John Niedzwiecki II

Angular 5 Companion Guide

Get up and running with the latest features of Angular 5

Packt

Angular 5 Companion Guide

Get up and running with the latest features of Angular 5

John Niedzwiecki II

BIRMINGHAM - MUMBAI

Angular 5 Companion Guide

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Kunal Chaudhari
Content Development Editor: Onkar Wani
Technical Editor: Murtaza Tinwala
Copy Editor: Shaila Kusanale
Proofreader: Safis Editing
Production Coordinator: Melwyn Dsa

First published: December 2017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78883-856-6

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 3,500 industry professionals
- Learn better with Skill Plans built especially for you
- Get a DRM-free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the Author

John Niedzwiecki II is your friendly neighborhood kilted coder. He is an experienced senior engineer with a passion for creating UI, a love of JavaScripting all the things, and a drive for both teaching and learning. John has a love of Angular, has lead the development of applications across large datasets, visualizing data, and loves trying out experimental ideas. He's a geeky ginger kilted running dad who codes because he loves it. John currently works at TINT as a senior engineer, bringing user and employee generated content to brands to build a trusted voice, powering online campaigns and events, with deep integrations and creating enterprise level analytics to measure their engagement and impact.

He has spoken at a number of conferences, including AngularMix. You can find him out and about in northern Virginia, rambling on his blog, hanging around on Twitter, and if he's lucky, at his happy place of Walt Disney World.

Packt is Searching for Authors Like You

If you're interested in becoming an author for Packt please visit `http://authors.packtpub.com/` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are searching for an author for, or submit your own idea.

Table of Contents

Part 1

What did I miss in Angular 4?

1
Changes in Version 4.x

While this book is primarily about Angular 5, the Angular team has worked hard on version 4 and released some great features throughout the minor releases, which will benefit you with the latest version of Angular. If you're coming from version 2 or want the highlights of everything added during the 4.x releases, you can start here.

Enhanced syntax for *ngIf

One of the first features added in the version 4.0 release was an enhanced syntax for `ngIf`. Two things added to the syntax were if/else syntax and local variable assignment. The first created ability was to use an if/else syntax with a template binding. This is a great addition and saves from writing multiple if statements, thus allowing for a simple programming structure. The second change allows you to assign a local variable. This is useful when unrolling an observable and to allow for simpler names to be used within the HTML:

```
<div *ngIf="userObservable | async; else loading; let user">
  {{ user.name }}<br />
  Email:
  <span *ngIf="user.email"; else unknown>
    {{user.email }}
  <span><br />
  Phone:
  <span *ngIf="user.phone"; else unknown>
    {{user.phone }}
  <span>
</div>

<ng-template #loading>Please wait...</ng-template>
<ng-template #unknown>Unknown</ng-template>
```

Introduction of ng-template

The Angular team introduced the `<ng-template>` element. This change was introduced in order to operate better with web components that may use the standard `<template>` tag by avoiding any conflicts over the use of this tag by Angular. This is a small change, but sees the deprecation of the template tag and attribute, which will result in a deprecation warning while running 4.x in development mode:

```
<ng-template #awesome>
  My <span class="awesome">awesome</span> template content!
</ng-template>
```

Deprecation of emulated /deep/ CSS selector

Angular previously allowed for a shadow-piercing descendant combinator with the CSS selector `/deep/` in order to allow a component to force a style down through a child component tree. It gets applied to both view children and content children. It also has aliases of `>>>` and `::ng-deep`. Due to browsers dropping support for the shadow-piercing descendant combinator, Angular has deprecated all three (`/deep/`, `>>>`, and `::ng-deep`) with the intention to remove. Until it is removed, any uses of deep should use `::ng-deep` for broader compatibility.

New lifecycle events for Guards and Resolver

Angular 4.3 saw the addition of router-level events for both GuardsCheck and Resolver. The newly added events are `ResolveStart`, `ResolveEnd`, `GuardsCheckStart`, and `GuardsCheckEnd`. These events allow you to know when Guards and Resolves start and end for each route navigation. One use for these new events would be metrics, to be able to better know how long guards and resolves take during navigation. A long resolve can result in poor user experience.

New HTTPClient with Interceptors

Angular 4.3 introduced a `new HttpClient` module. The new module is a rewrite of the old HTTP module but added some improvements and missing features that you may have used in Angular 1.x. This was one of the biggest new additions during the version 4 release cycle. The `new HttpClient` is available from `https://angular.io/api/common/http`.

First, the `new HttpClient` automatically maps responses to JSON by default. This keeps you from needing to call `response.json()` on every request. This value is configurable for cases when mapping to JSON may not be appropriate, such as an API with a plain text response:

```
// old way
http.get(url)
  .map(response => response.json() as Items)
  .subscribe( ... );

// new HttpClient, with type parameter of Items
http.get<Items>(url)
  .subscribe( ... );

// text not JSON data
// will return Observable<string>
http.get(urlToText, {responseType: 'text'})
  .subscribe( ... );
```

The second change is the inclusion of the `HttpInterceptor` interface. If you've worked with interceptors in Angular 1.x, you'll be familiar with them. Interceptors allow you to intercept requests and responses to perform additional functionality or modify them globally. You can modify the outgoing request or transform the response event stream.

To create an interceptor, you must declare a class that implements the `HttpInterceptor` interface. The following code creates an authorization interceptor. Its purpose is to add an authorization header to send with every request. We include an authorization service that will return us the value to add. For example, you could be using JWT to manage authentication, and the service would return the token from the `getAuthValue()` call:

```
import { Injectable } from '@angular/core';
import {
  HttpEvent,
  HttpHandler,
  HttpInterceptor,
  HttpRequest
} from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
```

```
import { AuthService } from './auth/auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(
    private auth: AuthService
  ) { }

  /*
   * Intercept function to get request before it is made.
   * Gets request and adds header for authorization.
   * @returns Observable from modified request.
   */
  intercept(request: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    // get request and add header
    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getAuthValue()}`
      }
    });
    return next.handle(request);
  }
}
```

To use your interceptor, it needs to be added in the `providers` in the application's module to the `HTTP_INTERCEPTORS` array:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './../auth/auth.interceptor';

