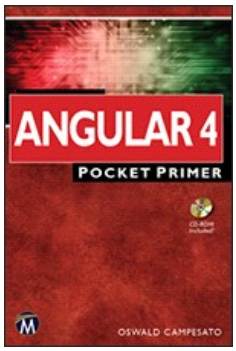

Angular 4 Pocket Primer

by Oswald Campesato

Chapter 1: Quick Introduction To Angular

Overview

This chapter provides a fast introduction to developing Web applications in Angular. After covering some of the high-level aspects of Angular, you can quickly grasp many of the code samples that are discussed in later chapters (but some code samples are more extensive). At the same time, keep in mind that you need to invest additional time and effort to acquire a deeper understanding of Angular.

Note The Angular code samples in this book are based on the Angular production code that was released in March 2017.

Another important consideration is your learning style: you might prefer to read the details regarding the "scaffolding" for Angular applications before you delve into the first code sample. However, it's perfectly acceptable to skim the introductory portion of this chapter, then quickly "get into the weeds" with the first Angular sample code in this chapter, and later review the initial portion again.

The first part of this chapter discusses the design goals of Angular and some new features, such as components, modules, and one-way data binding. The second part of this chapter discusses the Angular command-line interface (CLI), which is a command-line tool for generating Angular applications.

Note The Angular projects in this book are based on version `1.0.0` of `ng` (which is the Angular CLI) for creating Angular 4 applications.

There are several points to keep in mind before you read this chapter. First, this book provides many examples of Angular applications, so the details about Angular concepts, design goals, and architecture are "lighter" than what you might find in 500-page books. However, you can find useful information in online articles by Victor Savkin (and other people).

Second, you can create Angular applications using ECMA5, ES6, and TypeScript. However, the main recommendation is to develop Angular applications using ES6 or TypeScript; in fact, all the code samples in this book use TypeScript, which makes the transpilation process (performed "behind the scenes") straightforward. Third, no previous experience with Angular is required, but some knowledge can obviously be helpful.

Note DVD When you copy a project directory from the companion disc, if the node_modules directory is not present, then copy the top-level node_modules directory that has been soft-linked inside that project directory (which is true for most of the sample applications).

Supported Versions of Angular: How It Works

In case you didn't already know, the Angular landscape is moving fast, starting from the production release of Angular 4 in March 2017 and extending through Angular 7, which is scheduled for September 2018.

Each upcoming release (starting with Angular 6 in 2018) will probably support only two earlier versions of Angular. For example, Angular 6 will support Angular 4 and Angular 5, whereas Angular 7 will support Angular 6 and Angular 5 (but not Angular 4).

In addition, there is a sort of "duality" in the naming convention for Angular: upcoming Angular releases specify a version number, but Angular *without* a version number is the official name for Angular 2 and beyond (whereas AngularJS is the official name for version 1.x).

Another important goal of the Angular core team during this development process is to minimize breaking changes and ensure that such changes will have minimal impact with respect to upgrading to new versions of Angular.

Note Despite the version numbers for future production code releases, Angular 2 and beyond is officially known as Angular.

The Reason for Skipping Angular 3

The core Angular libraries are in github.com/angular/angular. All of them are versioned the same way, but distributed as different Node-based packages:

```
@angular/core          v2.3.0
@angular/compiler      v2.3.0
@angular/compiler-cli  v2.3.0
@angular/http          v2.3.0
@angular/router        v3.3.0
```

As you can see, the version number for the router package differs from the version number of the other packages. The simplest way to make the version numbers identical was to use version 4 instead of version 3, which in turn resulted in skipping Angular 3.

TypeScript Version

Upgrading the TypeScript dependency from v1.8 to v2.1 (or higher) does involve a breaking change. Fortunately, upgrading from Angular version 2 to version 4 (and beyond) does not involve a major rewrite, just a change in a few core libraries.

The Angular team has developed an automatic upgrade process, and the tool that they use might also become available for general use (possibly during 2017).

Note The Angular 2 applications in this book use version 2.2.2 of the TypeScript compiler.

Moving from Angular 2 to Angular

This section is optional if you are working with the latest release of Angular, which automatically includes all the features that were introduced in Angular 2.x beyond Angular 2.0.

A brief synopsis of several Angular 2.x releases is provided below because the new features were added to Angular 2 (but they are not part of Angular 2.0). In addition, the version of TypeScript has changed.

First, Angular 2.2 was released in November 2016; this version provides ahead-of-time (AOT) compilation compatibility and is discussed in more detail in Chapter 10. Next, Google released Angular 2.3 in December 2016, which includes the Angular Language Service for integrating with integrated development environments (IDEs). This service provides type completion and error-checking with Angular Templates. Moreover, object inheritance for components is featured in this service.

Regarding the version of TypeScript: Angular 2 used TypeScript 1.8, whereas the initial production release of Angular 4 uses TypeScript 2.1.6 (or higher), which involves some relatively small breaking changes.

