sitepoint

LEARN ANGULAR: RELATED TOOLS & SKILLS

LEVEL UP YOUR ANGULAR SKILLS

Learn Angular: Related Tools & Skills

Copyright © 2018 SitePoint Pty. Ltd.

Ebook ISBN: 978-0-6483315-9-9

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You’ll find a stack of information on JavaScript, PHP, design, and more.

Preface

So, why Angular? Well, because it’s supported on various platforms (web, mobile, desktop native), it’s powerful, modern, has a nice ecosystem, and it’s just cool. Not convinced? Let's be a bit more eloquent, then:

- **Angular presents you not only the tools but also design patterns to build your project in a maintainable way.** When an Angular application is crafted properly, you don’t end up with a tangle of classes and methods that are hard to modify and even harder to test. The code is structured conveniently and you won’t need to spend much time in order to understand what is going on.
- **It’s JavaScript, but better.** Angular is built with TypeScript, which in turn relies on JS ES6. You don’t need to learn a totally new language, but you still receive features like static typing, interfaces, classes, namespaces, decorators etc.
- **No need to reinvent the bicycle.** With Angular, you already have lots of tools to start crafting the application right away. You have directives to give HTML elements dynamic behavior. You can power up the forms using `FormControl` and introduce various validation rules. You may easily send asynchronous HTTP requests of various types. You can set up routing with little hassle. And there are many more goodies that Angular can offer us!
- **Components are decoupled.** Angular strived to remove tight coupling between various components of the application. Injection happens in NodeJS-style and you may replace various components with ease.
- **All DOM manipulation happens where it should happen.** With Angular, you don’t tightly couple presentation and the application’s logic making your markup much cleaner and simpler.
- **Testing is at the heart.** Angular is meant to be thoroughly tested and it supports both unit and end-to-end testing with tools like Jasmine and Protractor.
- **Angular is mobile and desktop-ready**, meaning you have one framework for multiple platforms.
- **Angular is actively maintained** and has a large community and ecosystem. You can find lots of materials on this framework as well as many useful third-party tools.

So, we can say that Angular is not just a framework, but rather a *platform* that empowers developers to build applications for the web, mobile, and the desktop.

This book provides an overview of some essential Angular tools, as well as outlining some must-have TypeScript tips.

Who Should Read This Book?

This book is for all front-end developers who want to get proficient with Angular and its related tools. You’ll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1> <p>It was a lovely day for a walk in the park. The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() { ⋮ new_variable = "Hello"; }
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web- ➥design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: The Ultimate Angular CLI Reference Guide

by Jurgen Van de Moere

In this article, we'll have a look at what Angular CLI is, what it can do for you, and how it performs some of its magic behind the scenes. Even if you already use Angular CLI, this article can serve as a reference to better understand its inner workings.

Technically, you're not required to use Angular CLI to develop an Angular application, but its many features can highly improve the quality of your code and save you a lot of time along the way.

Some History

On September 15, 2016, Angular Final was released.

Where AngularJS 1.x was limited to a framework, Angular has grown into an ambitious platform that allows you to develop fast and scalable applications across all platforms such as web, mobile web, native mobile and even native desktop.

With this transition to a platform, tooling has become more important than ever. However, setting up and configuring tooling is not always easy. To make sure Angular developers can focus on building applications with as little friction as possible, the Angular team is putting a lot of effort into providing developers with a high-quality development toolset.

Part of that toolset are close integrations with a wide array of IDEs and editors. Another part of that toolset is Angular CLI.

So let's get started!

What Is Angular CLI?

Angular CLI is a command-line interface (CLI) to automate your development workflow. It allows you to:

- create a new Angular application
- run a development server with LiveReload support to preview your application during development
- add features to your existing Angular application
- run your application's unit tests
- run your application's end-to-end (E2E) tests
- build your application for deployment to production.

Before we have a look at each of the above in detail, let's first see how you can install Angular CLI.

Prerequisites

Before you can use Angular CLI, you must have Node.js 6.9.0 and npm 3.0.0 or higher installed on your system.

You can download the latest version of Node.js for your operating system and consult the latest installation instructions on the official Node.js website.

If you already have Node.js and npm installed, you can verify their version by running:

```
$ node -v # => displays your Node.js version
$ npm -v # => displays your npm version
```

Once you have Node.js installed, you can use the `npm` command to install TypeScript:

```
$ npm install -g typescript@2.2.0
```

Although TypeScript is technically not an absolute requirement, it’s highly recommended by the Angular team, so I recommend you install it to make working with Angular as comfortable as possible.

Now that you have Node.js and TypeScript installed, you can install Angular CLI.

Installing Angular CLI

To install Angular CLI, run:

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

This will install the `ng` command globally on your system.

To verify whether your installation completed successfully, you can run this:

```
$ ng version
```

This displays the version you have installed:

```
@angular/cli: 1.0.0
node: 6.10.0
os: darwin x64
```

Now that you have Angular CLI installed, let’s use it to create a new application.

Creating a New Angular Application

There are two ways to create a new application using Angular CLI:

- `ng init`: create a new application in the current directory
- `ng new`: create a new directory and run `ng init` inside the new directory.

So `ng new` is similar to `ng init`, except that it also creates a directory for you.

Assuming you haven’t created a directory yet, let’s use `ng new` to create a new project:

```
$ ng new my-app
```

Behind the scenes, the following happens:

- a new directory `my-app` is created
- all source files and directories for your new Angular application are created based on the name you specified (`my-app`) and best-practices from the official Angular Style Guide
- npm dependencies are installed
- TypeScript is configured for you
- the Karma unit test runner is configured for you
- the Protractor end-to-end test framework is configured for you
- environment files with default settings are created.

You’ll learn more about each of these aspects in the following sections.

At this point you have a working Angular application and your new directory `my-app` looks like this:

```
.
├── README.md
├── e2e
│   ├── app.e2e-spec.ts
│   ├── app.po.ts
│   └── tsconfig.e2e.json
├── karma.conf.js
├── package.json
├── protractor.conf.js
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts

│   │   ├── app.component.ts

│   │   └── app.module.ts

│   ├── assets

│   ├── environments

│   │   ├── environment.prod.ts

│   │   └── environment.ts

│   ├── favicon.ico

│   ├── index.html

│   ├── main.ts

│   ├── polyfills.ts

│   ├── styles.css

│   ├── test.ts
│   ├── tsconfig.app.json

│   ├── tsconfig.spec.json

│   └── typings.d.ts

├── tsconfig.json

└── tslint.json
```

Available Options

- `--dry-run`: boolean, default `false`, perform dry-run so no changes are written to filesystem
- `--verbose`: boolean, default `false`
- `--link-cli`: boolean, default `false`, automatically link the `@angular/cli` package (more info)
- `--skip-install`: boolean, default `false`, skip `npm install`
- `--skip-git`: boolean, default `false`, don’t initialize git repository
- `--skip-tests`: boolean, default `false`, skip creating tests
- `--skip-commit`: boolean, default `false`, skip committing the first git commit
- `--directory`: string, name of directory to create, by default this is the same as the application name
- `--source-dir`: string, default `'src'`, name of source directory

- `--style`: string, default `'css'`, the style language to use (`'css'`, `'less'` or `'scss'`)

- `--prefix`: string, default `'app'`, the prefix to use when generating new components

- `--mobile`: boolean, default `false`, generate a Progressive Web App application (see section on upcoming features)

- `--routing`: boolean, default `false`, add module with routing information and import it in main app module

- `--inline-style`: boolean, default `false`, use inline styles when generating the new application

- `--inline-template`: boolean, default `false`, use inline templates when generating the new application.

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Let’s see how you can start your application so you can see it in action.

Running Your Application

To preview your new application in your browser, navigate to its directory:

```
$ cd my-app
```

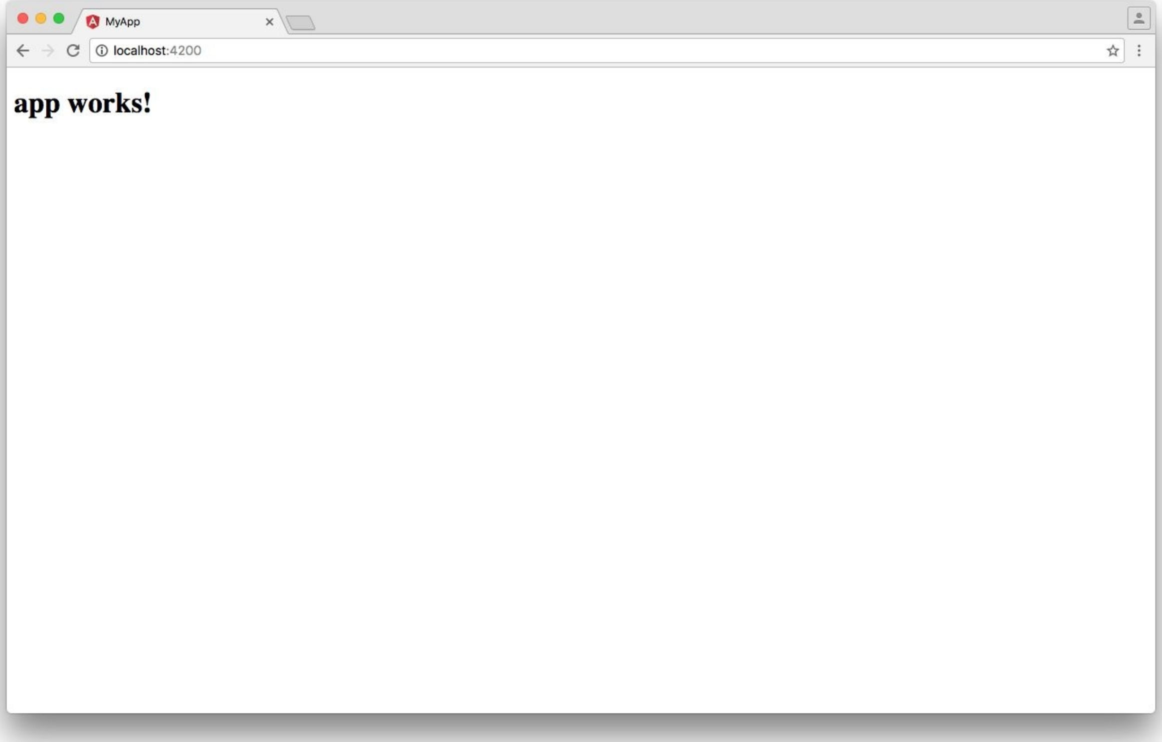
and run:

```
$ ng serve
```

to start the built-in development server on port 4200:

```
** NG Live Development Server is running on http://localhost:4200
**
Hash: 09fb2ad840c1472e5885
Time: 6230ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map
(polyfills) 158 kB {4}
➡[initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.js.map (main) 3.62 kB {3}
[initial]
➡[rendered]
chunk    {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77
kB {4} [initial]
➡[rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.37
MB [initial]
➡[rendered]
chunk    {4} inline.bundle.js, inline.bundle.js.map (inline) 0
bytes [entry]
➡[rendered]
webpack: Compiled successfully.
```

You can now navigate your favorite browser to http://localhost:4200/ to see your application in action:



app works!

Behind the scenes, the following happens:

1. Angular CLI loads its configuration from `.angular-cli.json`
2. Angular CLI runs Webpack to build and bundle all JavaScript and CSS code
3. Angular CLI starts [Webpack dev server](#) to preview the result on `localhost:4200`.

Notice that the `ng serve` command doesn't exit and return to your terminal prompt after step 3.

Instead, because it includes LiveReload support, the process actively watches your `src` directory for file changes. When a file change is detected, step 2 is repeated and a notification is sent to your browser so it can refresh automatically.

To stop the process and return to your prompt, press `ctrl-c`.

Adding Features to Your Angular Application

You can use the `ng generate` command to add features to your existing application:

- `ng generate class my-new-class`: add a class to your application
- `ng generate component my-new-component`: add a component to your application
- `ng generate directive my-new-directive`: add a directive to your application
- `ng generate enum my-new-enum`: add an enum to your application
- `ng generate module my-new-module`: add a module to your application
- `ng generate pipe my-new-pipe`: add a pipe to your application
- `ng generate service my-new-service`: add a service to your application

The `generate` command and the different sub-commands also have shortcut notations, so the following commands are similar:

- `ng g cl my-new-class`: add a class to your application
- `ng g c my-new-component`: add a component to your application
- `ng g d my-new-directive`: add a directive to your application
- `ng g e my-new-enum`: add an enum to your application
- `ng g m my-new-module`: add a module to your application
- `ng g p my-new-pipe`: add a pipe to your application
- `ng g s my-new-service`: add a service to your application.

Each of the different sub-commands performs a different task and offers different options and parameters.

Let’s have a look at each of them.

Adding a new class

To add a class called `UserProfile`, run:

```
$ ng generate class user-profile
installing component
  create src/app/user-profile.ts
```

Angular CLI will automatically adjust the letter case of the file name and class name for you, so the following commands have the same effect:

```
# All three commands are equivalent
$ ng generate class user-profile
$ ng generate class userProfile
$ ng generate class UserProfile
```

Behind the scenes, the following happens:

- a file `src/app/user-profile.ts` is created that exports an empty class named `UserProfile`

Available options

- `--spec`: boolean, default `false`, generate spec file with unit test

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Generate class 'UserProfile'
$ ng generate class user-profile

# Generate class 'UserProfile' with unit test
$ ng generate class user-profile --spec
```

Adding a new component

To add a component with a selector `app-site-header`, run:

```
$ ng generate component site-header
installing component
  create src/app/site-header/site-header.component.css
  create src/app/site-header/site-header.component.html
  create src/app/site-header/site-header.component.spec.ts
  create src/app/site-header/site-header.component.ts
  update src/app/app.module.ts
```

Angular CLI will automatically adjust the letter case of the file name and component name for you and apply the prefix to the component selector, so the following commands have the same effect:

```
# All three commands are equivalent
$ ng generate component site-header
$ ng generate component siteHeader
$ ng generate component SiteHeader
```

Behind the scenes, the following happens:

- a directory `src/app/site-header` is created
- inside that directory four files are generated:
  - a CSS file for the component styles
  - an HTML file for the component template
  - a TypeScript file with a component class named `SiteHeaderComponent` and selector `app-site-header`
  - a spec file with a sample unit test for your new component
- `SiteHeaderComponent` is added as a declaration in the `@NgModule` decorator of the nearest module, in this case the `AppModule` in `src/app/app.module.ts`.

Available options

- `--flat`: boolean, default `false`, generate component files in `src/app` instead of `src/app/site-header`
- `--inline-template`: boolean, default `false`, use an inline template instead of a separate HTML file
- `--inline-style`: boolean, default `false`, use inline styles instead of a separate CSS file
- `--prefix`: boolean, default `true`, use prefix specified in `.angular-cli.json` in component selector
- `--spec`: boolean, default `true`, generate spec file with unit test
- `--view-encapsulation`: string, specifies the view encapsulation strategy
- `--change-detection`: string, specifies the change detection strategy.

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Generate component 'site-header'
```

```
$ ng generate component site-header

# Generate component 'site-header' with inline template and inline styles
$ ng generate component site-header --inline-template --inline-style
```

Adding a new directive

To add a directive with a selector `appAdminLink`, run:

```
$ ng generate directive admin-link
installing directive
  create src/app/admin-link.directive.spec.ts
  create src/app/admin-link.directive.ts
  update src/app/app.module.ts
```

Angular CLI will automatically adjust the letter case of the file name and directive name for you and apply the prefix to the directive selector, so the following commands have the same effect:

```
# All three commands are equivalent
$ ng generate directive admin-link
$ ng generate directive adminLink
$ ng generate directive AdminLink
```

Behind the scenes, the following happens:

- a file `src/app/admin-link.directive.ts` is created that exports a directive named `AdminLinkDirective` with a selector `appAdminLink`
- a file `src/app/admin-link.directive.spec.ts` is created with a unit test for the directive
- `AdminLinkDirective` is added as a declaration in the `@NgModule` decorator of the nearest module, in this case the `AppModule` in `src/app/app.module.ts`.

Available options

- `--flat`: boolean, default `true`, generate directive files in `src/app` instead of `src/app/admin-link`
- `--prefix`: boolean, default `true`, use prefix specified in `.angular-cli.json` in directive selector

- `--spec`: boolean, default `true`, generate spec file with unit test

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Generate directive 'adminLink'
$ ng generate directive admin-link

# Generate directive 'adminLink' without unit test
$ ng generate directive admin-link --spec=false
```

Adding a new enum

To add an enum called `Direction`, run:

```
$ ng generate enum direction
installing enum
  create src/app/direction.enum.ts
```

Angular CLI will automatically adjust the letter case of the file name and enum name for you, so the following commands have the same effect:

```
# Both commands are equivalent
$ ng generate enum direction
$ ng generate enum Direction
```

Behind the scenes, the following happens:

- a file `src/app/direction.enum.ts` is created that exports an enum named `Direction`

Available options

There are no command line options available for this command.

Adding a new module

To add a new module to your application, run:

```
$ ng generate module admin
installing module
  create src/app/admin/admin.module.ts
```

Behind the scenes, the following happens:

- a directory `src/app/admin` is created
- an `AdminModule` module is created inside `src/app/admin/admin.module.ts`.

Notice that the `AdminModule` module is not added automatically to your main module `AppModule` in `src/app/app.module.ts`. It's up to you to import the module where you need it.

To import your new module in another module, you can specify is as an import in an `@NgModule` definition. For example:

```
import { AdminModule } from './admin/admin.module';

@NgModule({
  // ...
  imports: [
    AdminModule
  ]
})
export class AppModule { }
```

Available options

- `--routing`: boolean, default `false`, generate an additional module `AdminRoutingModule` with just the routing information and add it as an import to your new module
- `--spec`: boolean, default `false`, add `src/app/admin/admin.module.spec.ts` with a unit test that checks whether the module exists.

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Add module 'admin'
```

```
$ ng generate module admin

# Add module 'admin' with additional module containing routing
information
$ ng generate module admin --routing
```

Adding a new pipe

A pipe is the Angular equivalent of a filter in AngularJS 1.x and allows you to transform a displayed value within a template.

