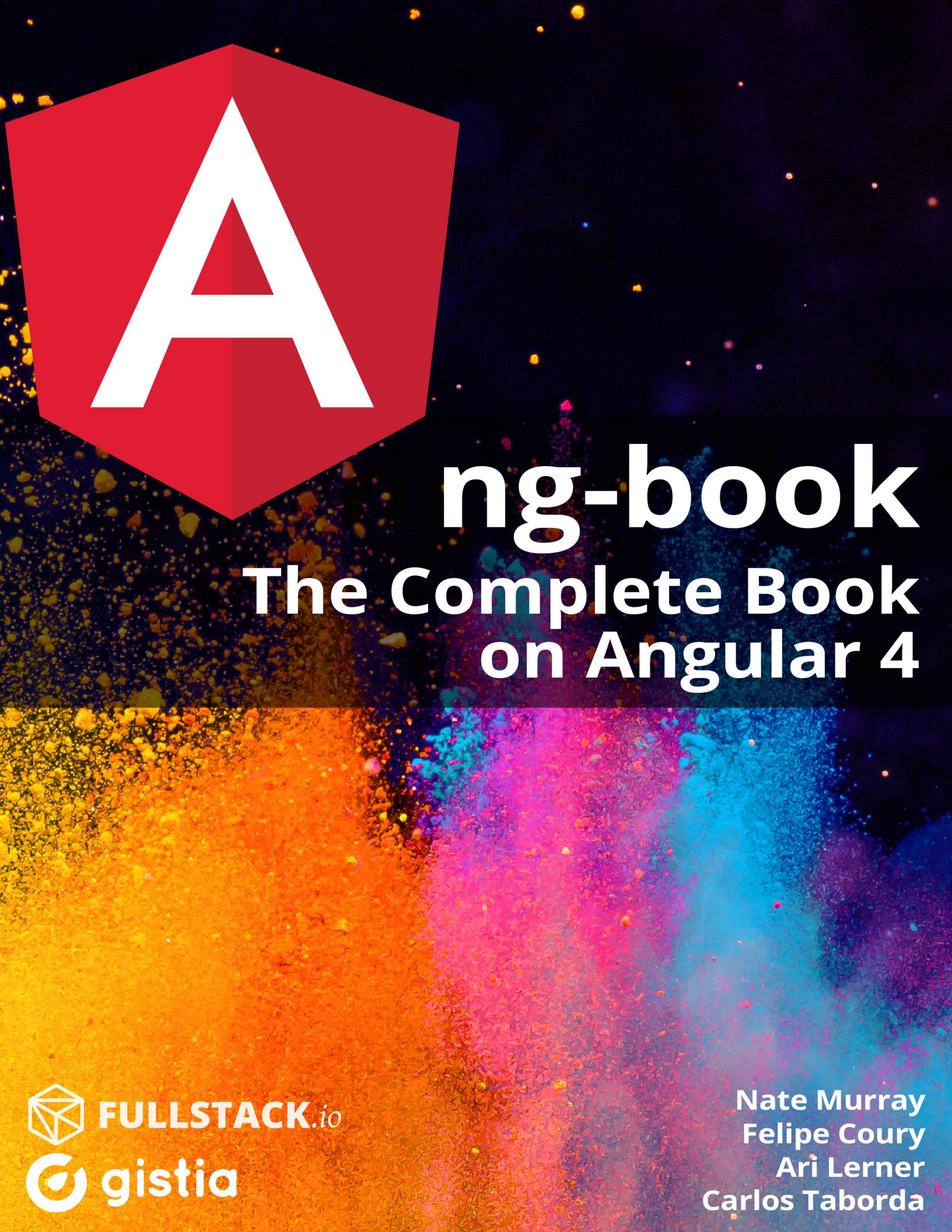

ng-book
The Complete Book
on Angular 4
FULLSTACK.io
gistia
Nate Murray
Felipe Coury
Ari Lerner
Carlos Taborda

# ng-book

***The Complete Guide to Angular***

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



Published in San Francisco, California by Fullstack.io.

We'd like to thank:

- Our technical editors: Frode Fikke, Travas Nolte, Daniel Rauf
- Nic Raboy, and Burke Holland for contributing the NativeScript chapter

# Contents


Book Revision . . . 1
Bug Reports . . . 1
Chat With The Community! . . . 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 . . . 3
A Word on Versioning . . . 4
Getting Help . . . 4
Emailing Us . . . 5
Technical Support Response Time . . . 5
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 . . . 6
Writing Application Code . . . 10
Running the application . . . 10
Making a `Component` . . . 12
Importing Dependencies . . . 13
Component Decorators . . . 14
Adding a template with `templateUrl` . . . 14
Adding a `template` . . . 15

Adding CSS Styles with `styleUrls` . . . 15
Loading Our Component . . . 16
Adding Data to the Component . . . 17
Working With Arrays . . . 20
Using the User Item Component . . . 23
Rendering the `UserItemComponent` . . . 24
Accepting Inputs . . . 25
Passing an Input value . . . 25
Bootstrapping Crash Course . . . 27
`declarations` . . . 29
`imports` . . . 29
`providers` . . . 29
`bootstrap` . . . 29
Expanding our Application . . . 30
Adding CSS . . . 31
The Application Component . . . 32
Adding Interaction . . . 34
Adding the Article Component . . . 38
Rendering Multiple Rows . . . 47
Creating an Article class . . . 47
Storing Multiple `Articles` . . . 52
Configuring the `ArticleComponent` with `inputs` . . . 53
Rendering a List of Articles . . . 55
Adding New `Articles` . . . 57
Finishing Touches . . . 58
Displaying the Article Domain . . . 58
Re-sorting Based on Score . . . 59
Deployment . . . 60
Building Our App for Production . . . 61
Uploading to a Server . . . 62
Installing `now` . . . 62
Full Code Listing . . . 62
Wrapping Up . . . 63
Getting Help . . . 63

**TypeScript . . . 64**
Angular 4 is built in TypeScript . . . 64
What do we get with TypeScript? . . . 65
Types . . . 66
Trying it out with a REPL . . . 67
Built-in types . . . 68
Classes . . . 70
Properties . . . 70

Methods . . . 71
Constructors . . . 73
Inheritance . . . 74
Utilities . . . 76
Fat Arrow Functions . . . 76
Template Strings . . . 78
Wrapping up . . . 79

**How Angular Works . . . 80**
Application . . . 80
The Navigation Component . . . 81
The Breadcrumbs Component . . . 81
The Product List Component . . . 82
How to Use This Chapter . . . 84
Product Model . . . 85
Components . . . 86
Component Decorator . . . 88
Component `selector` . . . 88
Component `template` . . . 88
Adding A `Product` . . . 89
Viewing the `Product` with Template Binding . . . 91
Adding More Products . . . 92
Selecting a `Product` . . . 93
Listing products using `<products-list>` . . . 94
The `ProductsListComponent` . . . 97
Configuring the `ProductsListComponent` `@Component` Options . . . 98
Component inputs . . . 98
Component outputs . . . 100
Emitting Custom Events . . . 101
Writing the `ProductsListComponent` Controller Class . . . 103
Writing the `ProductsListComponent` View Template . . . 104
The Full `ProductsListComponent` Component . . . 106
The `ProductRowComponent` Component . . . 108
`ProductRowComponent` Configuration . . . 109
`ProductRowComponent` `template` . . . 110
The `ProductImageComponent` Component . . . 111
The `PriceDisplayComponent` Component . . . 111
The `ProductDepartmentComponent` . . . 112
`NgModule` and Booting the App . . . 113
Booting the app . . . 115
The Completed Project . . . 116
Deploying the App . . . 116
A Word on Data Architecture . . . 117

**Built-in Directives** . . . . . **119**
Introduction . . . . . 119
`NgIf` . . . . . 119
`NgSwitch` . . . . . 120
`NgStyle` . . . . . 122
`NgClass` . . . . . 125
`NgFor` . . . . . 128
Getting an index . . . . . 133
`NgNonBindable` . . . . . 134
Conclusion . . . . . 135

**Forms in Angular** . . . . . **136**
Forms are Crucial, Forms are Complex . . . . . 136
`FormControls` and `FormGroups` . . . . . 136
`FormControl` . . . . . 136
`FormGroup` . . . . . 137
Our First Form . . . . . 138
Loading the `FormsModule` . . . . . 139
Simple SKU Form: @Component Decorator . . . . . 140
Simple SKU Form: `template` . . . . . 140
Simple SKU Form: Component Definition Class . . . . . 144
Try it out! . . . . . 144
Using `FormBuilder` . . . . . 146
Reactive Forms with `FormBuilder` . . . . . 147
Using `FormBuilder` . . . . . 147
Using `myForm` in the view . . . . . 148
Try it out! . . . . . 149
Adding Validations . . . . . 151
Explicitly setting the `sku` `FormControl` as an instance variable . . . . . 152
Custom Validations . . . . . 158
Watching For Changes . . . . . 160
ngModel . . . . . 161
Wrapping Up . . . . . 163

**Dependency Injection** . . . . . **164**
Injections Example: `PriceService` . . . . . 165
Dependency Injection Parts . . . . . 169
Playing with an Injector . . . . . 170
Providing Dependencies with `NgModule` . . . . . 174
Providers are the Key . . . . . 176
Providers . . . . . 176
Using a Class . . . . . 176
Using a Factory . . . . . 181

Dependency Injection in Apps . . . 184
More Resources . . . 185

**HTTP . . . 186**
Introduction . . . 186
Using `@angular/http` . . . 187
`import` from `@angular/http` . . . 187
A Basic Request . . . 188
Building the `SimpleHttpComponent` Component Definition . . . 189
Building the `SimpleHttpComponent` `template` . . . 189
Building the `SimpleHttpComponent` Controller . . . 190
Full `SimpleHttpComponent` . . . 192
Writing a `YouTubeSearchComponent` . . . 192
Writing a `SearchResult` . . . 194
Writing the `YouTubeSearchService` . . . 195
Writing the `SearchBoxComponent` . . . 200
Writing `SearchResultComponent` . . . 207
Writing `YouTubeSearchComponent` . . . 208
`@angular/http` API . . . 212
Making a `POST` request . . . 212
`PUT` / `PATCH` / `DELETE` / `HEAD` . . . 213
`RequestOptions` . . . 214
Summary . . . 215

