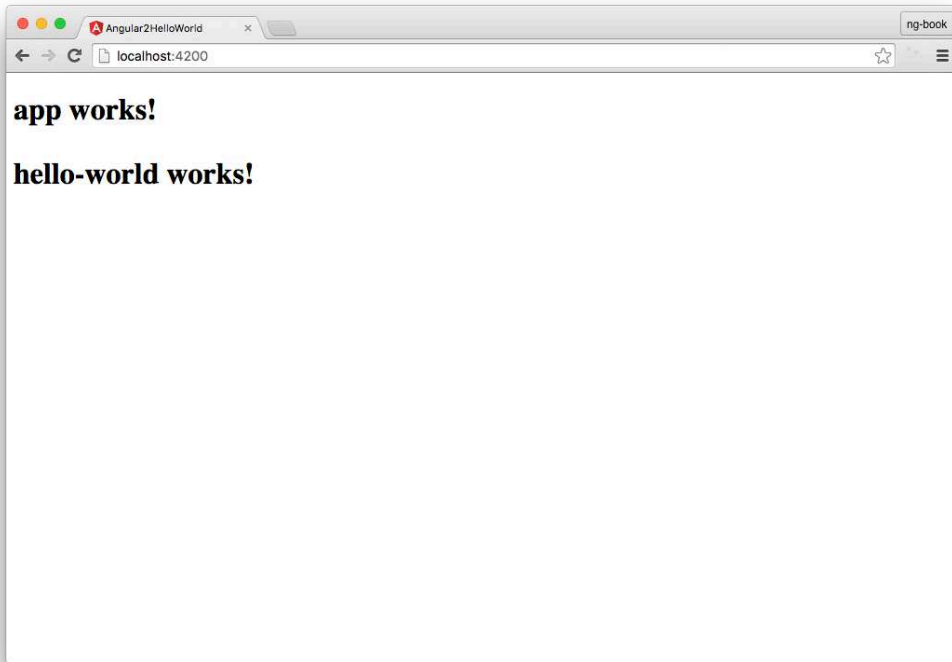
Remember that because we configured our `HelloWorldComponent` with the selector `app-hello-world`, we can use the `<app-hello-world></app-hello-world>` in our template. Let's add the `<app-hello-world>` tag to `app.component.html`

Delete the content in `app.component.html` and replace it with:

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

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

Now refresh the page and take a look:



Hello world works

It works!

Adding Data to the Component

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Notice that we've changed two things :

1. name Property

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

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

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

2. A Constructor

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

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

When we write:

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

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

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

Rendering The Template

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

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

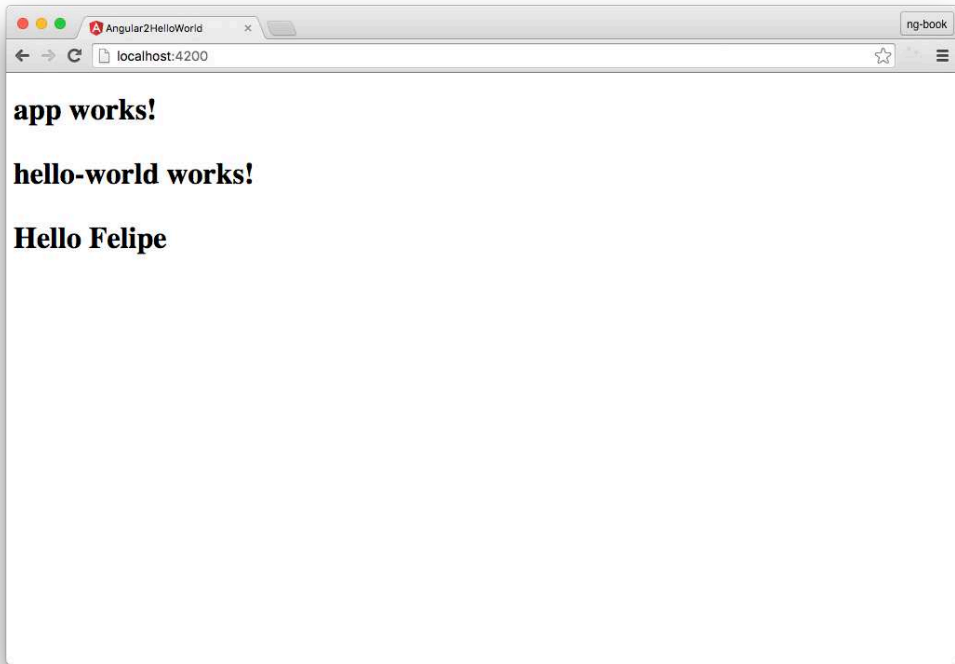
```
1 <p>
2   Hello {{ name }}
3 </p>
```