What You Need to Learn for Angular Applications

Angular applications in this book are written in TypeScript, but you also need to acquire a basic proficiency in the following technologies:

- NodeJS (npm)
- ECMA5, ES6, and TypeScript
- Webpack 2.2 (or higher)

The following subsections contain more information about the items in the preceding list.

NodeJS

The code samples in this book require `node` v6.x.x (or higher) and `npm` 4.x.x (or higher). Determine the version on your machine with the following commands in a terminal:

```
node -v
npm -v
```

If necessary, navigate to the NodeJS home page to download a more recent version of the node executable. If you have not worked with Node, read an online tutorial to understand the purpose of the following commands:

- `npm install –g webpack`
- `npm install webpack@latest –save`
- `npm start`

ECMA5, ES6, and TypeScript

You need to learn the basic concepts of ES6 and TypeScript, and their respective home pages contain plenty of information to help you get started. In particular, learn about arrow functions, classes, template strings, and module loaders. Other useful features include the spread and rest operators (you will encounter them in Chapter 7). As you will see in subsequent chapters, Angular applications rely heavily on dynamic templates, which frequently involve interpolation (via the "{{}}" syntax) of variables.

Knowledge of ES6 is helpful if you plan to write Angular applications with TypeScript. Fortunately, the following website provides an online "playground" and links for documentation and code samples for TypeScript:

https://www.typescriptlang.org/play/

Familiarity with ECMA5 is also useful: for example, the `filter()` function is handy (e.g., with Angular Pipes), and the `map()` function can be useful when you combine `Observables` with HTTP requests in Angular applications. Other functions, such as `merge()` and `flatten()`, can also be useful, and you can learn about them and other functions on an as-needed basis.

You also need a basic understanding of Promises and Observables. Angular with TypeScript favors Observables, as do the code samples in this book, but you will encounter online code samples that use Promises. Avail yourself of online resources regarding `ECMA5`, Promises, and `Observables`.

Angular takes advantage of ES6 features such as components and classes, as well as features that are part of TypeScript, such as annotations and its type system. TypeScript is preferred over `ECMA5`or ES6 because (1) TypeScript supports all the features of ES6, and (2) TypeScript provides an optional type inferencing system that can catch many errors for you.

As for scaffolding tools, Webpack has become the de facto standard for Angular applications. Webpack (version 2.2 or higher) has become

the de facto standard scaffolding tool for Angular applications, and its home page is located here:

https://webpack.github.io/

If you have not worked with Webpack, learn about the features of version 2.x (and bypass version 1.x).

Some additional relevant details: You can develop Angular applications in Electron, Webstorm, and Visual Studio Code. Check their respective websites for pricing and feature support. Finally, the following link includes style-related guidelines as well as "best practices" for developing Angular applications:

https://github.com/mgechev/angular2-style-guide

Glance through the preceding link to familiarize yourself with its content, and then you will know when to read the relevant sections as you progress through the chapters in this book.

A High-Level View of Angular

Angular was designed as a platform that supports Angular applications in a browser and supports server-side rendering and Angular applications on mobile devices. The first aspect—rendering Angular applications in browsers—is the focus of the chapters in this book. The second aspect—Angular Universal (aka server-side rendering)—is discussed briefly in Chapter 6. In essence, server-side rendering creates the "first view" of an Angular application on a server instead of a browser. Because browsers do not need to construct this view, they can render a view more quickly and create a faster perceived load time. The third aspect—Angular applications on mobile devices—is discussed in Chapter 8.

Angular also has a component-based architecture, where components are organized in a treelike structure (the same is true of Angular modules, as you will see in Chapter 5). Angular also supports powerful technologies that you will learn to become proficient in writing Angular applications. The simplest way to create an Angular application is to use the Angular CLI (discussed in detail later), which generates the required files for an Angular application from the command line.

Some of the important features of Angular are listed here:

- One-way data binding
- "Tree shaking"
- Change detection
- Style encapsulation

The first two features are briefly discussed below and the third feature is discussed in Chapter 2, where the code sample will make more sense to you than in this chapter.

Because the production version of Angular 4 was in March 2017, and additional "point" releases have been released (e.g., 4.1), it's possible that the latter releases will cause breaking changes in some of the code samples in this book. Consequently, you might need to modify application programming interface (API) calls in the affected code samples or change some `import` statements. You can always consult the online documentation regarding changes between consecutive releases of Angular:

https://angular.io/

One-Way Data Binding in Angular

Angular provides declarative one-way binding as the default behavior (but you can switch to two-way binding if you wish to do so). One-way binding acts as a unidirectional change propagation that provides two advantages over two-way data binding: (1) an improvement in performance because of eliminating the `$digest` cycle and watchers in Angular 1.x, and (2) a reduction in code complexity. Angular also supports stateful, reactive, and immutable models. The meaning of the previous statement will become clearer as you work with Angular applications.

