ANGULAR 2

A PRACTICAL INTRODUCTION TO THE NEW **WEB DEVELOPMENT PLATFORM**

Sebastian Eschweiler

Angular 2

A Practical Introduction to the new Web Development Platform Angular 2 (Angular.js, Angular.js 2, AngularJS, AngularJS 2, ng2)

Sebastian Eschweiler

Tweet This Book!

Please help Sebastian Eschweiler by spreading the word about this book on Twitter!

The suggested hashtag for this book is #angular2book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#angular2book

Contents

Chapter 01: Introduction

Angular 2 is the next major version of Google's popular JavaScript-based web framework. Angular 2 is designed for building complex applications for the browser. In contrast to version 1.x Angular 2 introduces a complete new concept of building web applications. In this book you will learn the basic concepts and explore the new building blocks of Angular 2.

Why Angular 2 Rocks

Since AngularJS 1.x was first released in 2009 Google's web framework has become increasingly popular. Many web and mobile developers have been using the framework in that last six years to implement applications of different sizes and complexity. Within the six years the frameworks has constantly evolved but the basic concepts remained. After six years it's now time to make a big move and change some of the basic concepts to make Angular a "modern" web framework again. In order to do so the Angular team at Google decided to make that big move by breaking with a lot of concepts from AngularJS 1.x and implement something completely new. This step becomes necessary because otherwise they would not have been able really make big improvements and break with things like the complex syntax for writing directives, multiple different approaches to implement services and struggling with the various scopes an AngularJS application has to deal with. Angular 2 has now evolved to a framework which is fully component oriented. Every Angular 2 application is built up of a tree of components which makes it much easier to reuse functionality and test single functional units. In addition Angular 2 is now based on the object-oriented ECMAScript 2015 and on top of that uses TypeScript. Angular will be more easy to use, simpler to learn and will make it easier to manage even complex application.

Who This Book Is For

You have some experience with AngularJS 1.x and now want to make the step to Angular 2 and learn more about how you can apply the new concepts specifically to developing robust web and mobile applications.

You have already worked with other modern web frameworks like Ember, Backbone or Knockout and now you would like to become familiar in Angular 2.

You have no framework experience at all but you have a profound understanding of HTML and JavaScript and now you would like to know how to bring your web and mobile applications to the next level by using Angular 2.