To add a pipe with a name `convertToEuro`, run:

```
$ ng generate pipe convert-to-euro
installing pipe
  create src/app/convert-to-euro.pipe.spec.ts
  create src/app/convert-to-euro.pipe.ts
  update src/app/app.module.ts
```

Angular CLI will automatically adjust the letter case of the file name and pipe name for you, so the following commands have the same effect:

```
# All three commands are equivalent
$ ng generate pipe convert-to-euro
$ ng generate pipe convertToEuro
$ ng generate pipe ConvertToEuro
```

Behind the scenes, the following happens:

- a file `src/app/convert-to-euro.pipe.ts` is created that exports a pipe class named `ConvertToEuroPipe`
- a file `src/app/convert-to-euro.pipe.spec.ts` is created with a unit test for the pipe
- `ConvertToEuroPipe` is added as a declaration in the `@NgModule` decorator of the nearest module, in this case the `AppModule` in `src/app/app.module.ts`.

Available options

- `--flat`: boolean, default `true`, generate component files in `src/app` instead of `src/app/site-header`
- `--spec`: boolean, default `true`, generate spec file with unit test.

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Generate pipe 'convertToEuro' with spec and in /src/app directory
$ ng generate pipe convert-to-euro

# Generate pipe 'convertToEuro' without spec and in
/src/app/convert-to-euro
# directory
$ ng generate pipe convert-to-euro --spec=false --flat=false
```

Adding a new service

To add a service with a dependency injection token `BackendApiService`, run:

```
$ ng generate service backend-api
installing service
  create src/app/backend-api.service.spec.ts
  create src/app/backend-api.service.ts
  WARNING Service is generated but not provided, it must be
provided to be used
```

Angular CLI will automatically adjust the letter case of the file name and pipe name for you, so the following commands have the same effect:

```
# All three commands are equivalent
$ ng generate service backend-api
$ ng generate service backendApi
$ ng generate service BackendApi
```

Behind the scenes, the following happens:

- a file `src/app/backend-api.service.ts` is created that exports a service class named `BackendApiService`
- a file `src/app/backend-api.service.spec.ts` is created with a unit test for your new service.

Notice how Angular CLI warns that the service is generated but not provided anywhere yet. It’s up to you to register the service as a provider by adding it to the `providers: []` array where you need it (e.g. in a module or component). For example:

```
import { BackendApiService } from './backend-api.service';

@NgModule({
  // ...
  providers: [BackendApiService],
  bootstrap: [AppComponent]
})
```

Available options

- `--flat`: boolean, default `true`, generate service file in `src/app` instead of `src/app/backend-api`
- `--spec`: boolean, default `true`, generate spec file with unit test

Run `$ ng generate --help` to see all available options of your locally installed Angular CLI.

Examples:

```
# Generate service with DI token 'BackendApiService' in /src/app
directory
$ ng generate service backend-api

# Generate service with DI token 'BackendApiService' in
/src/app/backend-api
# directory
$ ng generate service backend-api --flat=false
```

Special note

Angular CLI does not just blindly generate code for you. It uses static analysis to better understand the semantics of your application.

For example, when adding a new component using `ng generate component`, Angular CLI finds the closest module in the module tree of your application and integrates the new feature in that module.

So if you have an application with multiple modules, Angular CLI will automatically integrate the new feature in the correct module, depending on the directory where you run the command from.

Running Your Unit Tests

Angular CLI automatically configures the Karma test runner for you when your application is initially created.

When adding a feature to your application, you can use the `--spec` option to specify whether you want Angular CLI to also create a corresponding `.spec.ts` file with a sample unit test for your new feature.

Spec files are created in the same directory of their corresponding feature in the `src` directory. This allows you to easily locate them when working on a feature.

Running all unit tests of your application thus implies running all unit tests specified in all files ending in `.spec.ts` in all directories inside your `src` directory.

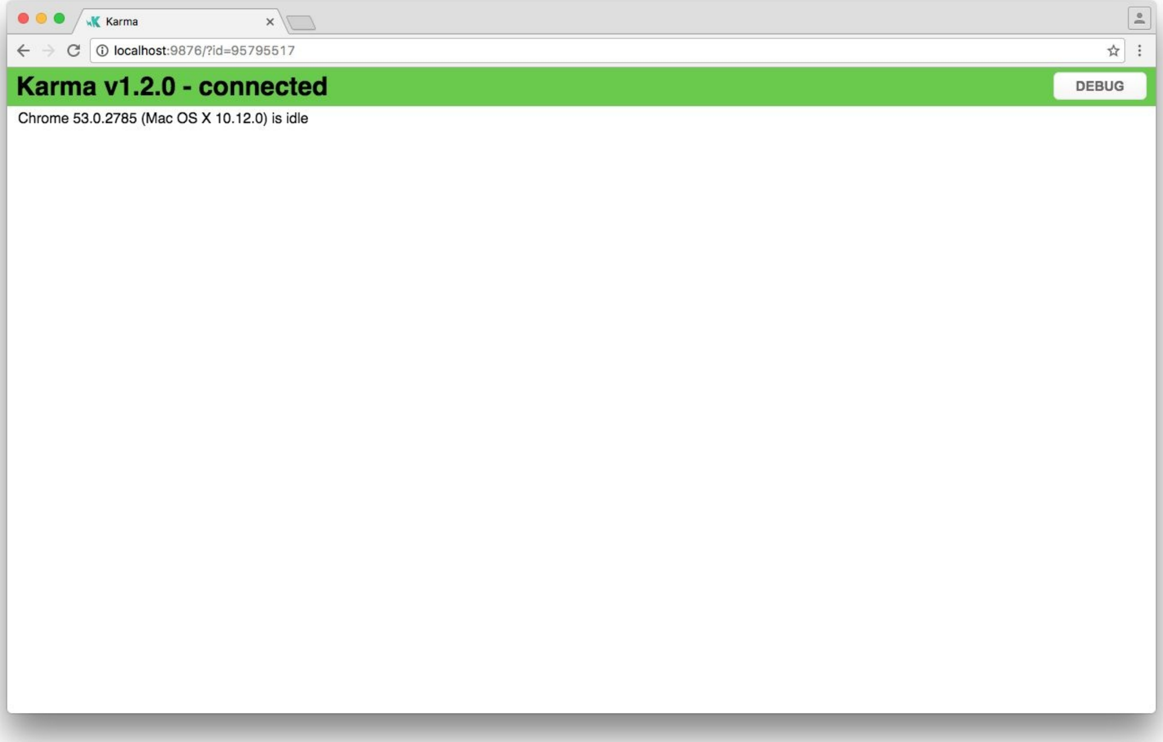
To run all unit tests, run:

```
$ ng test
```

The following output will appear in your console:

```
$ ng test
[karma]: No captured browser, open http://localhost:9876/
[karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
[launcher]: Launching browser Chrome with unlimited concurrency
[launcher]: Starting browser Chrome
[Chrome 57.0.2987 (Mac OS X 10.12.0)]: Connected on socket
4OBzlsVyIPZyE1AYAAAA with id 41455596
Chrome 57.0.2987 (Mac OS X 10.12.0): Executed 3 of 3 SUCCESS (0.132
secs / 0.121 secs)
```

A special browser instance will also be launched:



Karma v1.2.0 - connected

Chrome 53.0.2785 (Mac OS X 10.12.0) is idle

Here’s what happens behind the scenes:

1. Angular CLI loads `.angular-cli.json`.
2. Angular CLI runs Karma with the configuration specified in `.angular-cli.json`. By default this is `karma.conf.js` located in the root directory of your application.
3. Karma opens the browser specified in the Karma configuration. By default the browser is set to Google Chrome.
4. Karma then instructs the browser (Chrome) to run `src/test.ts` using the testing framework specified in the Karma config. By default this is the Jasmine framework. The file `src/test.ts` is automatically created when your application is created. It’s pre-configured to load and configure the code that’s needed to test your Angular application and run all spec files ending in `.spec.ts` in your `src` directory.
5. Karma reports the result of the test run to the console.
6. Karma watches the `src` file for changes and repeats step 4 and 5 when a file change is detected.

To end the process, you can press `ctrl-c`.

If you want to learn more about testing your Angular code, you can check out the Official Angular Testing Guide.

Running Your End-to-end (E2E) Tests

Angular CLI automatically configures Protractor for you when your application is initially created.

To run your E2E tests, run:

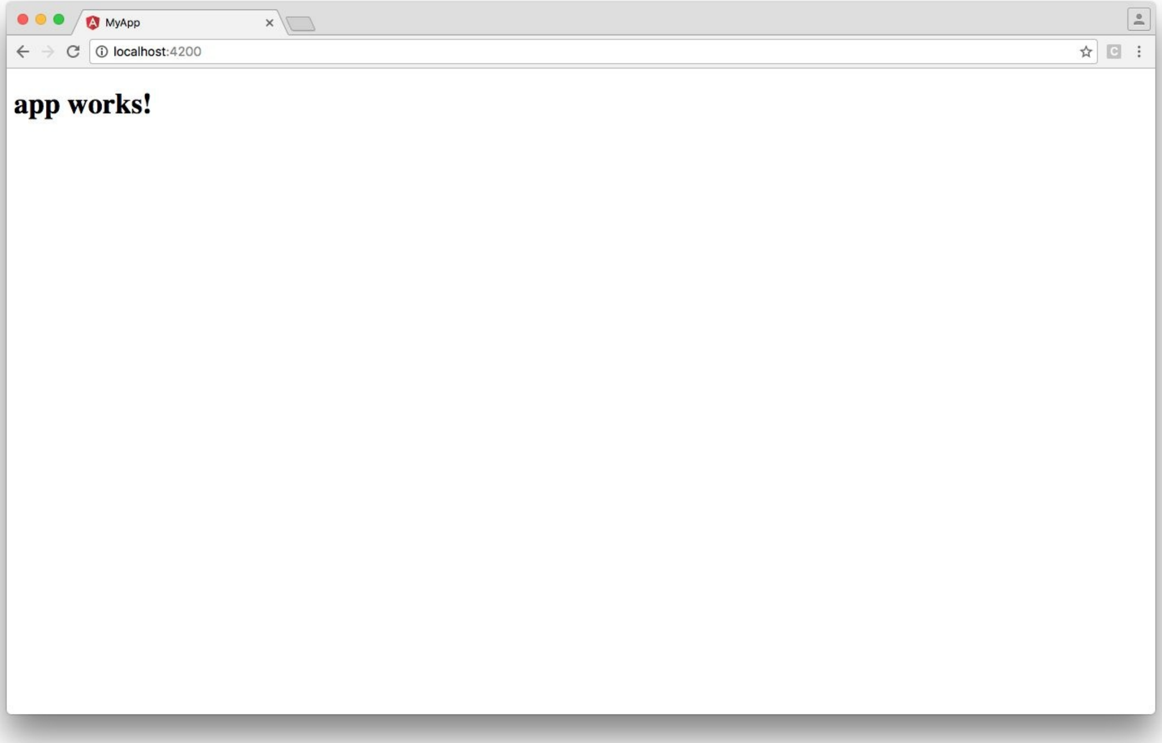
```
$ ng e2e
```

The following output will appear in your console:

```
** NG Live Development Server is running on http://localhost:49152
**
Hash: e570d23ac26086496e1d
Time: 6087ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map
(polyfills) 158 kB {4} [initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.js.map (main) 3.62 kB {3}
[initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77
kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.37
MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.js.map (inline) 0
bytes [entry] [rendered]
webpack: Compiled successfully.
I/file_manager - creating folder /Users/jvandemo/Projects/test/my-
app/node_modules/protractor/node_modules/webdriver-manager/selenium
I/downloader - curl -o /Users/jvandemo/Projects/test/my-
app/node_modules/protractor/node_modules/webdriver-
manager/selenium/chromedriver_2.29.zip
https://chromedriver.storage.googleapis.com/2.29/chromedriver_mac64.

I/update - chromedriver: unzipping chromedriver_2.29.zip
I/update - chromedriver: setting permissions to 0755 for
/Users/jvandemo/Projects/test/my-
app/node_modules/protractor/node_modules/webdriver-
manager/selenium/chromedriver_2.29
I/launcher - Running 1 instances of WebDriver
I/direct - Using ChromeDriver directly...
Spec started

  my-app App
    ✓ should display message saying app works
```

```
Executed 1 of 1 spec SUCCESS in 0.523 sec.
I/launcher - 0 instance(s) of WebDriver still running
I/launcher - chrome #01 passed
```

A special browser instance will also be launched:



app works!

Here’s what happens behind the scenes:

1. Angular CLI loads `.angular-cli.json`.
2. Angular CLI runs Protractor with the configuration specified in `.angular-cli.json`. By default this is the `protractor.conf.js` file located in the root directory of your application.
3. Protractor opens the browser specified in the Protractor configuration. By default the browser is set to Google Chrome.
4. Protractor then instructs the browser (Chrome) to run all spec files ending in `.e2e-spec.ts` in your `e2e` directory.
5. Protractor reports the result of the test run to the console.

The process then exits automatically after step 5.

If you want to learn more about E2E testing your Angular code, you can check out the Official Angular Testing Guide and the Protractor documentation.

Building Your Application for Production

Running `ng serve` builds and bundles your Angular application automatically to a virtual filesystem during development.

However, when your application is ready for production, you’ll need real files that you can deploy to your server or to the cloud.

To build and bundle your application for deployment, run:

```
$ ng build
```

The output of the command is sent to your console:

```
Hash: 59aaa9ef8eac5d46cdf8
Time: 5433ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map
(polyfills) 158 kB {4} [initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.js.map (main) 3.61 kB {3}
[initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77
kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.07
MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.js.map (inline) 0
bytes [entry] [rendered]
```

This is what happens behind the scenes:

1. Angular CLI loads its configuration from `.angular-cli.json`.
2. Angular CLI runs Webpack to build and bundle all JavaScript and CSS code.
3. The result is written to the `outDir` directory specified in your Angular CLI configuration. By default, this is the `dist` directory.

Available options

- `--aot`: enable ahead-of-time compilation
- `--base-href`: string, the base href to use in the index file
- `--environment`: string, default `dev`, environment to use
- `--output-path`: string, directory to write the output to

- `--target`: string, default `development`, environment to use
- `--watch`: boolean, default `false`, watch files for changes and rebuild when a change is detected.

Targets

Specifying a target impacts the way the build process operates. Its value can be one of the following:

- `development`: default mode, do not minify or uglify code
- `production`: minify and uglify code.

Building your application in production mode:

```
$ ng build --target=production
```

This results in bundles that are minified, uglified and have hashes in their names:

```
Hash: 4dea6adc9ac01de3c11b
Time: 5588ms
chunk    {0} polyfills.2d45a4c73c85e24fe474.bundle.js (polyfills)
158 kB {4} [initial] [rendered]
chunk    {1} main.a64b48e56248eb808195.bundle.js (main) 20.8 kB {3}
[initial] [rendered]
chunk    {2} styles.d41d8cd98f00b204e980.bundle.css (styles) 69
bytes {4} [initial] [rendered]
chunk    {3} vendor.205c7417e47c580a2c34.bundle.js (vendor) 1.1 MB
[initial] [rendered]
chunk    {4} inline.310ccba0fff49a724c8f.bundle.js (inline) 0 bytes
[entry] [rendered]
```

Environments

Environments let you specify settings to customize your application behavior.

You can define your own environments in the `.angular-cli.json` file. The default ones are:

- `source`: use settings defined in `environments/environment.ts`
- `dev`: use settings defined in `environments/environment.ts`
- `prod`: use settings defined in `environments/environment.prod.ts`.

Here, `environments/environment.ts` equals:

```
export const environment = {
  production: false
};
```

And `environments/environment.prod.ts` equals:

```
export const environment = {
  production: true
};
```

The build process will use the `dev` environment by default.

If you specify a different environment, the build process will use the corresponding environment:

```
# Uses environments/environment.ts
$ ng build

# Also uses environments/environment.ts
$ ng build --environment=dev

# Uses environments/environment.prod.ts
$ ng build --environment=prod
```

As you can see in `src/main.ts`, you can access the environment settings from your code by importing `environments/environment`:

```
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 build process will make sure the right environment is provided when you import it.

Deploying Your Application

As of February 9, 2017, the `ng deploy` command has been removed from the core of Angular CLI. Read more here.

Ejecting your application

As of v1.0, Angular CLI provides a command to decouple your application from Angular CLI.

By default, Angular CLI manages the underlying webpack configuration for you so you don’t have to deal with its complexity.

If, at any given time, you wish to manually configure Webpack and you no longer want to use Angular CLI with your Angular application, you can run: `$ ng eject`

This will generate the following output in your console:

```
====================================================================

Ejection was successful.

To run your builds, you now need to do the following commands:
    - "npm run build" to build.
    - "npm run test" to run unit tests.
    - "npm start" to serve the app using webpack-dev-server.
    - "npm run e2e" to run protractor.

Running the equivalent CLI commands will result in an error.

====================================================================

Some packages were added. Please run "npm install".
```

This is what happens behind the scenes:

1. A property `ejected: true` is added to the `.angular-cli.json` file
2. A `webpack.config.js` file is generated in the root of your application with a standalone Webpack configuration so you can build your project without Angular CLI
3. The `build` script in your `package.json` is updated so you can run `npm run build` to build your project
4. The `test` script in your `package.json` is updated so you can run `npm run test` or `npm test` to run your unit tests
5. The `start` script in your `package.json` is updated so you can run `npm run`

`start` or `npm start` to start a development server

6. The `e2e` script in your `package.json` is updated so you can run `npm run e2e` to run your end-to-end tests.

After ejecting your application, you can manually update the Webpack configuration to your liking and the Angular CLI commands will no longer work.

So if, for some reason, you want to move away from Angular CLI, the eject command has you covered.

A Glimpse into the Future

Angular CLI’s roadmap contains many exciting upcoming features:

- **progressive web app (PWA) support** Angular CLI will be able to create a Web Application Manifest for you, generate an App Shell and provide Service Worker scripts to cache your application data.

  Learn more about mobile support here.
- **Refactoring Support** In addition to adding features to your application, Angular CLI will also let you refactor existing features — for example, moving a component or renaming a service.
- **Upgrade Support** Angular CLI will be able to help you with Angular upgrades. Because Angular CLI has a semantic understanding of your application, it will be able to help you with code changes that are required when upgrading to a newer version of Angular.
- **Extensibility** Improved extensibility support so you can create your own custom add-ons and deployment scripts.
- **Increased Performance** Improved performance so Angular CLI becomes faster.
- **Library Developer Mode** In addition to creating a new application, Angular CLI will also let you create a new library that can be consumed by others.

Some of these features are already partially available. Check out the GitHub repository if you want to play around with them.

Summary

Angular CLI is a command-line interface (CLI) to automate your development workflow.