Angular applications involve defining a top-level ("root") module that references a `Component` that in turn specifies an HTML element (via a mandatory `selector` property) that is the "parent" element of the `Component`. The definition of the `Component` involves a so-called "decorator," which includes a `selector` property and a `template` property (or a `templateUrl` property).

The `template` property includes a mixture of HTML and custom markup (which can be placed in a separate file and then referenced via the `templateUrl` property). In addition, the `Component` is immediately followed by a TypeScript class definition that includes "backing code" that is associated with component-related variables that appear in the `template` property. These details will become much clearer after you have worked with some Angular applications.

Note The `templateUrl` property and `styleUrls` property refer to files, whereas the `template` property and `styles` property refer to inline code.

Tree Shaking an Angular Application

Tree shaking refers to "pruning" the unnecessary files from an Angular project (other technologies have the same concept), just like shaking

the dead branches from a tree, in order to create a production version of an Angular application. This production version can be significantly smaller, and hence require less loading time. Webpack 2 and Angular AOT support tree shaking, and you will see an example with AOT in Chapter 10.

A High-Level View of Angular Applications

Angular applications consist of a combination of built-in components and custom components (the latter are written by you), each of which is typically defined in a separate TypeScript file (with a `ts` extension). Each component specifies its dependencies via `import` statements. There are various types of dependencies available in Angular, such as directives and pipes (discussed later in this chapter).

A *custom directive* is essentially the contents of a TypeScript file that defines a component. Thus, a custom directive consists of `import` statements, a `Component` decorator, and an exported TypeScript class.

Angular provides *built-in directives*, such as `*ngIf` (for "if" logic) and `*ngFor` (for loops). These two directives are also called *structural directives* because they modify the content of an HTML page.

Angular *built-in pipes* include date and numeric items (currency, decimal, number, and percent), whereas *custom pipes* are defined by you.

In addition, TypeScript classes use a *decorator* (which is a built-in function) that provides metadata to a class, its members, or its method arguments. Decorators are easy to identify because they always have the @ prefix. Angular provides a number of built-in decorators, such as `@Component()` and `@NgModule`.

This concludes the high-level introduction to Angular features. The next portion of this chapter introduces the Angular CLI, which is used throughout this book to create Angular applications.

The Angular CLI

During the beta releases of Angular, developers used a manual process to create applications, or they used a "starter" or "seed" project (often available on GitHub). However, those projects are often out of date after new releases of Angular are available. Hence, the code samples in this book are based on the Angular CLI, which is the official Angular application generator from Google.

The Angular CLI is a command-line tool called `ng` that generates complete Angular applications, including test-related code, and (by default) also launches `npm install` to install required files in `node_modules`. The home page for the Angular CLI is located here:

https://cli.angular.io

The Angular CLI generates a configuration file called `package.json` to manage the necessary dependencies and their version numbers. After generating an Angular application, navigate to the `node_modules` subdirectory, and you will see an assortment of Angular subdirectories that contain files that are required for Angular applications.

The Angular CLI is a superior alternative to using "starter" projects or creating projects manually, and its feature set will continue to improve over time.

Installing the Angular CLI

You need to perform several steps to install the Angular CLI. First uninstall the previous CLI (if you installed an older version) with this command:

```
sudo npm uninstall -g angular-cli
npm cache clean
```

Next, install the new CLI with this command (note the new package name):

```
[sudo] npm install -g @angular/cli
```

Create a new project called `hello-world` with Angular 4 as follows:

```
ng new hello-world
```

The Angular CLI provides everything except your custom code, and also requires noticeably more time to install than starter applications. Second, the Angular CLI enables you to generate new components, routers, and so forth, which are possible with starter applications. Third, the Angular CLI is based purely on TypeScript, and the generated application includes the JavaScript Object Notation (JSON) files `tsconfig.json`, `tslint.json`, `typedoc.json`, and `typings.json`. On the other hand, the starter applications tend to use Webpack, which involves the configuration file `webpack.config.js`, which includes information for Angular applications.

Features of the Angular CLI

The `ng` executable supports various options, and some command-line invocations are shown here:

```
ng new app-root-name
ng build
ng deploy
```

```
ng e2e
ng generate <component-type>
ng generate route
ng generate ...
ng lint
ng serve
ng test
ng x18n
```

The `ng g` option is equivalent to the `ng generate` option, which enables you to generate an Angular custom `Component`, an Angular `Pipe` (discussed in Chapter 5), and so forth. The `ng x18n` option extracts i18n messages from source code. The next section shows you an example of generating an Angular custom `Component` in an application, and the contents of the files that are automatically generated for you.

The default prefix is `app` for components (e.g., `<app-root></app-root>`), but you can specify a different prefix with this invocation:

```
ng new app-root-name –prefix abc
```

Note Angular applications created via `ng` always contain the `src/app` directory.