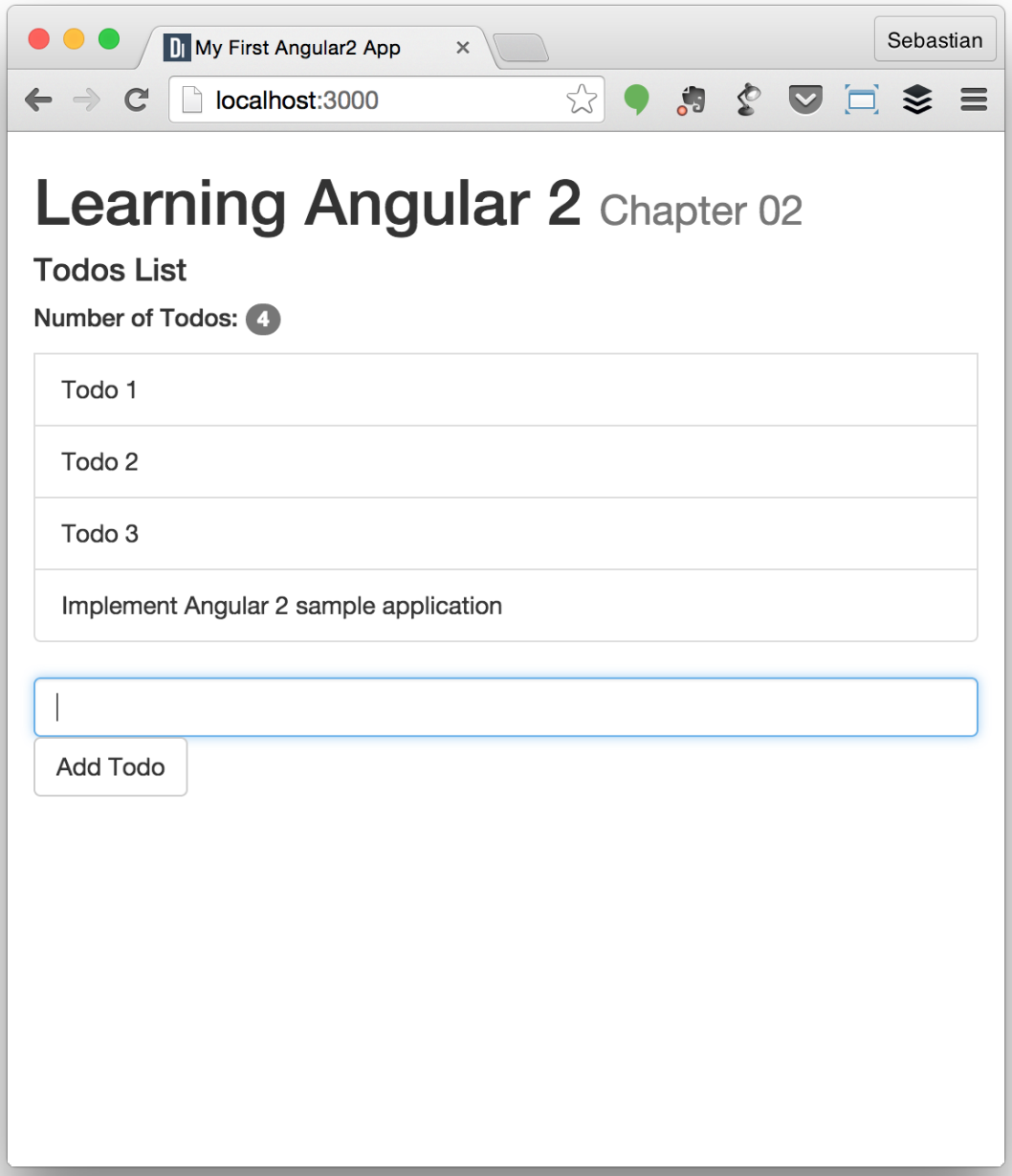
Code Examples On GitHub

As the subtitle of this book *A Practical Introduction to the new Web Development Platform* promises, the book will make use of many practical code examples to illustrate the new concepts of Angular 2. The code is available on GitHub. At the end of every chapter you'll find the link to the GitHub repository containing the sample code of this chapter.

Chapter 02: Writing your First Angular 2 Application

In this chapter we are going through the process of writing a first Angular 2 application from scratch. This will give you a first idea of the framework and provide you with a basic understanding of the concepts behind.

You can see a screenshot of the final application we are going to built in this chapter in the following:



Sample application of chapter 2

As you can see the sample application is a simple Todo application. The user is able to add new todo items by using the input field and clicking on button *Add Todo*. The output will be updated automatically after a new todo item has been added. The item will be printed out in the list and the total number of todos will increase.

Preparing your Development Environment

Before starting with the implementation you need to prepare your development environment. For the following steps we assume that you have *NPM* (*Node.js Package Manager*) already installed on your system. If this is not the case please refer to https://docs.npmjs.com/getting-started/installing-node.

Installing Project Dependencies

If NPM is installed, we can continue to load all libraries by using this tool. The easiest way is to create a new file *package.json* in a new project directory and insert the following JSON code:

The package.json file contains all dependencies of the project

```
{
  "name": "app02-01",
  "version": "1.0.0",
  "scripts": {
    "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "lite": "lite-server",
    "postinstall": "typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "license": "ISC",
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",
    "@angular/upgrade": "2.0.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23",
    "angular2-in-memory-web-api": "0.0.20",
    "bootstrap": "^3.3.6"
```

```
30    },
31    "devDependencies": {
32      "concurrently": "^2.2.0",
33      "lite-server": "^2.2.2",
34      "typescript": "^2.0.2",
35      "typings":"^1.3.2"
36    }
37  }
```