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

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

Try It Out

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



Application with Data

Working With Arrays

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

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



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

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

a new component:

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

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

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

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

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

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

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

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

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

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

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

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

Here's what our new template will look like:

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

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

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

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

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

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



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

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

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

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

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



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

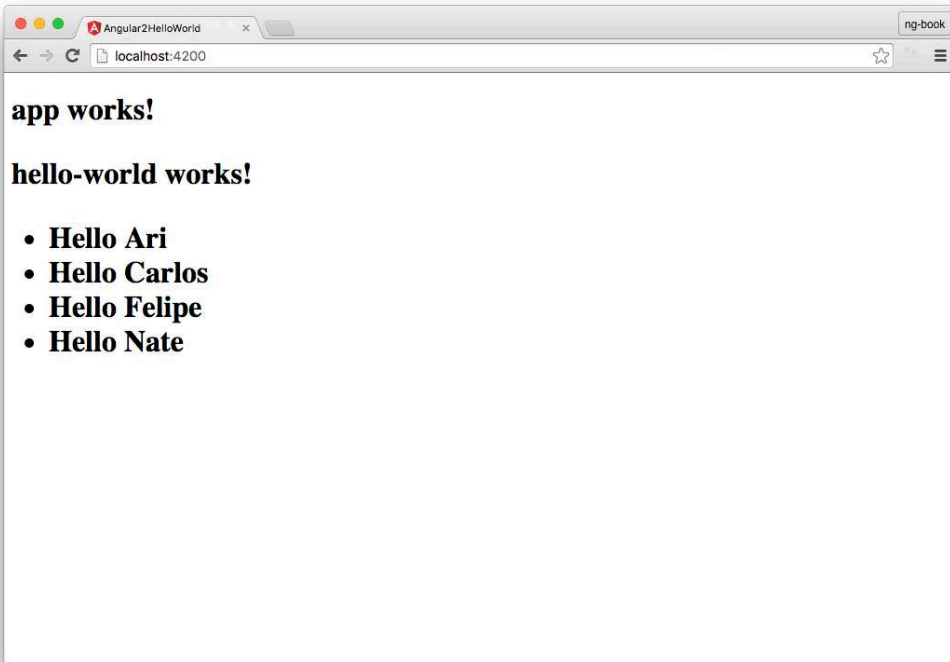
Note that the capitalization here isn't a typo: `NgFor` is the capitalization of the *class* that implements the logic and `ngFor` is the "selector" for the attribute we want to use.



If you're feeling adventurous you can learn a lot about how the Angular core team writes Components by reading the source directly. For instance, you can find the source of the [NgFor directive here](https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.ts)¹⁶.

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

¹⁶https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.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 rendering the text `Hello` and the name directly, we should let our `UserItemComponent` specify the template (and functionality) of **each item in the list**.