Information about upgrading the Angular CLI is located here:

https://github.com/angular/angular-cli

Documentation for the Angular CLI is located here:

http://cli.angular.io

Now that you have an understanding of some of the features of the `ng` utility, let's create our first Angular application, which is the topic of the next section.

A "Hello World" Application via the Angular CLI

Navigate to a convenient directory and create an Angular application called `myapp` with the following command:

```
ng new myapp
```

Note Earlier versions of ng require an `--ng4` switch.

After the preceding command has completed, navigate inside the new project:

```
cd myapp
```

Launch the Angular application with the following command:

```
ng serve
```

The preceding command automatically launches a browser session at the following URL:

```
localhost:4200
```

You will see the following displayed in your screen:

app works!

The preceding string is specified in the `template` property in the file app/app.component.ts.

Note The `template` property (or the `templateUrl` property) is where you will place custom code that will generate `HTML` output, which is then inserted into the `<app-root>` element in the `index.html` Web page.

The Structure of an Angular Application

When you invoke the `ng` command to create an Angular application, here are the files that are automatically created:

```
angular-cli.json
dist/ (details omitted)
e2e/  (details omitted)
karma.conf.js
node_modules
package.json
protractor.conf.js
README.md
src/app/app.component.css
src/app/app.component.html
src/app/app.component.spec.ts
src/app/app.component.ts
```

```
src/app/app.module.ts
src/environments/environment.prod.ts
src/environments/environment.ts
src/favicon.ico
src/index.html
src/main.ts
src/polyfills.ts
src/styles.css
src/test.ts
src/tsconfig.json
tslint.json
```

The `dist` subdirectory includes the compiled project and JavaScript bundle, whereas the `e2e` subdirectory includes files for end-to-end testing (invoke the command `ng test` to execute the tests).

The file `app.component.ts` (shown in bold) includes custom TypeScript code in the code samples in this book, and (to a lesser extent) you might need to update the file `app.module.ts`. You will modify `index.html` when you need to include JavaScript `<script>` elements (for jQuery, Bootstrap, and so forth), and the `styles.css` file is for global Cascading Style Sheets (CSS) style rules (if any).

Finally, the files `angular-cli.json`, `package.json`, and `tslint.json` are configuration files for `ng`, `npm`, and `tslint`, respectively. Note that when you invoke the command `ng lint`, it invokes the `tslint` utility, which in turn relies on the file `tslint.json`.

The Naming Convention for Angular Project Files

The files in the `app` subdirectory have a naming convention that comprises three parts: the type of functionality, whether it's a component or module, and a suffix that indicates the type of code. For example, the TypeScript file `app.component.ts` includes component-related code for the application, whereas the TypeScript file `app.module.ts` includes module-related code. In addition, the file `app.component.css` contains CSS selectors for the component, and `app.component.html` contains HTML markup for the same component.

The next portion of this chapter contains two sections: The first part discusses the contents of `index.html` and various JavaScript configuration related files, and the second part discusses application-related files that are contained in the `app` subdirectory. Alas, reading these sections is a "long slog," and although it's recommended that you read them, feel free to skim this section and return after you have launched your first Angular application. There are trade-offs with both reading styles, so proceed with this material in the manner that best suits your learning style.

Now let's take a look at the contents of the HTML Web page `index.html`, which is the main Web page for our Angular application.

The index.html Web Page

Listing 1.1 displays the contents of `index.html` for a new Angular application that is generated from the command line via the `ng` utility.

Listing 1.1: index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular Application</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>
```

Listing 1.1 is minimalistic: Only the custom `<app-root>` element (which you will see in the `selector` property in app/app.component.ts) gives you an indication that this Web page is part of an Angular application.

Note The JavaScript dependencies are dynamically inserted in `index.html` by the Angular CLI during the "build" of the project.

Before we delve into the TypeScript files in an Angular application, let's take a quick detour to understand how `import` statements work in Angular applications. Feel free to skip the next section if you are already familiar with `import` and `export` statements in Angular.

Exporting and Importing Packages and Classes (Optional)

Keep in mind the following point: Every TypeScript class that is imported in a TypeScript file must be exported in the TypeScript file where that class is defined. You will see many examples of `import` and `export` statements; in fact, this is true of every Angular application in this book.

There are 2 common types of `import` statements: one type involves importing packages from Angular modules, and the other type involves importing custom classes (written by you). Here is the syntax for both types:

```
import {some-package-name} from 'some-angular-module';
import {some-class }       from 'my-custom-class';
```

Here is an example of both types of `import` statements:

```
import { NgModule }        from '@angular/core';
import {EmpComponent}      from './emp.component';
```

In the preceding code snippet, the `NgModule` package is imported from the `@angular/core` module that is located in the `node_modules` directory. The `EmpComponent` class is a custom class that is defined and exported in the TypeScript file `emp.component.ts`.

In the second `import` statement, the "`./`" prefix is required when a custom class is imported from a TypeScript file, and notice the omission of the "`.ts`" suffix.

The next several sections discuss three application-related TypeScript files in the `src/app` subdirectory: `main.ts`, `app.component.ts`, and `app.module.ts`. These files are the bootstrap file, the main module, and the main component class, respectively, for this Angular application.

Here is the condensed explanation of the purpose of these three files: Angular uses `main.ts` as the initial "entry point" to bootstrap the Angular module `AppModule` (defined in `app.module.ts`), which in turn includes the component `AppComponent` (defined in `app.component.ts`), as well as any other custom components (and modules) that you have imported into `AppModule`.

Moreover, these three files are located in the `src/app` subdirectory, which is also where you place custom components (and modules), or some suitable subdirectory of `src/app` whose name is based on its feature.

The Bootstrap File `main.ts`

Listing 1.2 displays the contents of `main.ts` in the `src` subdirectory (not the `src/app` subdirectory) that imports and bootstraps the top-level Angular module `AppModule`.

Listing 1.2: main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-
                                          browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

The first line of code in Listing 1.2 is an `import` statement that is needed for the conditional logic later in the code listing. The second `import` statement that you will see in many Angular code samples, and it's necessary for launching Angular applications on desktops and laptops.

The third `import` statement involves the top-level module of Angular applications, which in turn contains all the custom components and services that are included in this Angular module. The fourth `import` statement contains environment-related information that is used in the next conditional logic snippet: if the current application is in production mode, the `enableProdMode()`function is executed.

The final line of code is the actual bootstrapping process, which involves rendering the code in `app.component.ts` in a browser.

The Top-Level Module File `app.module.ts`

Listing 1.3 displays the contents of `app.module.ts` (located in the `src/app` subdirectory) that exports the top-level Angular module `AppModule`.

Listing 1.3: app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
```

```
import { NgModule }      from '@angular/core';
import { FormsModule }   from '@angular/forms';
import { HttpModule }    from '@angular/http';
import { AppComponent }  from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 1.3 includes two `import` statements that import `BrowserModule`, `NgModule`, `FormsModule`, `HttpModule`, and `BrowserModule`, all of which are part of Angular. The last `import` statement imports the class `AppComponent`, which is the top-level component illustrated in Listing 1.4.

Note Angular dependencies always contain the @ symbol, whereas custom dependencies specify a relative path to TypeScript files.

Next, the `@NgModule` decorator includes an object with various properties (discussed in the next section). These properties specify metadata for the class `AppModule`, which is exported in the final line of code. The metadata in `AppModule` involves the following properties, each of which is an array of values: `imports`, `providers`, `declarations`, `exports`, and `bootstrap`.

In Listing 1.3, the array properties `declarations`, `imports`, and `bootstrap` are non-null, whereas the `providers` property is an empty array. This metadata is required for Angular to bootstrap the code in `AppComponent`, which in turn contains the details of what is rendered (e.g., an <h1> element) and where it is rendered (e.g., the `app-root` element in `index.html`).

Now let's take a closer look at the purpose of each array-based property in the `@NgModule` decorator to understand their purpose.

The MetaData in `@NgModule`

The `imports` array includes *modules* that are required for this application, such as `BrowserModule`, and some optional modules (e.g., `FormModule`) for this application. The `imports` array is not transitive: if module A imports module B and module B imports module C, then module C is not imported into module A.

Next, the `providers` array is an array of application-wide services for this Angular application. A service is something that provides behind-the-scenes functionality that is not visible in the application. The `providers` array includes any injectable services that have been defined via the `@Injectable` decorator (discussed later).

The `declarations` array consists of *components* that are required for this application, such as the `AppComponent`, and also custom components, directives, and pipes, all of which have (module-level) private scope. Specifically, *private scope* means that everything in the `declarations` array is accessible only via other components, directives, and pipes that are declared in the same module.

Keep in mind that components listed in the `declarations` property *must* be exported from their respective component-related files. For example, `AppComponent` is exported from `app.component.ts` (shown later), which enables you to import it in `app.module.ts` and also specify it in the `declarations` property.

The `exports` array includes components, directives, and pipes that are required in other components in the application.

The `schemas` array includes the value `CUSTOM_ELEMENTS_SCHEMA`, which provides additional information that is "external" to the application; see the example in Chapter 2 and the example in Chapter 5.

Finally, the `bootstrap` array includes the name of the component that will be "bootstrapped" when the Angular application is launched in a browser.

Note In general, it's advisable to bootstrap only one component via the `bootstrap` property.

The Top-Level Component File `app.component.ts`

Listing 1.4 displays the contents of `app.component.ts` (located in the `src/app` subdirectory) that exports the top-level Angular component `AppComponent`.

Listing 1.4: app.component.ts

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

@Component({
  selector:    'app-root',
  templateUrl: './app.component.html',
  styleUrls:   ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Listing 1.4 starts with an `import` statement for the Angular `@Component` decorator to define metadata for the class `AppComponent`. At a minimum, the metadata involves two properties: `selector` and `template`. Except for routing-related components, both of these properties are required in custom components. In this example, the `selector` property specifies the custom element `app-root` (which you can change) that you saw in the HTML Web page `index.html`.

The `template` property specifies the HTML markup that will be inserted in the custom element `app-root`. In this example, the markup is an `<h1>` element containing some text. The final line of code in Listing 1.4 is an `export` statement that makes the `AppComponent` class available for import in other TypeScript files, such as `app.module.ts`, which is shown in Listing 1.3.

Note The files `main.ts`, `app.component.ts`, and `app.module.ts` are in the `src/app` subdirectory for all the Angular projects in this book.

Launch your Angular application by navigating to the root directory and entering the following command from a shell:

```
ng serve
```

The next section discusses Angular template syntax, which you will use in your custom code in the `template` property.

A Simple Angular Template

As you saw in Listing 1.4, the file `app.component.ts` includes a `template` property with an `<h1>` element that contains a single line of text. However, Angular enables you to specify multiple lines of text when you place everything inside a pair of matching backticks ("``"). This syntax (introduced in ES6) is used heavily in Angular applications, and it conveniently supports variable interpolation.

Listing 1.5 displays the contents of a new `app.component.ts` file, which has more content than Listing 1.4. This code sample illustrates how to use interpolation in a `template` property.

Listing 1.5: app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
             <h3>Hello everyone!</h3>
             <h3>My name is {{emp.fname}} {{emp.lname}}</h3>
             `
})
export class AppComponent {
  public emp = {fname:'John',lname:'Smith',city:'San Francisco'};
}
```

Listing 1.5 includes two `<h3>` elements; the first is a simple text string and the second includes "curly braces" to reference data values that are defined in the class `AppComponent`. Angular uses something called *interpolation* with the elements of the literal object `emp` and then substitutes the variables inside the curly braces with their actual values in the second `<h3>` element.

Working with Components in Angular

An Angular application is a tree of nested components, where the top-level component is the application. The components define the user interface (UI) elements, screens, and routes. In general, organize Angular applications by placing each custom component in a TypeScript file and then import that same TypeScript file in the "main" file (which is often named `app.component.ts`), which includes the top-level component.

The Metadata in Components

Angular components are often a combination of an `@Component` decorator and a class definition that can optionally contain a constructor. A simple example is shown here:

```
import { Component }  from '@angular/core';
import {EmpComponent} from './emp/emp.component';

@Component({
    selector:  'app-container',
    template:  `{{message}}<tasks></tasks>`,
    directives: [EmpComponent]
})
```

The preceding `@Component` decorator includes several properties, some of which are mandatory and others are optional. Let's look at both types in the preceding code block.

The `selector` property is mandatory, and it specifies the HTML element (whether it's an existing element or a custom element) that serves as the "root" of an Angular application.

Next, the `template` property (or a `templateUrl` property) is mandatory, and it includes a mixture of markup, interpolated variables, and TypeScript code. One important detail: the `template` property requires backticks when its definition spans multiple lines.

The `directives` property is an optional property that specifies an array of components that are treated as nested components. In this example, the `directives` property specifies the component `EmpComponent,` which is also imported (via an `import` statement) near the beginning of the code block. Notice that the `import` statement does not contain the @ symbol, which means that `EmpComponent` is a custom component defined in the file `emp/emp.component.ts`.

Stateful versus Stateless Components in Angular

In high-level terms, a *stateful* component retains information that is relevant to other parts of the same Angular application. On the other hand, stateless components do not maintain an application state, nor do they request or fetch data: they are passed data via property bindings from another component (such as its parent).

The code samples in this book are usually a combination of stateful components, stateless components, and sometimes "value objects," which are instances of custom classes that model different entities (such as an employee, customer, student, and so forth).

You will see an example of a presentational component in Chapter 2. In the meantime, a good article that delves into stateful and stateless components is located here:

https://toddmotto.com/stateful-stateless-components# stateful

Generating Components with the Angular CLI

Earlier you saw some of the options for the `ng` utility, and this section discusses how to use the `generate` option. In addition, here are some self-explanatory examples:

```
ng generate component  mycomp
ng generate directive  mydir
ng generate pipe       mypipe
ng generate route      myroute
ng generate service    mycomp
```

The following commands are equivalent to their counterparts in the preceding list, except that they use short aliases:

```
ng g c mycomp
ng g d mydir
ng g p mypipe
ng g r myroute
ng g s mycomp
```

The preceding commands create new files that are located in new subdirectories of `src/app`. For example, the command `ng g c mycomp` creates the subdirectory `src/app/mycomp` and populates it with files (discussed later).

If you do not want to create a new subdirectory, use the option `–flat` and the new files will be placed in `src/app` instead of `src/app/mycomp`.