As you can see, the *package.json* file is not only containing dependencies, but also some general project information (*name*, *version*) and a *scripts* section. Within the *scripts* section we are defining command shortcuts which can be executed via the *npm* command. We will get back to that section and see how we can make use of these commands at the end of this chapter. The dependencies in *package.json* are split up into two sections: *dependencies* and *devDependencies*. The first section contains all dependencies which are used in our application to implement the functionality we want to have. For example you can find entries for the Angular and the Bootstrap framework here. In contrast, the *devDependencies* section is listing the dependencies which are required at development time only. In the example from above you can see that four development dependencies are listed: *concurrently*, *lite-server*, *typescript* and *typings*.

- The *typescript* package contains the TypeScript compiler that enables us to use TypeScript elements in our code. The compiler runs before the application files are sent to the browser and transforms TypeScript to valid JavaScript code, so that the browser is able to interpret our scripts.
- The *lite-server* package contains a lightweight development *Node.js* server that serves a web app, opens it in the browser, refreshes when html or javascript change, injects CSS changes using sockets, and has a fallback page when a route is not found.
- With the *concurrently* package installed, we can execute two *npm* commands at the same time. This is great, especially for Angular 2, because we always need to perform two steps: run the TypeScript compiler and then execute the *lite-server* to deliver our application to the browser. Instead of opening up two terminals and executing the two commands seperatly, *concurrently* lets us do this in one step by using the *concurrent* command.
- The *typings* package installs the *typings* command which helps to manage TypeScript definition files for your project. The command uses the typings.json configuration file.

Now, that all of the dependencies of our project are listed in *package.json* we can perform the installation by only executing one single command in the project directly:

```
$ npm install
```

That's it! No more steps needed. The command will read in the dependencies defined in *package.json* and will automatically install one after another in the *node_modules* subfolder in your project.

TypeScript Compiler Configuration

After having installed all of the project dependencies via *npm* there is one more step to complete before actually starting to write code. As we would like to make use of TypeScript in our project we need to store a TypeScript compiler configuration file in our project.

Before attaching the TypeScript configuration to our project, let's first take a look at what TypeScript is and why do we want to make use of it in our project? TypeScript is a superset of ECMAScript 2015 (aka ECMAScript 6) which introduces types. Of course you can also write Angular 2 projects without using TypeScript or without using features which are new in ECMAScript 2015. Using types in your JS code is very easy. E.g. a declaration of a variable of type *string* can be done with the following syntax in TypeScript:

```
todo: string;
```

This TypeScript configuration file is named *tsconfig.json* and is stored in root project directory. So create that file and insert the following JSON code:

The tsconfig.json file contains TypeScript compiler configuration parameters

```
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "module": "commonjs",
5      "moduleResolution": "node",
6      "sourceMap": true,
7      "emitDecoratorMetadata": true,
8      "experimentalDecorators": true,
9      "removeComments": false,
10     "noImplicitAny": false
11   }
12 }
```

The JSON structure consists of two sections. The first section is named *compilerOptions* and contains command line options which are passed to the TypeScript compiler every time the program is executed. Please note, that we use the compiler option *target* to specify that we want to get our TypeScript code compiled to *ECMAScript 5*. Herewith it is ensured that the resulting JavaScript code can be interpreted by modern browsers even if they are not supporting *ECMAScript 2015* yet. If you want to have additional information about the meaning of the various TypeScript compiler options please take a look at https://github.com/Microsoft/TypeScript/wiki/Compiler-Options.

SystemJS Configuration

SystemJS is a dynamic module loader that also works with ECMAScript 2015 modules and is a good choice for Angular2 projects. If you prefer another loader you can skip SystemJS and install the loader you like most (e.g. *webpack*).