**Routing . . . 216**
Why Do We Need Routing? . . . 216
How client-side routing works . . . 217
The beginning: using anchor tags . . . 218
The evolution: HTML5 client-side routing . . . 218
Writing our first routes . . . 219
Components of Angular routing . . . 219
Imports . . . 219
`Routes` . . . 220
Installing our Routes . . . 221
`RouterOutlet` using `<router-outlet>` . . . 222
`RouterLink` using `[routerLink]` . . . 224
Putting it all together . . . 224
Creating the Components . . . 226
`HomeComponent` . . . 226
`AboutComponent` . . . 227
`ContactComponent` . . . 227
Application Component . . . 228
Configuring the Routes . . . 229

Routing Strategies . . . 231
Running the application . . . 232
Route Parameters . . . 234
`ActivatedRoute` . . . 235
Music Search App . . . 236
First Steps . . . 238
The `SpotifyService` . . . 239
The `SearchComponent` . . . 240
Trying the search . . . 249
`TrackComponent` . . . 251
Wrapping up music search . . . 253
Router Hooks . . . 253
`AuthService` . . . 254
`LoginComponent` . . . 255
`ProtectedComponent` and Route Guards . . . 257
Nested Routes . . . 264
Configuring Routes . . . 265
`ProductsModule` . . . 266
Summary . . . 270

**Data Architecture in Angular 4 . . . 271**
An Overview of Data Architecture . . . 271
Data Architecture in Angular 4 . . . 272

**Data Architecture with Observables - Part 1: Services . . . 273**
Observables and RxJS . . . 273
Note: Some RxJS Knowledge Required . . . 273
Learning Reactive Programming and RxJS . . . 273
Chat App Overview . . . 275
Components . . . 276
Models . . . 277
Services . . . 278
Summary . . . 278
Implementing the Models . . . 279
`User` . . . 279
`Thread` . . . 279
`Message` . . . 280
Implementing `UsersService` . . . 281
`currentUser` stream . . . 282
Setting a new user . . . 283
`UsersService.ts` . . . 284
The `MessagesService` . . . 285
the `newMessages` stream . . . 285

the messages stream . . . 287
The Operation Stream Pattern . . . 287
Sharing the Stream . . . 289
Adding Messages to the messages Stream . . . 290
Our completed MessagesService . . . 293
Trying out MessagesService . . . 296
The ThreadsService . . . 298
A map of the current set of Threads (in threads) . . . 298
A chronological list of Threads, newest-first (in orderedthreads) . . . 303
The currently selected Thread (in currentThread) . . . 303
The list of Messages for the currently selected Thread (in currentThreadMessages) . . . 305
Our Completed ThreadsService . . . 308
Data Model Summary . . . 310

**Data Architecture with Observables - Part 2: View Components . . . 311**
Building Our Views: The AppComponent Top-Level Component . . . 311
The ChatThreadsComponent . . . 314
ChatThreadsComponent template . . . 315
The Single ChatThreadComponent . . . 315
ChatThreadComponent Controller and ngOnInit . . . 317
ChatThreadComponent template . . . 317
The ChatWindowComponent . . . 318
The ChatMessageComponent . . . 328
The ChatMessageComponent template . . . 330
The ChatNavBarComponent . . . 331
The ChatNavBarComponent @Component . . . 331
The ChatNavBarComponent template . . . 333
Summary . . . 334

**Introduction to Redux with TypeScript . . . 336**
Redux . . . 337
Redux: Key Ideas . . . 337
Core Redux Ideas . . . 338
What's a *reducer*? . . . 338
Defining Action and Reducer Interfaces . . . 339
Creating Our First Reducer . . . 339
Running Our First Reducer . . . 340
Adjusting the Counter With *actions* . . . 341
Reducer switch . . . 342
Action "Arguments" . . . 344
Storing Our State . . . 345
Using the Store . . . 346
Being Notified with subscribe . . . 346

The Core of Redux . . . 350
A Messaging App . . . 351
Messaging App `state` . . . 351
Messaging App `actions` . . . 352
Messaging App `reducer` . . . 353
Trying Out Our Actions . . . 356
Action Creators . . . 357
Using Real Redux . . . 359
Using Redux in Angular . . . 361
Planning Our App . . . 362
Setting Up Redux . . . 362
Defining the Application State . . . 362
Defining the Reducers . . . 363
Defining Action Creators . . . 364
Creating the Store . . . 364
Providing the Store . . . 366
Bootstrapping the App . . . 367
The `AppComponent` . . . 368
`imports` . . . 368
The template . . . 369
The `constructor` . . . 370
Putting It All Together . . . 372
What's Next . . . 373
References . . . 373

**Intermediate Redux in Angular . . . 375**
Context For This Chapter . . . 375
Chat App Overview . . . 376
Components . . . 376
Models . . . 377
Reducers . . . 378
Summary . . . 378
Implementing the Models . . . 379
`User` . . . 379
`Thread` . . . 379
`Message` . . . 380
App State . . . 381
A Word on Code Layout . . . 381
The Root Reducer . . . 382
The `UsersState` . . . 382
The `ThreadsState` . . . 383
Visualizing Our `AppState` . . . 384
Building the Reducers (and Action Creators) . . . 385

Set Current User Action Creators . . . 385
`UsersReducer` - Set Current User . . . 386
Thread and Messages Overview . . . 387
Adding a New Thread Action Creators . . . 387
Adding a New Thread Reducer . . . 388
Adding New Messages Action Creators . . . 389
Adding A New Message Reducer . . . 390
Selecting A Thread Action Creators . . . 392
Selecting A Thread Reducer . . . 393
Reducers Summary . . . 394
Building the Angular Chat App . . . 394
The top-level `AppComponent` . . . 396
The `ChatPage` . . . 398
Container vs. Presentational Components . . . 399
Building the `ChatNavBarComponent` . . . 400
Redux Selectors . . . 402
Threads Selectors . . . 403
Unread Messages Count Selector . . . 404
Building the `ChatThreadsComponent` . . . 405
`ChatThreadsComponent` Controller . . . 406
`ChatThreadsComponent` template . . . 408
The Single `ChatThreadComponent` . . . 409
`ChatThreadComponent` template . . . 410
Building the `ChatWindowComponent` . . . 411
The `ChatMessageComponent` . . . 418
Setting `incoming` . . . 419
The `ChatMessageComponent` template . . . 420
Summary . . . 421

**Advanced Components . . . 423**
Styling . . . 423
View (Style) Encapsulation . . . 426
Shadow DOM Encapsulation . . . 430
No Encapsulation . . . 432
Creating a Popup - Referencing and Modifying Host Elements . . . 435
Popup Structure . . . 435
Using `ElementRef` . . . 437
Binding to the `host` . . . 439
Adding a Button using `exportAs` . . . 442
Creating a Message Pane with Content Projection . . . 444
Changing the Host's CSS . . . 445
Using `ng-content` . . . 445
Querying Neighbor Directives - Writing Tabs . . . 447

ContentTabComponent . . . 448
ContentTabsetComponent Component . . . 449
Using the ContentTabsetComponent . . . 451
Lifecycle Hooks . . . 453
OnInit and OnDestroy . . . 454
OnChanges . . . 458
DoCheck . . . 464
AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked . . . 477
Advanced Templates . . . 484
Rewriting ngIf - ngBookIf . . . 485
Rewriting ngFor - NgBookFor . . . 487
Change Detection . . . 493
Customizing Change Detection . . . 497
Zones . . . 504
Observables and OnPush . . . 505
Summary . . . 509

**Testing . . . 510**
Test driven? . . . 510
End-to-end vs. Unit Testing . . . 510
Testing Tools . . . 511
Jasmine . . . 511
Karma . . . 512
Writing Unit Tests . . . 512
Angular Unit testing framework . . . 512
Setting Up Testing . . . 513
Testing Services and HTTP . . . 515
HTTP Considerations . . . 516
Stubs . . . 516
Mocks . . . 517
Http MockBackend . . . 518
TestBed.configureTestingModule and Providers . . . 518
Testing getTrack . . . 519
Testing Routing to Components . . . 526
Creating a Router for Testing . . . 526
Mocking dependencies . . . 529
Spies . . . 530
Back to Testing Code . . . 532
fakeAsync and advance . . . 535
inject . . . 536
Testing ArtistComponent's Initialization . . . 536
Testing ArtistComponent Methods . . . 537
Testing ArtistComponent DOM Template Values . . . 539

Testing Forms . . . 541
Creating a ConsoleSpy . . . 544
Installing the ConsoleSpy . . . 545
Configuring the Testing Module . . . 546
Testing The Form . . . 546
Refactoring Our Form Test . . . 548
Testing HTTP requests . . . 552
Testing a POST . . . 552
Testing DELETE . . . 554
Testing HTTP Headers . . . 555
Testing YouTubeSearchService . . . 556
Conclusion . . . 563

**Converting an AngularJS 1.x App to Angular 4 . . . 564**
Peripheral Concepts . . . 564
What We're Building . . . 565
Mapping AngularJS 1 to Angular 4 . . . 566
Requirements for Interoperability . . . 568
The AngularJS 1 App . . . 568
The ng1-app HTML . . . 570
Code Overview . . . 571
ng1: PinsService . . . 571
ng1: Configuring Routes . . . 573
ng1: HomeController . . . 574
ng1: / HomeController template . . . 574
ng1: pin Directive . . . 575
ng1: pin Directive template . . . 575
ng1: AddController . . . 577
ng1: AddController template . . . 579
ng1: Summary . . . 582
Building A Hybrid . . . 582
Hybrid Project Structure . . . 583
Bootstrapping our Hybrid App . . . 585
What We'll Upgrade . . . 587
A Minor Detour: Typing Files . . . 590
Writing ng2 PinControlsComponent . . . 593
Using ng2 PinControlsComponent . . . 595
Downgrading ng2 PinControlsComponent to ng1 . . . 596
Adding Pins with ng2 . . . 598
Upgrading ng1 PinsService and $state to ng2 . . . 599
Writing ng2 AddPinComponent . . . 600
Using AddPinComponent . . . 606
Exposing an ng2 service to ng1 . . . 606

Writing the `AnalyticsService` . . . 607
Downgrade ng2 `AnalyticsService` to ng1 . . . 607
Using `AnalyticsService` in ng1 . . . 608
Summary . . . 609
References . . . 610