Today, you can already use Angular CLI to:

- create a new Angular application
- run a development server with LiveReload support to preview your application during development
- add features to your existing Angular application
- run your application's unit tests
- run your application's end-to-end (E2E) tests
- build your application for deployment to production
- deploy your application to a server.

Although it's technically not required to use Angular CLI to develop an Angular application, it can definitely improve the quality of your code and save you a lot of time and effort.

Because of the many exciting features that will be added over time, Angular CLI will probably become an indispensable tool for developing Angular applications.

To learn more about Angular CLI, check out the official website and the GitHub repository.

Chapter 2: 10 Essential TypeScript Tips and Tricks for Angular Devs

by Florian Rappl

In this article, we’ll dive into a set of tips and tricks that should come in handy in every Angular project and beyond when dealing with TypeScript.

In recent years, the need for static typing in JavaScript has increased rapidly. Growing front-end projects, more complex services, and elaborate command-line utilities have boosted the need for more defensive programming in the JavaScript world. Furthermore, the burden of compiling an application before actually running it hasn’t seen as a weakness, but rather as an opportunity. While two strong parties (TypeScript and Flow) have emerged, a lot of trends actually indicate that only one may prevail — TypeScript.

Besides the marketing claims and commonly known properties, TypeScript has an amazing community with very active contributors. It also has one of the best teams in terms of language design behind it. Led by Anders Hejlsberg, the team has managed to fully transform the landscape of large-scale JavaScript projects to be nearly an exclusively TypeScript-powered business. With very successful projects such as VSTS or Visual Studio Code, Microsoft themselves is a strong believer in this technology.

But it’s not only the features of TypeScript that make the language appealing, but also the possibilities and frameworks that TypeScript is powering. Google’s decision to fully embrace TypeScript as their language of choice for Angular 2+ has proven to be a win-win. Not only did TypeScript gain more attention, but also Angular itself. Using static typing, the compiler can already give us informative warnings and useful explanations of why our code will not work.

TypeScript Tip 1: Supply Your Own Module Definitions

TypeScript is a superset of JavaScript. As such, every existing npm package can be utilized. While the TypeScript eco-system is huge, not all libraries are yet delivered with appropriate typings. Even worse, for some (smaller) packages not even separate declarations (in the form of `@types/{package}`) exist. At this point, we have two options:

1. bring in legacy code using TypeScript tip 7
2. define the API of the module ourselves.

The latter is definitely preferred. Not only do we have to look at the documentation of the module anyway, but typing it out will prevent simple mistakes during development. Furthermore, if we're really satisfied with the typings that we just created, we can always submit them to `@types` for including them on npm. As such, this also rewards us with respect and gratefulness from the community. Nice!

What's the easiest way to supply our own module definitions? Just create a `module.d.ts` in the source directory (or it could also be named like the package — for example, `unknown-module.d.ts` for an npm package `unknown-module`).

Let's supply a sample definition for this module:

```
declare module 'unknown-module' {
  const unknownModule: any;
  export = unknownModule;
}
```

Obviously, this is just the first step, as we shouldn't use `any` at all. (There are many reasons for this. TypeScript tip 5 shows how to avoid it.) However, it's sufficient to teach TypeScript about the module and prevent compilation errors such as "unknown module 'unknown-module'". The `export` notation here is meant for the classic `module.exports = ...` kind of packages.

Here's the potential consumption in TypeScript of such a module:

```
import * as unknownModule from 'unknown-module';
```

As already mentioned, the whole module definition is now placed in the type declaration of the exported constant. If the exported content is a function, the declaration could look like this:

```
declare module 'unknown-module' {
  interface UnknownModuleFunction {
    (): void;
  }
  const unknownModule: UnknownModuleFunction;
  export = unknownModule;
}
```

Of course, it’s also possible to use packages that export functionality using the ES6 module syntax:

```
declare module 'unknown-module' {
  interface UnknownModuleFunction {
    (): void;
  }
  const unknownModule: UnknownModuleFunction;
  export const constantA: number;
  export const constantB: string;
  export default unknownModule;
}
```

TypeScript Tip 2: Enum vs Const Enum

TypeScript introduced the concept of enumerations to JavaScript, which did represent a collection of constants. The difference between `const Foo = { A: 1, B: 2, };`

and

```
enum Foo {
  A = 1,
  B = 2,
}
```

is not only of syntactical nature in TypeScript. While both will be compiled to an object (i.e., the first one will just stay as is, while the latter will be transformed by TypeScript), the TypeScript `enum` is protected and contains only constant members. As such, it would not be possible to define its values during runtime. Also, changes of these values will not be permitted by the TypeScript compiler.

This is also reflected in the signature. The latter has a constant signature, which is similar to `interface EnumFoo { A: 1; B: 2; }`

while the object is generalized:

```
interface ConstFoo {
  A: number;
  B: number;
}
```

Thus we wouldn't see the values of these "constants" in our IDE. What does `const enum` now give us? First, let's look at the syntax: `const enum Foo { A = 1, B = 2, }`

This is actually the same — but note, there's a `const` in front. This little keyword makes a giant difference. Why? Because under these circumstances, TypeScript won't compile anything. So we have the following cascade:

- objects are untouched, but generate an implicit generalized shape declaration (interface)
- `enum` will generate some boilerplate object-initializer along with a

specialized shape declaration

- `const enum` doesn’t generate anything beside a specialized shape declaration.

Now how is the latter then used in the code? By simple replacements. Consider this code:

```
enum Foo {
  A = 1,
  B = 2
}

const enum Bar {
  A = 1,
  B = 2
}

console.log(Bar.A, Foo.B);
```

Here we end up in JavaScript with the following outcome:

```
var Foo;
(function (Foo) {
  Foo[Foo["A"] = 1] = "A";
  Foo[Foo["B"] = 2] = "B";
})(Foo || (Foo = {}));
console.log(1 /* A */, Foo.B);
```

Note that 5 lines alone have been generated for `enum Foo`, while `enum Bar` only resulted in a simple replacement (constant injection). Thus `const enum` is a compile-time only feature, while the original `enum` is a runtime + compile-time feature. Most projects will be well suited for `const enum`, but there may be cases where `enum` is preferred.

TypeScript Tip 3: Type Expressions

Most of the time, we’re satisfied with using `interface` for defining new shapes of objects. However, there are cases when a simple interface isn’t sufficient any more. Consider the following example. We start with a simple interface:

```
interface StatusResponse {
  issues: Array<string>;
  status: 'healthy' | 'unhealthy';
}
```

The notation in `'healthy' | 'unhealthy'` means either a constant string being `healthy` or another constant string equal to `unhealthy`. Alright, this is a sound interface definition. However, now we also have a method in our code, that wants to mutate an object of type `StatusResponse`:

```
function setHealthStatus(state: 'healthy' | 'unhealthy') {
  // ...
}
```

So far, so good, but changing this now to `'healthy' | 'unhealthy' | 'unknown'` results in two changes already (one in the interface definition and one in the definition of argument type in the function). Not cool. Actually, the expressions we looked at until now are already type expressions, we just did not “store” them — that is, give them a name (sometimes called *alias*). Let’s do that:

```
type StatusResponseStatus = 'healthy' | 'unhealthy';
```

While `const`, `var`, and `let` create objects at runtime from JS expressions, `type` creates a type declaration at compile-time from TS expressions (so-called type expressions). These type declarations can then be used:

```
interface StatusResponse {
  issues: Array<string>;
  status: StatusResponseStatus;
}
```

With such aliases in our tool belt, we can easily refactor the type system at will. Using TypeScript’s great type inference just propagates the changes accordingly.

TypeScript Tip 4: Use Discriminators

One of the uses of type expressions is the formerly introduced union of several (simple) type expressions — that is, type names or constants. Of course, the union is not restricted to simple type expressions, but for readability we should not come up with structures such as this:

```
type MyUnion = {
  a: boolean,
  b: number,
} | {
  c: number,
  d: {
    sub: string,
  }
} | {
  (): void;
};
```

Instead, we want a simple and straightforward expression, such as this:

```
type MyUnion = TypeA | TypeB | TypeC;
```

Such a union can be used as a so-called discriminated union if all types expose at least one member with the same name, but a different (constant) value. Let’s suppose we have three types, such as these:

```
interface Line {
  points: 2;
  // other members, e.g., from, to, ...
}

interface Triangle {
  points: 3;
  // other members, e.g., center, width, height
}

interface Rectangle {
  points: 4;
  // other members, e.g., top, right, bottom, left
}
```

A discriminated union between these types could be this:

```
type Shape = Line | Triangle | Rectangle;
```

This new type can now be used in functions, where we can access specific members using some validation on the discriminator, which would be the `points` property. For example:

```
function calcArea(shape: Shape) {
  switch (shape.points) {
    case 2:
      // ... incl. return
    case 3:
      // ... incl. return
    case 4:
      // ... incl. return
    default:
      return Math.NaN;
  }
}
```

Naturally, `switch` statements come in quite handy for this task, but other means of validation can also be used.

Discriminated unions come in handy in all kinds of scenarios — for example, when traversing an AST-like structure of when dealing with JSON files that have a similar branching mechanism in their schema.

TypeScript Tip 5: Avoid Any Unless It Really Is Any

We’ve all been there: we know exactly what code to write, but we’re unable to satisfy the TypeScript compiler to accept our data model for the code. Well, luckily for us we can always fall back to `any` for saving the day. But we shouldn’t. `any` should only be used for types that can in fact be any. (For example, it’s on purpose that `JSON.parse` returns `any`, as the outcome could be anything depending on the string we’re parsing.)

For instance, in one of our data stores we defined explicitly that a certain field `custom` will hold data of type `any`. We don’t know what will be set there, but the consumer is free to choose the data (and thus the data type). We neither wanted nor could prevent this from happening, so the type `any` was for real.

However, in most scenarios (that is, in all scenarios that are exclusively covered by our code) `any` is usually one or multiple types. We only need to find out what type exactly we expect and how to construct such a type to give TypeScript all the necessary information.

Using some of the previous tips — for example, TypeScript tip 4 and TypeScript tip 3 — we can already solve some of the biggest problems:

```
function squareValue(x: any) {
  return Math.pow(x * 1, 2);
}
```

We’d much rather constrain the input as much as possible:

```
function squareValue(x: string | number) {
  return Math.pow(+x, 2);
}
```

Now the interesting part is that the former expression `x * 1` is allowed with `any`, but disallowed in general. However, the `+x` gives us the forced cast to a `number` as wanted. To check if our cast works with the given types, we need to be specific. The question “what types can enter here?” is a legit one that we need to answer before TypeScript can supply us with useful information.

TypeScript Tip 6: Use Generics Efficiently

TypeScript means static typing, but static typing doesn’t mean explicit typing. TypeScript has powerful type inference, which has to be used and fully understood before one can be really productive in TypeScript. Personally, I think I’ve become far more productive in TypeScript than plain JavaScript, as I don’t spend much time on my typings, yet everything seems to be in place and almost all trivial errors are already detected by TypeScript. One of the drivers behind this productivity boost is generics. Generics gives us the ability to bring in types as variables.

Let’s consider the following case of a classic JS helper function:

```
function getOrUpdateFromCache(key, cb) {
  const value = getFromCache(key);

  if (value === undefined) {
    const newValue = cb();
    setInCache(key, newValue);
    return newValue;
  }

  return value;
}
```

Translating this directly to TypeScript leaves us behind with two `any`s: one is the data retrieved from the callback, and one from the function itself. However, this doesn’t need to look like that, since we obviously know the type (we pass in `cb`):

```
function getOrUpdateFromCache<T>(key: string, cb: () => T) {
  const value: T = getFromCache(key);

  if (value === undefined) {
    const newValue = cb();
    setInCache(key, newValue);
    return newValue;
  }

  return value;
}
```

The only troublesome position in the code above is the explicit type assignment to the result of calling the `getFromCache` function. Here we must trust our code for the moment to consistently only use the same types for the same keys. In TypeScript tip 10 we learn how to improve this situation.

Most of the time the use of generics is just to “pass through” a type — that is, to teach TypeScript about the relation between certain argument types (in the former case the type of the result is connected to the return type of the callback). Teaching TypeScript about such relations can also be subject for further constraints, which are then put in place by TypeScript.

While generics is easy to use together with interfaces, types, classes, and standard functions, they may not seem so approachable with arrow functions. These functions are anonymous by definition (they need to be assigned to a variable to be accessed via a name).

As a rule of thumb, we can follow this approach: just think of a normal, but anonymous function declaration. Here only the name is gone. As such the `<T>` is naturally just placed before the parentheses. We end up with:

```
const getOrUpdateFromCache = <T>(key: string, cb: () => T) => /*
...*/;
```

However, once we would introduce this in a TSX file (for whatever reason), we would end up with an error *ERROR : unclosed `T` tag*. It’s the same problem that appears with casts (solved there by using the `as` operator). Now our workaround is to tell TypeScript explicitly that the syntax was intended for generics use:

```
const getOrUpdateFromCache = <T extends {}>(key: string, cb: () =>
T) => /* ...*/;
```

TypeScript Tip 7: Bring in Legacy Code

The key for migrating existing code to TypeScript has been a set of well-adjusted TypeScript configuration parameters — for example, to allow implicit `any` and to disable strict mode. The problem with this approach is that transformed code goes from a legacy state into a freeze state, which also impacts the new code that's being written (since we disabled some of the most useful compiler options).

A better alternative is to just use `allowJs` in the `tsconfig.json` file, next to the usual (quite strong) parameters: `{ "compilerOptions": { "allowJs": true, // ... } }`

Now instead of already renaming existing files from `.js` to `.ts`, we keep existing files as long as possible. We will only rename if we seriously can tackle the content in such a way that the code is fully transformed from JavaScript to a TypeScript variant that satisfies our settings.

TypeScript Tip 8: Create Functions with Properties

We already know that using interfaces to declare the shape of a function is a sound way. Furthermore, this approach allows us to attach some properties to the given function type. Let's first see how this may look in practice:

```
interface PluginLoader {
  (): void;
  version: string;
}
```

Defining this is straightforward, but unfortunately, working with it isn't. Let's try to use this interface as intended by creating an object that fulfills the interface:

```
const pl: PluginLoader = () => {};
pl.version = '1.0.0';
```

Ouch: we can't get past the declaration. TypeScript (correctly) complains, that the `version` property are missing. Okay, so how about the following workaround:

```
interface PluginLoaderLight {
  (): void;
  version?: string;
}

const pl: PluginLoaderLight = () => {};
pl.version = '1.0.0';
```

Perfect. This works, but it has one major drawback: even though we know that past the `pl.version` assignment the `version` property will always exist at `pl`, TypeScript doesn't know that. So from its point of view, any access to `version` could be wrong and needs to be checked against `undefined` first. In other words, in the current solution the interface we use for producing an object of this type has to be different from the interface used for consuming. This isn't ideal.

Fortunately, there is a way around this problem. Let's return to our original `PluginLoader` interface. Let's try it with a cast that states to TypeScript "Trust me, I know what I'm doing".

```
const pl = <PluginLoader>(() => {});
pl.version = '1.0.0';
```

The purpose of this is to tell TypeScript, “See this function, I know it will be of this given shape (`PluginLoader`)”. TypeScript still checks if this *can* be still fulfilled. Since there are no clashing definitions available, it will accept this cast. Casts should be our last line of defense. I don’t consider `any` a possible line of defense: either the type is `any` for real (can always be — we just accept anything, totally fine), or it should not be used and has to be replaced by something specific (see TypeScript tip 5).

While the way of casting may solve problems such as the described one, it may not be feasible in some non-Angular environment (for example, React components). Here, we need to choose the alternative variant of casting, namely the `as` operator:

```
const pl = (() => {}) as PluginLoader;
pl.version = '1.0.0';
```

Personally, I would always go for `as`-driven casts. Not only do they *always* work, they’re also quite readable even for someone who doesn’t have a TypeScript background. For me, consistency and readability are two principles that should always be at the core of every codebase. They can be broken, but there have to be good reasons for doing so.

TypeScript Tip 9: The keyof Operator

TypeScript is actually quite good at — well — handling types. As such, it gives us some weapons that can be used to boilerplate some code for actually generating the content of an interface. Likewise, it also offers us options for iterating through the content of an interface.

Consider the following interface:

```
interface AbstractControllerMap {
  user: UserControllerBase;
  data: DataControllerBase;
  settings: SettingsControllerBase;
  //...
}
```

Potentially, in our code we have an object with a similar structure. The keys of this object are magic: its strings are used in many iterations and thus on many occasions. Quite likely we use these keys as arguments somewhere.

Obviously, we could just state that a function could look like this:

```
function actOnAbstractController(controllerName: string) {
  // ...
}
```

The downside is that we definitely have more knowledge, which we do not share with TypeScript. A better version would therefore be this:

```
function actOnAbstractController(controllerName: 'user' | 'data' |
'settings') {
  // ...
}
```

However, as already noted in TypeScript tip 3, we want to be resilient against refactorings. This is not resilient. If we add another key (that is, map another controller in our example above), we’ll need to edit the code in multiple locations.

A nice way out is provided by the `keyof` operator, which works against any type. For instance, aliasing the keys of the `AbstractControllerMap` above looks as follows:

```
type ControllerNames = keyof AbstractControllerMap;
```

Now we can change our function to truly become resilient against refactorings on the original map.

```
function actOnAbstractController(controllerName: ControllerNames) {
  // ...
}
```

The cool thing about this is that `keyof` will actually respect interface merging. No matter where we place the `keyof`, it will always work against the “final” version of the type it’s applied to. This is also very useful when thinking about factory methods and efficient interface design for them.

TypeScript Tip 10: Efficient Callback Definitions

A problem that appears more often than anticipated is the typing of event handlers. Let’s look at the following interface for a second:

```
interface MyEventEmitter {
  on(eventName: string, cb: (e: any) => void): void;
  off(eventName: string, cb: (e: any) => void): void;
  emit(eventName: string, event: any): void;
}
```

Looking back at all the previous tricks, we know that this design is neither ideal nor acceptable. So what can we do about it? Let’s start with a simple approximation to the problem. A first step is certainly to define all possible event names. We could use type expressions as introduced in TypeScript tip 3, but even better would be a mapping to the event type declarations like in the previous tip.