In the examples of this book we'll be using SystemJS. SystemJS needs to be configured. Therefore the file *systemjs.config.js* must be added to the root of your project and the following configuration code needs to be included:

SystemJS configuration in file systemjs.config.js

```
/**
 * System configuration for Angular 2 samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',
      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platfo\
rm-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynami\
c/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
      // other libraries
      'rxjs':                      'npm:rxjs',
      'angular2-in-memory-web-api': 'npm:angular2-in-memory-web-api',
    },
    // packages tells the System loader how to load when no filename and/or no e\
xtension
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
```

```
        defaultExtension: 'js'
      },
      'angular2-in-memory-web-api': {
        main: './index.js',
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

The configuration code is a bit complicated to read. The first think you should notice is the a map object is created and filled with properties.

Now, that we have everything installed and configured correctly we're ready to start the implementation of our first Angular 2 application. Herewith we tell SystemJS where to look when we import modules. E.g. if we're importing from *@angular* in our components we're actually importing from the *node_modules/@angular* folder.

The next thing the configuration code is doing is to register all packages which may be used in the project. The packages are stored in the *packages* object.

Finally, both configuration objects (*map* and *packages*) are added the *config* object. As a parameter this object is passed to the call of *System.config*.

Implementation

First of all, we need to implement the main HTML file of the project and include the external libraries needed. Create the *index.html* in the root project folder and insert the following HTML code:

Basic structure of file index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">
    <!— 1. Load libraries -->
    <!— Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
```

```
13
14      <!– Bootstrap -->
15      <script src="node_modules/bootstrap/dist/js/bootstrap.min.js"></script>
16      <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="styleshe\
17  et">
18
19      <!– 2. Configure SystemJS -->
20      <script src="systemjs.config.js"></script>
21      <script>
22        System.import('app').catch(function(err){ console.error(err); });
23      </script>
24    </head>
25
26    <body>
27      <div class="container">
28        <h1>Learning Angular 2 <small>Chapter 02</small></h1>
29        <my-app>Loading ...</my-app>
30      </div>
31    </body>
32  </html>
```

As you can see we're using various `<script>` tags point to external libraries in subfolder *node_modules*. These are the libraries which we installed via *NPM* in the last section and now include in our web application to make use of them. Furthermore you find an additional script element in the header section which included the SystemJS configuration file which we've created in the previous step.

Furthermore the *app* module is imported and run.

Implementing AppModule

Create the project subfolder *app* and start by using the following code to implement the application module for our application in file *app/app.module.ts*:

AppModule implementation in file app.module.ts

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { AppComponent } from './app.component';
4
5  @NgModule({
6    declarations: [AppComponent],
7    imports: [BrowserModule],
8    bootstrap: [AppComponent],
9  })
10 export class AppModule {}
```

By using Angular 2 modules it is possible to consolidate components, directives and pipes into cohesive blocks of functionality. Each Angular 2 application has at least one module: the application module or root module. This application module has the conventional name of *AppModule* and is implemented in the file *app.module.ts*. As every module the application module is implemented by using a class and attaching the *@NgModule* decorator as you can see in the code listing. This decorator is imported from *@angular/core*. Within the decorator block you can find the following properties which are used for configuration:

- *declarations*: The array which is assigned to the *declarations* property contains all components, directives and pipes which belong to that module.
- *imports*: The imports array is used to list all dependencies (other modules) which needs to be made available to the module. Every application module needs to import *BrowserModule*. By importing this standard module we get access to browser specific renderers and Angular 2 standard directives like *ngIf* and *ngFor*.
- *bootstrap*: The *bootstrap* property is only used in application / root modules and contains the component of the module which should be loaded first when bootstrapping the app. The example from above defines that *AppComponent* should be loaded first. Of course this is no surprise as this is the only component of our sample app.

Furthermore a module definition can also contain the following properties:

- *exports*: This property can be used when declaring function modules. Function modules can be imported by the application module or other function modules and comprise components, directives and pipes (and even services) which belong to a certain functionality. The *exports* property contains a list of components, directives and pipes the importing module can use. In fact that is the module's public API.
- *providers*: A list of dependency injection providers (e.g. service classes) which are part of the module. These providers are automatically registered with the root injector of the module's execution context. Don't be confused by the terms *dependency injection*, *providers* and *injectors*. We'll cover those things in chapter chapter 8 in detail.

Implementing AppComponent

Next, use the following code to implement the basic structure of our main Angular 2 component in file *app/app.component.ts*:

Main Angular component in file app.component.ts