**NativeScript: Mobile Applications for the Angular Developer . . . 611**
What is NativeScript? . . . 611
Where NativeScript Differs from Other Popular Frameworks . . . 612
What are the System and Development Requirements for NativeScript? . . . 613
Creating your First Mobile Application with NativeScript and Angular . . . 615
Adding Build Platforms for Cross Platform Deployment . . . 615
Building and Testing for Android and iOS . . . 615
Installing JavaScript, Android, and iOS Plugins and Packages . . . 616
Understanding the Web to NativeScript UI and UX Differences . . . 617
Planning the NativeScript Page Layout . . . 617
Adding UI Components to the Page . . . 619
Styling Components with CSS . . . 620
Developing a Geolocation Based Photo Application . . . 621
Creating a Fresh NativeScript Project . . . 622
Creating a Multiple Page Master-Detail Interface . . . 623
Creating a Flickr Service for Obtaining Photos and Data . . . 627
Creating a Service for Calculating Device Location and Distance . . . 632
Including Mapbox Functionality in the NativeScript Application . . . 636
Implementing the First Page of the Geolocation Application . . . 637
Implementing the Second Page of the Geolocation Application . . . 643
Try it out! . . . 644
NativeScript for Angular Developers . . . 645

**Changelog . . . 646**
Revision 63 - `2017-08-02` . . . 646
Revision 62 - `2017-06-23` . . . 646
Revision 61 - `2017-05-24` . . . 646
Revision 60 - `2017-04-27` . . . 646
Revision 59 - `2017-04-07` . . . 647
Revision 58 - `2017-03-24` . . . 647
Revision 57 - `2017-03-23` . . . 647
Revision 56 - `2017-03-22` . . . 647
Revision 55 - `2017-03-17` . . . 648
Revision 54 - `2017-03-10` . . . 648
Revision 53 - `2017-03-01` . . . 648
Revision 52 - `2017-02-22` . . . 648
Revision 51 - `2017-02-14` . . . 649

Revision 50 - 2017-02-10 . . . . . 649
Revision 49 - 2017-01-18 . . . . . 649
Revision 48 - 2017-01-13 . . . . . 649
Revision 47 - 2017-01-06 . . . . . 649
Revision 46 - 2017-01-03 . . . . . 649
Revision 45 - 2016-12-05 . . . . . 649
Revision 44 - 2016-11-17 . . . . . 649
Revision 43 - 2016-11-08 . . . . . 650
Revision 42 - 2016-10-14 . . . . . 650
Revision 41 - 2016-09-28 . . . . . 650
Revision 40 - 2016-09-20 . . . . . 651
Revision 39 - 2016-09-03 . . . . . 651
Revision 38 - 2016-08-29 . . . . . 651
Revision 37 - 2016-08-02 . . . . . 651
Revision 36 - 2016-07-20 . . . . . 651
Revision 35 - 2016-06-30 . . . . . 651
Revision 34 - 2016-06-15 . . . . . 652
Revision 33 - 2016-05-11 . . . . . 652
Revision 32 - 2016-05-06 . . . . . 652
Revision 31 - 2016-04-28 . . . . . 653
Revision 30 - 2016-04-20 . . . . . 653
Revision 29 - 2016-04-08 . . . . . 653
Revision 28 - 2016-04-01 . . . . . 653
Revision 27 - 2016-03-25 . . . . . 653
Revision 26 - 2016-03-24 . . . . . 653
Revision 25 - 2016-03-21 . . . . . 653
Revision 24 - 2016-03-10 . . . . . 653
Revision 23 - 2016-03-04 . . . . . 654
Revision 22 - 2016-02-24 . . . . . 654
Revision 21 - 2016-02-20 . . . . . 654
Revision 20 - 2016-02-11 . . . . . 655
Revision 19 - 2016-02-04 . . . . . 655
Revision 18 - 2016-01-29 . . . . . 655
Revision 17 - 2016-01-28 . . . . . 655
Revision 16 - 2016-01-14 . . . . . 655
Revision 15 - 2016-01-07 . . . . . 656
Revision 14 - 2015-12-23 . . . . . 656
Revision 13 - 2015-12-17 . . . . . 656
Revision 12 - 2015-11-16 . . . . . 657
Revision 11 - 2015-11-09 . . . . . 657
Revision 10 - 2015-10-30 . . . . . 658
Revision 9 - 2015-10-15 . . . . . 658

Revision 8 - 2015-10-08 . . . . . 658
Revision 7 - 2015-09-23 . . . . . 658
Revision 6 - 2015-08-28 . . . . . 659
Revision 5 - 2015-08-01 . . . . . 659
Revision 4 - 2015-07-30 . . . . . 659
Revision 3 - 2015-07-21 . . . . . 659
Revision 2 - 2015-07-15 . . . . . 659
Revision 1 - 2015-07-01 . . . . . 659

# Book Revision

Revision 63 - Covers up to Angular 4 (4.3.2, 2017-08-02)

# Bug Reports

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

# Chat With The Community!

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

# 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[3].

# Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow @fullstackio[4]

# We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[5].

[1] mailto:us@fullstack.io?Subject=ng-book%202%20feedback
[2] https://gitter.im/ng-book/ng-book
[3] https://fullstackio.canny.io/ng-book
[4] https://twitter.com/fullstackio
[5] mailto:us@fullstack.io?Subject=ng-book%202%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`[6] 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.

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.

[6] https://www.npmjs.com/

## Angular CLI

With a couple of minor exceptions, every project in this book was built on Angular CLI[7]. 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 works!';
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`.

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.

[7] https://github.com/angular/angular-cli

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 4*, 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[8]. 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.

# 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[9] 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. Our preferred way to receive this would be for you to send us a Plunkr link by using this URL[10].

That URL contains a runnable, boilerplate Angular app. If you can copy and paste your code into that project, reproduce your error, and send it to us **you'll greatly increase the likelihood of a prompt, helpful response**.

When you've written down these things, email us at us@fullstack.io[11]. We look forward to hearing from you.

## Technical Support Response Time

We perform our free, technical support **once per week**.

If you need a faster response time, and help getting **any** of your team's questions answered, then you may consider our premium support option[12].

[8] https://gitter.im/ng-book/ng-book

[9] http://mattgemmell.com/what-have-you-tried/

[10] https://angular.io/resources/live-examples/quickstart/ts/eplnkr.html

[11] mailto:us@fullstack.io

[12] mailto:us@fullstack.io?Subject=Angular%20Premium%20Support&Body=Hello%21%20I%27m%20interested%20in%20premium%20Angular%20support%20for%20our%20team

# 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 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 4+. 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 (@eigenjoy[13])

[13] 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[14] or Product Hunt[15].

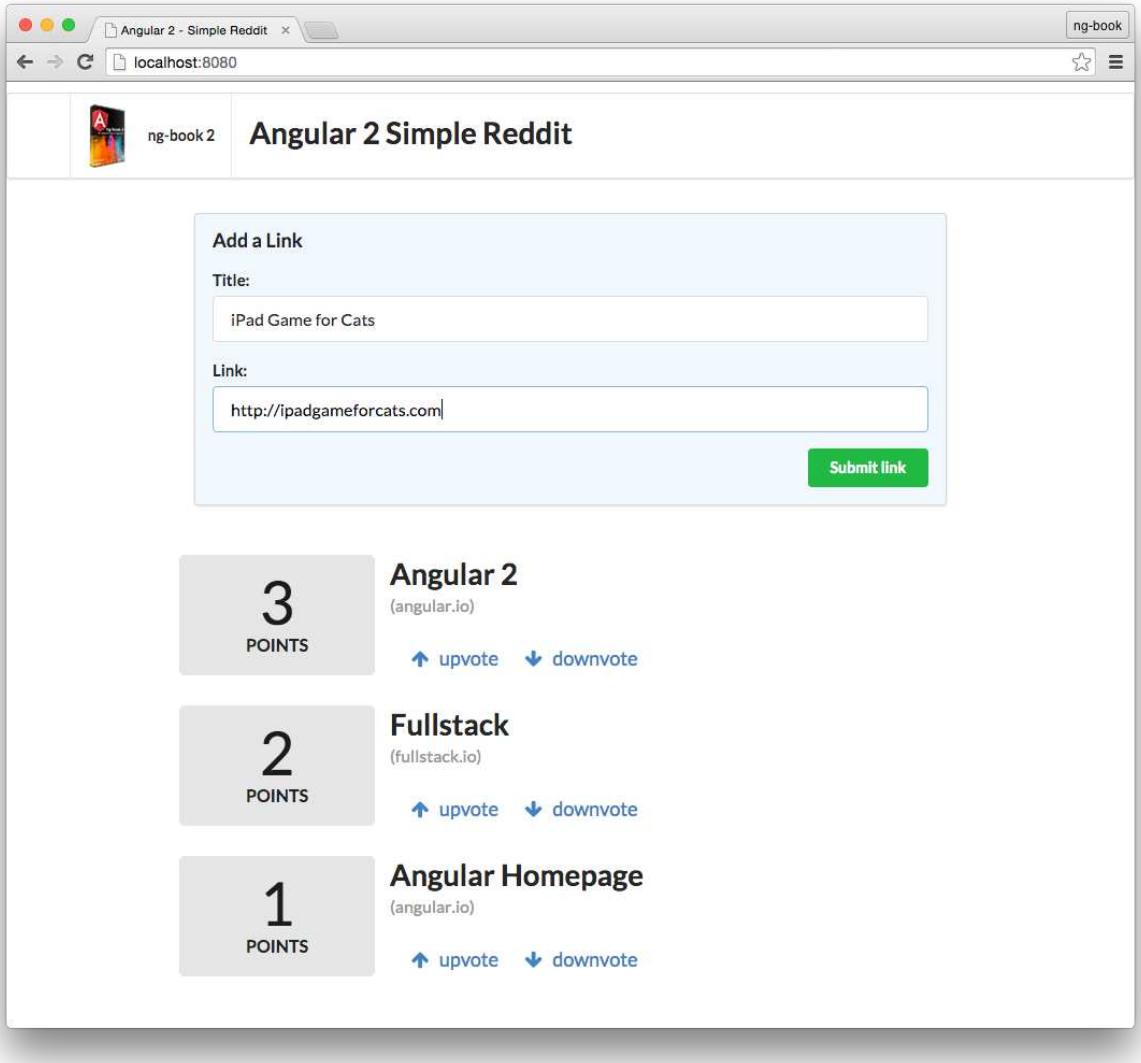
In this simple app we're going to cover most of the essentials of Angular including:

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

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

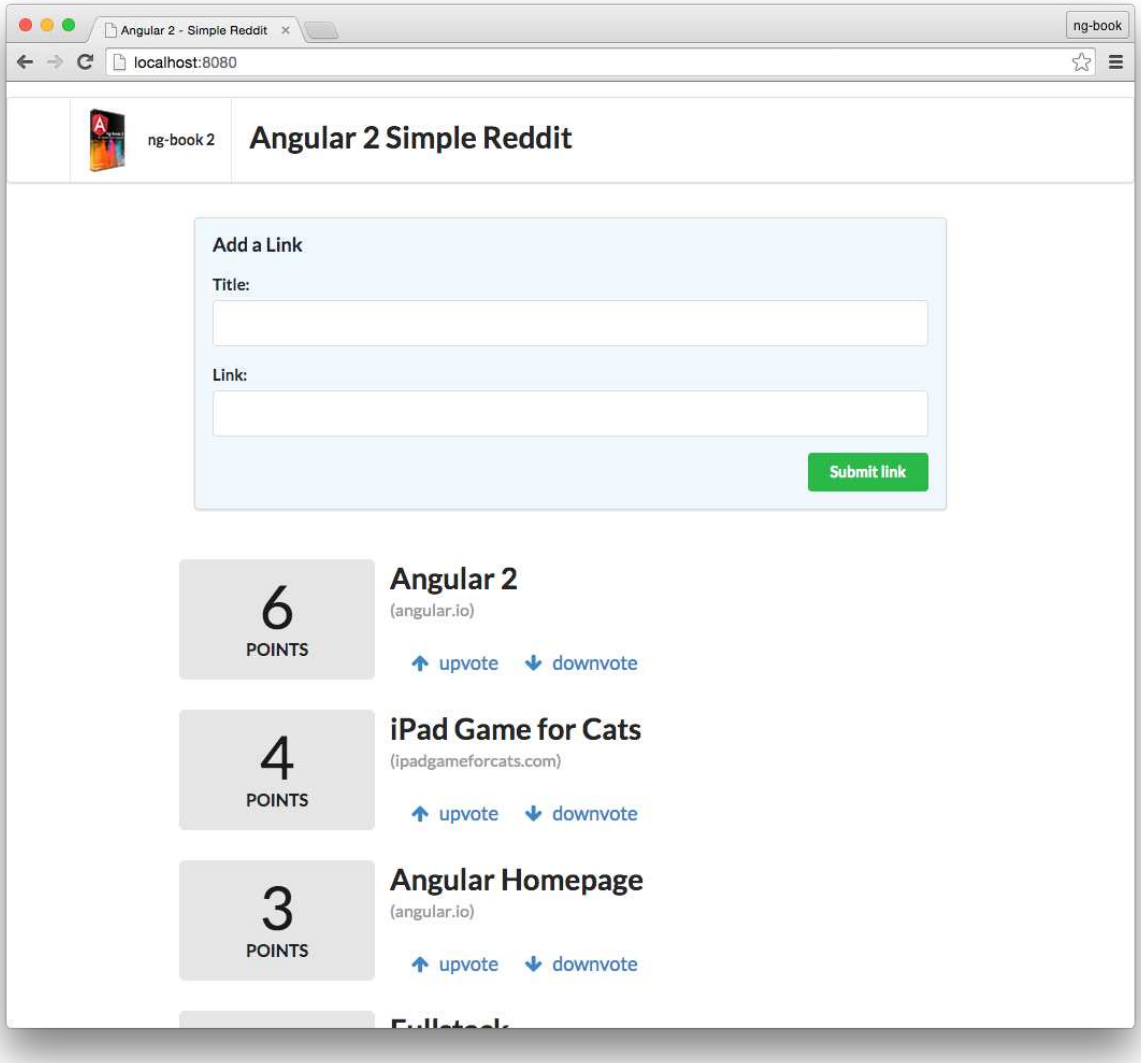
Here's a screenshot of what our app will look like when it's done:

[14] http://reddit.com
[15] http://producthunt.com




**Completed application**

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




**App with new article**

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

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

# Getting started

## Node.js and `npm`

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

**Make sure you install Node `6.9.0` or higher**.

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

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

```
$ npm -v
```

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

Your `npm` version should be `3.0.0` or higher.

## TypeScript

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

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

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

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

## Browser

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

# 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 -1p`).

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, just run the following command:

```
1 $ npm install -g @angular/cli@1.0.0
```

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

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

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

[17] https://www.google.com/chrome/

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

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

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

```
$ brew install watchman
```

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

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

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

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

```
$ 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:

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

Let's break it down a bit:

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

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

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</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**

```
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

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

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

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

## Writing Application Code

# Running the application

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

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

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

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

http://localhost:4200[18]

[18] http://localhost:4200

Note that if you get the message:

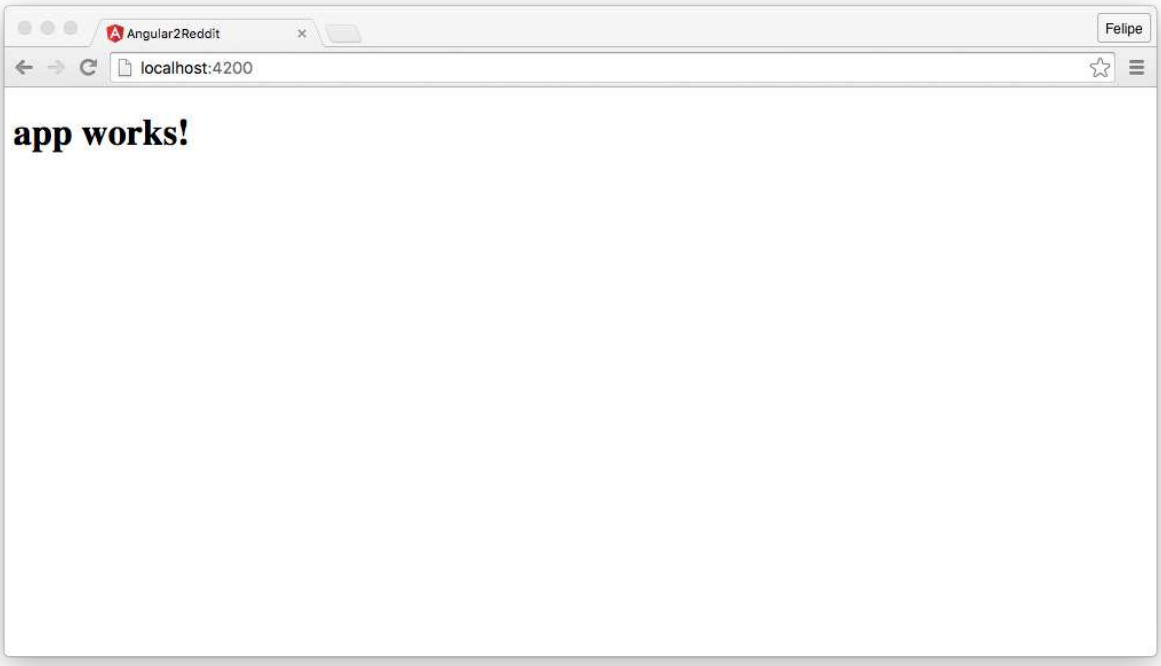
```
Port 4200 is already in use. Use '--port' to specify a different port
```

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:

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

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

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




**Running application**

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

## Making a `Component`

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

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

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

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

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

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

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

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

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

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

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

1. A `Component` 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**

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

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

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

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

## Adding a `template`

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

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

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

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

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

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

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

## Adding CSS Styles with `styleUrls`

Notice the key `styleUrls`:

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

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

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

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

## Loading Our Component

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

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

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

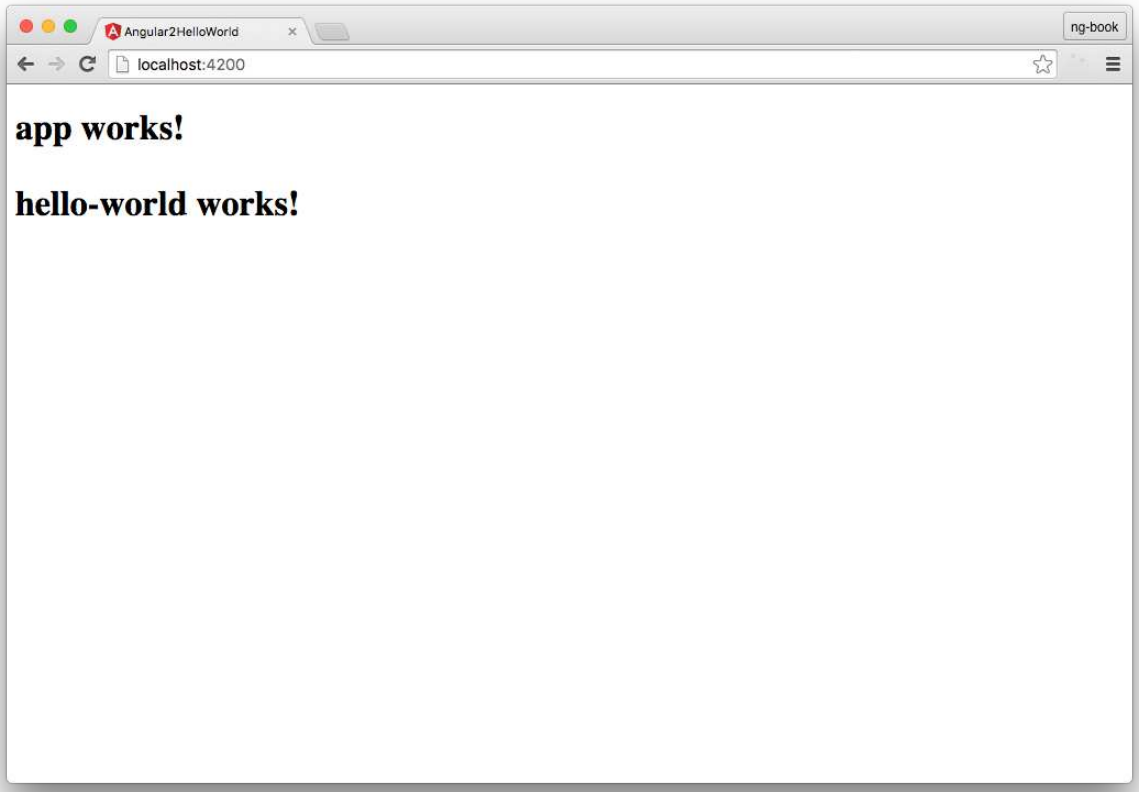
Remember that because we configured our `HelloWorldComponent` with the selector `app-hello-world`, we can use the `<app-hello-world></app-hello-world>` in our template. Let's add the `<app-hello-world>` tag to `app.component.html`:

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

```
<h1>
  {{title}}

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

```
<h1>
  {{title}}

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

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

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

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

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

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

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

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

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

  ngOnInit() {
  }

}
```

Notice that we've changed two things:

### 1. `name` Property

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

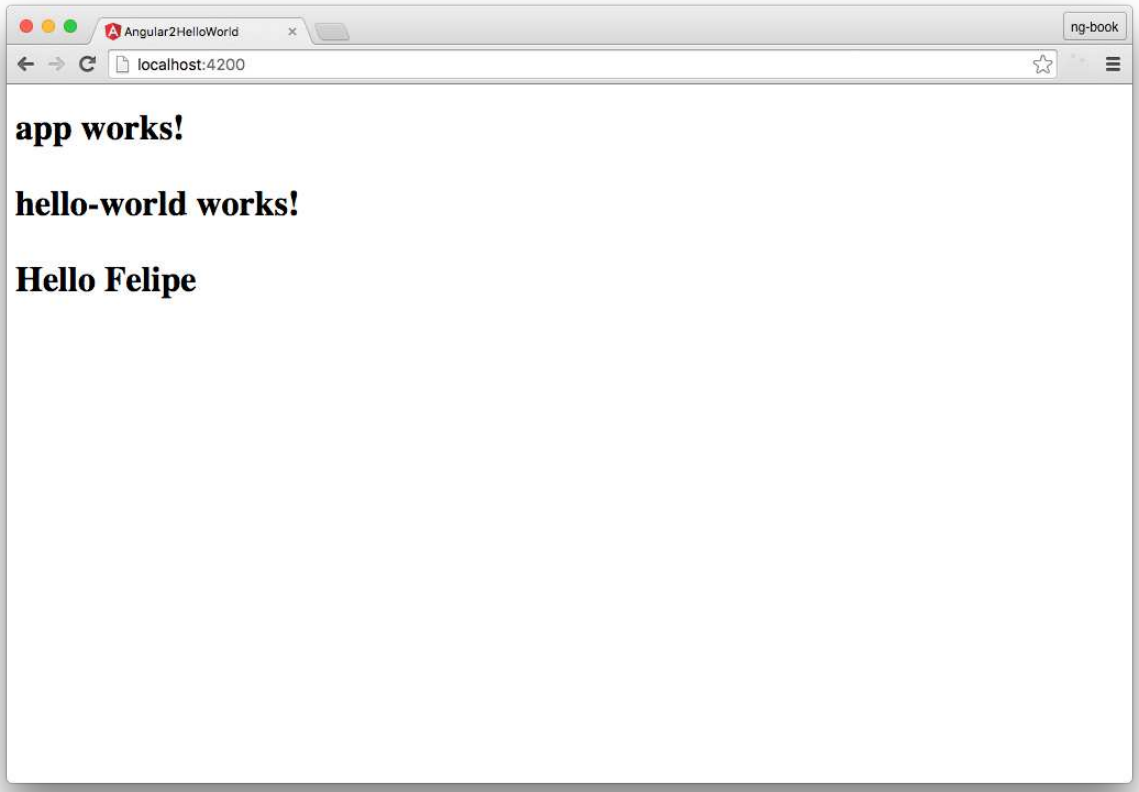
```
1  <p>
2    Hello {{ name }}
3  </p>
```

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**

```
<h1>
  {{title}}

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

  <app-user-list></app-user-list>
</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**

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

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

  ngOnInit() {
  }

}
```

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

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

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

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

Here's what our new template will look like:

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

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

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

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

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

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

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

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

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

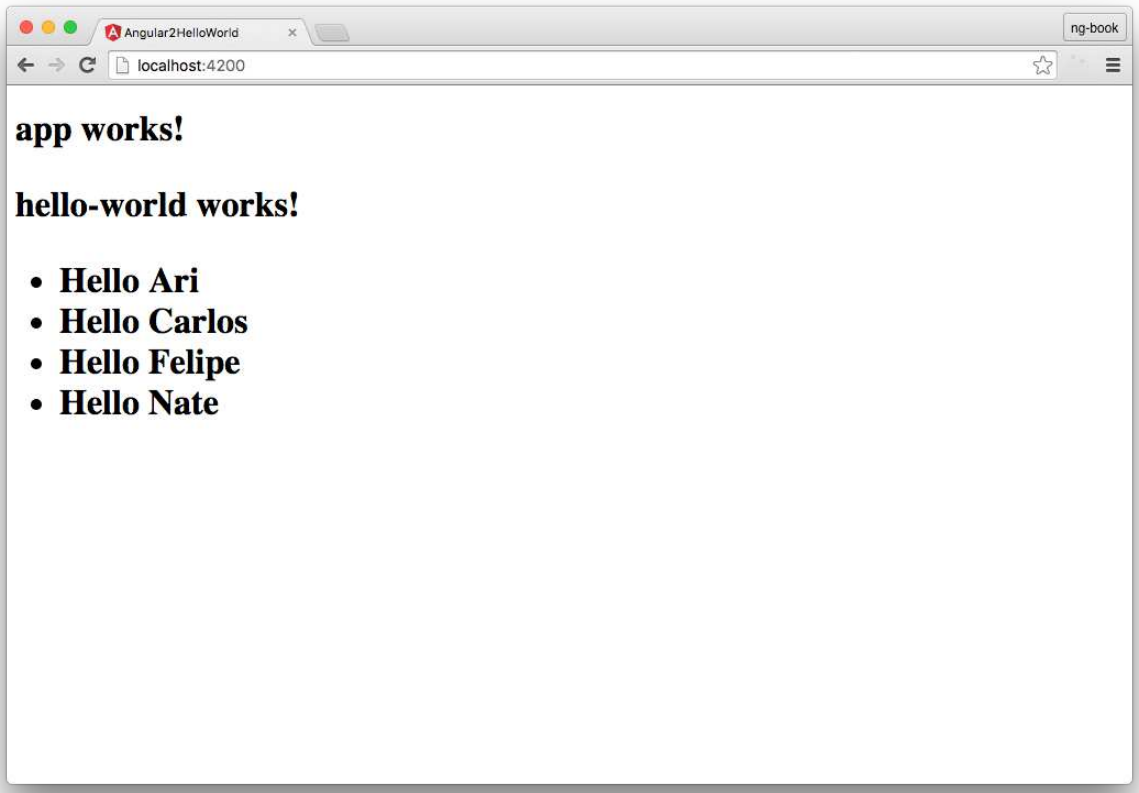
`NgFor` repeats the element that the `ngFor` is called. That is, we put it on the `li` tag and **not** the `ul` tag because we want to repeat the list element (`li`) and not the list itself (`ul`).

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[19].

[19] https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_for_of.ts

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




**Application with Data**

## Using the User Item Component

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

To do this, we need to do three things:

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

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

## Rendering the `UserItemComponent`

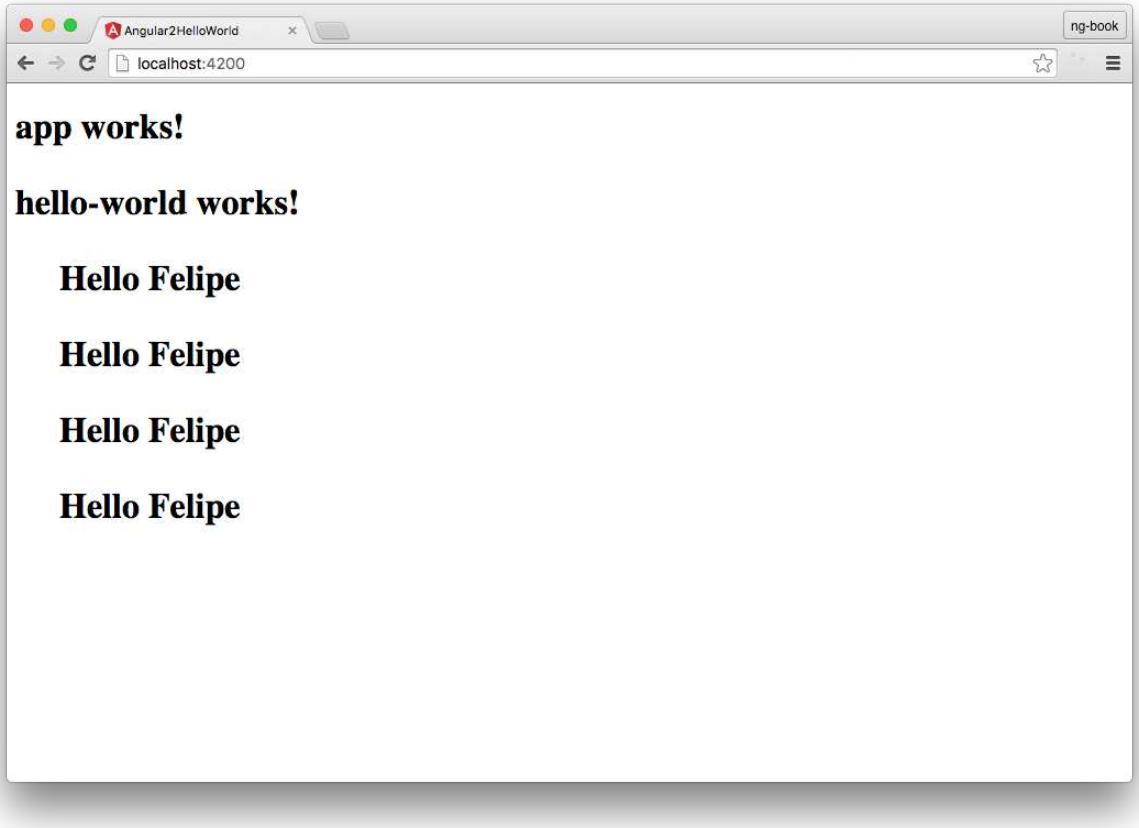
Our `UserItemComponent` specifies the selector `app-user-item` - let's add that tag to our template:

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

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

```
import {
  Component,
  OnInit,
  Input      // <--- added this
} from '@angular/core';