So we start with our map and apply TypeScript tip 9 to obtain the following:

```
interface AllEvents {
  click: any;
  hover: any;
  // ...
}

type AllEventNames = keyof AllEvents;
```

This has already some effect. The previous interface definition now becomes:

```
interface MyEventEmitter {
  on(eventName: AllEventNames, cb: (e: any) => void): void;
  off(eventName: AllEventNames, cb: (e: any) => void): void;
  emit(eventName: AllEventNames, event: any): void;
}
```

A little bit better, but we still have `any` on all interesting positions. Now TypeScript tip 6 can be applied to make TypeScript a little bit more knowledgeable about the entered `eventName`:

```
interface MyEventEmitter {
  on<T extends AllEventNames>(eventName: T, cb: (e: any) => void):
void;
```

```
  off<T extends AllEventNames>(eventName: T, cb: (e: any) => void):
void;
  emit<T extends AllEventNames>(eventName: T, event: any): void;
}
```

This is good, but not sufficient. TypeScript now knows about the exact type of `eventName` when we enter it, but we’re unable to use the information stored in `T` for anything. Except, we can use it with another powerful type expressions: index operators applied to interfaces.

```
interface MyEventEmitter {
  on<T extends AllEventNames>(eventName: T, cb: (e: AllEvents[T])
=> void): void;
  off<T extends AllEventNames>(eventName: T, cb: (e: AllEvents[T])
=> void): void;
  emit<T extends AllEventNames>(eventName: T, event: AllEvents[T]):
void;
}
```

This seems to be powerful stuff, except that our existing declarations are all set to `any`. So let’s change this.

```
interface ClickEvent {
  leftButton: boolean;
  rightButton: boolean;
}

interface AllEvents {
  click: ClickEvent;
  // ...
}
```

The real powerful part is now that interface merging still works. That is, we can extend our event definitions out of place by using the same interface name again:

```
interface AllEvents {
  custom: {
    field: string;
  };
}
```

This makes type expressions even more powerful, as the extensibility is integrated in a wonderful and elegant way.

Further Reading

- (Original, 2012) Introducing TypeScript — JavaScript on Steroids
- Introduction to TypeScript
- TypeScript GitBook on Discriminated Unions
- The Official TypeScript Blog
- Getting Started with Angular 2 using TypeScript

Conclusion

Hopefully one or more of these TypeScript tips were new to you or at least something you wanted to see in a closer write up. The list is far from complete, but should give you a good starting point to avoid some problems and increase productivity.

Chapter 3: Using Angular Augury to Debug Your Code

by Ilya Bodrov-Krukowski

Augury is an open-source tool allowing developers to profile and debug Angular applications.

Modern web browsers provide developer consoles to inspect various elements on the page, which is really handy when trying to debug markup, styles, and scripts. However, this console isn't enough to debug Angular applications that usually have lots of components, events, attributes, and a separate routing system.

Augury is a tool designed specifically for Angular apps. It's an open-source debugging and profiling tool for Angular 2+ applications.

Augury is just a Chrome extension that's quite simple to use, so you won't need to spend hours and hours learning how to utilize this tool. We're going to build a sample Angular app and then see Augury in action by exploring various parts of our project. So, let's get started!

Hello, Augury!

Augury visualizes your app's structure in a form of a tree, showing how components and their dependencies relate to each other. It also allows you to inspect properties of your objects and change them on the fly. On top of that, you can easily view the source code of a specific component, insert breakpoints as needed, work with events, and more. Lastly, you can browse the application's routing system, as well as view the full list of all utilized modules.

Augury is only available as a Chrome extension (there's no Firefox support yet, unfortunately) and installing it is as simple as going to this page and pressing the *Install* button. After that, you may open the developer tools by pressing `Ctrl` + `Shift` + `I` (Windows/Linux) or `Cmd` + `Opt` + `I` (macOS). You'll note that a new tab called *Augury* has appeared. After switching to this tab, you'll either see the application's structure or the phrase "This application is not an Angular application". I've noticed that sometimes it may be required to re-open the Developer Console in order for Augury to analyze the page properly, so watch out.

Now that we have Augury installed, let's proceed to the next section and prepare the sample application that we'll use as a playground!

Building a Sample App

In order to see Augury in action, we need something to debug, right? In this section, I’m going to quickly guide you through the process of creating a very simple application (loosely based on the sample app from the Angular’s official tutorial) listing some users and allowing you to edit them.

Example Code

Alternatively, you may grab the source code from my GitHub repo.

Before getting started, install Angular CLI on your machine if you don’t have it yet:

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

Next, create the skeleton of our new project:

```
ng new sitepoint-augury
```

Change the application’s title by tweaking the *src/app/app.component.ts* file:

```
// ...

export class AppComponent {
  title = 'Augury Demo';
}
```

Tweak the `src/app/app.component.html` by removing all the links to documentation added automatically by code generator and add an `<app-users></app-users>` line:

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>

<app-users></app-users>
```

Of course, we need a `User` component, so generate it now by running:

```
ng generate component users
```

Change the `src/app/users/user.component.ts` file in the following way:

```
import { Component, OnInit } from '@angular/core';
import { User } from './user.model'; // <--- 1
import { UserService } from './user.service'; // <--- 2

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

  selectedUser: User;

  onSelect(user: User): void { // <--- 3
    this.selectedUser = user;
  }

  constructor(private userService: UserService) { } // <--- 4

  ngOnInit() {
    this.getUsers(); // <--- 5
  }

  getUsers(): void { // <--- 6
    this.users = this.userService.getUsers();
  }

}
```

Main things to note here:

1. We are importing a `User` model that will be created in a moment.
2. We're also importing a `UserService`. It will simply return a list of hardcoded users, but let's pretend they're being fetched from some remote location.
3. We're allowing the users to be selected by clicking on them. The currently selected user is stored in a separate `selectedUser` attribute.
4. Hook up the `userService` using the dependency injection mechanism.
5. Load the list of users once the component is initialized.
6. In order to get users, we're utilizing our `userService`.

That’s pretty much it for this component.

Next, let’s create a model in a `src/app/users/user.model.ts` file. Each user is going to have an ID, a first and a last name:

```
export class User {
  id: number;
  first: string;
  last: string;
}
```

Nothing complex.

Now let’s proceed to the `UserService` that’s going to be defined in the `app/src/users/user.service.ts` file:

```
import { Injectable } from '@angular/core';
import { User } from './user.model';

@Injectable()
export class UserService {

  constructor() { }

  getUsers(): User[] {
    return [
      {
        id: 1,
        first: 'John',
        last: 'Doe'
      },
      {
        id: 2,
        first: 'Margaret',
        last: 'Brown'
      }
    ]
  }
}
```

The `getUsers` method simply returns an array of users with hardcoded data.

Now let’s display our users with the help of `ngFor`. Also, we’re going to add a click event listener and fire `onSelect` whenever a user is clicked on. When this happens, a form to edit the chosen user should be displayed (which is going to be done with the help of `ngIf`). Modify the `src/app/users/user.component.html` file like this:

```
<div *ngFor="let user of users" (click)="onSelect(user)"
[class.selected]="user === selectedUser">
  <p>{{user.last}}, {{user.first}} (ID: {{user.id}})</p>
</div>

<div *ngIf="selectedUser">
  <h3>Edit</h3>
  <label for="first">First</label>
  <input [(ngModel)]="selectedUser.first" placeholder="First name"
id="first">

  <label for="last">Last</label>
  <input [(ngModel)]="selectedUser.last" placeholder="Last name"
id="last">
</div>
```

We're assigning a `.selected` CSS class to the chosen user, so let's add some simple styling for it inside the `src/app/users/user.component.css` file:

```
.selected {
  font-weight: bold;
}
```

Lastly, we have to import `FormsModule` and `UserService` inside the `src/app/app.module.ts` file:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms'; // <---
import { UserService } from './users/user.service'; // <---

import { AppComponent } from './app.component';
import { UsersComponent } from './users/users.component';
```

`FormsModule` should be listed in the `imports` section in the `app.module.ts`, whereas `UserService` goes to the `providers`:

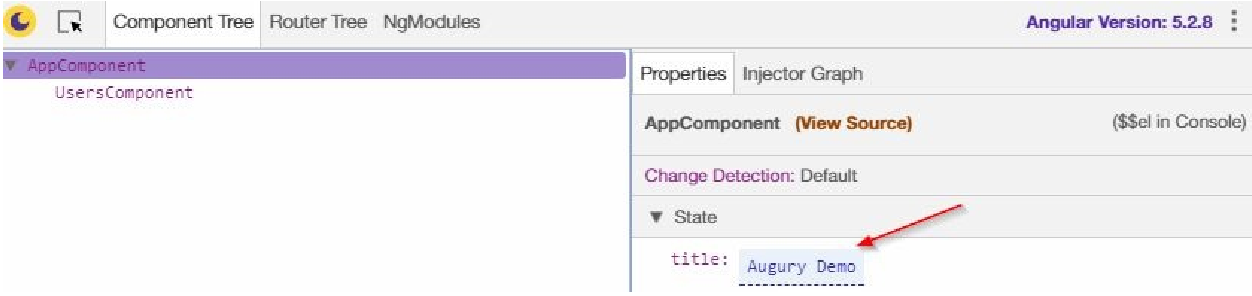
```
@NgModule({
  declarations: [
    AppComponent,
    UsersComponent
  ],
  imports: [
    BrowserModule,
    FormsModule // <---
  ],
  providers: [
    UserService // <---
  ],
  bootstrap: [AppComponent]
})
```

That’s it! Our sample application is finished, and you can open it by running the following command:

```
ng serve --open
```

Component View

All right, so now let’s take a look at the goodies that Augury has to offer for us. Open Chrome and go to `http://localhost:4200`. A list of users should be displayed. Press `Ctrl` + `Shift` + `I` or `Cmd` + `Opt` + `I`, switch to the Augury tab, and press *AppComponent* in the left pane, under the *Component Tree*:



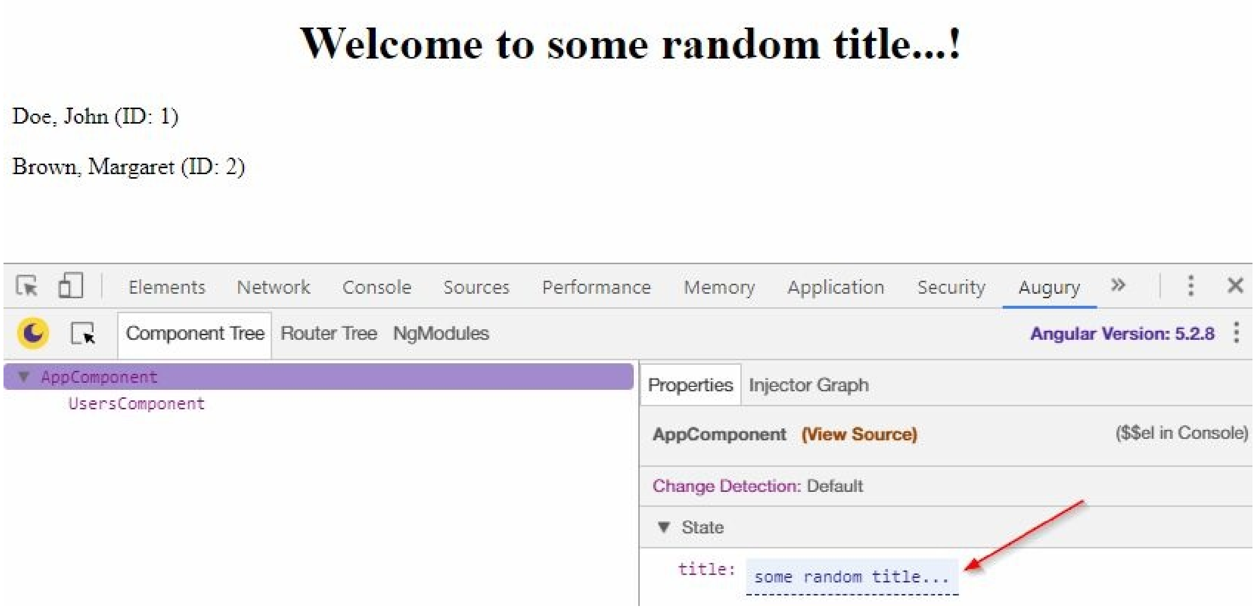
Change Detection: Default

title: Augury Demo

There are two panes here:

- To the left, you can see the *Component Tree* that has a list of the application’s components, properly nested.
- To the right, there’s a *Properties* pane for the selected component. Our `AppComponent` has only one property `title`, which is displayed alongside its value (annotated with an arrow).

What’s interesting about the *Properties* pane is that you can change the values as needed and instantly observe the changes. For example, try changing the `title`:



Welcome to some random title...!

Doe, John (ID: 1)

Brown, Margaret (ID: 2)

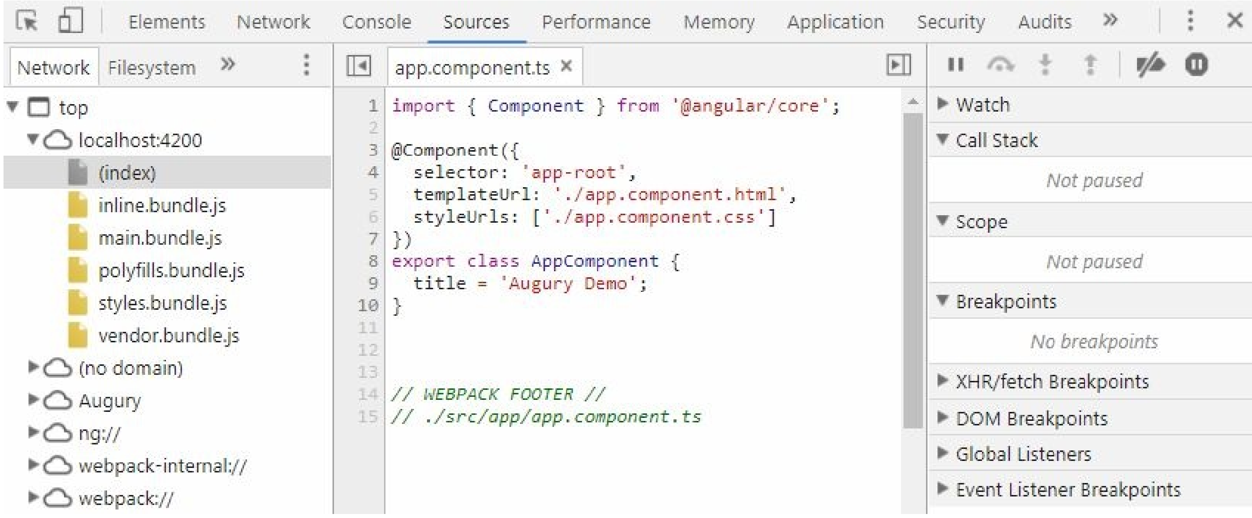
AppComponent (View Source) ($$el in Console)

Change Detection: Default

▼ State

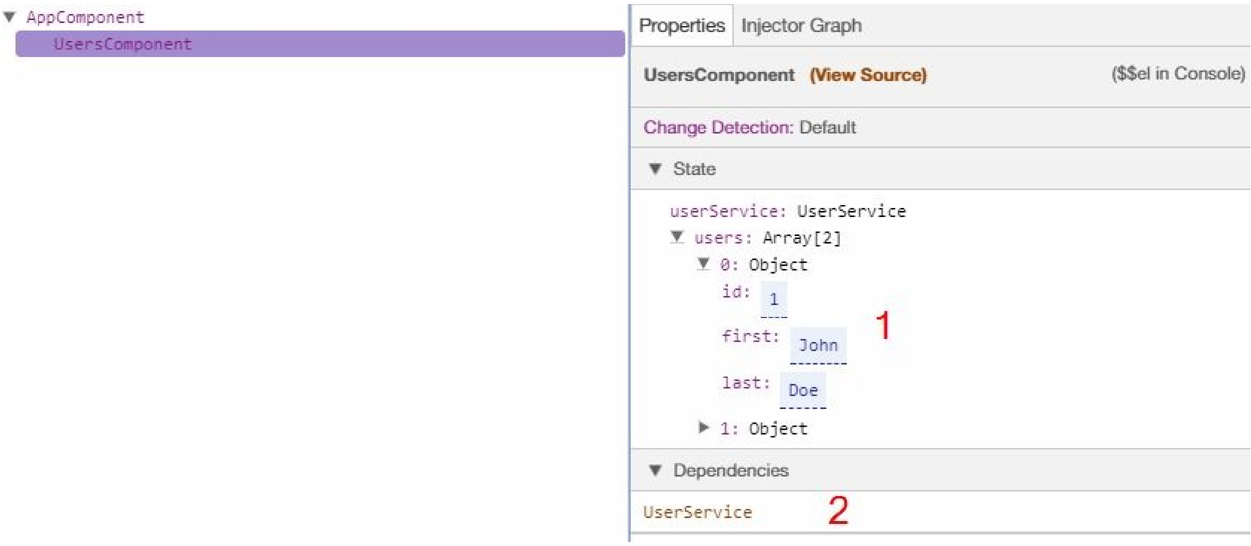
title: some random title...