```
1  import {Component} from '@angular2/core'
2
3  @Component({
4    selector: 'my-app',
5    template: ``
6  })
7  export class AppComponent {
8    constructor() {
9    }
10 }
```

The first line of the code shows an ECMAScript 2015 import statement which is used to include the standard Angular *Component* decorator. With the decorator available, the Angular 2 component is implemented as a class. The class only contains an empty constructor and the decorator is added (*@Component*). If you are familiar with Angular 1.x you can compare components with the concept of directives in Angular 1.x. Every Angular 2 application is built by implementing components. Components are organized in a hierarchical way, so that the Angular 2 application always starts with one main component. Further components are used by the main components or by child components of the main components, so that we end up with a tree of components in an typical Angular 2 application. Inside the *@Component*-Annotation we use the *selector* property to define the name of the HTML-Element which should be used to include this component. In this current example we set the value for the *selector* property to 'my-app'. With this setting in place we are able to use the component in our HTML document like so:

```
<my-app></my-app>
```

Furthermore the *@Component* annotation is containing the *template* properties. The value which is assigned to the template property is an empty string. In the next step we will replace the empty string with the HTML template code which is needed to generate the output of the component.

Implementing the View

The template code of a component is used to generate the component's HTML output. The code is assigned to the *template* property of the *@Component* annotation of the component's class. The value which is assigned is just an empty string so far. We will change that now to:

Template code of main component

```
template: `
  <h4>Todos List</h4>
  <h5>Number of Todos: <span class="badge">{{todos.length}}</span></h5>
  <ul class="list-group">
    <li *ngFor="let todo of todos" class="list-group-item">
      {{todo}}
    </li>
  </ul>
  <div class="form-inline">
    <input class="form-control" #todotext>
    <button class="btn btn-default" (click)="addTodo(todotext.value)">Add Todo</\
button>
  </div>
`
```

The value which is assigned to the *template* property is now a string containing the HTML template code which should be used as the component's view. We are using the new EcmaScript 2015 template string feature to assign a multiline string containing the template code. In order to make use of this feature the multiline string has to start and end with backticks (‘). The first information which is printed out by the template is the total number of *todo* items available. All *todo* items will be stored in an array of strings named *todos*. To get the total number of items we can use the *length* property of the *todos* array. To access the property value the expression is included in the template by using double curly braces:

```
{{todos.length}}
```

This is the syntax which is used to include the result of an expression in the HTML output which is generated when the component is used in the application. Next, a list of all *todo* items is generated by using a *<ul>* element. Each list item is represented by a *<li>* child-element. The *ngFor* directive helps us to generate *<il>* elements for each data item by iterating over the todos array:

```
<li *ngFor="let todo of todos" class="list-group-item">
  {{todo}}
</li>
```

The syntax which is used here is `*ngFor="let todo of todos"` and is applied on the `<li>` element. For every element found in todos array the output of the element (and its content) is repeated. For each iteration the current *todo* item is available through the variable *todo*, so that we can access the todo text directly by using again the expression syntax `{{todo}}`. Please be aware: the *ngFor* directive is available because we've imported *BrowserModule* in our root application module *AppModule*. This makes Angular 2 standard directives available in all templates of components

listed in the declarations property of the *@NgModule* decorator. After the list of todos the template contains an input element and a button so that the user is able to add new todo elements. Notice that the *<input>* element is containing the attribute *#todotext*. This is the syntax which is used in Angular 2 to create a variable *todotext* which is accessible in the component class and gives us access to the HTML element. The `<button>` element is containing another special attribute: *(click)*. This notation is used in Angular 2 to describe events. The value which is assigend to the click event is a string containing an event handler method named *addTodo*. This method is not yet available and will be implemented in the next step. The *addTodo* method gets one parameter: *todotext.value*. Here we are using the previously assigned variable *todotext* and access the user input value by using the property *value*.

So the view implementation for our component is ready now. The last thing we have to do is to adapt the *MyApp* component class a little bit to meet the expectations the template code has set. First we need to have an *todos* array in place containing all our *todo* items:

Initializing the todos array in AppComponent