@NgModule({
  bootstrap: [ AppComponent ],
  imports: [ ... ],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    }
  ]
})
export class AppModule { }
```

You can also intercept the response before it is returned. The setup is the same for the interceptor, but we add our logic to the `next.handle` portion of the code before it is run. In the following code, we'll create an interceptor to help us capture several 4xx error codes:

```
import { Injectable } from '@angular/core';
import {
  HttpEvent,
  HttpHandler,
  HttpInterceptor,
  HttpRequest
} from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/do';

export class ErrorInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    return next.handle(req).do((event: HttpEvent<any>) => {
      if (event instanceof HttpResponse) {
        // modify successful response if you want
      }
    }, (err: any) => {
      // handle error states
      if (err instanceof HttpErrorResponse {
        // handle error codes
        if (err.status === 401) {
          // redirect to / show login
        } else if (err.status === 402) {
          // redirect to payment page
        }
      }
    });
  }
}
```

The new `HttpClient` also allows you to track the progress of uploads and downloads for when you're working with transfers of large amounts of data. You can configure an instance of `HttpRequest` with the `reportProgress` option to enable progress events:

```
const uploadRequest = new HttpRequest(
  'POST',
  '/upload/file',
  file,
  {
```

```
      reportProgress: true
    }
  );

  http.request(uploadRequest).subscribe(event => {
    // Get upload progress event from raw event stream
    if (event.type === HttpEventType.UploadProgress) {
      // Compute percentage
      const percentDone = Math.round(100 * event.loaded / event.total);
      console.log(`File is ${percentDone}% uploaded.`);
    } else if (event instanceof HttpResponse) {
      console.log('File is done uploading!');
    }
  });
```

Separation of the Animations package

With the 4.0 release of Angular, animations were separated into their own package. This was done for several reasons. This means that if you don't use animations, the code for animations will not be a part of your production bundles, unlike prior to version 4. With the view engine change, the animation runtime was no longer tree shakeable, so it made sense to separate it. In addition, it makes it easier to find documentation and autocompletion. To use animations, you can import them into your main `NgModule` yourself, or they will be imported automatically if you use libraries such as `Material`, where animations are used. You will still need to first install the package via NPM in both cases.

In addition to their own module, the minor release of Angular 4 has seen a large number of improvements and additions to animations. Let's take a look at some of these changes.

The first feature added is the ability to configure options and set input variables within animations. This allows you to configure and override options for the various step-based animation methods. The two options are `options.delay` and `options.params`. The first option, `options.delay`, does exactly what it says-it allows you to delay the start of the animation. This does not support percentages or negative values, though negative values are planned for a future release. The second option, `options.params`, is for adding input parameters. These inputs allow you to pass in data to be used for style properties and timing values within an animation sequence. Any key/value pair provided through `options.params` can be used with `{{ binding }}` to access the value:

```
transition('* => *', [
  style({ opacity: 0 }),
  animate("{{ duration }}",
    style({ opacity: "{{ endOpacity }}" }),
```

```
], {
  duration: "5s",
  endOpacity: "1"
})
```

Both of these options can be set by a new method parameter (as seen in the preceding code example) or can be passed into an animation binding value. It should be noted that these values do not update once an animation starts. It will evaluate the values at the start to build the animation and then, it will not change while the animation is executing.

The second feature added provides the ability to create a reusable animation. The `animation()` helper method lets you define and package animations that can be used elsewhere, while supporting the previously mentioned inputs. The following code will create a reusable animation that also accepts inputs for customization of the animation:

```
import {animation, style, animate} from "@angular/animations";

export var fadeAnimation = animation([
  style({ opacity: "{{ startOpacity }}" }),
  animate("{{ duration }}", style({ opacity: "{{ endOpacity }}" }))
], { startOpacity: "0", endOpacity: "1", time: "1s" })
```

You've now created a `fadeAnimation` variable that can be used throughout our application, fully definable by input parameters. We've also defined the default values for the inputs. To use the animation, you invoke it with the `useAnimation()` function and specify the input parameters:

```
import {useAnimation, transition} from "@angular/animations";
import {fadeAnimation} from "./animations";

transition('* => *', [
  useAnimation(fadeAnimation, {
    startOpacity: 0,
    endOpacity: 1,
    time: '3s'
  })
]);
```

The third feature added gives the ability to create really powerful animations. The `query()` function allows you to select child elements and animate them separately from each other in parallel within the element with the animation trigger. This allows you to create multielement animations triggered together:

```
animations: [
  trigger('groupAnimation', [
    transition('* => *', group([
      // start elements hidden
      query('*', style({ opacity: 0 })),

      // use fadeAnimation to fade in all divs with groupHeader class
      query('div.groupHeader', [
        useAnimation(fadeAnimation, {
          endOpacity: 1,
          time: '2s'
        })
      ]),

   // fade in and move divs with groupItem class or id of importantItem
      query('div.groupItem, #importantItem', [
        animate('2s', style({ opacity: 1, transform: 'translateX(50px)'}))
      ]);
     ])
  ])
];
```

The provided code does several things. The key part is the query selector. It allows for several values to be used. The first values you can use are regular CSS selectors that can return one or more items matching the selector string, as seen with `div.groupHeader`. Additionally, query can use `query(':enter')` for new nodes and `query(':leave')` for nodes marked to be removed, within the container. Additionally, query finds elements with animation triggers through `query('@triggerName')` or all items with animation triggers with `query('@*')`. Query can find all elements that are currently animating using `query(':animating')`. Finally, the container element itself can be selected with `query(':self')`, which is useful if you want the container itself to have animation in addition to child elements selected through queries.

The query function will throw an error if it doesn't find any elements, by default. You can override this with the query option of `{ optional: true }`. In addition, you can limit the number of elements to be selected by setting the `limit` option value.

The query function is powerful for allowing us to do multiple animations in one. It gets even more powerful when we pair it with the new `stagger()` animation helper function. Stagger allows you to stagger animations for elements, spacing them out so that the animations start with a gap of time in between. Let's create a staggered animation for a list with an `ngFor` that has a container wrapped around it:

```
<div [@listAnimation]="items.length">
  <div *ngFor="let item of items">
    {{ item }}
  </div>
</div>
```

You can query those inner elements of our `div`s added with the `ngFor` and animate them as they are added, but stagger them so that they start their animation with a gap in between:

```
trigger('listAnimation', [
  transition('* => *', [
    // starts elements off to side
    query(':enter', style({ transform: 'translateX(-100%)' })),
    // starts animations with 100ms in between
    query(':enter', stagger('100ms', [
      animate('1s', style({ transform: 'translateX(0)' }))
    ])
  ])
])
```

Queried elements trigger their own animation.

The next feature allows you to take animations one step further and apply them to routes. We can have an animation activated on the route change and animate the components that come and leave the router-outlet.

To begin, we need to wrap our outlet in an element to trigger our animation, as we cannot add the trigger directly to the `router-outlet` because of how it works. We also need to get a hold of the details of the route switching in and out through a function and a route local variable:

```
<div [@routeAnimation]="getRouteAnimation(route)">
  <router-outlet #route="outlet"></router-outlet>
</div>
```

Next, our component must define the `getRouteAnimation` function. This will define the state value to be used by the `routeAnimation` animation. The animation itself will fade in the new route and fade out the old. By using a group, we can keep the animations simultaneous:

```
import { Component} from '@angular/core';
import {
  animate,
  query,
  state,
  style,
  transition,
  trigger
} from '@angular/animations';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  animations: [
    trigger('routeAnimation', [
      transition('* <=> *', [
        group([
          // fade in new route
          query(':enter', [
            style({ opacity: 0 }),
            animate('500ms', style({ opacity: 1 }))
          ]),
          // fade fade out old route
          query(':leave', [
            animate('500ms', style({ opacity: 0 }))
          ])
        ])
      ])
    ])
  ]
})
export class AppComponent {
  getRouteAnimation(outlet) {
    return outlet.activatedRouteData.animation
  }
}
```

You can see that the `getRouteAnimation` function takes in the outlet and returns a string value for the state of the animation. This will be determined by custom data of the current active route, which we must then define in our routing:

```
const routes: Routes = [
  {
    path: 'home',
    component: HomeComponent,
    data: {
      animation: 'home'
    }
  },
  {
    path: 'list',
    component: ListComponent,
    data: {
      animation: 'list'
    }
  }
];
```

In providing these values, you can use them to define custom animations between two particular routes, as opposed to our example that does one animation between all routes:

```
// home to list
transition('home => list', [
    // animation 1
]),
// list back to home
transition('list => home', [
    // animation 2
])
```

The final feature gives you the power to programmatically build animations through the `AnimationBuilder` service. This service allows you to create and define animations within a component. However, it requires you to define more aspects of the controls for the animation. Normally, triggers do a lot of the work for you to track the state of animations, which you will need to define yourself if using the `AnimationBuilder`.

To build an animation in a component, there are several key pieces. First, you build the animation with `factory = this.animationBuilder.build( ... )`. Second, you create an `AnimationPlayer` from your builder with `player = factory.create(someElement)`. Third, you tell the player to play with `player.play()`. Let's see a full example of a component to create a loading bar that animates to the width provided by an input:

```
import {
  AnimationBuilder,
  AnimationPlayer
} from '@angular/animations';
@Component({
  selector: 'loading-bar',
  template: `<div class="loading" #loading></div>`
})
class LoadingBarComponent {
  @ViewChild('loading')
  public loading;

  @Input('percentage')
  set percentage(per: number) {
    this._percentage = p;
    this.startLoading();
  }

  get percentage() {
    return this._percentage;
  }

  public player: AnimationPlayer;

  private _percentage: number = 0;

  constructor(
    private animationBuilder: AnimationBuilder
  ) { }
  startLoading() {
    // stop animation if ongoing
    if(this.player) {
      this.player.destroy();
    }

    // build animation
    const loadingAnimation = this.animationBuilder.build([
      style({ width: 0 }),
      animate('1s', style({ width: `${this._percentage}%` }))
    ]);
```

```
      // create player for animation
      this.player = loadingAnimation(loading.nativeElement);
      // play animation
      this.player.play();
    }
  }
```

To use this component with the built animation, you use it like any other component and provide it the input value of the percentage:

```
<loading-bar [percentage]="percentage"></loading-bar>
```

Multiple exportAs names

One of the latest added features during the Angular 4.4 release is the ability to specify multiple names in the `exportAs` attribute of a directive. This feature is useful for backward compatibility. It allows you to change the name of a current directive, but keep the old names still available:

```
@Directive({
  selector: '[my-super]',
  exportAs: 'super, superDooper'
})
export class SuperDirective { }

<!-- used as -->
<div my-super #foo="super"></div>
<!-- or -->
<div my-super #foo="superDooper"></div>
```

Source maps

The Angular team has also made additions to the tools to make life easier for developers. One of the features added was to create source maps for templates. This addition adds value when debugging during development. This was added to the template compiler, providing more contextual information when debugging in the browser, or from crash reports, than previously available.

Angular Universal

Angular Universal is the project created to handle running Angular on the server and provide server-side rendering. With the release of version 4.0, this project was adopted by the Angular core team and brought into the official code base. This brought the code into platform-server, making it readily available and integrated. The topic of server-side rendering is too large to be covered over here, but it's important to know that its inclusion into the core code base makes it easier to be used and integrated into your project.

Angular CLI

With the 4.0 release, much like Angular Universal, the **Angular CLI** was officially brought into the Angular project. The CLI reached its 1.0.0 release alongside the Angular version 4.0 release and has continued to progress. The improvements include the AoT compilation that will be discussed under the *New view Engine* section, upgrades to webpack that bundles the code, improved error messages, and all the scaffolding it can create for you. If you haven't been using the CLI for your Angular project, you should start using it. Get started with it and create your first new project using the following code:

```
ng new PROJECT-NAME
```

Performance increases to Angular

The Angular team is always working to increase the performance of Angular. In Angular 4, they've made advancements to the compilation process to make your bundled code smaller and perform better. Here are two of these updates, providing you increased performance with no or little work to you.

New view engine

The Angular team put a lot of work into changing things under the hood for **Ahead of Time** (**AOT**) compilation. They updated the view engine to reduce the size of the generated code after compilation. In their tests, the team found significantly smaller code sizes, 3x or 1.7x gzip, as well as speed increases in pure change detection (1.5x) and tree updated (1.34x) (source: `https://docs.google.com/document/d/195L4WaDSoI_kkW094LlShH6gT3B7K1GZpSBnnLkQR-g/preview`). This results in smaller bundle sizes for the application.

In addition, the Angular team improved performance by lessening the amount of work done by browser garbage collectors. The best part is that this comes with no work to you just using the Angular-CLI.

Removing template whitespace

One of the latest features added in the 4.4 release is an option to remove whitespace and blank text notes from compiled templates. This can result in saving of lot of size as a simple new line of white space can result in a full line of JavaScript in the compiled template. The trade-off in removing whitespace to save in size is that it can lead to layout changes and cause issues in the browser, due to the nature of how HTML and whitespace is interpreted by the browser. For this reason, the flag is optional and opt-in.

It can be enabled globally during Bootstrapping:

```
platformBrowserDynamic().bootstrapModule(AppModule, {
  preserveWhitespaces: false
});
```

Additionally, you can enable it for a single component:

```
@Component({
  selector: 'who-needs-whitespace',
  templateUrl: './whitespace.component.html',
  preserveWhitespaces: false
})
export class WhitespaceComponent { }
```

2
Keeping up with TypeScript

One of the great features of Angular is working in TypeScript. The benefits of TypeScript are numerous. Throughout the development of Angular versions, the Angular team has been keeping up with the latest updates and benefits from TypeScript. With the 4.0 release, the Angular team updated for support of TypeScript 2.1. During the 4.1 release, the team updated with support for TypeScript versions 2.2 and 2.3. This chapter will look at some of the additions to TypeScript that can benefit the code you write in your Angular application.

StrictNullChecks

With the 4.0 release of Angular, in addition to updating to TypeScript 2.1, the team also updated the Angular code base to be compliant with TypeScript's `StrictNullChecks setting`. This switches the TypeScript compiler to a strict null checking mode, where null and undefined values are not in the domain of every data type. The goal is to write safer code. The TypeScript compiler will catch code trying to use null values that are not explicitly stated to be nullable and catch the "possibly null" case. For example, the compiler will show an error if you try to access a property of an object that may be undefined, such as an optional parameter. This update allows you to enable to this flag (`–strictNullChecks`) in your own application without the Angular code base throwing errors.

Support for mixin classes

TypeScript 2.2 added support for the ECMAScript 2015 mixin class pattern and rules for using them. Mixins provides with the templates for classes and provides an abstract superclass that can be parameterized. It sits in the area between subclasses and superclasses.

In order to create a mixin class in TypeScript, we need to define it to accept a superclass, which it will extend to create a subclass:

```
let CustomMixin = (superclass) => class extends superclass {
  foo() {
    console.log('foo called in CustomMixin');
  }
```

Next, we create a class that uses the mixin within an `extends`:

```
class CustomClass extends CustomMixin(MyBaseClass) {
  /* ... */
}
```

Finally, if we use that class, we get the `foo()` defined in the mixin:

```
let cc = new CustomClass();
cc.foo(); // logs 'foo called in CustomMixin'
```

Better checking for null/undefined in operands of expressions

One of the benefits of using TypeScript is catching errors early. TypeScript 2.2 improved the catching of `null` and `undefined` in operands of expressions, catching more errors for you during compilation. This check occurs for many of the binary and unary operators if any of the operands are nullable.

Generators and iteration for ES5/ES3

Iterators are objects useful for iteration over data structures, such as elements of an Array or the keys of a Map. An Iterator needs to expose three methods: next, return, and throw in order to implement the interface properly:

```
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
```

Generators are functions used to compute results using an Iterator and the yield keyword. TypeScript 2.3 added full support for generators and iterators for targeting ES3 and ES5 with the use of the `--downlevelIteration` flag. When the flag is used, the compiler uses a new type check that will adjust the calls on an iterated object with `[Symbol.iterator]()` or create a synthetic array iterator if it cannot. Enabling this flag will also allow `for..of`, array destructuring, and spread elements to function in ES5/ES3.

Async iteration

TypeScript 2.3 also adds support for **async iterators** and **generators** to align with the current TC39 proposal. Part of what makes TypeScript great is adding support for the latest in JavaScript. An AsyncIterator is similar to an Iterator; however, the next, return, and throw methods return a Promise instead of an actual value result:

```
interface AsyncIterator<T> {
  next(value?: any): Promise<IteratorResult<T>>;
  return?(value?: any): Promise<IteratorResult<T>>;
  throw?(e?: any): Promise<IteratorResult<T>>;
}
```

Master --strict option

TypeScript offers a lot of checks to help catch more errors, making your code stronger. As the TypeScript team add more options, they usually leave them off by default to avoid breaking the existing projects. If you want to use all of these stricter options, you would need to manually opt-in for each. TypeScript 2.3 added an easier way, the `--strict` option. This will provide you with the maximum level of type safety and checking. As future safety checking options are added, the `--strict` option may be updated to then include them as well. Using the `--strict` option is the same as enabling all the following options:

```
--strictNullChecks
--noImplicitAny
--noImplicitThis
--alwaysStrict
```

Check for errors in .js files

TypeScript can always work with JavaScript files. However, it doesn't return any errors in the `.js` files, by default. TypeScript 2.3 added the ability to add type-checking errors to `.js` files with the `--checkJsoption`. If you want to skip some files with the option enabled, you can add `// @ts-nocheck` or ignore individual errors by adding `// @ts-ignore` on the previous line. If you do not want to enable the JavaScript checking everywhere, you can instead add checking to individual files by adding `// @ts-check`.

3
Migrating to Version 4

Migrating to Angular 4 was intended to be simple. The goal that was set out from the beginning of version 2 and semantic versioning was to keep version-to-version upgrades simple. Migrating to version 4 of Angular kept true to this vision, while providing a lot of extra great features.

To migrate to version 4, the first update you need to make is to update your dependencies. This update is simple; in doing this, you will need to add the Animations module, if you are already using them in your application or if you now want to with everything you read in Chapter 1.

To install, simply run this:

```
npm install @angular/animations
```

To use animations, you need to update or add the new imports that will use this new module:

```
import {
  animate,
  query,
  state,
  style,
  transition,
  trigger
} from '@angular/animations';
```

The next mostly likely update you'll want to make is to update your templates. As mentioned in Chapter 1, the Angular team introduced a new ng-template element. The purpose of this element is to replace the template element for Angular use. Due to this change, the use of template is deprecated; therefore, you should update all of your instances from template to ng-template:

```
<!-- old way -->
<template #loading>Old busted loading...</template>

<!-- new way -->
<ng-template #loading>New shiny loading...</ng-template>
```

The last update may not affect you, but if it does, it is a very small change and deals with the lifecycle events. For events such as `OnInit` and `OnDestroy`, you now need to use `implements`, not `extends`:

```
// if you have
MyComponent extends OnInit

// change to
MyComponent implements OnInit
```

Part 2

What's new in Angular 5?

4

New Features in Version 5

The version 5 release of Angular had the theme of "easier, smaller, faster." Many of the feature changes and additions were selected to help accomplish this mission. This section of the book will cover the features and changes that have been made in version 5 of Angular. The goal is to give you the information you need to hit the ground running with the new features.

Object map for HttpClient headers and params

One of the biggest features introduced during the Angular 4 release cycle was `HttpClient`. Chapter 1 discusses the abilities of the new `HttpClient` and `Interceptors`. One of the pain points with the `HttpClient` was that to define headers for a request would require creating an `HttpHeaders` object first, which could then be passed as part of the request:

```
const headers = new HttpHeaders({
  'Custom-Header': 'header value',
});
http.get('/url', { headers })
  .subscribe( ... );
```

The creation of an additional object is no longer necessary and can't be streamlined by creating a properly formatted object map. The end result is a much simpler request being made:

```
http.get('/url', { headers: { 'Custom-Header': 'header value' } })
  .subscribe( ... );
```

NgForm adds updateOn and ngFormOptions

Angular provides plenty of features to assist in creating powerful forms. The next feature adds a new `Input` property called—`ngFormOptions`. This allows you to pass in options, in this case, the default `updateOn` value. You can specify either `change`, `blur`, or `submit`, which will then be the default value for all children of the form, unless the form inputs explicitly specify their own `updateOn` value:

```
<form [ngFormOptions]="{ updateOn: submit }">
  <input name="name"  ngModel>  <!-- will update on submit -->
  <input name="email" ngModel
         type="email"
         [ngModelOptions]="{updateOn: 'blur'}"> <!-- will update on blur -
->
</form>
```

This also applies to working with `FormControl`, `FormGroup`, or `FormArray`. You can still set the `updateOn` value on the `FormGroup` and `FormControl` separately:

```
this.login = new FormGroup({
  name: new FormControl(),
  email: new FormControl(null, { updateOn: 'blur' }) // will update on blur
}, { updateOn: 'submit' }); // default will update on submit
```

Animations

The version 5 release of Angular has additional improvements to animations. One improvement provides additional error reporting. Animations will throw errors when invalid CSS properties are detected. This makes it easier to find problems if your animations aren't working as expected, as it will tell you whether a provided property is not supported. The Angular team has also worked to reduce the size of the bundle. They've accomplished this by removing `AST` classes. Animation has also added the ability for queries to handle negative numbers. This will match elements from the end instead of from the beginning. Allowing for negative numbers in queries aligns similarly to how many array functions work.

Router events

The next features provides additional router events. The changes add the ability to track the activation of individual routes as well as provide new events. You can now leverage the `ActivationStart` and `ActivationEnd`, `GuardsCheckStart` and `GuardsCheckEnd`, and `ResolveStart` and `ResolveEnd` events:

```
import {
  Router,
  ActivationStart,
  ActivationEnd
} from '@angular/router';
import 'rxjs/add/operator/filter';

@Component({
  selector: 'app-root',
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {
  constructor(
    private router: Router
  ) {
    router.events
      .filter(e =>
        e instanceof ActivationStart || e instanceof ActivationEnd)
      .subscribe(e => console.log( e.toString() ));
  }
}
```

If you want to track specific URLs, you can leverage the `RouterEvent` and then check the ID and URL values:

```
import {
  Router,
  RouterEvent
} from '@angular/router';

import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
  selector: 'app-root',
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {
  constructor(
    private router: Router
```

```
  ) {
    router.events
      .filter(e => e instanceof RouterEvent)
      .filter(e => e.url == '/tracked-route') // filter desired route
      .distinctUntilChanged((e1, e2) => e1.id == e2.id && e1.url == e2.url
)
        .subscribe(e => {
          console.log(e.id + ': ' e.url);
        });
  }
}
```

Pipes

The 5.0 release of Angular introduces a large number of changes in pipes. Due to multiple bugs and browser inconsistencies, the Angular team chose to drop the `intl` API for data exported from the **Unicode Common Locale Data Repository** (**CLDR**). This causes breaking changes for any i18n pipes, including date, number, currency, and percent.

The i18n pipes, by default, only contain local data for the language `en-US`. You can import new locale data for language and then set the value of `LOCALE_ID` to another locale. With this, you no longer need to use the `intl` poly fill. In addition, the old pipes do still exist, but the names are changed and found within `DeprecatedI18NPipesModule`. One important factor is that it must be imported after `CommonModule`:

```
import { NgModule } from '@angular/core';
import { CommonModule, DeprecatedI18NPipesModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule,
    DeprecatedI18NPipesModule
  ]
})
export class AppModule { }
```

The first pipe we'll look into the changes made to is the **Date pipe**. The predefined formats now use the patterns given by CLDR. This will mean that there are some changes to the formats and to the results of the predefined formats. The timezone format z will now fall back to `0` and display GMT+1 instead of the complete timezone name (such as Eastern Standard Time).

There are now some new predefined formats. These include `long`, `full`, `longTime`, and `fullTime`:

- `'long'` is shorthand for 'MMMM d, y, h:mm:ss a z', which will create January 31, 2018 at 9:45:02 AM GMT+1
- `'full'` is shorthand for 'EEEE, MMMM d, y, h:mm:ss a zzz', which will create Wednesday, January 31, 2018 at 9:45:02 AM GMT+01:00
- `'longTime'` is shorthand for 'h:mm:ss a z', which will create 9:45:02 am GMT+1
- `'fullTime'` is shorthand for 'h:mm:ss a zzzz', which will create 9:45:02 am GMT+01:00

There are also additional form values available to be used, thanks to the inclusion of CDLR. These are yyy (numeric, 3 digits + zero padded), standalone months (use L up to LLLLL), week of year (use w or ww), week of month (use W), fractional seconds (use S, SS, or SSS), extra day periods, and short non-localized timezones.

The **Currency pipe** has some small changes as well. The default value for `symbolDisplay` is now a symbol instead of code. This means you'll see $7.99 instead of USD7.99. This is the more natural display value for the default. The second parameter is no longer boolean with the Currency pipe. Instead, you can choose between `code`, `symbol`, or `symbol-narrow`. You can provide a boolean; however, this is now deprecated and will show a warning message in the console.

The **Percent pipe** has some default changes. If you don't specify the number of digits to round to, the local format will be used. This usually rounds numbers to 0 digits instead of the previous implementation, which would not round. For example, `{{ 2.71828 | percent }}` will output `271%` for local `en-US`. Previously, it would have shown `271.828`.

Watch mode

Compilation with `ngc` now has a watch mode. This makes it easier to use **Ahead of Time** (**AoT**) compilation, hence saving your headaches that you may find in AoT that are missed in **just in time** (**JIT**) compilation. The Angular team also hooked into transforms with TypeScript 2.3 to speed up the compilation pipeline. They have worked to try and make AoT compilation fast enough to be used all the time for development purposes. The new build does incremental AoT and saves 95% of the build time on the Angular team's machines. This will become the default in the future, so check out this feature in this release.

To run, you need to run a serve with the AoT flag turned on:

```
ng serve  --aot
```

Lazy loading

The Angular team has added additional functionality in version 5 to help the migration from AngularJS. The team has added the ability to lazy load an Angular Module into an AngularJS application. This helps you upgrade your application in place without needing to load all the Angular modules immediately when the application bootstraps.

Material Design

One key addition to the Material Design components in Angular 5 is the compatibility with Angular Universal and server-side rendering. This makes the components more accessible for production deployments requiring server-side rendering.

Service worker package

The Angular team has been making a push toward **Progressive Web Apps** (**PWA**). They want to make it easy to work out of the box. To support this, the team has introduced the `@angular/service-worker` package. This is based on a version from Angular **mobile-toolkit repo**, but rewritten for greater application support.

The package includes several entry points:

- The first is `@angular/service-worker`, for use within client applications to communicate with the service worker
- The second library is `@angular/service-worker/gen`, for generating the `ngsw.json` file from blog-based service worker config files
- The third is `@angular/service-worker/ngsw-worker.js`, which is the bundled service worker script
- Fourth is `@angular/service-worker/ngsw-cli.js`, a CLI tool for generating `ngsw.json` files from blog-based service worker config files

Together, this gives you the pieces to get started with implementing service workers into your Angular application. The next release of Angular CLI, version 1.6, will include the features needed to easily implement the service worker into your project to make an Angular PWA.

You can start playing with service workers now, but you'll have to do some manual set up to get it up and running. First, you need to install the Angular Service worker:

```
npm install @angular/service-worker --save
```

Next, you need to register the service worker in your main app module:

```
@NgModule({
  imports: [
    // add following to imports
    environment.production ? ServiceWorkerModule.register('/ngsw-
worker.js') : []
  ],
  // other configurations
})
export class AppModule { }
```

The service worker needs to be a manifest (control) file, which is based on a configuration file. In the upcoming version 1.6 release of the CLI, the current default created version of the `ngsw-config.json` is shown here:

```
{
  "index": "/index.html",
  "assetGroups": [{
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
        "/favicon.ico",
        "/index.html"
      ],
      "versionedFiles": [
        "/*.bundle.css",
        "/*.bundle.js",
        "/*.chunk.js"
      ]
    }
  }, {
    "name": "assets",
    "installMode": "lazy",
    "updateMode": "prefetch",
```

```
      "resources": {
        "files": [
          "/assets/**"
        ]
      }
    }]
}
```

To create the control file, you can use a CLI included as part of the package. The syntax for the command is `ngsw-config outputFolder sourceConfigFile baseHref`. The `outputFolder` is the destination to create the file. The `sourceConfigFile` specifies the location of the `ngsw-config.json` file. Finally, you can also specify a base HREF for your application, which is "/" by default. You'll only need this if you're already setting it as part of your Angular application compilation as well. An example of the command to run, using the `dist` folder and the JSON from the `src` directory, is shown here:

```
node_modules/.bin/ngsw-config dist ./src/ngsw-config.json
```

Lastly, to get everything up and running, you first need to do a production build, run the preceding command to create the config, and finally, copy the service worker code to your list folder as well. Finally, you need to run from that `dist` folder. The full set of commands you need to run would be as follows:

```
ng build --prod
node_modules/.bin/ngsw-config dist ./src/ngsw-config.json
cp node_modules/@angular/service-worker/ngsw-worker.js dist/
http-server dist -p 8080
```

Unfortunately, you can't currently use `ng serve` to run and test a PWA app. This should all become easier once we have full Angular CLI integration. From here, the work of creating a great Progressive Web App starts. You have two locations in the `ngsw-config.json` for caching items, `assetGroups` for resources related to the app version that may change, and `dataGroups` for resources independent from the app version, which is where you can cache API responses. For example, if we want to cache fonts, we can add a group called `fonts` and add any resources we want to cache in there related to fonts:

```
"assetGroups": [
  ...
  {
    "name": "fonts",
    "resources": {
      "urls": [
        "https://fonts.googleapis.com/**"
      ]
    }
```

```
    }
  ]
```

To cache API responses, we set up groups in the `dataGroups` section. Here, you need to specify the caching strategy. You can choose freshness for a network-first strategy or performance for a cache-first strategy. The following example will specify one API for each strategy:

```
"dataGroups": [
  {
    "name": "api-freshness",
    "urls": [
      "/feed"
    ],
    "cacheConfig": {
      "maxSize": 100,
      "maxAge": "2d",
      "timeout": "1m",
      "strategy": "freshness"
    }
  },
  {
    "name": "api-performance",
    "urls": [
      "/profile"
    ],
    "cacheConfig": {
      "maxSize": 100,
      "maxAge": "2d",
      "timeout": "1m",
      "strategy": "performance"
    }
  }
]
```

This will get you up and running with service workers. As you can see, there is a little bit to set up, but the next version of the CLI will make this easy out of the box.

Strict Injectable checking

The Angular team has added an additional compiler option. The `strictInjectionParameters` option will report errors for parameters of an Injectable that cannot be determined. This will default to false, where it will simply return a warning. This is planned to be switched to a true default value with Angular version 6.0.

Performance improvement

The Angular team is always looking for performance improvements. You can now use the native `addEventListener` for faster rendering. This allows angular to bypass Zone's `addEventListener`, because Angular can make assumptions about its event handlers, but not for zones. In addition, the team has made it possible to bypass zones entirely if you have an application that is performance focused. You can bypass it with `'noop'` as your `ngZone`:

```
platformBrowserDynamic().bootstrapModule(AppModule, {ngZone: 'noop'}).then(
ref => {} )
```

As mentioned when discussing watch mode, the team has also sped up the compilation process significantly by reusing the TypeScript typecheck for template typechecking. With this, they've added a new option, `fullTemplateTypeCheck`. This adds extra checks in templates, including checking expressions inside templatized content, checking the arguments of calls to the transform function of pipes, and checking references to directives that were exposed as variables via `exportAs`.

5
New Features in Typescript 2.4

String enums

TypeScript 2.4 has added support for string enums. Enums are useful for mapping out allowable values. TypeScript has added the ability for enums to contain string values for initializing:

```
enum Hand {
  Rock: "ROCK",
  Paper: "PAPER",
  Scissors: "SCISSORS"
```

It should be noted that string-initialized enums, such as the preceding one, cannot be reversed-mapped. This means you cannot use `Hand["Scissors"]` to get the string `Scissors` as the original enum member name.

Improved inference for generics

TypeScript 2.4 introduced changes around how generics are inferred. Inferences can now be used as a return type of a call. This is designed to help improve experience and catch errors. The following code shows how it can be used and catch an error on the return value:

```
let result = Promise<string> = new Promise(resolve => {
  resolve(20);
```

You can see that `20` is a number being resolved for the `Promise`, not a string as defined, so this code will throw an error.

TypeScript now provides better checks for type parameter inference from contextual types. The problem is that values on the right will not always gain types from the left.

```
let ct: <T>(x: T) => T = y => y;
```

This code will leave `y` previously with a type of any. Now, with TypeScript 2.4, the right implicitly gains type parameters. This will make the code more type safe.

In addition, TypeScript has added stricter checking for generic functions. It will try to unify type parameters between two single-signature types. This will give stricter checks and may catch some more bugs:

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];

function f(a: A, b: B) {
    a = b;  // Error
    b = a;  // Ok
}
```

Weak type detection

TypeScript 2.4 adds *weak types*, a new concept to TypeScript. If a type has all optional properties, then it is considered to be *weak*:

```
interface WeakSettings {
  maxValue?: number,
  name?: string;
  options?: string[]
```

None of the properties are required, making `WeakSettings` as *weak*. TypeScript 2.4 will now show an error if there is no overlap with the optional properties:

```
let settings = {
  minValue: 5,
  unchecked: true
}

function setConfig(settings: WeakSettings) {
  // ...
}
setConfig(setting); // error
```

6
Migrating to Version 5

The goal has always been to make the migration between versions simple. This holds true for the latest release, with the biggest change coming in pipes. You can choose to keep using the old pipes with `DeprecatedI18NPipesModule`; however, the names have changed. The better choice is to upgrade the pipes, making any changes needed to keep the output of the pipes to match what you expect (such as with date pipes).

With Angular 4.3, the team introduced the new `HttpClient`. With version 5, the old HTTP is deprecated, so you'll want to migrate your code to the new client, as deprecation means it is marked for deletion in a future major version. You should move from `@angular/http` to `@angular/common/http`. For more information on the new `HttpClient`, refer to Chapter 1.

Version 4 replaced the `<template>` tag with `<ng-template>` and deprecated its use. The compiler option of `enableLegacyTemplate` is available, but it's disabled by default. Both `enableLegacyTemplate` and `<template>` are scheduled to be removed in version 6, so if you are still using them, you should plan to move away at this time.

There is a sizable list of things deprecated since version 4 that have now been removed. This will list some of the most common or likely ones, but you should check the full release notes if you are using any deprecated features. You should also move away from anything deprecated, as they will eventually make their way to this list. `NgFor` was removed, and you should use `NgForOf` instead. This does not impact `*ngFor` used in templates. `OpaqueToken` was removed with version 5, and you should change to using `InjectionToken`. The `RouterOutlet` has dropped some properties from use, including `locationInjector` and `locationFactoryResolver`. The last we'll mention here is that the `TrackbyFn` was removed, and you can simply use `TrackByFunction` instead.

Part 3

What to expect in Angular 6

7
Vision for Version 6

With the release of version 5, our attention turns towards the future of Angular. We currently do not know a lot of what exactly will be a part of this release, but we do have a good view into the future of Angular. At *AngularMix* in Orlando, Florida, the Angular team announced *Angular Labs*, the home of experimental features for Angular. These may not be a part of the next major release, but they do offer a view into the future of Angular as a whole.

Component Dev Kit

The **Component Dev Kit** (**CDK**) is a current initiative that is now a part of Angular Labs. As a part of the Angular Material team's work, they have been extracting some of the core pieces into common problems. These commonalities have been exposed as the CDK. This allows you to write your own components or UI component library leveraging the pieces of the CDK that have solved reusable pieces. The CDK includes everything from handling overlays to trapping user focus, and handling key navigation. They are continuously adding solutions to more problems at the high level, such as dialogs and generic helpers such as drag and drop.

Schematics

The Angular CLI's current initiative for custom schematics is also now a part of Angular Labs. Schematics are your own custom blueprints for generation from the CLI. This means that you can write your own custom generators. Instead of calls simply like `ng generate service DataService`, you will be able to make calls such as `ng generate ngrxService NgrxDataService`, which describes your own custom element to create.