Also, there’s a *View Source* link next to the component’s name. Pressing it will display the source code for the chosen component, where you can easily insert breakpoints to stop code execution at an arbitrary point:



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

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



// WEBPACK FOOTER //
// ./src/app/app.component.ts
```

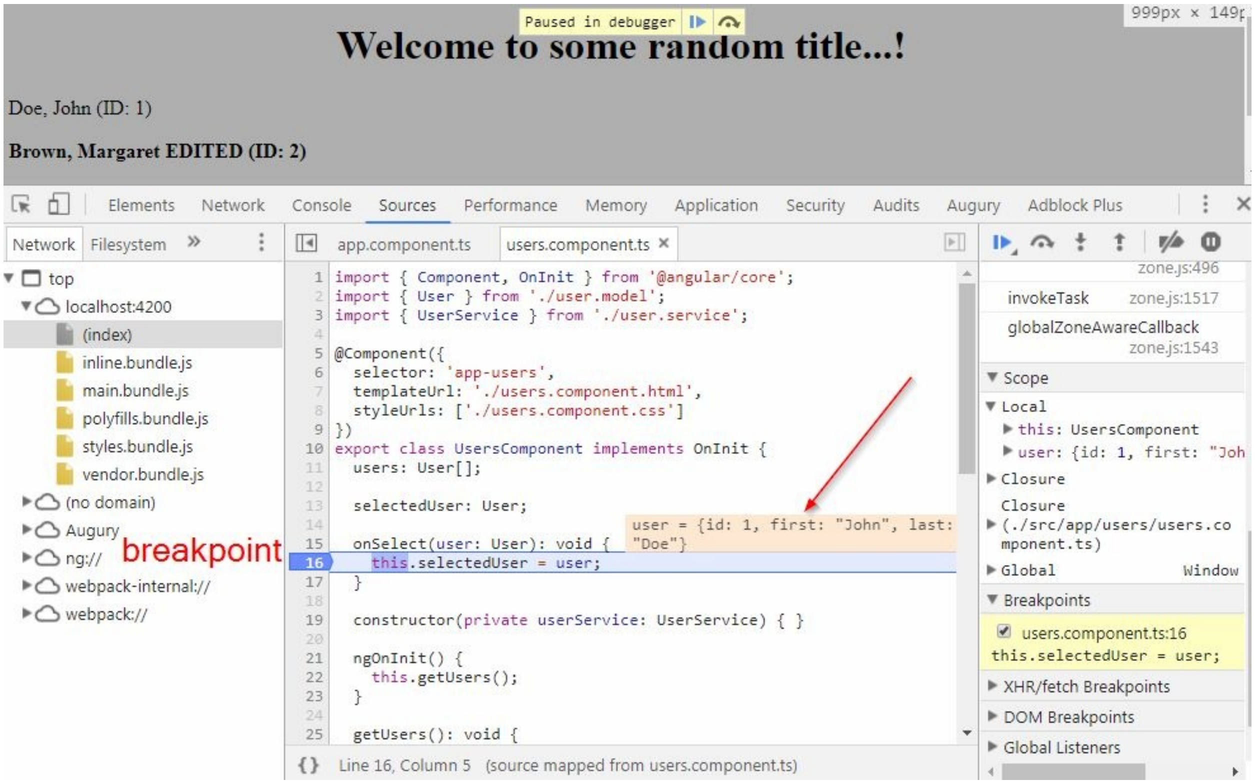
Now let’s return to the *Component Tree* and select `UsersComponent`. In the *Properties* tab we’ll see an array of users (point #1 in the screenshot below) as well as `UserService` listed in the *Dependencies* section (#2):



```
userService: UserService
▼ users: Array[2]
  ▼ 0: Object
      id: 1
      first: John
      last: Doe
  ▶ 1: Object
```

UserService

Next, let’s try to view the source for the `UsersComponent`. When the corresponding tab opens, you may insert a breakpoint at, for example, line 16 that says `this.selectedUser = user;`. To insert a breakpoint, simply click on the line number. Next, try selecting a user and you’re going to see the following:



```
1  import { Component, OnInit } from '@angular/core';
2  import { User } from './user.model';
3  import { UserService } from './user.service';
4
5  @Component({
6    selector: 'app-users',
7    templateUrl: './users.component.html',
8    styleUrls: ['./users.component.css']
9  })
10 export class UsersComponent implements OnInit {
11   users: User[];
12
13   selectedUser: User;
14
15   onSelect(user: User): void {  user = {id: 1, first: "John", last: "Doe"}
16     this.selectedUser = user;
17   }
18
19   constructor(private userService: UserService) { }
20
21   ngOnInit() {
22     this.getUsers();
23   }
24
25   getUsers(): void {
```

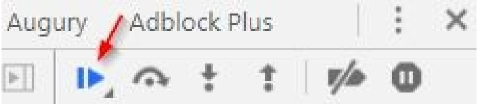
Line 16, Column 5 (source mapped from users.component.ts)

```
▶ this: UsersComponent
▶ user: {id: 1, first: "Joh
▶ Closure
  Closure
▶ (./src/app/users/users.component.ts)
```

☑ users.component.ts:16
this.selectedUser = user;

So, the code execution is paused and the currently selected user is displayed in that orange box. Once you’re done debugging this line, you can press the

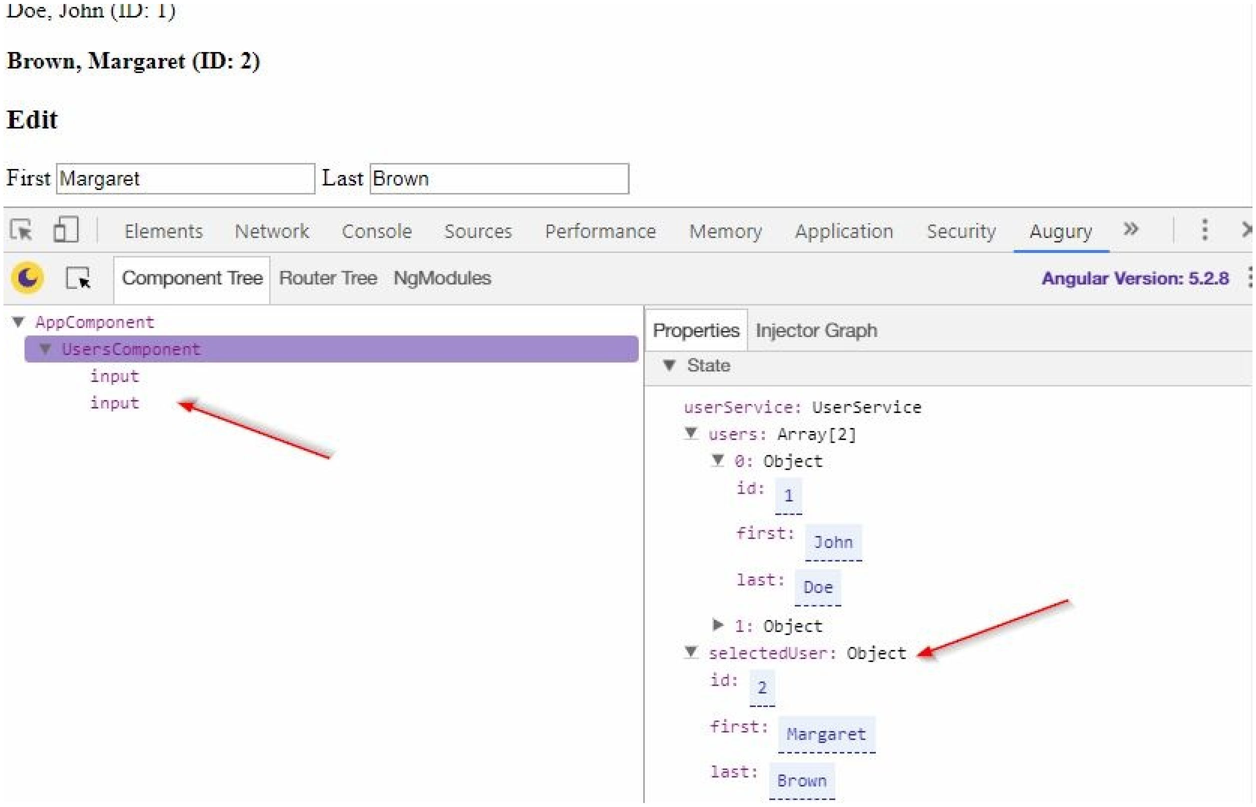
Resume Script Execution button:



The program will proceed and either stop at the next breakpoint or finish its job. Breakpoints can come in very handy if you suspect that the application is misbehaving due to the data being processed incorrectly at some specific method. You may have as many breakpoints as needed, which allows you to gradually explore the script and understand what's going on with your data at various steps.

Forms

Note that after you select a user, the `selectedUser` property will be instantly added. Also, you're going to see two `inputs` added to the `UsersComponent`:



Doe, John (ID: 1)

Brown, Margaret (ID: 2)

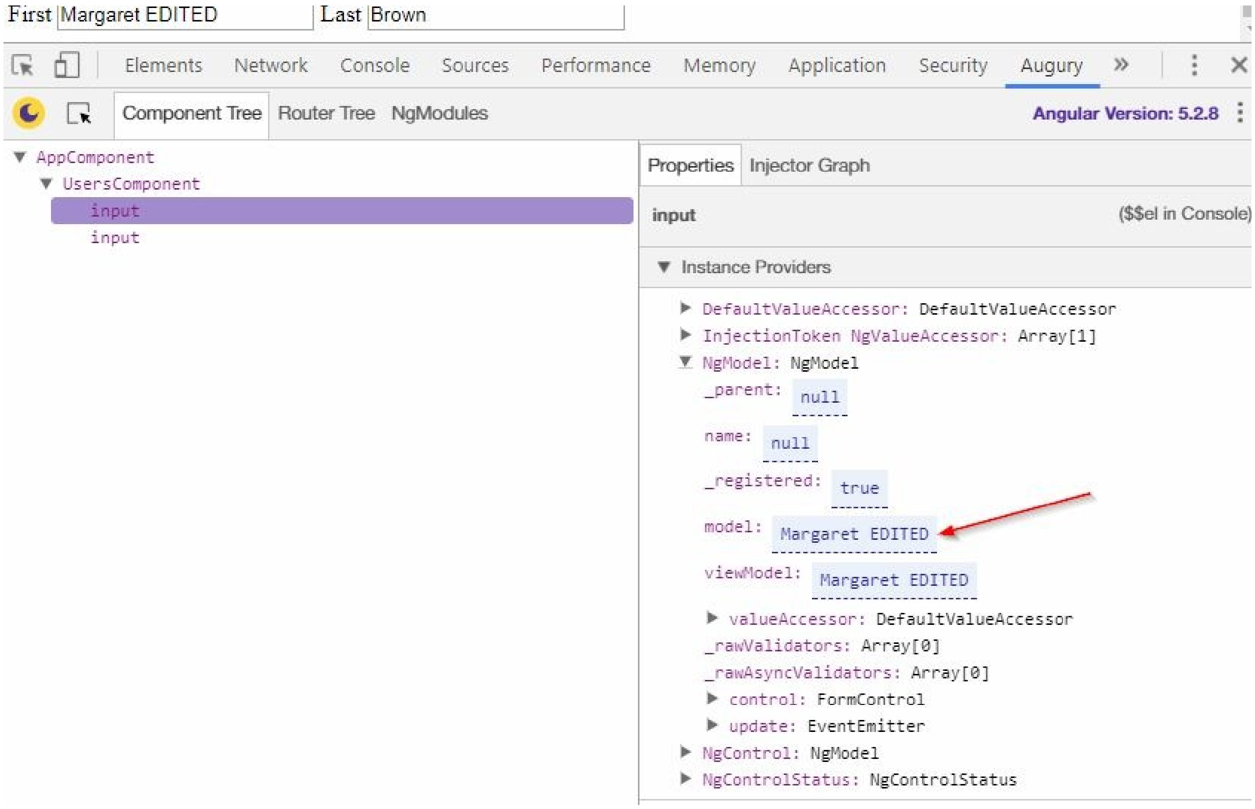
Edit

First Margaret Last Brown

```
▼ AppComponent
  ▼ UsersComponent
      input
      input
```

```
userService: UserService
▼ users: Array[2]
  ▼ 0: Object
      id: 1
      first: John
      last: Doe
  ▶ 1: Object
▼ selectedUser: Object
    id: 2
    first: Margaret
    last: Brown
```

If we select one of these inputs, we're going to see some quite detailed information about this control — specifically, what model it relates too. If I change the input's value, the model is going to be updated as well:



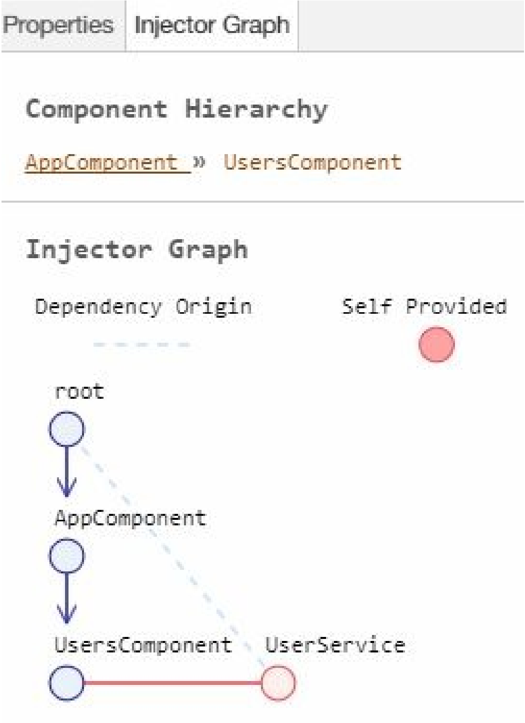
First Margaret EDITED Last Brown