Now let's invoke the following command in an Angular application to generate a `student` component:

```
ng g c student
```

After the preceding command has completed, you will see the following output:

```
installing component
  create src/app/student/student.component.css
```

```
  create src/app/student/student.component.html
  create src/app/student/student.component.spec.ts
  create src/app/student/student.component.ts
```

As you can see, the preceding output displays four new files: a CSS file, an HTML file, a TypeScript test file, and a component definition file.

Listing 1.6 displays the contents of `student.component.ts` that contains the code for the `student` component.

Listing 1.6: student.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }
}
```

Note When you use `ng` to create a custom component, `ng` inserts an `import` statement in `app.module.ts` and updates the `declarations` property with a reference to the new custom component.

For your convenience, `ng` also generates a test file when you generate a component from the command line. In particular, Listing 1.7 displays the contents of the test file called `student.component.spec.ts`.

Listing 1.7: student.component.spec.ts

```
/* tslint:disable:no-unused-variable */

import { TestBed, async } from '@angular/core/testing';
import { StudentComponent } from './student.component';

describe('Component: Student', () => {
  it('should create an instance', () => {
    let component = new StudentComponent();
    expect(component).toBeTruthy();
  });
});
```

Listing 1.7 includes `import` statements to make various components available to this test file, including the `StudentComponent` that is defined in Listing 1.6.

The autogenerated contents of `student.component.html` are shown here:

```
<p>
  student works!
</p>
```

As you can see, the file `student.component.html` is minimalistic, and you are free to add application-related HTML markup in this file.

Note that the file `student.component.css` is currently empty; add any CSS selectors that you need for styling purposes.

One other detail: `ng` enables you to create a component inside an existing component. For example, the following command creates a `profile` component inside the `student` component:

```
ng g component student/profile
```

You must invoke the preceding command from the `student` subdirectory, otherwise you will see the following error message:

```
You have to be inside an angular-cli project in order to use
the generate command.
```

When you are ready, create a production build of an Angular application with the following command:

```
ng build --prod
```

Keep in mind that you can significantly reduce the size of an Angular application via AOT, which is discussed in more detail in Chapter 10.

Syntax, Attributes, and Properties in Angular

Angular introduced the square brackets "[]" notation for attributes and properties, as well as the parentheses "()" notation for functions that handle events. This new syntax is actually valid HTML5 syntax. Here is an example of a code snippet that specifies an attribute and a function:

```
<foo [bar]= "x+1" (baz)="doSomething()">Hello World</foo>
```

An example that specifies a property and a function is shown here:

```
<button [disabled]="!inputIsValid" (click)="authenticate()"
                                              >Login </button>
```

An example of a data-related element with a custom element is shown here:

```
<my-chart [data]="myData" (drag)="handleDrag()"></my-chart>
```

The new syntax in the preceding code snippet eliminates the need for many built-in directives, as you will see later in this chapter.

Attributes versus Properties in Angular

Keep in mind the following distinction: A property can specify a complex model, whereas an attribute can only specify a string. For example, in Angular 1.x you can write the following:

```
<my-directive foo="{{something}}"></my-directive>
```

The corresponding code in Angular (which does not require interpolation) is shown here:

```
<my-directive [foo]="something"></my-directive>
```

The new architecture for Angular provides improved performance and a mechanism for developing "cleaner" Angular applications that can be developed, enhanced, and maintained more quickly.

The next section contains a code sample involving a `<button>` element, which is probably one of the most common UI controls in HTML Web pages.

Displaying a Button in Angular

After having soldiered through all the code listings in this chapter, and also reading explanations about their purpose, you might be wondering if application development in Angular is going to be a long and tedious process. Fortunately, you can create many basic applications with a small amount of code. When you are ready to create medium-sized applications, you can take advantage of the component-based nature of Angular applications to incrementally add new components (and modules).

As a simple example, the file `app.component.ts` in this section includes all the custom code for this Angular application. The `ng` utility was used to generate all the files in this code sample, and you won't need to tinker manually with those other files.

DVD Copy the directory `ButtonClick` from the companion disc into a convenient location. Listing 1.8 displays the contents of `app.component.ts` that illustrates how to render a `<button>` element and respond to click events by displaying the number of times that users have clicked the `<button>` element during the current session.

Listing 1.8: app.component.ts

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

@Component({
   selector: 'app-root',
   template: `<div>
               <button (click)="clickMe()">ClickMe</button>
               <p>Click count is now {{clickCount}}</p>
              </div>`,
   styles: [` button {
                 color: red;
              }`
           ]
})
export class AppComponent {
   clickCount = 0;

   clickMe() {
      ++this.clickCount;
```

```
      console.log("click count: "+this.clickCount);
   }
}
```

Listing 1.8 starts with an `import` statement that appears in `app.component.ts`, which you will see in every code sample in this book. This statement gives you access to the `Component` decorator, which injects metadata into the TypeScript class called `AppComponent`. When the TypeScript compiler transpiles (converts) the TypeScript code into ECMA5, the metadata will also be included to run the Angular application in modern browsers.