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.css']
})
export class UserItemComponent implements OnInit {
  @Input() name: string; // <-- added Input annotation

  constructor() {
    // removed setting name
  }

  ngOnInit() {
  }

}
```

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

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

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

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

## Passing an Input value

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

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

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

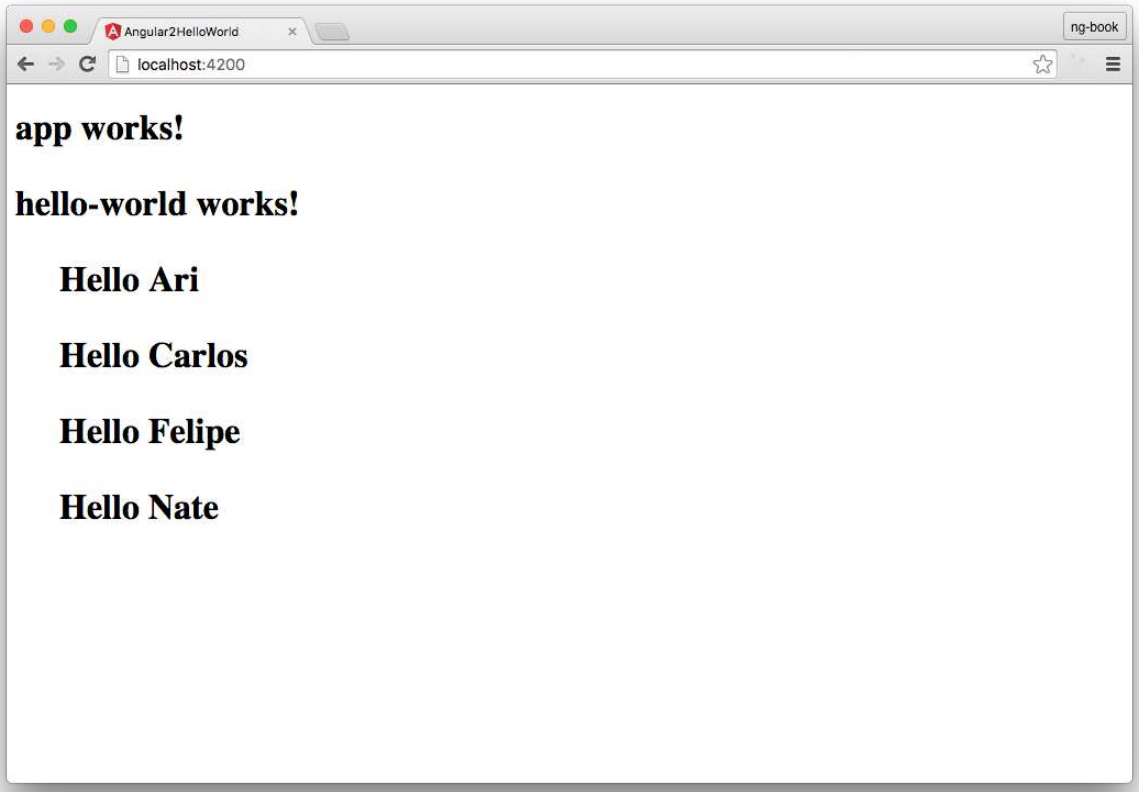
```
  <li *ngFor="let individualUserName of names">
    <app-user-item [name]="individualUserName"></app-user-item>
  </li>
```

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

At a high level, it looks like this:

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

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

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

Take a look at this code:

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

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

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

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

### `declarations`

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

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

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

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

### `imports`

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

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

**`import` vs. `imports`?**

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

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

### `providers`

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

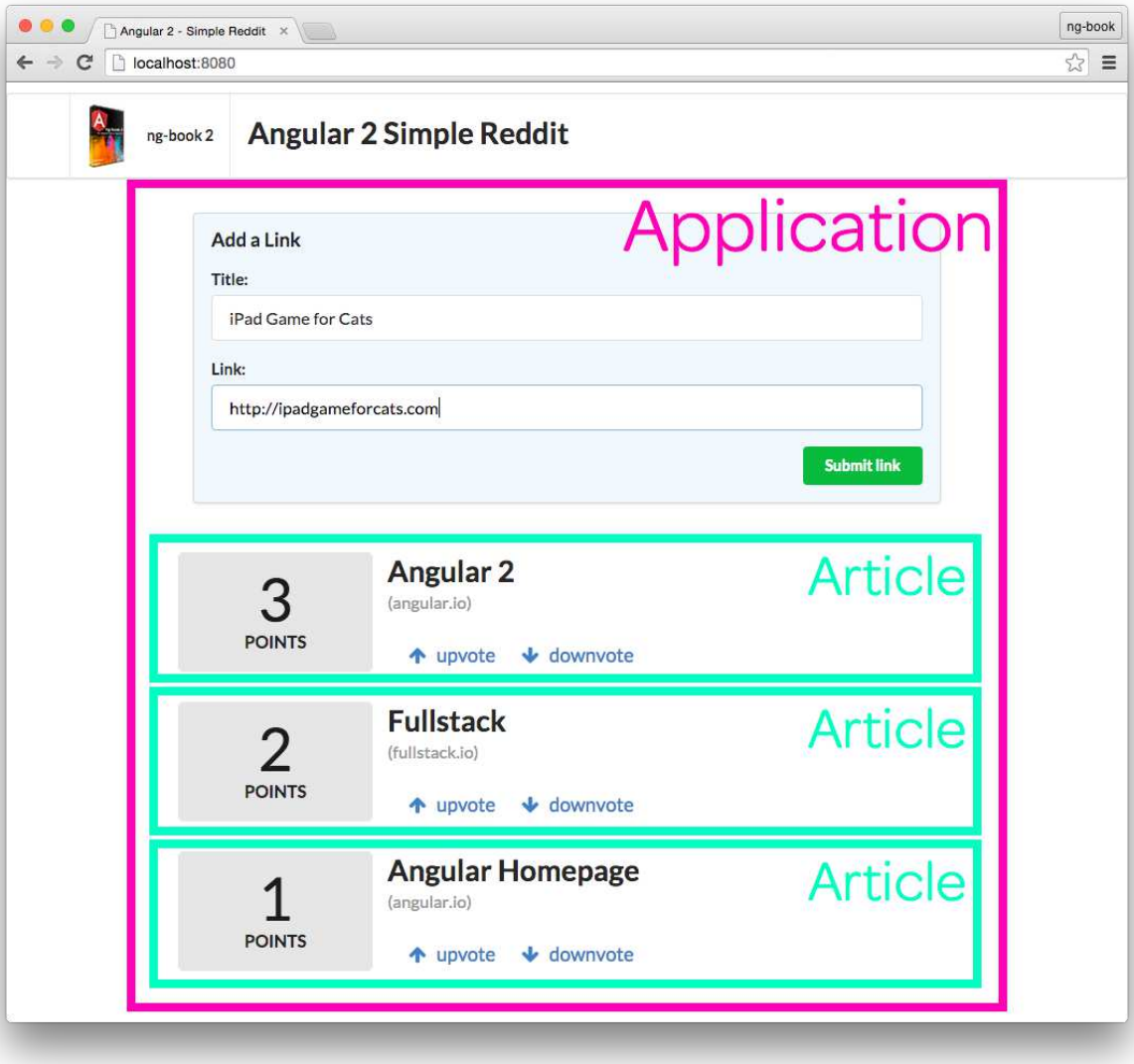
Learn more about this in the section on `Dependency Injection`.

### `bootstrap`

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

# Expanding our Application

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




**Application with Data**

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

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

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

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

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

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

We've 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[20] to help with the styling. Semantic-UI is a CSS framework, similar to Zurb Foundation[21] or Twitter Bootstrap[22]. We've included it in the sample code download so all you need to do is copy over the files specified above.

[20] http://semantic-ui.com/
[21] http://foundation.zurb.com
[22] http://getbootstrap.com

## The Application Component

Let's now build a new component which will:

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

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

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

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

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

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

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

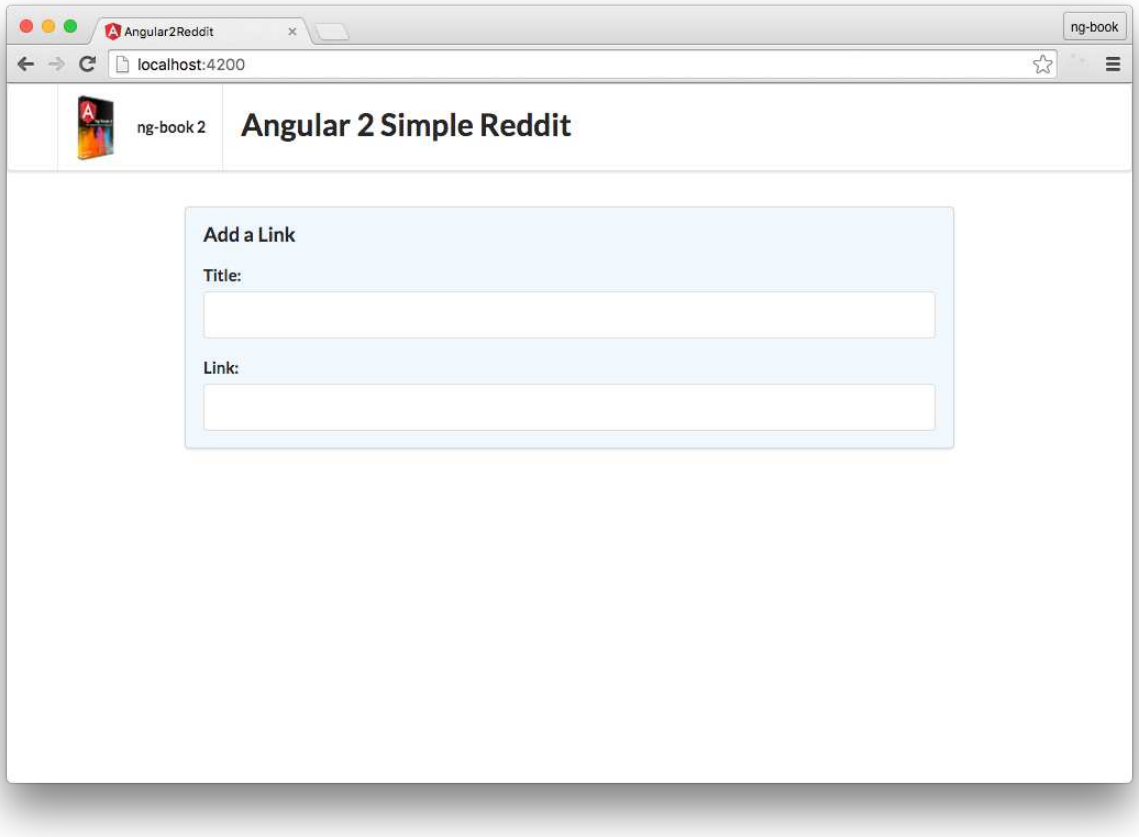
**code/first-app/angular-reddit/src/app/app.component.html**

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

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

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

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




**Form**

## Adding Interaction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Binding `input`s to values

Notice in our first input tag we have the following:

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

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

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

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

## Binding actions to events

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

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

All together:

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

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

## Defining the Action Logic

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

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

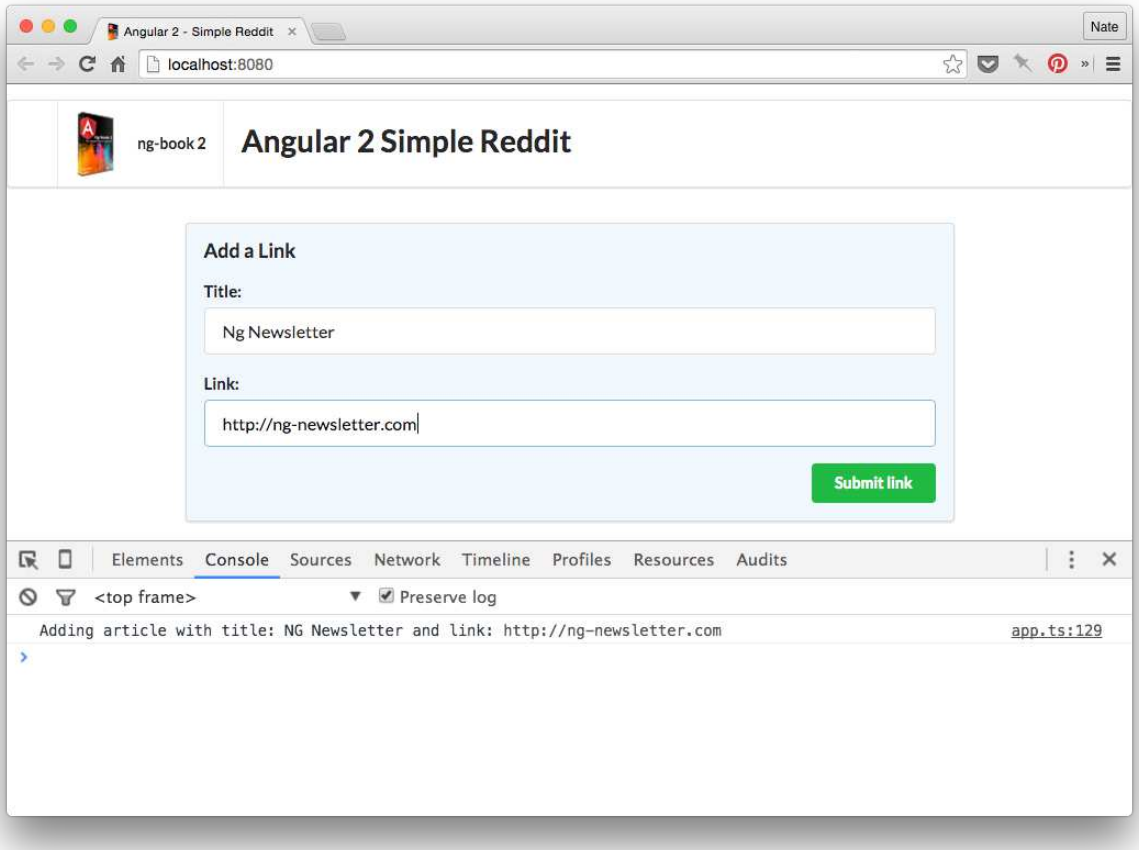
```
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
```

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

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

## Try it out!

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




**Clicking the Button**

## Adding the Article Component

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

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


**A reddit-article**

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