To do this, we need to do three things:

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

3. Configure the `UserListComponent` template to **pass the name** to the `UserItemComponent`.

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

Rendering the `UserItemComponent`

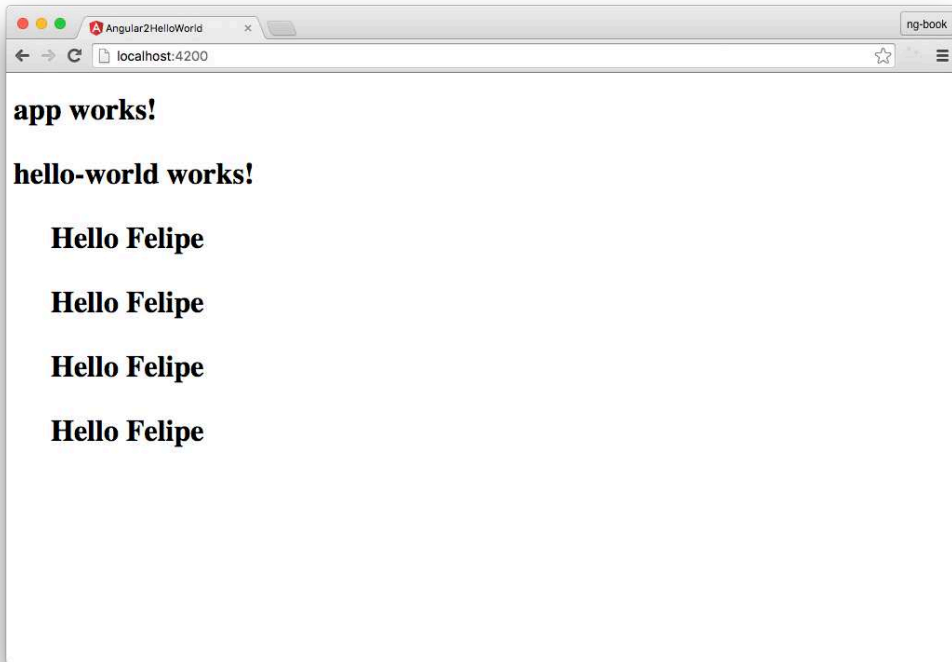
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template:

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

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

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

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



Application with Data

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

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

Accepting Inputs

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

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

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

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

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

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

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

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

Passing an Input value

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

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

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

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

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

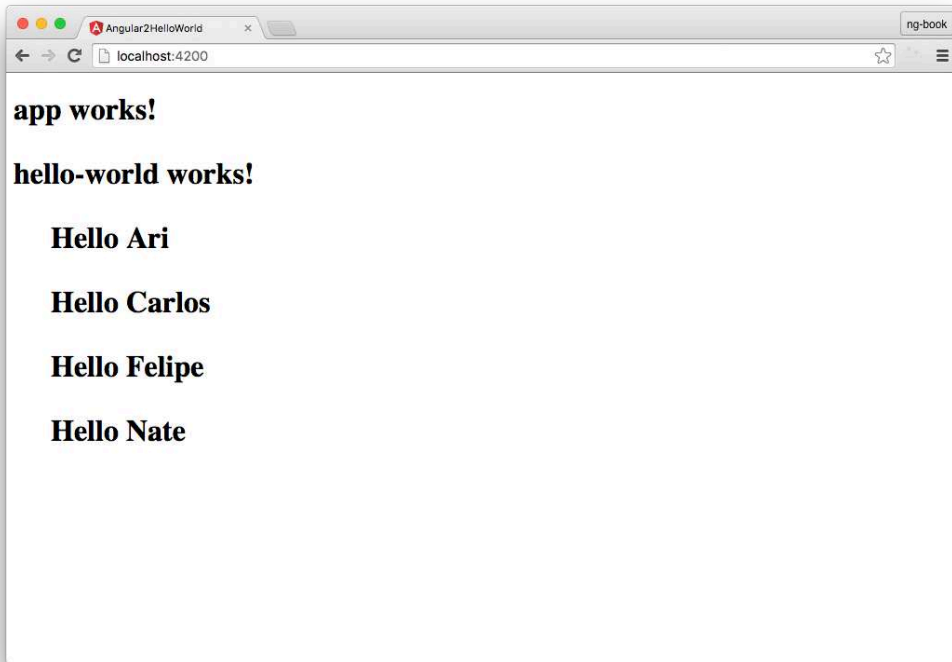
```
<li *ngFor="let individualUserName of names">
  <app-user-item [name]="individualUserName"></app-user-item>
</li>
```

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

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

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

Now our list of names works!