```
▼ AppComponent
  ▼ UsersComponent
      input
      input
```

```
input

▼ Instance Providers
   ▶ DefaultValueAccessor: DefaultValueAccessor
   ▶ InjectionToken NgValueAccessor: Array[1]
   ▼ NgModel: NgModel
      _parent: null
      name: null
      _registered: true
      model: Margaret EDITED
      viewModel: Margaret EDITED
      ▶ valueAccessor: DefaultValueAccessor
      _rawValidators: Array[0]
      _rawAsyncValidators: Array[0]
      ▶ control: FormControl
      ▶ update: EventEmitter
   ▶ NgControl: NgModel
   ▶ NgControlStatus: NgControlStatus
```

Dependency Injection

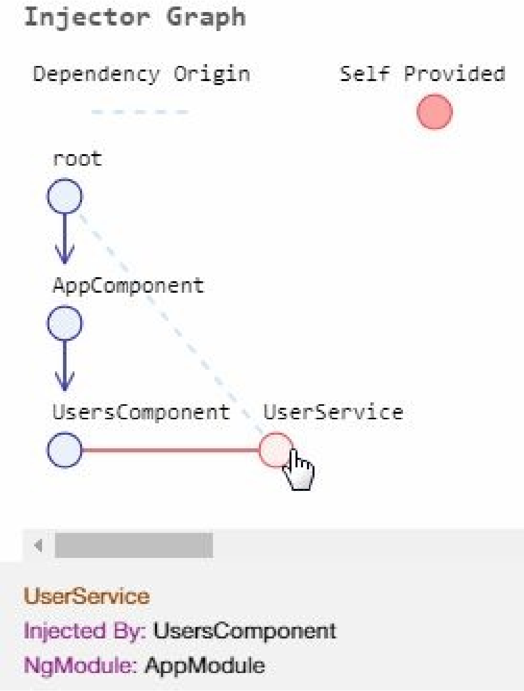
As you remember, we have the `UserService` injected into the `UsersComponent`, right? This can be easily confirmed by selecting the `UsersComponent` and opening an “Injector Graph”:



So, here we can see the Component Hierarchy and the Injector Graph itself. This graph illustrates the dependency tree:

- `root` is our `NgModule` defined in the `app.module.ts` file
- Next, there’s an `AppComponent`
- Lastly, there’s the `UsersComponent` itself.

Also, note that here we can see the `UserService` connected to the `UsersComponent` with a red line. This confirms that the service is definitely being injected into the component. You can also hover the mouse pointer over the `UserService` node to see more detailed information at the bottom of the tab:

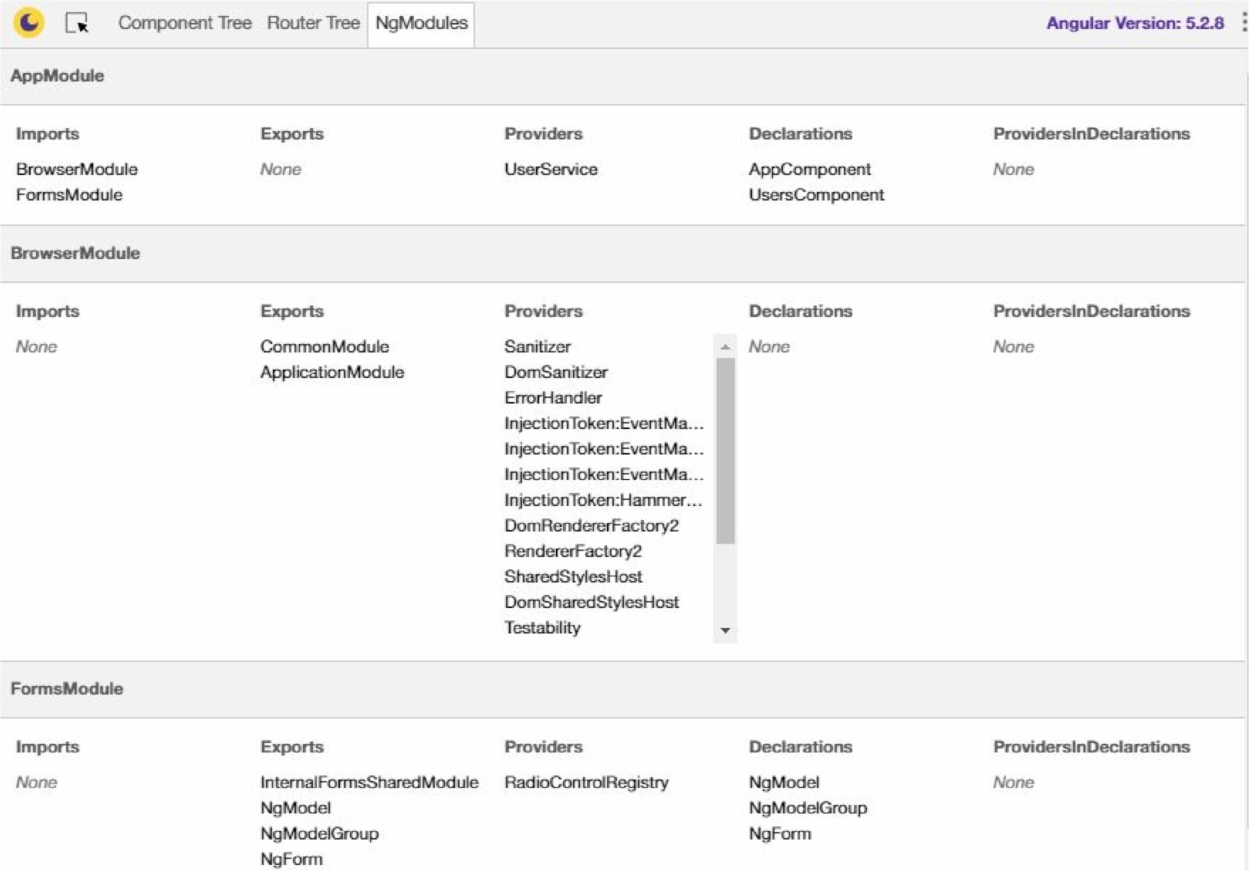


UserService
Injected By: UsersComponent
NgModule: AppModule

Dependency injection can help you to make sure that all the dependencies are hooked up properly, as many bugs arise simply because you forget to import some module.

NgModules

It's also possible to view a full list of all modules utilized in the app as well as information about the providers, exports, and other stuff. It's as simple as opening the *NgModules* tab:



AppModule

Imports	Exports	Providers	Declarations	ProvidersInDeclarations
BrowserModule FormsModule	*None*	UserService	AppComponent UsersComponent	*None*

BrowserModule

Imports	Exports	Providers	Declarations	ProvidersInDeclarations
None	CommonModule ApplicationModule	Sanitizer DomSanitizer ErrorHandler InjectionToken:EventMa… InjectionToken:EventMa… InjectionToken:EventMa… InjectionToken:Hammer… DomRendererFactory2 RendererFactory2 SharedStylesHost DomSharedStylesHost Testability	*None*	*None*

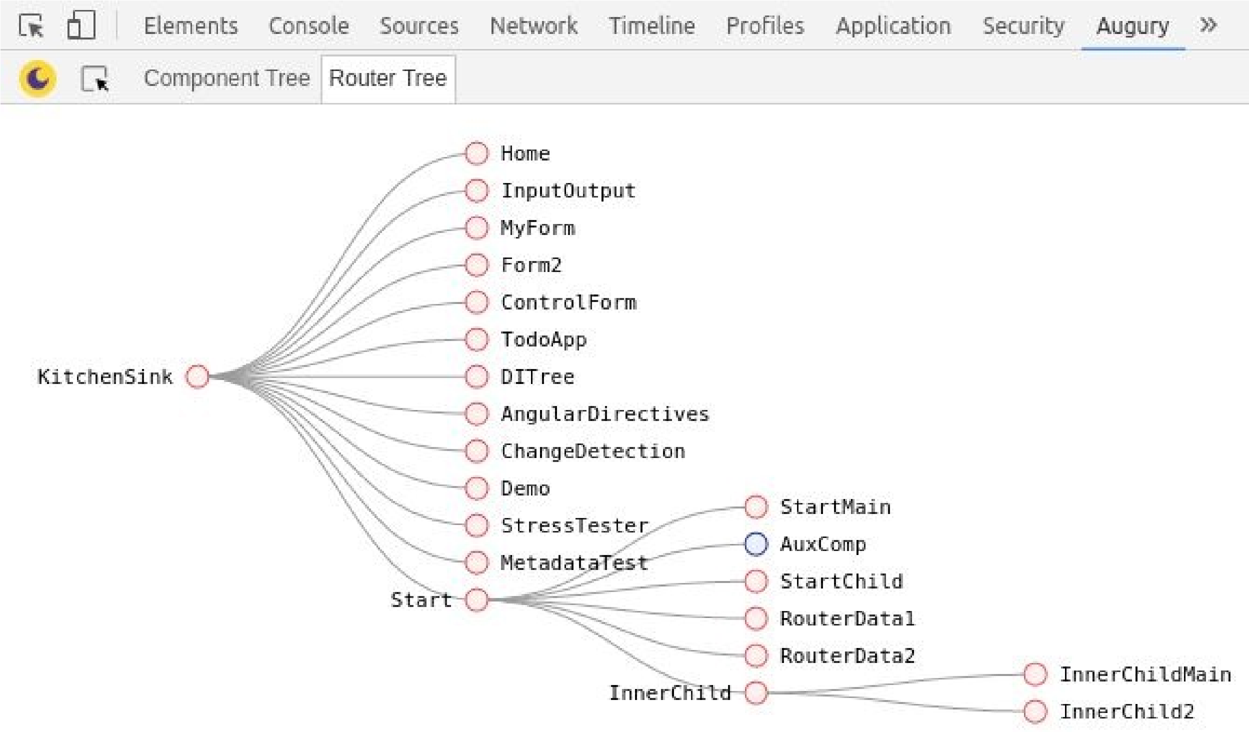
FormsModule

Imports	Exports	Providers	Declarations	ProvidersInDeclarations
None	InternalFormsSharedModule NgModel NgModelGroup NgForm	RadioControlRegistry	NgModel NgModelGroup NgForm	*None*

Unfortunately, this tab isn’t interactive, so you can’t select a module to view more detailed information about it. Still, it can come in really handy.

Routes

The last Augury feature is the ability to inspect the routing system of your application. We don’t have any routes in the demo app, but here’s an image taken from the official documentation that nicely illustrates this feature:



We can see all the application’s routes with ease. Even if a route is being lazy-loaded, this schema will be automatically updated when the corresponding route appears. How cool is that?

Conclusion

In this chapter, we’ve looked at the Augury profiler and debugger for Angular 2+ applications. We’ve seen how to install and launch this tool and discussed all its main features, including component tree, injection graph, breakpoints, and routing debugging.

Chapter 4: Top Angular Plugins for Sublime Text

by Jeff Smith

This chapter covers a number of Angular plugins for the Sublime Text text editor.

Whether you’re new to Angular development, and looking to get started with it and integrate it more closely with your code editor of choice, or you’re an old hand at Angular development and you’re trying your hand with Sublime Text as a new editor, integrating the two has never been easier.

There are lots of options for Angular plugins that can smooth your way developing Angular apps using Sublime Text. You’ll need to set up the Sublime Text package manager (called “Package Control”) prior to getting started, and then you can take a look at the list of plugins here and see if any meet your needs!

Setting up Package Control

In order to use most of the Angular plugins that will be discussed in this article, you'll first need to set up Package Control in Sublime Text. This is a fairly painless process. The easiest way involves copy-pasting a configuration code.

1. Use the hotkey `CTRL` + ` or use the *View* > *Show Console* menu to bring up the integrated Sublime Text console.

2. Paste the block of Python code into the console, which can be copied from the Package Control Installation page and follow the instructions there.

3. Once this is done, the Package Control menus will be set up! Now, all you'll need to do is find and install your packages.

Installing a Sublime Text Package via Package Control Menus

Using Package Control is a breeze. You'll need to open Package Control, select the *install* menu, and then choose and install a package:

1. Use the shortcut `CMD` + `Shift` + `P` (`CTRL` + `Shift` + `P`, depending on OS) to open the command palette.

2. Type `install package` and then press `Enter`, which brings you to the list of available packages.

3. Search for the package by name, select the package, then press `Enter` to install.

Angular 2 HTML Package

The Angular 2 HTML plugin provides augmented HTML syntax for Sublime Text. You'll be able to use Angular attributes without syntax highlighting being broken. Additionally, the JavaScript parts of your pages will also highlight as JavaScript. A small but incredibly useful addition for Angular developers.

Angular 2 HTML Package Setup

At the time of this writing, this plugin was not present in Package Control, but can be installed manually via the following steps.

1. Close Sublime Text and navigate via the Command Line to your Sublime Text 3 "Packages" folder in your application installation. In macOS, this is at `/Users/{user}/Library/Application Support/Sublime Text 3/Packages`.

2. Clone the repository into your Packages folder with the following:

   ```
   git clone https://github.com/princemaple/ngx-html-syntax
   ```

3. Re-open Sublime Text 3, and check in the *View* > *Syntax* menu to ensure that Ngx HTML is an option.

Additionally, you can have Sublime Text automatically highlight `.component.html` files with Angular 2 HTML Syntax:

1. Open a `.component.html` file.

2. Choose *View* > *Syntax* > *Ngx HTML*.

3. Go to *Preferences* > *Settings* > *Syntax Specific*. Because your current file is using the Ngx HTML syntax, it should open the syntax-specific settings file for Ngx HTML. If so, add an extensions entry to the settings panel on the right:

   ```
   "extensions":
   [
     "component.html"
   ```

```
]
```

4. Restart Sublime Text. Now, all `.component.html` files should automatically use the Ngx Syntax plugin!

Angular 2 Snippets

Angular 2 Snippets is a Sublime Text plugin that aims to provide users with snippets and code completion for Angular. It brings with it most of the autocompletion features that you would need for your Angular development. Autocompletion of code is a bit of a hotbed issue with many developers, who feel that it’s as much a curse as a blessing, but nonetheless, it can be extremely useful.

At the time of writing, the following snippets and completion categories were available. (For a current list, or for more details about the items in each category, see the plugin’s GitHub README.)

Angular Plugins: Snippet Categories

- Component
- Directive
- Service
- Pipe
- RouteConfig
- Route
- Test

Angular Plugins: Completion Categories

- Annotations
- Decorators
- Lifecycle

- Routing
- Directives
- Attributes
- Pipes

Angular CLI

The Angular CLI plugin is fantastic for any Angular devs out there, allowing Angular CLI commands to be run from within Sublime Text. Once installed, Angular CLI commands can be found from the command palette. The Angular CLI can be used for a variety of purposes:

- to generate components, classes, routes, and more
- for testing
- for linting
- for starting a development server

… and much more!

Angular Plugins: Typescript

TypeScript provides a typed layer over standard JavaScript, and is the preferred language for many Angular projects. It compiles to valid JavaScript, so the true value is to the developer. The TypeScript package for Sublime Text offers an improved experience for TypeScript users in the editor.

- TypeScript formatting for a line, a selection of code, or an entire document at the tap of a shortcut.
- Quick navigation to a symbol or piece of text, easy viewing of TypeScript errors.
- Project handling is a breeze with the plugin. It allows for the creation of new, configured TypeScript projects. It will also automatically create “inferred projects” when editing a single TypeScript file, pulling in imported files to make them available to you.

Angular Plugins: Linting

Who doesn’t need code listing? When you’re working with Angular and TypeScript, you’ll want your editor to lint your code. SublimeLinter-contrib-tslint is a Sublime package that interfaces between the `tslint` software and Sublime Text. In order to use it, you’ll need two things previously installed:

1. Install `tslint` (`npm install -g tslint`).
2. Install the SublimeLinter 3 plugin via Package Control.
3. Use the SublimeLinter Documentation. (Follow the steps “Finding a linter executable” through “Validating your path” to ensure that SublimeLinter can see your tslint package.)
4. Finally, install the Sublime Plugin `SublimeLinter-contrib-tslint` via Package Control.

You’re all set. Now you just need to configure your linter. You can use the tslint docs to get a handle on configuration options for tslint (which you’ll want to include in a `tslint.json` file in your project). Also look at the SublimeLinter documentation about setting up linter-specific settings.

DocBlockr

DocBlockr is a package for Sublime Text that supports JavaScript and a variety of flavors of it, including TypeScript. While DocBlockr is not unique to TypeScript or Angular, it’s definitely usable there and provides a service that shouldn’t be discounted. DocBlockr is a plugin that assists developers in writing docblocks. It autocompletes the syntax for docblocks in order to save your precious time, and even goes so far as to autogenerate function and variable docblock templates as well. It’s a fantastic way both to save time and to ensure that your code gets documented in a uniform way.

Conclusion

Hopefully this set of plugins will provide you with at least some of the Angular plugins you need to make your Angular development efforts in Sublime a success.

Chapter 5: Boosting Your Workflow with Angular 5 Snippets and VS Code

by Michael Wanyoike

In this chapter, we'll focus on how to use Angular 5 snippets in Visual Studio Code to improve our workflow. We'll first start with the basics of using and creating snippets. We'll look at how we can use Angular snippets for VS Code in an Angular project. Then we'll later look at how we can create our own snippets and share them with others.

For anyone who's become proficient in a programming language, you know how boring it is to type the same code over and over again. You eventually start copying and pasting pieces of your code to spare your fingers the agony of writing another `for` loop.

What if your editor could help you insert this common code for you automatically as you type? That would be awesome, right?

Well, you probably know them as **snippets**. It's a feature that's common in every modern IDE currently available. Even Notepad++ supports them (though not enabled by default).

Prerequisites

Before you start, you’ll need to have the latest version of Visual Studio Code installed on your machine. We’ll also be looking at Angular 5 snippets. So having at least a basic knowledge of Angular will be helpful, but not necessary.

You’ll need to use an existing or new Angular 5 project in order to experiment with snippets. I assume you have the latest version of Node.js, or at least a version that’s not older than Node.js 6. Here’s the command for installing the Angular CLI tool if you haven’t:

```
npm install -g @angular/cli

# or
yarn global add @angular/cli
```

Snippets Explained

Snippets are templates that allow you to easily insert repetitive code. When you first install VS Code, it comes with snippets for JavaScript pre-installed. To check them out, just open an existing JavaScript file or create a new one in your workspace and try out typing these prefixes:

- `log`
- `if`
- `ifelse`
- `forof`
- `settimeout`

As you type, a popup list will appear giving you options to autocomplete your code. As soon as the right snippet is highlighted, just press `enter` to insert the snippet. In some cases, such as `log` and `for`, you may need to press the down key to select the right snippet.

You’re probably wondering where these JavaScript snippets are coming from exactly. Well, you can find the exact definitions in these locations:

- Windows - `C:\Program Files\Microsoft VS Code\resources\app\extensions\javascript\snippets\javascript.jsc`
- Linux - `/usr/share/code/resources/app/extensions/javascript/snippets/jav`

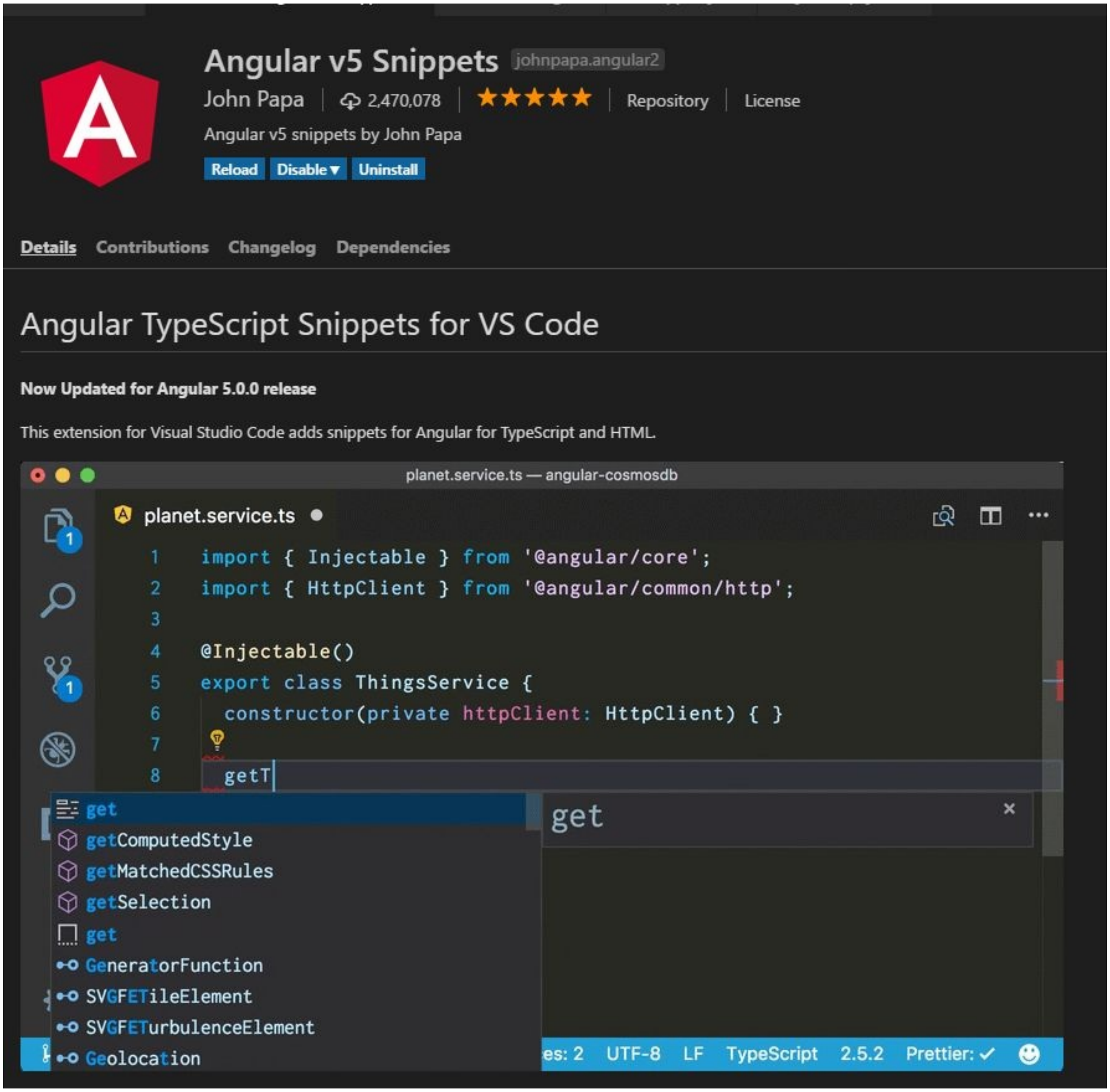
We’ll look into how we can create our own snippets, but first let’s explore some third-party snippets.

How to Use Snippets

At the time of writing, the Visual Studio Marketplace listed 934 extensions in the snippet category. You'll find snippets for every programming language created, including Pascal! This means you probably should check out the marketplace before creating your own snippets. As mentioned earlier, we'll be looking at Angular 5 snippets. Locate an existing Angular 5 project in your workspace or just create a new one:

```
ng new snippets-demo
```

Once the project is set up, open it in VS Code. Next, click the extension icon on the activity bar to open the *Extensions* panel. Search for `Angular 5`. The search results should list several Angular extensions. Install the one that has the author **'John Papa'**. After installation is done, click the reload button to restart VS Code. This Angular 5 snippets extension comes with over 50 snippets. About 70% of the snippets are written for TypeScript, and the rest are mostly for HTML. There are some Docker snippets in there too.



Angular v5 Snippets johnpapa.angular2

John Papa | 2,470,078 | ★★★★★ | Repository | License

Angular v5 snippets by John Papa

Angular TypeScript Snippets for VS Code

Now Updated for Angular 5.0.0 release

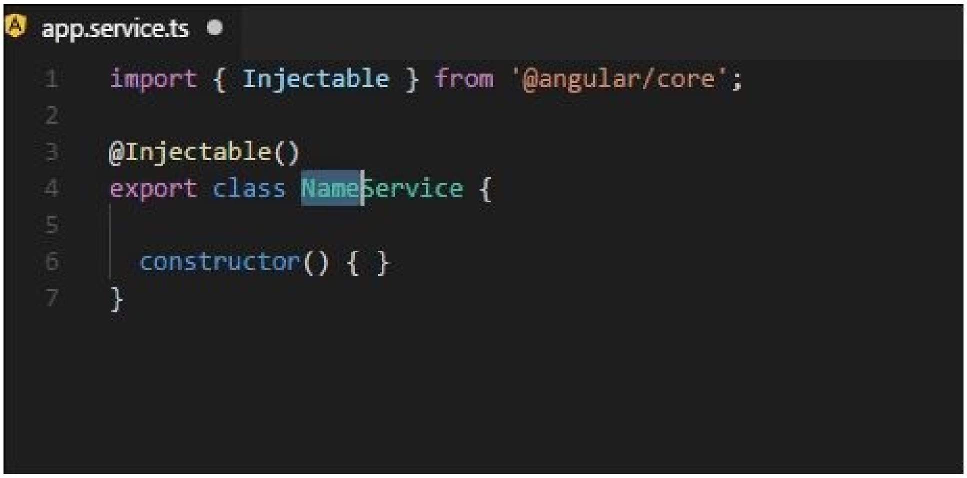
This extension for Visual Studio Code adds snippets for Angular for TypeScript and HTML.

planet.service.ts — angular-cosmosdb

planet.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ThingsService {
  constructor(private httpClient: HttpClient) { }

  getT
```

Let’s try a few of these Angular 5 snippets. Create a new file called `app.service.ts` inside the app folder. Open the file and start typing “a-service”. A pop-up list will appear as you type. Even before you finish typing, you should have the correct option highlighted. Press `enter` to insert the template. Once the snippet is entered, take note that the generated class name is highlighted for you to change.