```
1  export class AppComponent{
2    todos: Array<string>;
3    constructor() {
4      this.todos = ["Todo 1", "Todo 2", "Todo 3"];
5    }
6    addTodo(todo: string) {
7      this.todos.push(todo);
8    }
9  }
```

As you can see we are using TypeScript here to declare a typed array of type `Array<string>`. The class constructor is used to initialize the *todos* array with three items. If you try to assign anything other as a string to the array the TypeScript compiler will run into an error because the type check fails. The *addTodo* method is implemented in class *AppComponent*, so that we can use this method to handle the click event of the button. Again, we are using TypeScript to declare that the *todo* parameter must be of type *string*. The method takes this string and calls the *push* method to extend the *todos* array with this new item. Another thing to notice here is that the *class* keyword is preceded by *export*. Herewith we make the *AppComponent* available as a module, so that we can import *AppComponent* in another file by using the SystemJS module loader.

Loading Components and Bootstrapping the Application

Now we have nearly all pieces together to get our Angular 2 application up and running except one. By using the method *System.import* in *index.html* we have included the file *app/main.js*. Now we need to create the corresponding TypeScript file *app/main.ts* and add the missing code pieces which are needed to load the root application module and startup the Angular application by loading the first component from that module. Create the file and insert the following lines of code:

Importing modules and bootstrapping application in file main.ts

```
1 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2 import { AppModule } from './app.module';
3
4 platformBrowserDynamic().bootstrapModule(AppModule);
```

As you can see the file *main.ts* is quite comprehensive, only three lines of code are needed. First we're importing *platformBrowserDynamic* and *AppModule*. By importing *platformBrowserDynamic* we get access to the Just-In-Time (JIT) compiler which compiles the application in the browser. By executing

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

we're telling the Just-In-Time compiler to start the application by first compiling and loading *AppModule*. To determine what needs to be displayed to the user, the compiler looks for the value which has been passed to the AppModule's *bootstrap* property:

```
bootstrap: [AppComponent]
```

In our case we've been passing *AppComponent*. Now this tells the Just-In-Time compiler that *AppComponent* needs to loaded first when bootstrapping with the *AppModule*. The code of the component is compiled to JavaScript which can be interpreted by the Browser directly and the corresponding template is used to generate the HTML output which is displayed to the user.

Running the Application

Running the application basically consists of two steps:

- Running the TypeScript compiler to convert TypeScript to JavaScript code
- Running a web server to serve the project files

Luckily we have already set up *NPM* scripts for those tasks in *package.json*:

```
...
"scripts": {
  "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
  "lite": "lite-server",
  "postinstall": "typings install",
  "tsc": "tsc",
  "tsc:w": "tsc -w",
  "typings": "typings"
}
...
```

Here you can see that the *scripts* section of *package.json* contains *NPM* scripts definitions. Most of the NPM scripts are executed by using the *npm* command in the following way:

```
$ npm run [script]
```

Some scripts (like start) don't rquire the run keyword and can be executed in the following way:

```
$ npm start
```

In the following you can find an overview of the NPM scripts defined:

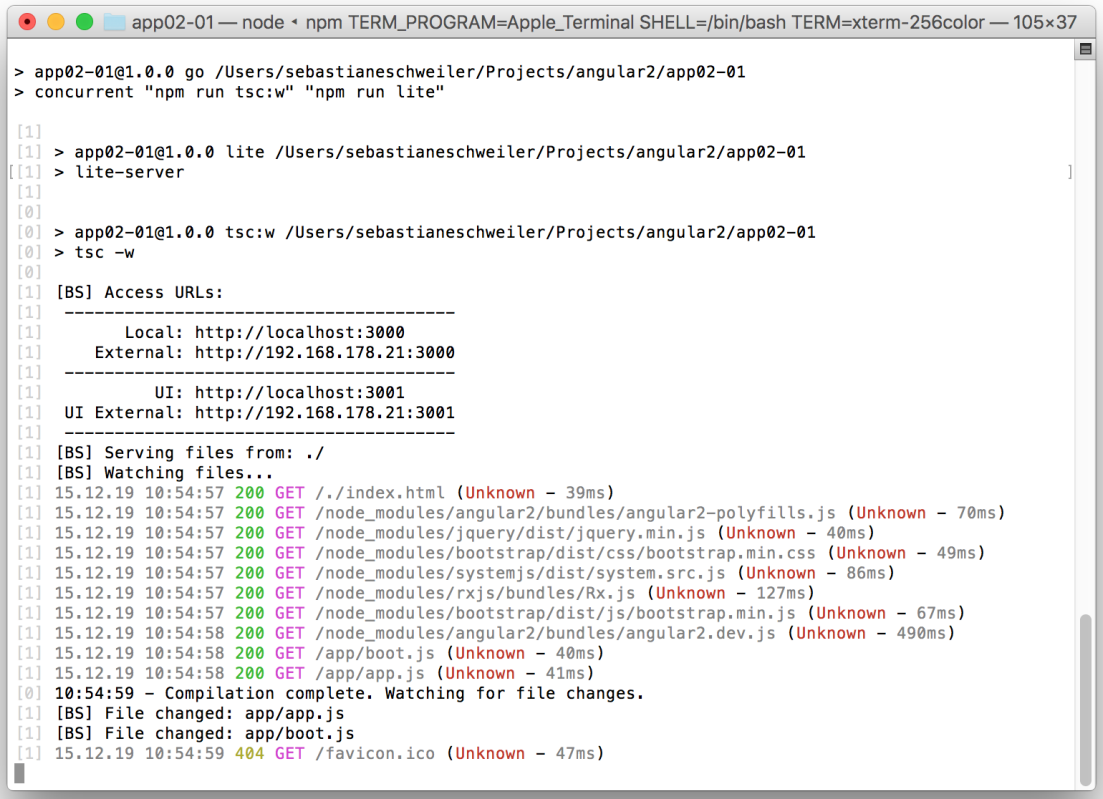
- *npm run tsc* - Runs the TypeScript compiler one time.
- *npm run tsc:w* - Runs the TypeScript compiler with option *-w* which enables the watch mode. After the initial compilation the compiler process stays active and listens for file changes.
- *npm run lite* - Runs the static file web server *lite-server*.
- *npm start* - Uses concurrently to run the *tsc:w* script and the *lite* script at the same time.
- *npm run typings* - Runs the typings tool
- *npm run postinstall* - The script is called automatically by NPM after having complete installation of NPM packages successfully. This scripts installs the TypeScript definition files defined in *typings.json*.

Start the sample project by entering the project root folder on the command line and execute the NPM start script:

In order to start our project with one single command (TypeScript compiler and *lite-server* at the same time) we simply need to execute:

```
$ npm start
```

Executing this command should deliver a result similar to what you can see in the following screenshot:



Sample application of chapter 2

The output shows that both, the TypeScript compiler and the web server, are running in parallel. Every line which is printed on the console is prefixed with either *[0]* or *[1]*. With this information available you can see from which process the outputted line is generated. Everything which starts with *[0]* is outputted from the TypeScript server which is executed in watch mode. Everything which starts with *[1]* is generated by *lite-server*.

The application is now up and running and can be opened by accessing URL *http://localhost:3000* in your browser. If you change HTML or TypeScript code in your project changes are processed by the running TypeScript compiler in the background and pushed to the browser automatically without the need of doing a manual refresh of the webpage.

To stop both processes simply hit keys CTRL+C in your terminal windows.

GitHub

The source code of this chapter is available on GitHub: https://github.com/seeschweiler/angular2-book-chapter02