```
ng generate component article
```

We have three parts to defining this new component:

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

Let's talk through each part in detail:

### Creating the `ArticleComponent` template

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

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

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

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

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


**A Single reddit-article Row**

We have two columns:

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

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

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

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

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

### Creating the `ArticleComponent`

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

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

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**

```
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  votes: number;
  title: string;
  link: string;

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

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

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

```
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[23] called `row`.

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

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

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

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

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

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

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

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

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

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

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

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

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

### Using the `app-article` Component

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

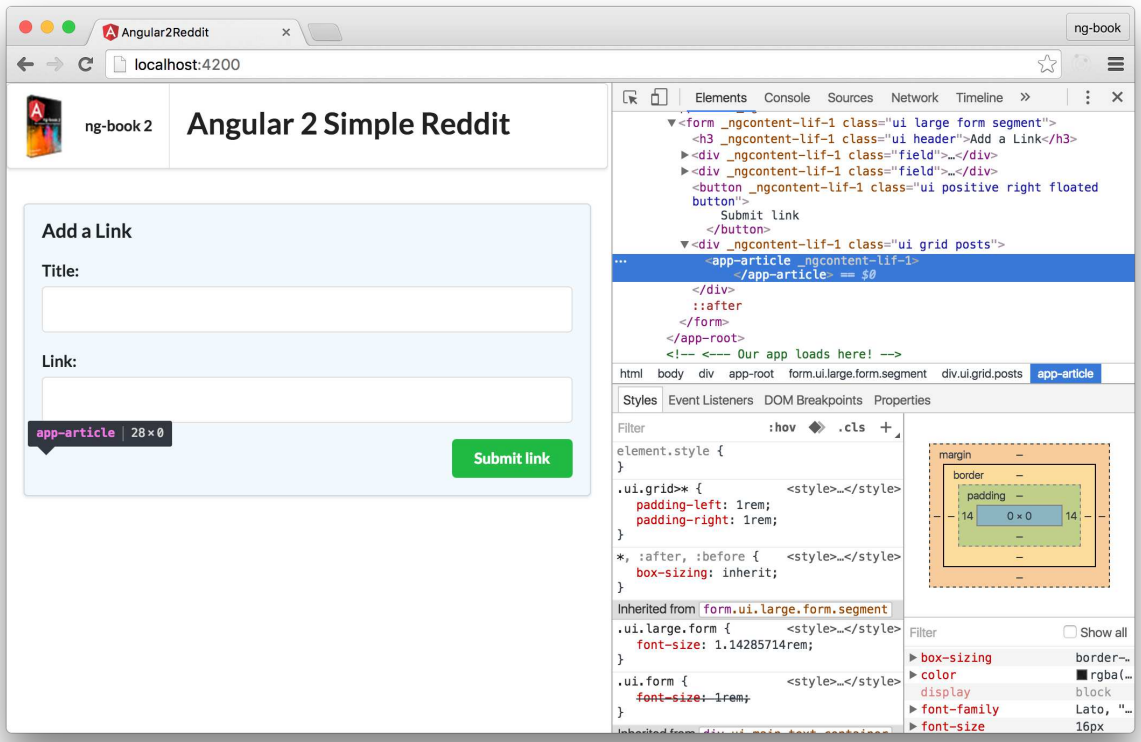
In this case, we want the `AppComponent` to render this new component, so let's update the code in that component. Add the `<app-article>` tag to the `AppComponent`'s template right after the closing `</form>` tag:

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

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

If we 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.

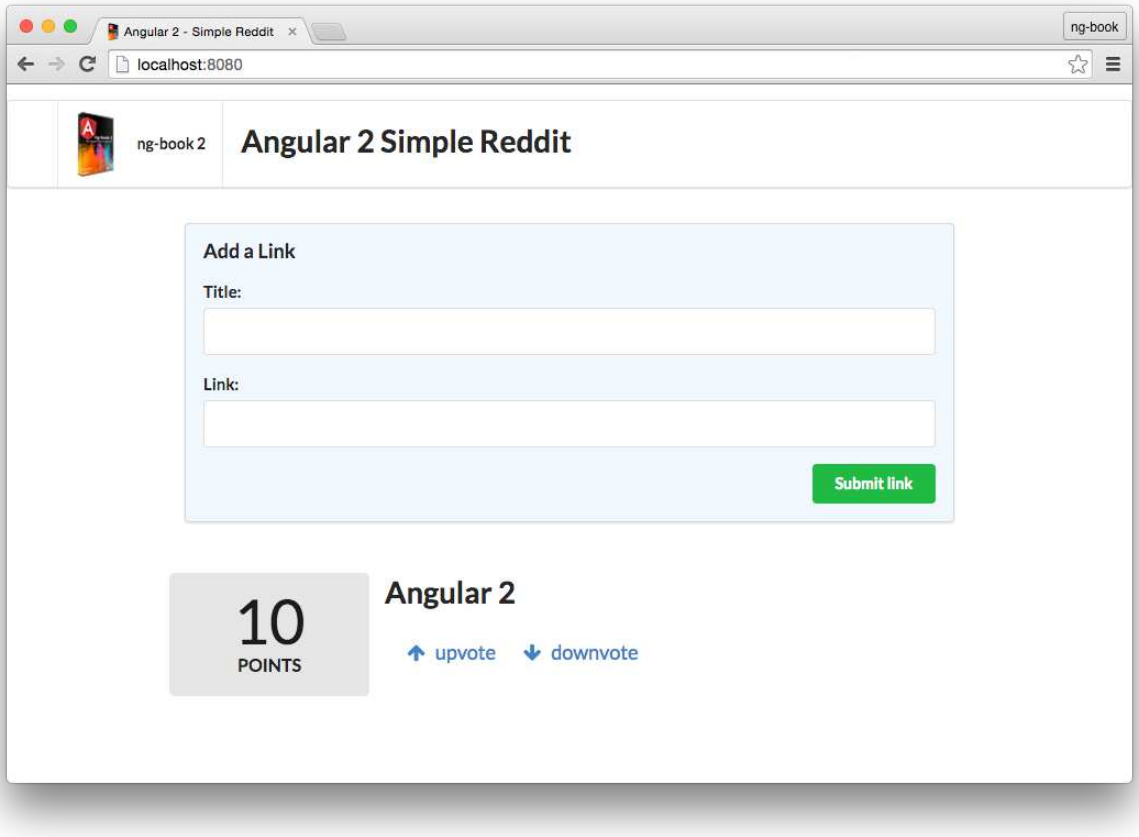
**code/first-app/angular-reddit/src/app/app.module.ts**

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

See here that we are:

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

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




**Rendered ArticleComponent component**

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

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

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

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

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

# Rendering Multiple Rows

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

## Creating an Article class

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

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

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

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

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

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

First let's import the class:

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

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

Then let's use it:

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

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

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

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

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

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

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

Reload the browser and everything still works.

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

`voteUp` and `voteDown` currently break the Law of Demeter[24] 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

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

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

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

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

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

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

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

  // domain() is a utility function that extracts
  // the domain from a URL, which we'll explain shortly
  domain(): string {
    try {
      // e.g. http://foo.com/path/to/bar
      const domainAndPath: string = this.link.split('//')[1];
      // e.g. foo.com/path/to/bar
      return domainAndPath.split('/')[0];
    } catch (err) {
      return null;
    }
  }
}
```

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

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

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

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

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

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

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

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

Checkout our `ArticleComponent` component definition now: it's so short! We've moved a lot of logic **out** of our component and into our models. The corresponding MVC guideline here might be Fat Models, Skinny Controllers[25]. The idea is that we want to move most of our logic to our models so that our components do the minimum work possible.