The required `selector` property specifies a value of `app-root`, which is the custom element (listed in `index.html`) that serves as the "container" where the content of the `template` element is rendered.

In this example, the `template` property includes a `<button>` element that responds to click events and a `<p>` element whose contents are updated when users click the `<button>` element. As you can see, the value of the term `(click)` is the `clickMe()` function (defined in the `AppComponent` class), which increments and then displays the value of the `clickCount` variable.

In addition, the `styles` property specifies a value of `red` for the `<button>` element. The `styles` property is an example of component style, which means that the styles only apply to the template of the given component.

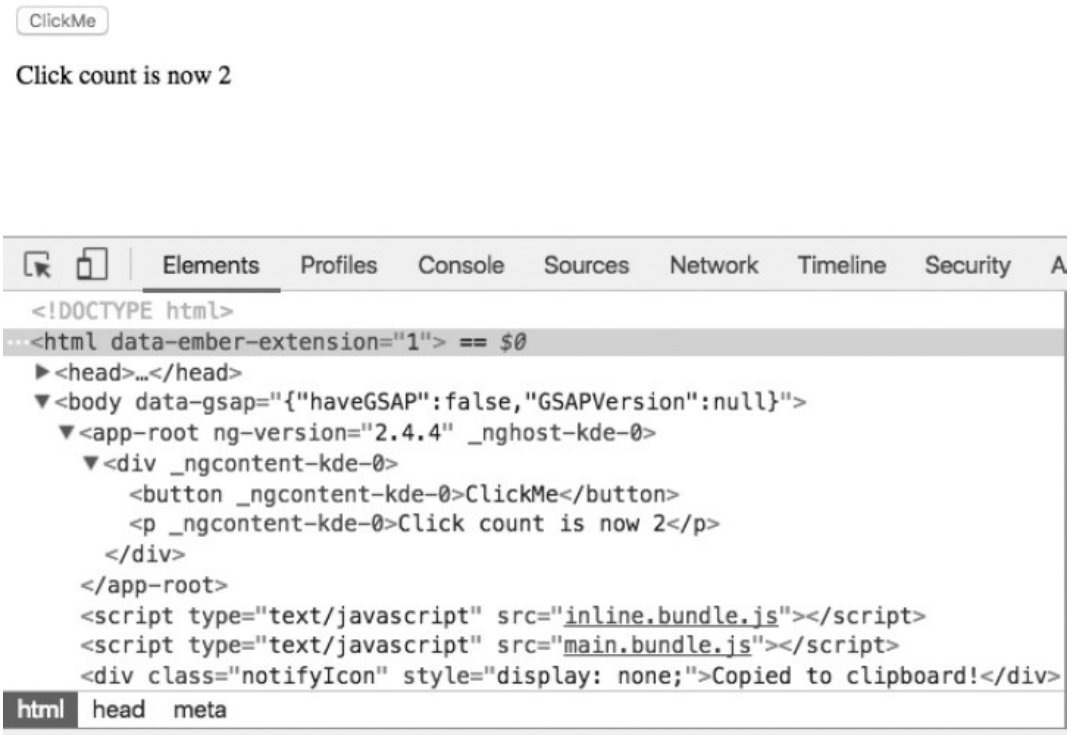

Figure 1.1: A `<button>` Element that Responds to Click Events.

In effect, Angular applies CSS locally instead of globally by generating unique attributes that are visible when you click the `Elements` tab in Chrome Web Inspector.

More detailed information regarding component styles in Angular is located here:

https://angular.io/docs/ts/latest/guide/component-styles. html

The next portion of Listing 1.8 is the definition of the `AppComponent` class that includes the `clickCount` variable, which is incremented in the `clickMe()` function.

Navigate to the `src` subdirectory of this application and invoke the following command:

```
ng serve
```

Figure 1.1 displays the browser output from this Angular application. You can also see the file `main.bundle.js`, which includes minified "tree shaken" code, which is discussed in more detail in Chapter 10.

Element versus Property

In Listing 1.8, the `selector` property matched the element `<app-root></app-root>` in the HTML page `index.html`:

```
selector: 'app-root'
```

However, you can also specify a property instead of an element. For example, suppose that `index.html` includes the following element:

```
<div app-root>Loading. . .</div>
```

You also need to modify the `selector` property as follows:

```
selector: '[app-root]'
```

Summary

This chapter started with a description of the Angular version numbers, prerequisites for Angular, and an overview of Angular and its hierarchical component-based structure.

Next you learned about the Angular CLI utility `ng` and how to create an Angular "Hello world" application with the `ng` utility. You also learned about the TypeScript files `main.ts`, `app.component.ts`, and `app.module.ts`, which contain TypeScript code for an Angular application. Next you learned about the reason for transpiling the code in an Angular application into ECMA5. Finally, you saw the code for an Angular application that displays a `<button>` element that also responds to click events.