Application with Names Working

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

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

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

Bootstrapping Crash Course

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

```
1 ng serve
```

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

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

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

Take a look at this code:

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

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

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

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

declarations

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

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

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

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

imports

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

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



import vs. imports?

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

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

providers

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



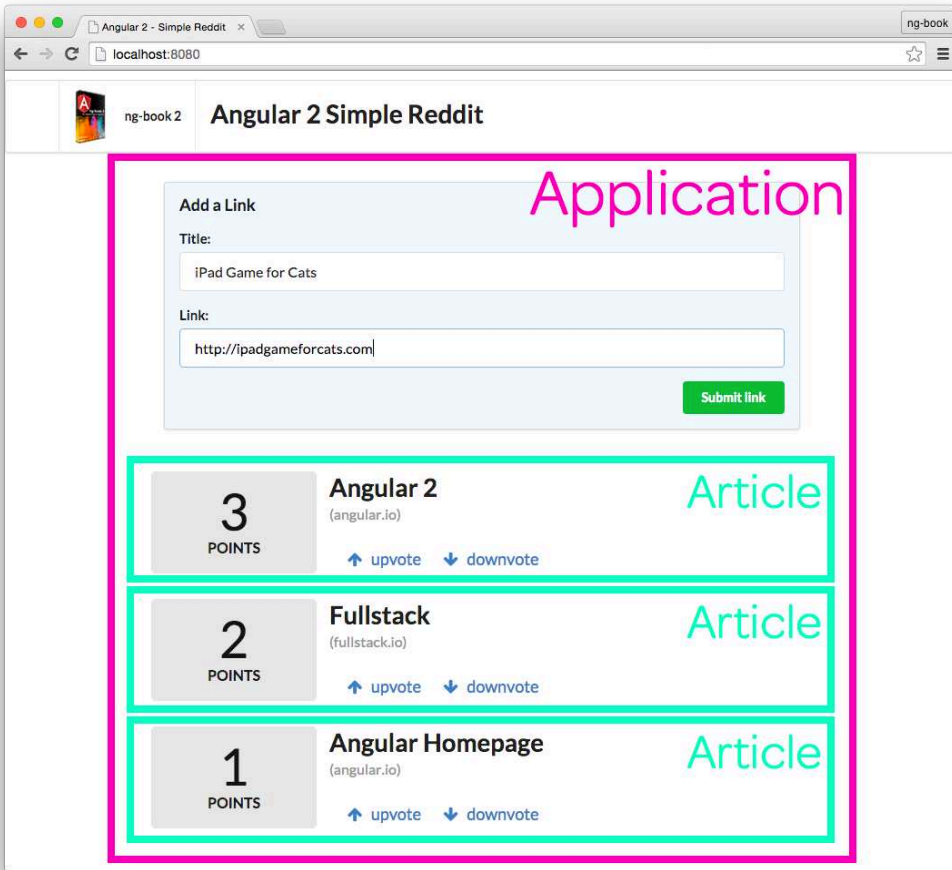
Learn more about this in [the section on Dependency Injection](#).

bootstrap

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

Expanding our Application

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



Application with Data

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

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



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

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

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

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



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

Adding CSS

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



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

Copy:

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

into your application's folder.

For this project we're going to be using [Semantic-UI](#)¹⁷ to help with the styling. Semantic-UI is a CSS framework, similar to [Zurb Foundation](#)¹⁸ or [Twitter Bootstrap](#)¹⁹. 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.