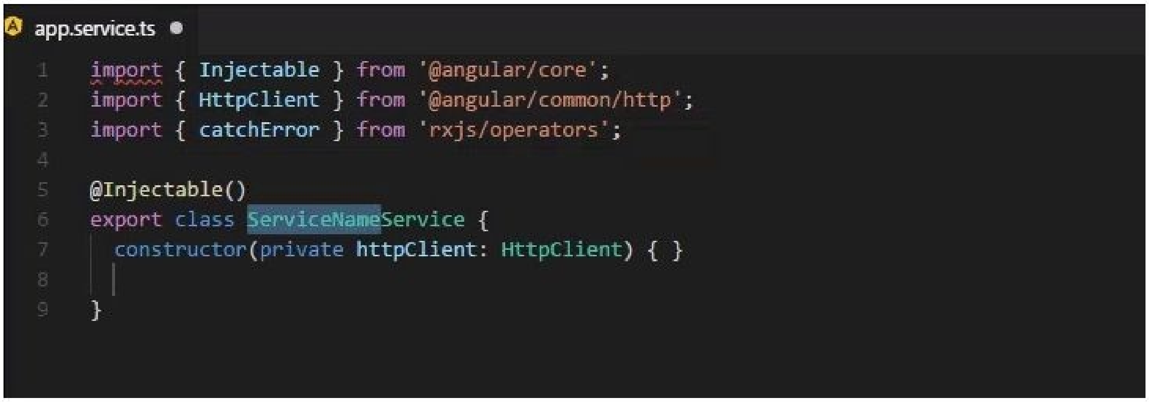


```
import { Injectable } from '@angular/core';

@Injectable()
export class NameService {

  constructor() { }
}
```

Just type “App” and the entire class name will read “AppService”. Pretty convenient, right? Let’s try something different. Delete the entire code in `app-service.ts` and type this prefix:

```
a-service-httpclient
```

You should get a service class definition, including imports and `HttpClient` injection in the class constructor. Just like before, rename the class name to `AppService`.



```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ServiceNameService {
  constructor(private httpClient: HttpClient) { }

}
```

Next, let’s use another snippet to define an HTTP GET request. Let’s define an empty GET method. For this piece of code, you have to type; don’t copy-paste:

```
getPosts(): Observable<any[]> {

}
```

As you start to type “Observable”, a snippet option for it will appear. Just press `enter` and the Observable class will be imported for you automatically.

Next, let’s use another snippet to complete the function. Start typing this prefix: “a-httpclient-get”. Pressing `enter` will insert this snippet for you:

```
return this.httpClient.get('url');
```

Change the URL to an imaginary path — let’s say `/posts`. Obviously our code won’t run, as we haven’t set up any APIs. Fix it by adding `<any>` after `get`. The complete method now looks like this.

```
getPosts(): Observable<any[]> {
  return this.httpClient.get<any[]>('/posts');
}
```

Now that we know a little bit about how to use Angular 5 snippets, let’s see how they’re created.

How to Create Angular 5 Snippets

The syntax used in Visual Studio Code is the same syntax defined in TextMate. In fact, you can copy a snippet from TextMate and it will work in VS Code. However, keep in mind that the “interpolated shell code” feature isn’t supported in VS Code.

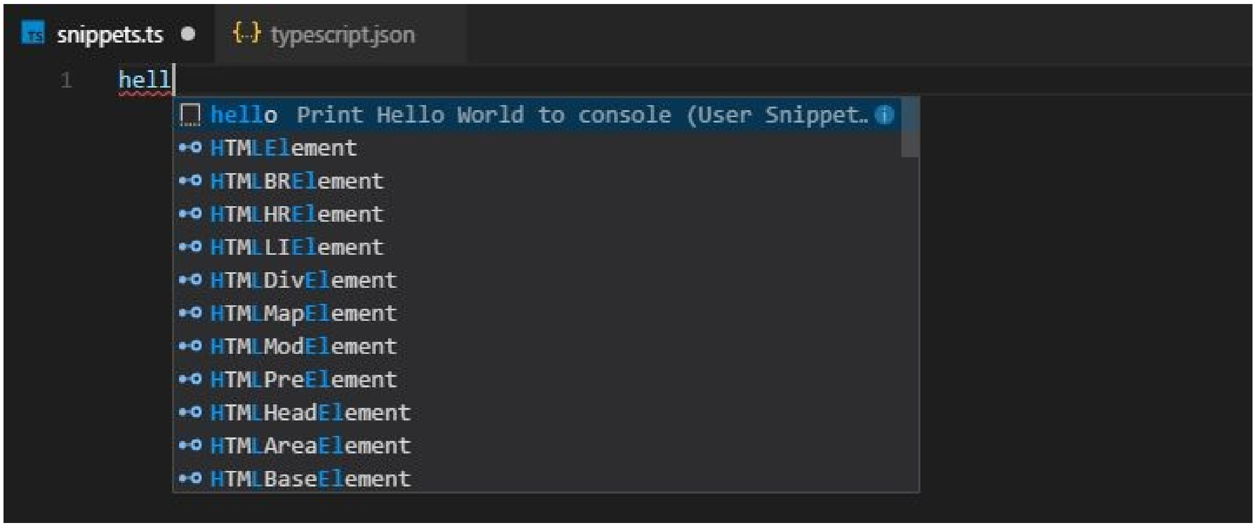
The easiest way to create a snippet is by opening `Preferences:Configure User Snippet` via the command palette (`ctrl` + `shift` + `p`). Select it, then choose a language you’d like to create a syntax for. Let’s create one now for TypeScript. Once you open `typescript.json`, place this snippet template right within the opening and closing braces:

```
"Print Hello World": {
  "prefix" : "hello",
  "body": [
    "console.log('Hello World!');"
  ],
  "description": "Print Hello World to console"
}
```

Let me go over the template structure:

- “Print Hello World” — title of the snippet template. You can put any title you want.
- prefix — trigger keyword for the snippet.
- body — the code the snippet will insert.
- description — no need to explain this one.

This is basically the simplest snippet template you can write. After you’ve saved the file, create a blank `typescript` and test the prefix `hello`. A pop list should appear as you type.



```
hell
hello  Print Hello World to console (User Snippet…
HTMLElement
HTMLBRElement
HTMLHRElement
HTMLLIElement
HTMLDivElement
HTMLMapElement
HTMLModElement
HTMLPreElement
HTMLHeadElement
HTMLAreaElement
HTMLBaseElement
```

Just hit `enter` once your snippet is highlighted. You should have this code inserted for you:

```
console.log('Hello World');
```

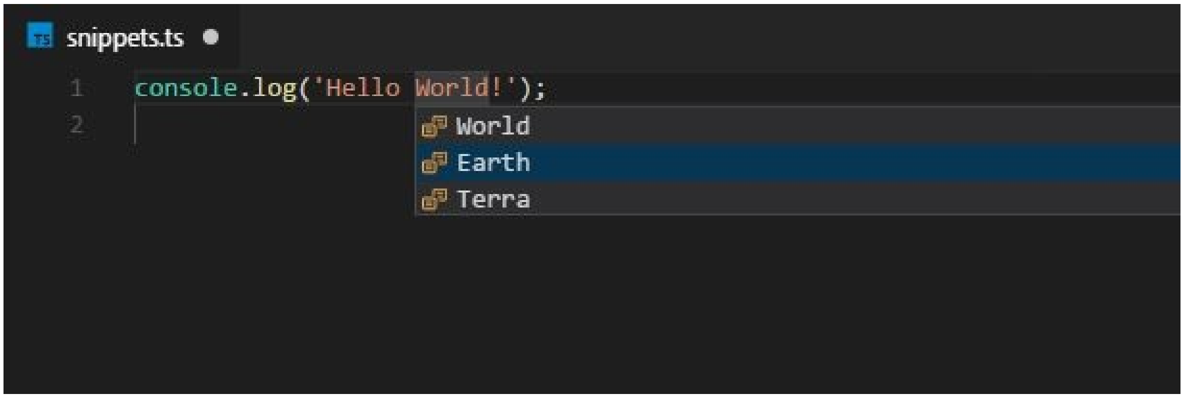
Awesome, right? Let’s now make our snippet template interactive by inserting some **Tab Stops**.

```
"Print Hello World": {
  "prefix" : "hello",
  "body": [
    "console.log('Hello $1!');",
    "$2"
  ],
  "description": "Print Hello World to console"
}
```

Now try your prefix again. This version allows you to insert your name. Once you finish typing, just press tab and your cursor will come to rest at a new line. `$0` indicates the final tab stop. You can use `$1` and greater to create multiple tabs stops. But what if we wanted to insert a default value? We can use a placeholder that looks like this: `${1:World}`. Here’s the full template:

```
"Print Hello World": {
  "prefix" : "hello",
  "body": [
    "console.log('Hello ${1:World}!');",
    "$0"
  ],
  "description": "Print Hello World to console"
}
```

Now try the snippet again. You can either change the default value, or tab out to leave it as is. Let’s now give developers a choice list to pick from:



```
console.log('Hello World!');
```

To implement the choice list, just replace the `console.log` line like this:

```
...
"console.log('Hello ${1|World,Earth,Terra|}!');",
...
```

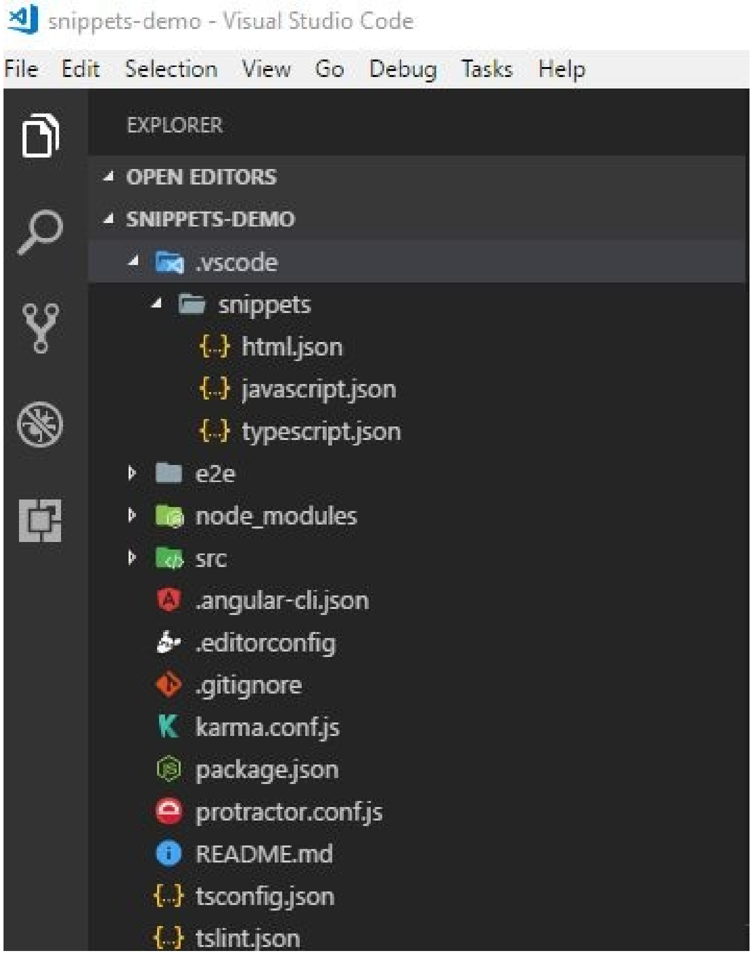
That’s enough examples for now. I haven’t covered variable names and transformations. To learn more, check out [John Papa’s Angular snippets]() on GitHub. Here’s a sneak preview:

```
"Angular HttpClient Service": {
    "prefix": "a-service-httpclient",
    "description": "Angular service with HttpClient",
    "body": [
      "import { Injectable } from '@angular/core';",
      "import { HttpClient } from '@angular/common/http';",
      "import { catchError } from 'rxjs/operators';",
      "",
      "@Injectable()",
      "export class ${1:ServiceName}Service {",
      "\tconstructor(private httpClient: HttpClient) { }",
      "\t$0",
      "}"
    ]
  }
```

This is the template we used earlier to create the `HttpClient` service. I recommend you go through the project, as it contains quite a number of highly useful templates you can learn from.

Summary

Now that you’ve learned how to create snippets, you probably want to share them with their team members, friends or the public. The easiest way to share the snippets with your teammates is to create a `.vscode` folder at your project root. Create a subfolder called `snippets` and place all your templates there, like this:



Make sure to commit this folder such that everyone can find it in the repository. To share with your friends and public, you have a number of options available to you:

- you can upload your snippets to a public server such as Google Drive, Dropbox or even Pastebin.
- you can publish your snippets as an extension on the VS Code Marketplace.

I hope this brief introduction to working with snippets will help make your programming life a bit easier!

Chapter 6: Top 12 Productivity Tips for WebStorm and Angular: Part 1

by Jurgen Van de Moere

This chapter on WebStorm and Angular was sponsored by JetBrains. Thank you for supporting the partners who make SitePoint possible.

In this two-part series on WebStorm and Angular, Google Developer Experts Jurgen Van de Moere and Todd Motto share their favorite productivity tips for developing Angular applications using WebStorm.

In this first part, Jurgen shares his personal top five WebStorm features that allow him to increase his productivity on a daily basis when working with WebStorm and Angular:

1. Use Angular CLI from within WebStorm
2. Navigate like a pro
3. Take advantage of Angular Language Service
4. Auto format your code
5. Optimize your imports

Each tip can tremendously increase your development productivity, so let’s dig into them a little deeper one by one.

WebStorm and Angular Tip 1: Use Angular CLI from Within WebStorm

Angular CLI is a command-line interface (CLI), written and maintained by the Angular team, to help automate your development workflow. You can use it to quickly create new Angular projects and add new features such as components, services and directives to existing Angular projects.

WebStorm and Angular integration using Angular CLI provides you with all Angular's power right from within WebStorm, without using the terminal.

To create a new Angular Project, choose *File* > *New* > *Project* and select *Angular CLI*.

Enter a project location and hit the *Create* button. WebStorm uses Angular CLI to create a new Angular project and install dependencies.

When your new Angular application is in place, you can easily add new Angular features. Right-click on *src/app* and choose *New* > *Angular CLI* to pick the type of feature you wish to add.

Once you've selected a feature, you can specify the name and optional parameters, just as you would with Angular CLI on the command line. What's really awesome is that WebStorm automatically adds the component to the right Angular module for you.

If your application has multiple Angular modules, right-click on the module you wish to add the feature to and choose *New* > *Angular CLI*. WebStorm will make sure the new files are created in the right location and that the new feature is added to the correct Angular module.

How sweet is that!

WebStorm and Angular Tip 2: Navigate Like a Pro

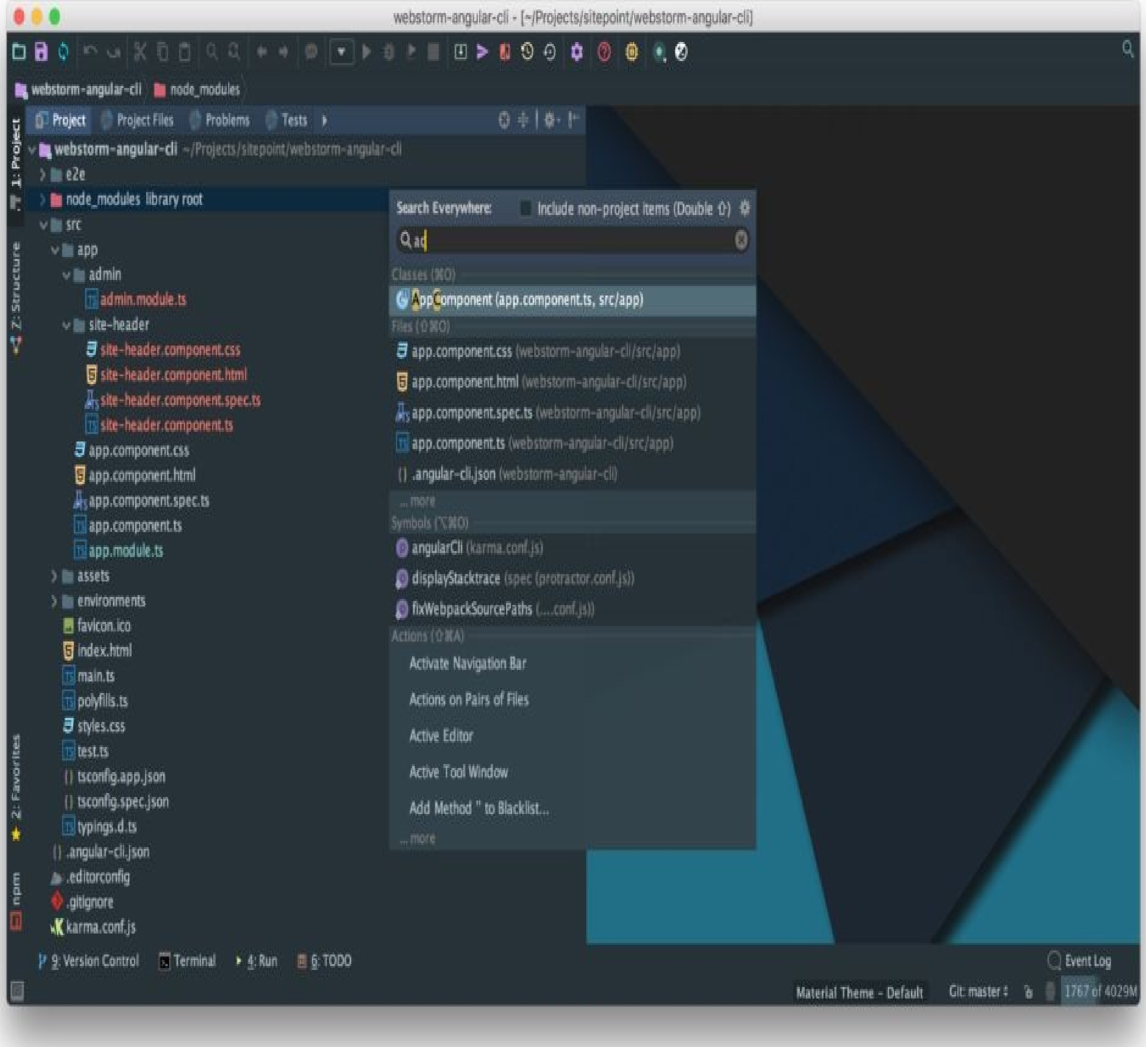
Use `cmd`-click or `cmd` + `B` to easily jump to any definition within your project.