ABC

The last of the current initiatives to be added to Angular Labs is **ABC**: **Angular** + **Bazel** + **Closure**. This is Google's effort to use **Bazel** + **Closure** to create lightning fast build tools. The goal is to have an **AoT** build that is as fast as the **Just-in-Time** development build. This is based off of Google's use of Bazel and Closure to build from their mono-repo. They are working to use closure optimizers to save more space as well as Bazel to build as a tool that can be used for the frontend and backend. This work is early, but can be played with now.

Angular Elements

Angular Elements is a new way to package Angular components. This change will allow components to be packaged on their own, with the ability to bootstrap themselves and be run outside of an Angular application. This "hosts" the Angular component inside of its own custom element. This will help expand the reach of your Angular code by allowing you to write standalone components that can be embedded in other applications, while still writing them in Angular. It creates the bridge between the DOM API and the Angular component. This is still at a very early stage, but promising.

As you can see, there are a lot of exciting large features coming to Angular. The team will also still keep pushing to improve the core, making it faster and easier.

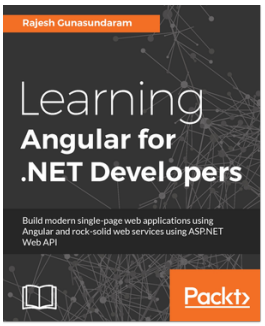
Other Books You May Enjoy

If you enjoyed this book, you may be interested in other books by Packt:

Expert Angular 4
Mathieu Nayrolles, Rajesh Gunasundaram, Sridhar Rao

ISBN: 978-1-78588-023-0

- Implement asynchronous programming using Angular
- Beautify your application with the UI components built to the material design specification
- Secure your web application from unauthorized users
- Create complex forms, taking full advantage of 2-way data binding
- Test your Angular applications using the Jasmine and Protractor frameworks for better efficiency
- Learn how to integrate Angular with Bootstrap to create compelling web applications
- Use Angular built-in classes to apply animation in your app



Learning Angular 4 for .NET Developers
Rajesh Gunasundaram

ISBN: 978-1-78588-428-3

- Create a standalone Angular application to prototype user interfaces
- Validate complex forms with Angular version 4 and use Bootstrap to style them
- Build RESTful web services that work well with single-page applications
- Use Gulp and Bower in Visual Studio to run tasks and manage JavaScript packages
- Implement automatic validation for web service requests to reduce your boilerplate code
- Use web services with Angular version 4 to offload and secure your application logic
- Test your Angular version 4 and web service code to improve the quality of your software deliverables

Essential Angular 4
Victor Savkin, Jeff Cross

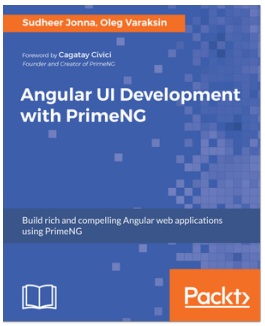
ISBN: 978-178829-376-1

- Understand why and how to use JIT and AOT compilation in Angular
- Bootstrap and inject NgModules
- Learn about the component lifecycle
- Understand the two phases of Change Detection
- Visualize and parse the Injector tree
- Understand advanced Lazy Loading
- Integrate and run different testing strategies on your code

Building Web Apps with Spring 5 and Angular 4
Ajitesh Shukla

ISBN: 978-1-78728-466-1

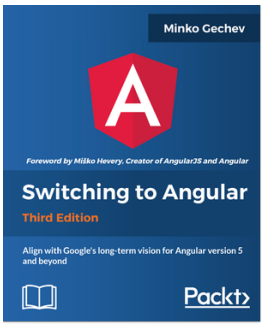
- Set up development environment for Spring Web App and Angular app.
- Process web request and response and build REST API endpoints.
- Create data access components using Spring Web MVC framework and Hibernate
- Use Junit 5 to test your application
- Learn the fundamental concepts around building Angular
- Configure and use Routes and Components.
- Protect Angular app content from common web vulnerabilities and attacks.
- Integrate Angular apps with Spring Boot Web API endpoints
- Deploy the web application based on CI and CD using Jenkins and Docker containers



Angular UI Development with PrimeNG
Sudheer Jonna, Oleg Varaksin

ISBN: 978-1-78829-957-2

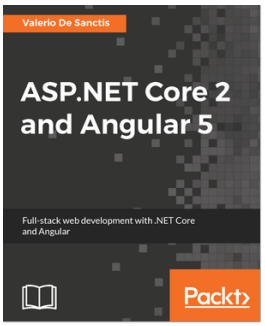
- Setup PrimeNG projects with SystemJS, Webpack, and Angular CLI.
- Use theming concepts and layouts with grid systems and Bootstrap.
- Work with enhanced input, select, button and panel components.
- Apply countless DataTable features: sorting, filtering, grouping, and templating.
- Meet data iteration components: DataList, DataGrid, Tree, and so on.
- Build endless menu variations: SlideMenu, TieredMenu, MegaMenu, and so on.
- Visualize your data representations with PrimeNG charts and GMap components.
- Adopt best practices such as state management with @ngrx/store.
- Write unit and end-to-end tests with Jasmine, Karma, and Protractor.



Switching to Angular - Third Edition
Minko Gechev

ISBN: 978-1-78862-070-3

- Align with Google's vision for Angular version 5 and beyond
- Confidently move forward with a long-term understanding of Angular
- Use stable APIs in Angular to build future-proof, blazingly fast enterprise applications
- Work with TypeScript to supercharge your Angular applications
- Understand the core concepts of Angular, aligned with the vision from Google
- Be ready with Angular from any direction—whether you're building new apps with the Angular and ASP.NET stack, or upgrading from AngularJS with ngUpgrade



ASP.NET Core 2 and Angular 5
Valerio De Sanctis

ISBN: 978-1-78829-360-0

- Use ASP.NET Core to its full extent to create a versatile backend layer based on RESTful APIs
- Consume backend APIs with the brand new Angular 5 HttpClient and use RxJS Observers to feed the frontend UI asynchronously
- Implement an authentication and authorization layer using ASP.NET Identity to support user login with integrated and third-party OAuth 2 providers
- Configure a web application in order to accept user-defined data and persist it into the database using server-side APIs
- Secure your application against threats and vulnerabilities in a time efficient way
- Connect different aspects of the ASP. NET Core framework ecosystem and make them interact with each other for a Full-Stack web development experience