¹⁷<http://semantic-ui.com/>

¹⁸<http://foundation.zurb.com>

¹⁹<http://getbootstrap.com>

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

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



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

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

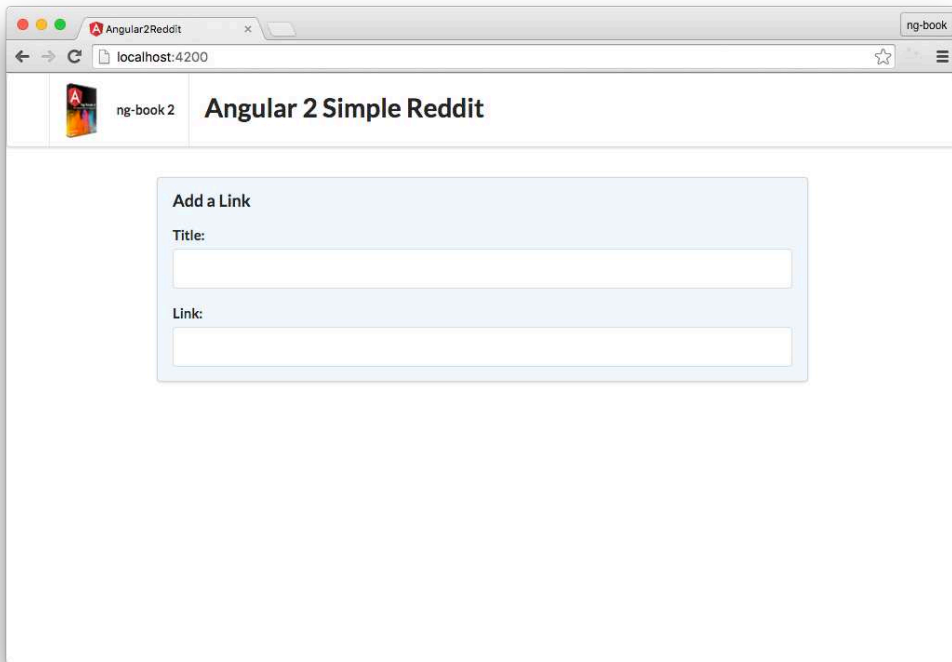
Let's change the template a bit to include a form for adding links. We'll use a bit of styling from the `semantic-ui` package to make the form look a bit nicer:

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

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

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

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



Form

Adding Interaction

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

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

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

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

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

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

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

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

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

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

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

```
15         class="ui positive right floated button">
16     Submit link
17 </button>
18
19 </form>
```

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

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

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

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

Binding inputs to values

Notice in our first `input` tag we have the following:

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

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

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

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

Binding actions to events

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

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

All together:

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



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

Defining the Action Logic

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

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

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

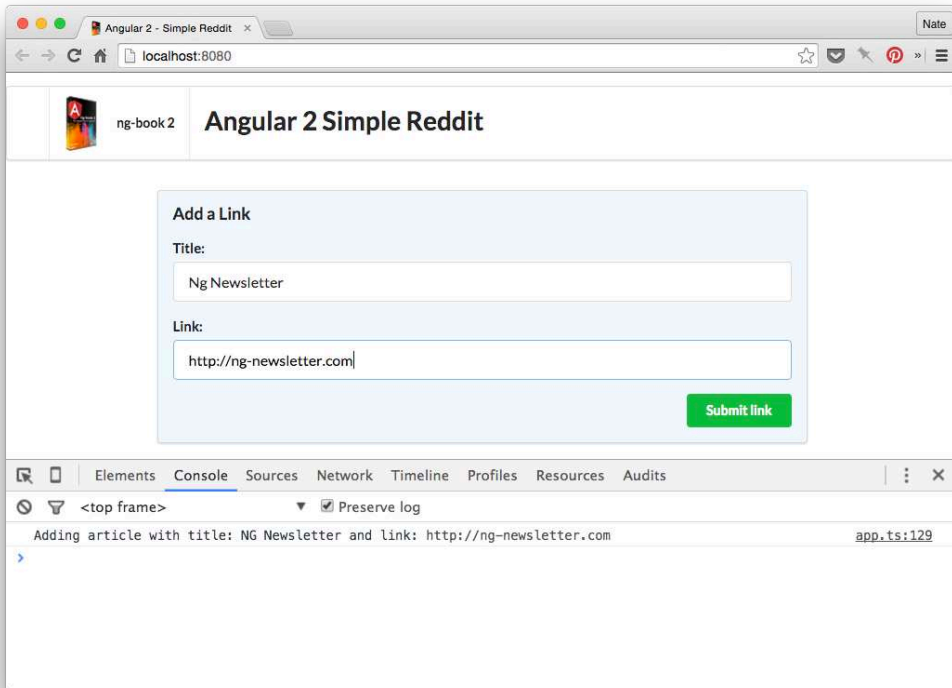


Notice that we're using backtick strings again. This is a really handy feature of ES6: backtick strings will expand template variables!