## Storing Multiple `Article`s

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

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

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

```
import { Component } from '@angular/core';
import { Article } from './article/article.model'; // <-- import this

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  articles: Article[]; // <-- component property

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

Notice that our `AppComponent` has the line:

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

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

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

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

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

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

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

## Configuring the `ArticleComponent` with `inputs`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
import {
  Component,
  OnInit,
  Input,        // <-- added,
  HostBinding
} from '@angular/core';
import { Article } from './article.model'; // <-- added

@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css']
})
export class ArticleComponent implements OnInit {
  @HostBinding('attr.class') cssClass = 'row';
  @Input() article: Article;

```

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

**Don't forget to `import`!**

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

## Rendering a List of Articles

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

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

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

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

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

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

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

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

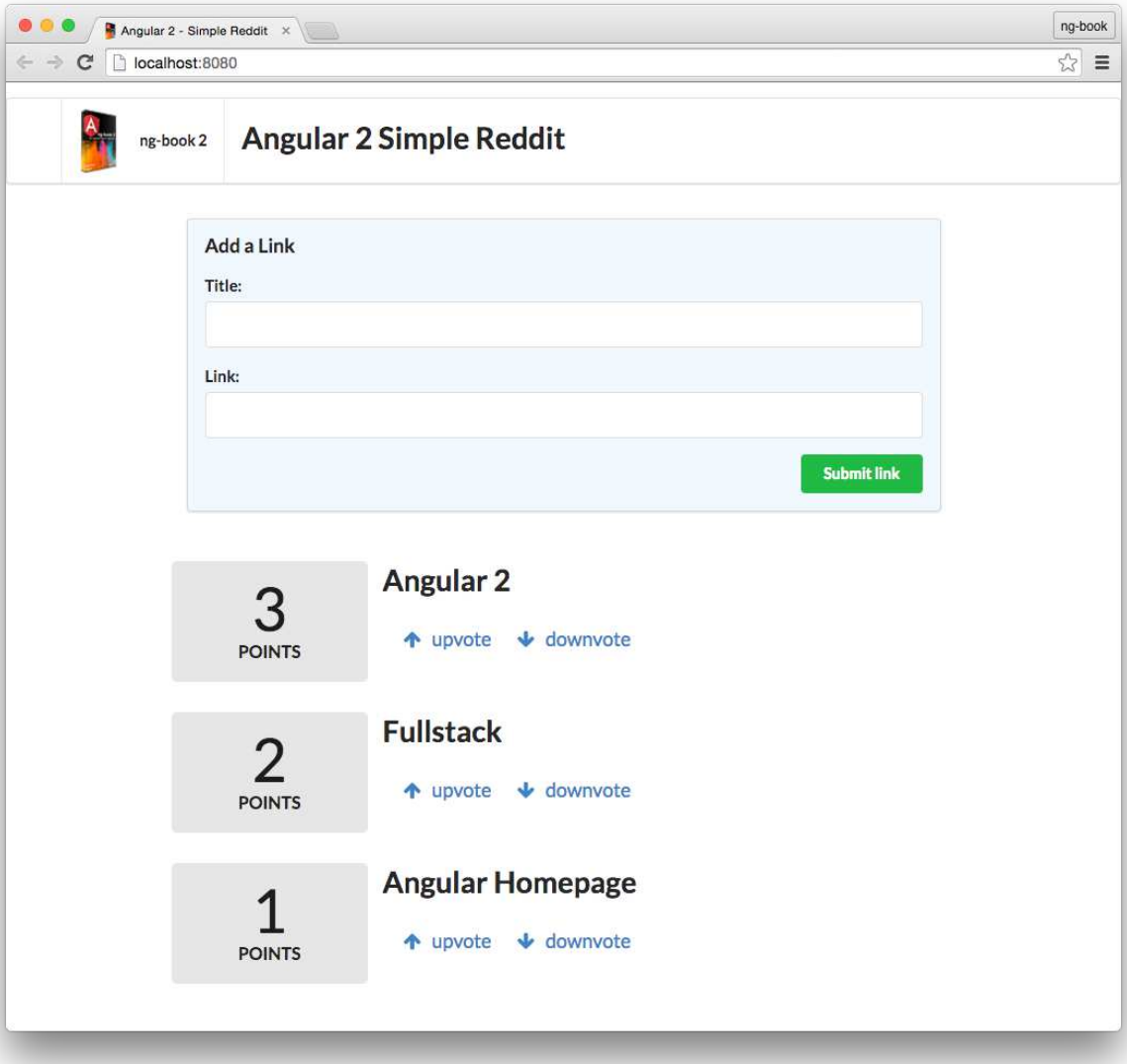
```
1   <app-article
2     *ngFor="let foobar of articles"
3     [article]="foobar">
4   </app-article>
```

So here we have three variables:

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

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

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




**Multiple articles being rendered**

## Adding New `Article`s

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

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

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

This will:

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

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

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

# Finishing Touches

## Displaying the Article Domain

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

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

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

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

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

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

And now when we reload the browser, we will see the domain name of each URL (note: URL must include *http://*).

## Re-sorting Based on Score

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

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

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

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

**ES6 Arrow Function**

The above code snippet uses "arrow" (=>) functions from ES6. You can read more about arrow functions here[26]

**`sort()`** We're also calling the `sort()` function, which is a built-in which you can read about here[27]

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

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

# Deployment

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

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

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

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

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

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

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

[26] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[27] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

## Building Our App for Production

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

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

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

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

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

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

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

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

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

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

## Uploading to a Server

There are **lots** of ways to host your HTML and JavaScript. For this demo, we're going to use the easiest way possible: now[28].

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

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

## Installing now

We can install now using npm:

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

To deploy a site with now is very easy:

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

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

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

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

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

# Full Code Listing

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

[28] https://zeit.co/now

# Wrapping Up

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

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

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

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

# Getting Help

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

- Come join our community and chat with us on Gitter[29]
- Email us directly at us@fullstack.io[30]

Onward!

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

# GET THE FULL BOOK

This is the end of the preview chapter!

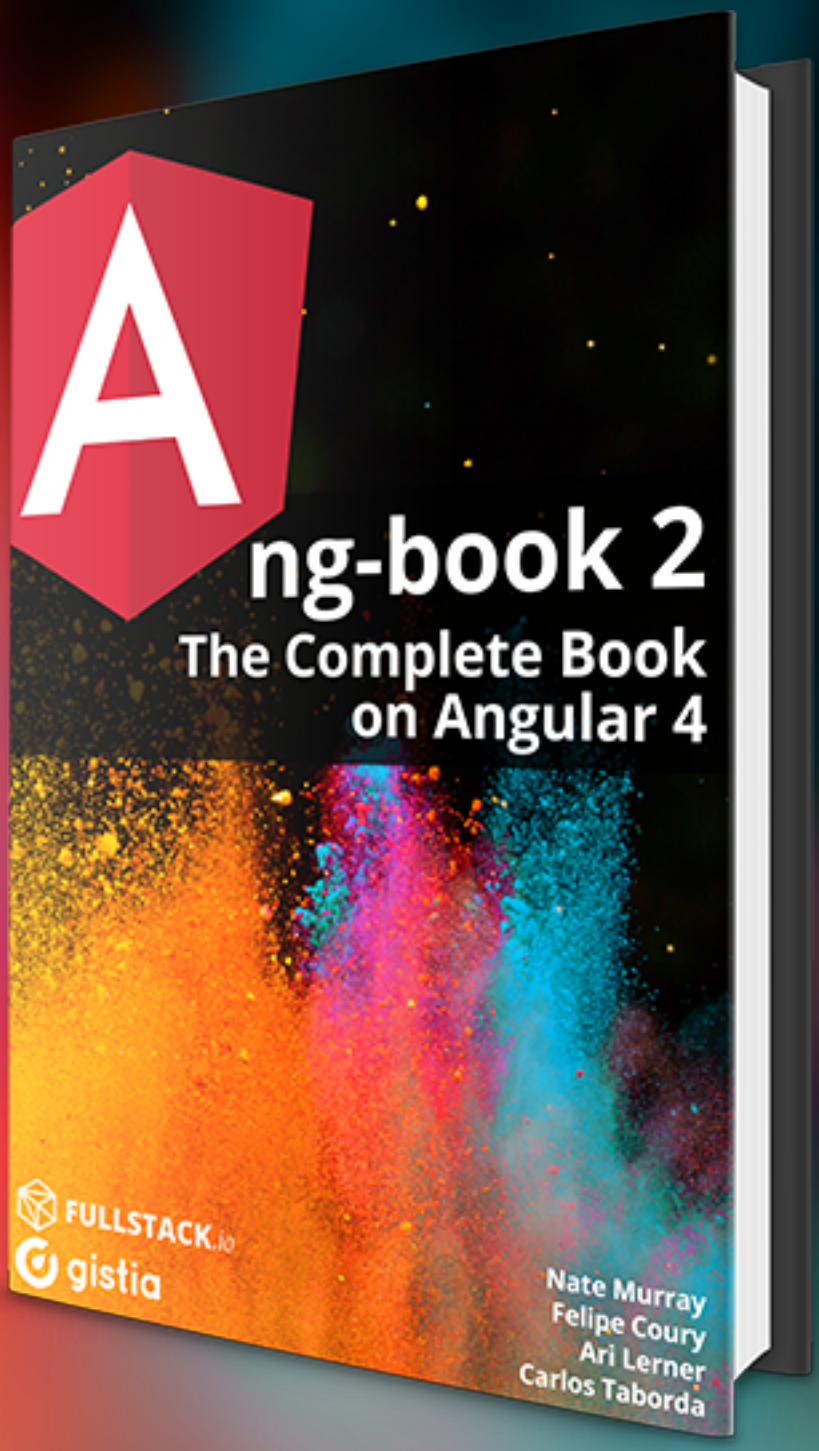
Head over to:

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**