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

Try it out!

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



Clicking the Button

Adding the Article Component

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

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



A reddit-article

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

```
ng generate component article
```

We have three parts to defining this new component:

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

Let's talk through each part in detail:

Creating the `ArticleComponent` template

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

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

```
1 <div class="four wide column center aligned votes">
2   <div class="ui statistic">
3     <div class="value">
4       {{ votes }}
5     </div>
6     <div class="label">
7       Points
8     </div>
9   </div>
10 </div>
11 <div class="twelve wide column">
12   <a class="ui large header" href="{{ link }}">
```

```

13     {{ title }}
14   </a>
15   <ul class="ui big horizontal list voters">
16     <li class="item">
17       <a href (click)="voteUp()">
18         <i class="arrow up icon"></i>
19         upvote
20       </a>
21     </li>
22     <li class="item">
23       <a href (click)="voteDown()">
24         <i class="arrow down icon"></i>
25         downvote
26       </a>
27     </li>
28   </ul>
29 </div>

```

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



A Single reddit-article Row

We have two columns:

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

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

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

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

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

Creating the ArticleComponent

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

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

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

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

```
<app-article>
</app-article>
```

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

Creating the ArticleComponent Definition Class

Finally, we create the ArticleComponent definition class:

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

```
12 export class ArticleComponent implements OnInit {
13   @HostBinding('attr.class') cssClass = 'row';
14   votes: number;
15   title: string;
16   link: string;
17
18   constructor() {
19     this.title = 'Angular';
20     this.link = 'http://angular.io';
21     this.votes = 10;
22   }
23
24   voteUp() {
25     this.votes += 1;
26   }
27
28   voteDown() {
29     this.votes -= 1;
30   }
31
32   ngOnInit() {
33   }
34
35 }
```

Here we create four properties on `ArticleComponent`:

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

We want each `app-article` to be on its own row. We’re using Semantic UI, and Semantic provides a [CSS class for rows](http://semantic-ui.com/collections/grid.html)²⁰ called `row`.

In Angular, a component *host* is **the element this component is attached to**. We can set properties on the host element by using the `@HostBinding()` decorator. In this

²⁰<http://semantic-ui.com/collections/grid.html>

case, we're asking Angular to keep the value of the host elements class to be in sync with the property `cssClass`.



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

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

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



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

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

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

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

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

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

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

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

Using the `app-article` Component

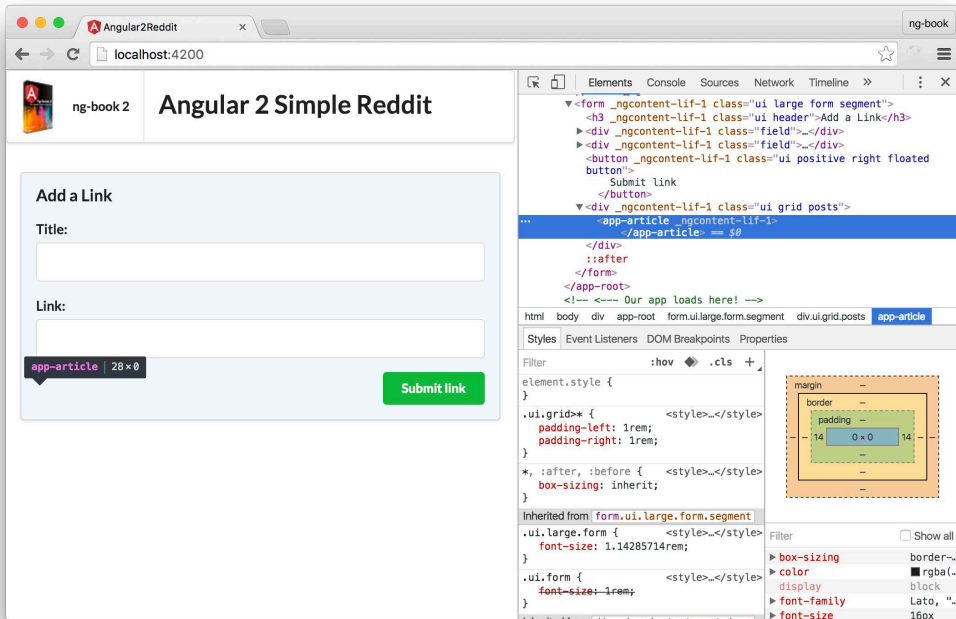
In order to use this component and make the data visible, we have to add a `<app-article></app-article>` tag somewhere in our markup.

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

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

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

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



Unexpanded tag when inspecting the DOM

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



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

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

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



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

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

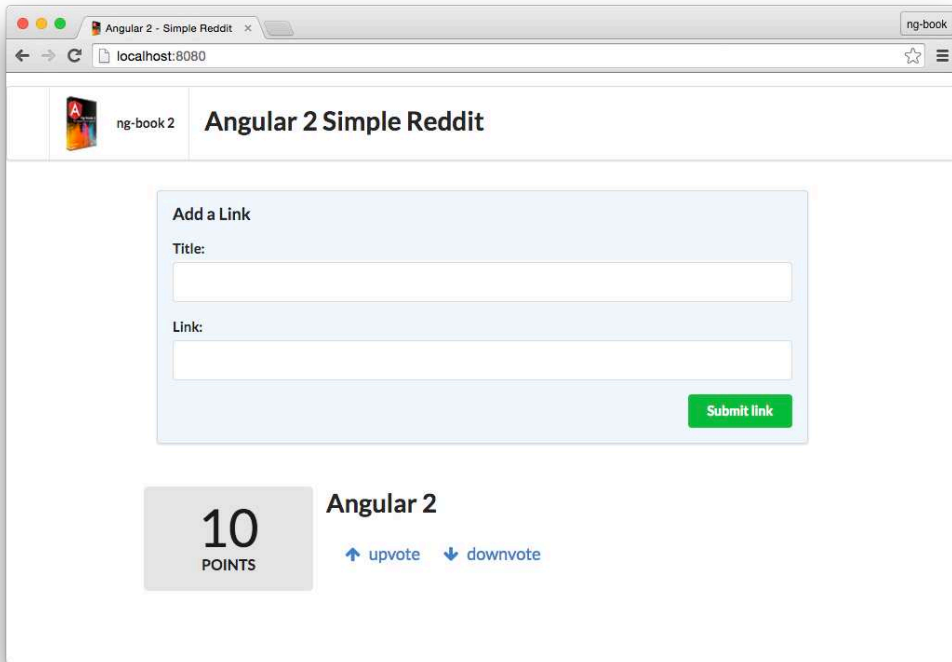
code/first-app/angular-reddit/src/app/app.module.ts

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

See here that we are:

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

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



Rendered ArticleComponent component

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

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

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

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

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

Rendering Multiple Rows

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

Creating an Article class

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

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

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

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

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

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

First let's import the class:

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

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

Then let's use it:

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

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

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

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

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

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

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

Reload the browser and everything still works.

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



`voteUp` and `voteDown` currently break the [Law of Demeter](http://en.wikipedia.org/wiki/Law_of_Demeter)²¹ 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 (we'll also add a `domain` function, which we'll talk about in a moment):

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

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

²¹http://en.wikipedia.org/wiki/Law_of_Demeter

GET THE FULL BOOK

This is the end of the preview chapter!

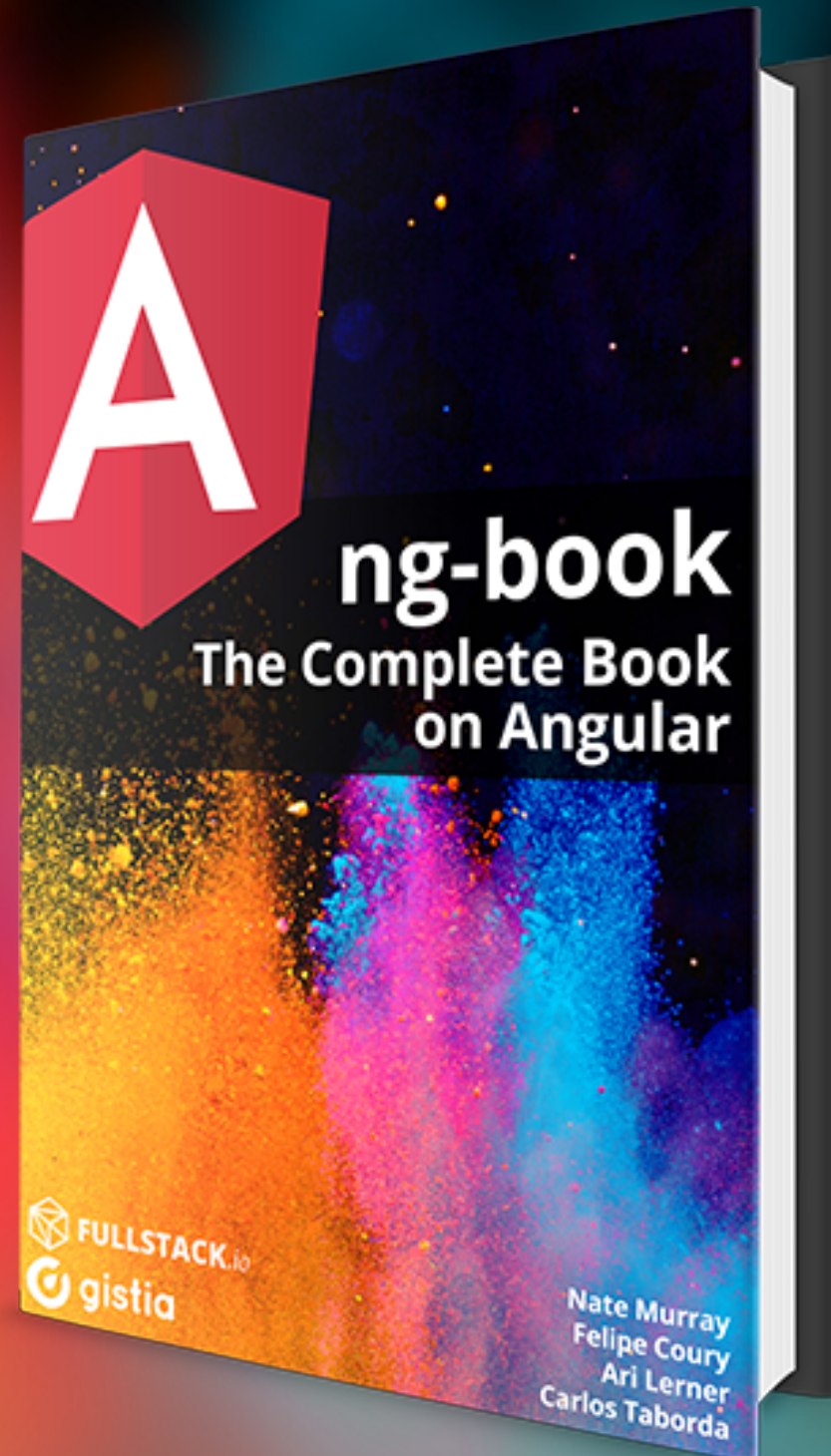
Head over to:

<https://ng-book.com/2>

to download the full package!

Learn how to use:

- Forms
- Routing
- Dependency Injection
- Advanced Components
- RxJX
- Redux
- NativeScript
- and more!



GET INSTANT ACCESS

<https://ng-book.com/2>