If you’re a keyboard user, just put your cursor on a term and hit `cmd` + `B`. If you’re a mouse user, hold down the `cmd` button and all terms you hover will turn into links to their definition.

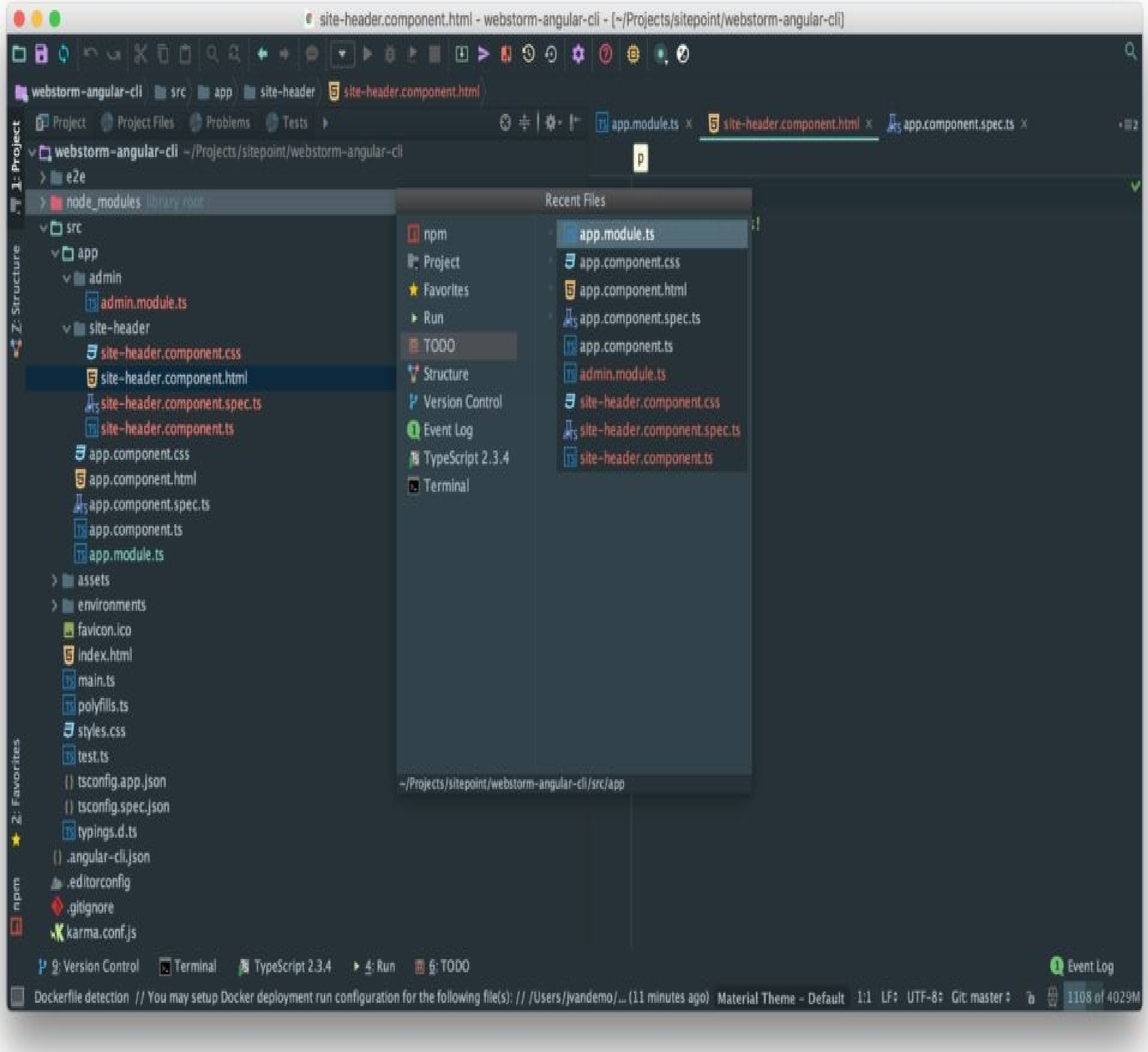
WebStorm automatically recognizes Angular components and directives in your HTML — links to stylesheets, links to templates, classes, interfaces and much more.

No need to open file(s) manually; just jump to any definition you need!

When looking for a file that you don’t have an immediate reference to, hit `shift` twice to open the *Search everywhere* dialog. You don’t have to type the entire search string. If you want to open `AppComponent`, just type the first letter of each part — for example, `ac` — and WebStorm will immediately narrow down the result list for you, so you can quickly pick the suggestion you wish to open:


webstorm-angular-cli - [~/Projects/sitepoint/webstorm-angular-cli]
webstorm-angular-cli
node_modules
Project
Project Files
Problems
Tests
1: Project
Z: Structure
2: Favorites
npm
webstorm-angular-cli ~/Projects/sitepoint/webstorm-angular-cli
e2e
node_modules library root
src
app
admin
admin.module.ts
site-header
site-header.component.css
site-header.component.html
site-header.component.spec.ts
site-header.component.ts
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
assets
environments
favicon.ico
index.html
main.ts
polyfills.ts
styles.css
test.ts
tsconfig.app.json
tsconfig.spec.json
typings.d.ts
.angular-cli.json
.editorconfig
.gitignore
karma.conf.js
Search Everywhere:
Include non-project items (Double ⇧)
ac
Classes (⌘O)
AppComponent (app.component.ts, src/app)
Files (⇧⌘O)
app.component.css (webstorm-angular-cli/src/app)
app.component.html (webstorm-angular-cli/src/app)
app.component.spec.ts (webstorm-angular-cli/src/app)
app.component.ts (webstorm-angular-cli/src/app)
.angular-cli.json (webstorm-angular-cli)
... more
Symbols (⌥⌘O)
angularCli (karma.conf.js)
displayStacktrace (spec (protractor.conf.js))
fixWebpackSourcePaths (....conf.js))
Actions (⇧⌘A)
Activate Navigation Bar
Actions on Pairs of Files
Active Editor
Active Tool Window
Add Method '' to Blacklist...
... more
9: Version Control
Terminal
4: Run
6: TODO
Event Log
Material Theme - Default
Git: master
1767 of 4029M

Another super useful navigation shortcut is `cmd` + `E`, which presents you with a list of recently edited files so you can easily navigate back and forth between them.



Dockerfile detection // You may setup Docker deployment run configuration for the following file(s): // /Users/jvandemo/... (11 minutes ago)

Knowing how to quickly navigate to the code you need will save you a tremendous amount of time every single day.

WebStorm and Angular Tip 3: Take Advantage of Angular Language Service

By default, WebStorm already provides great assistance for writing Angular code.

When editing a script, WebStorm automatically imports the required JavaScript modules so you don’t have to import them manually.

If you open up the TypeScript panel, WebStorm provides you with immediate feedback on the validity of your code, so you can quickly resolve issues before having to compile your project. When you edit a template, WebStorm provides you with smart code completion that recognizes components, directives and even input and output properties.

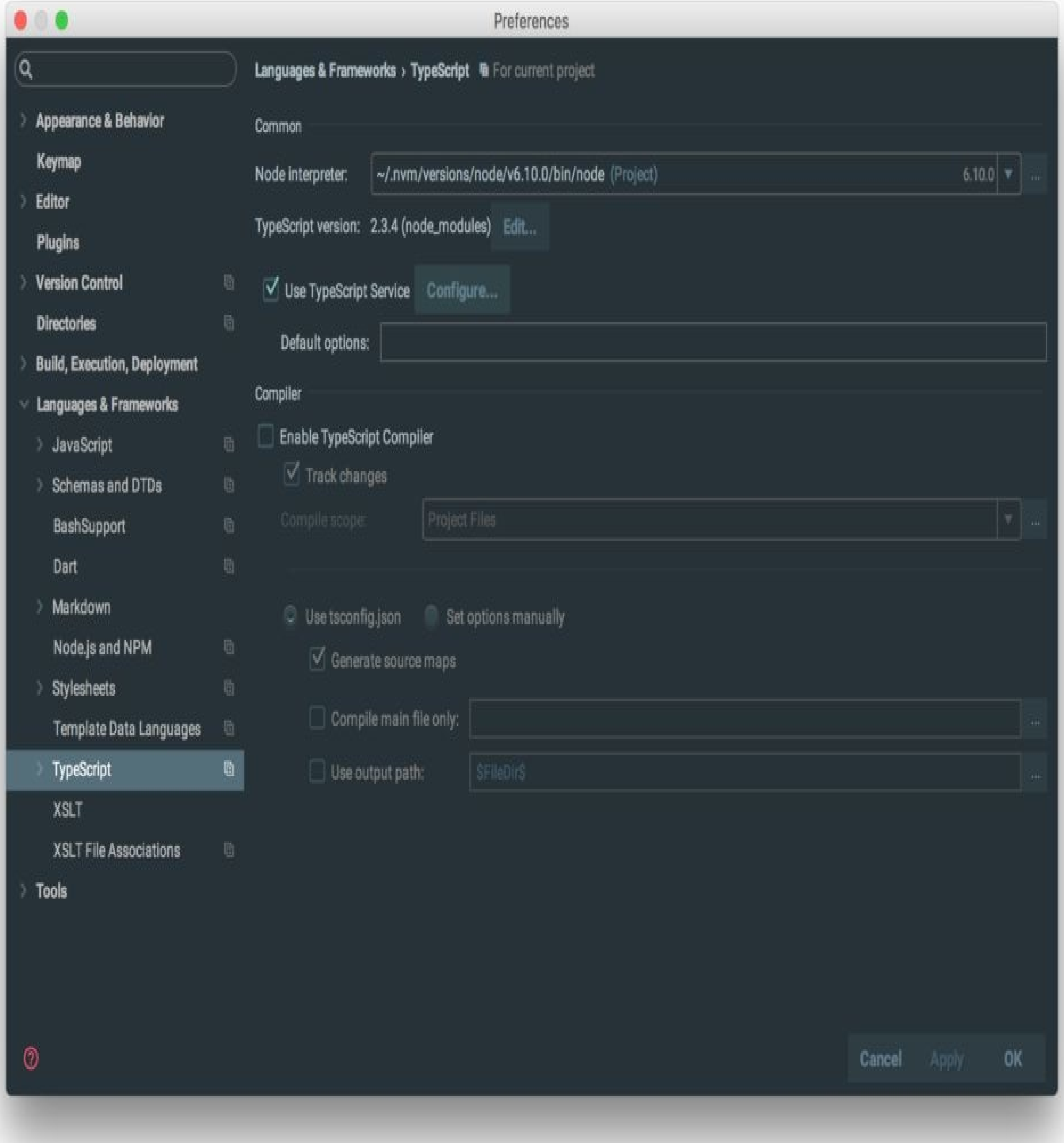
You can take things further by installing the Angular Language Service. This is a service, designed by the Angular Team, to provide IDEs with error checking and type completion within Angular templates.

WebStorm integrates with Angular Language Service to better understand your code. To enable Angular Language Service, first make sure it’s installed: `npm install @angular/language-service --save-dev`

With Angular CLI

If you use Angular CLI to generate an Angular application, Angular Language Service is automatically installed.

Next, go to *Preferences > Languages & Frameworks > TypeScript*, make sure *Use TypeScript Service* is checked and click *Configure…*:



Languages & Frameworks › TypeScript For current project

Common

Node interpreter: ~/.nvm/versions/node/v6.10.0/bin/node (Project) 6.10.0

TypeScript version: 2.3.4 (node_modules)

☑ Use TypeScript Service

Default options:

Compiler

☐ Enable TypeScript Compiler

☑ Track changes

Compile scope: Project Files

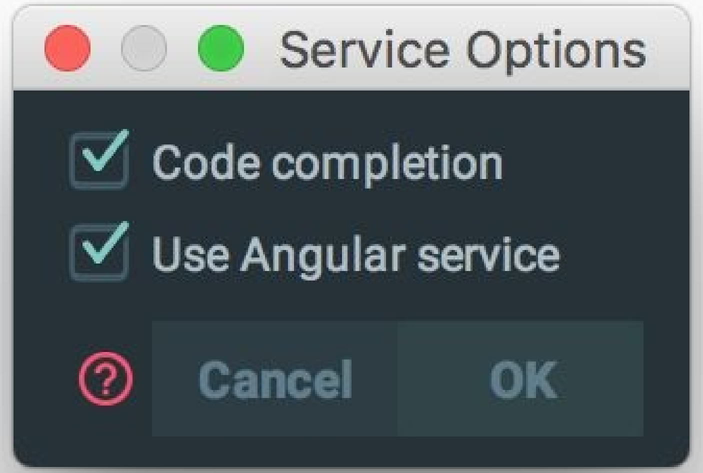
○ Use tsconfig.json ○ Set options manually

☑ Generate source maps

☐ Compile main file only:

☐ Use output path: $FileDir$

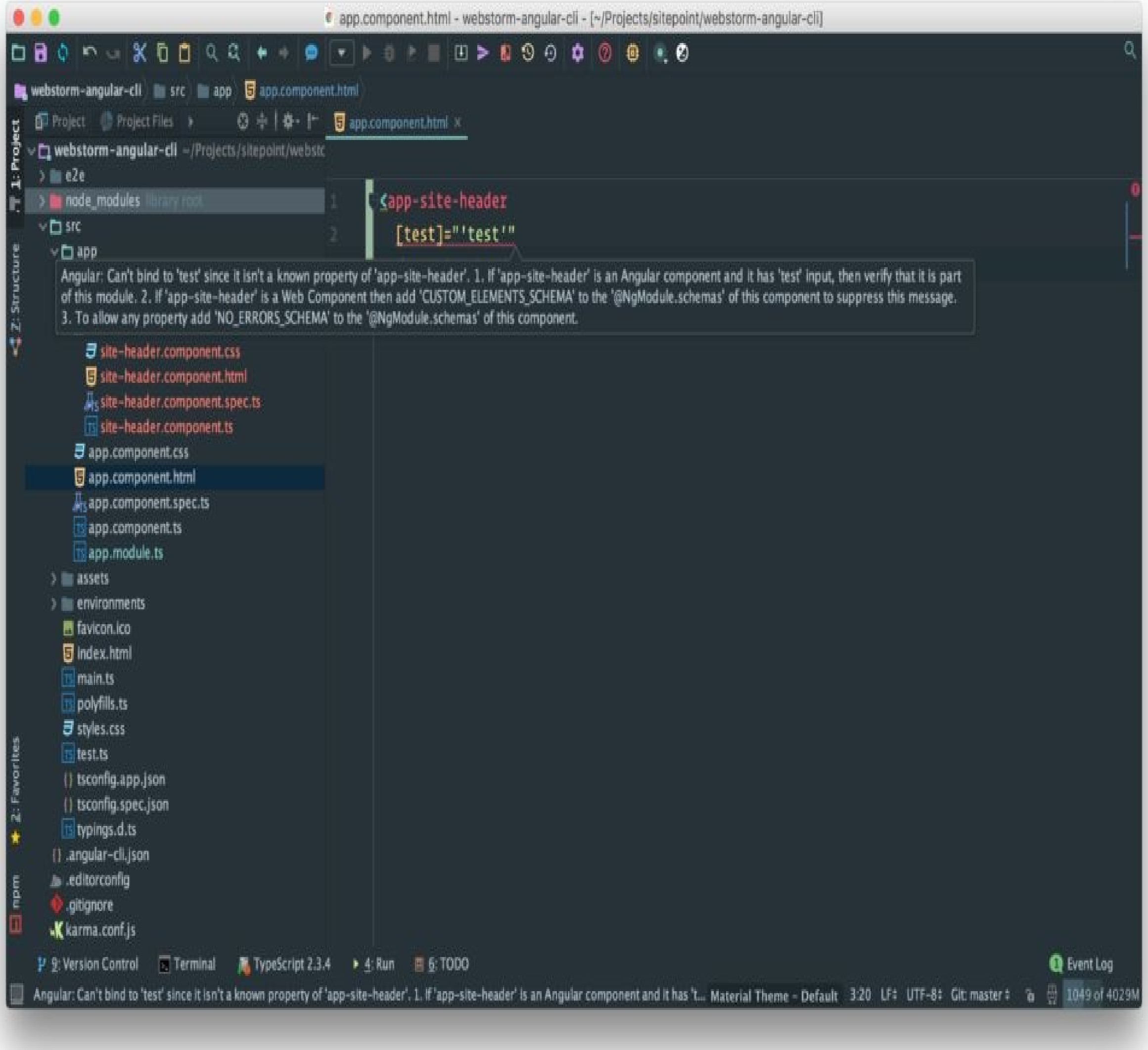
The *Service Options* modal will pop up. Enable *Use Angular service* and apply the changes:



☑ Code completion

☑ Use Angular service

With Angular Language Service enabled, WebStorm is able to improve code completion in template expressions and report template errors more precisely right inside your editor:



```
<app-site-header
  [test]="'test'"
```

Angular: Can't bind to 'test' since it isn't a known property of 'app-site-header'. 1. If 'app-site-header' is an Angular component and it has 'test' input, then verify that it is part of this module. 2. If 'app-site-header' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message. 3. To allow any property add 'NO_ERRORS_SCHEMA' to the '@NgModule.schemas' of this component.

Angular: Can't bind to 'test' since it isn't a known property of 'app-site-header'. 1. If 'app-site-header' is an Angular component and it has 't...

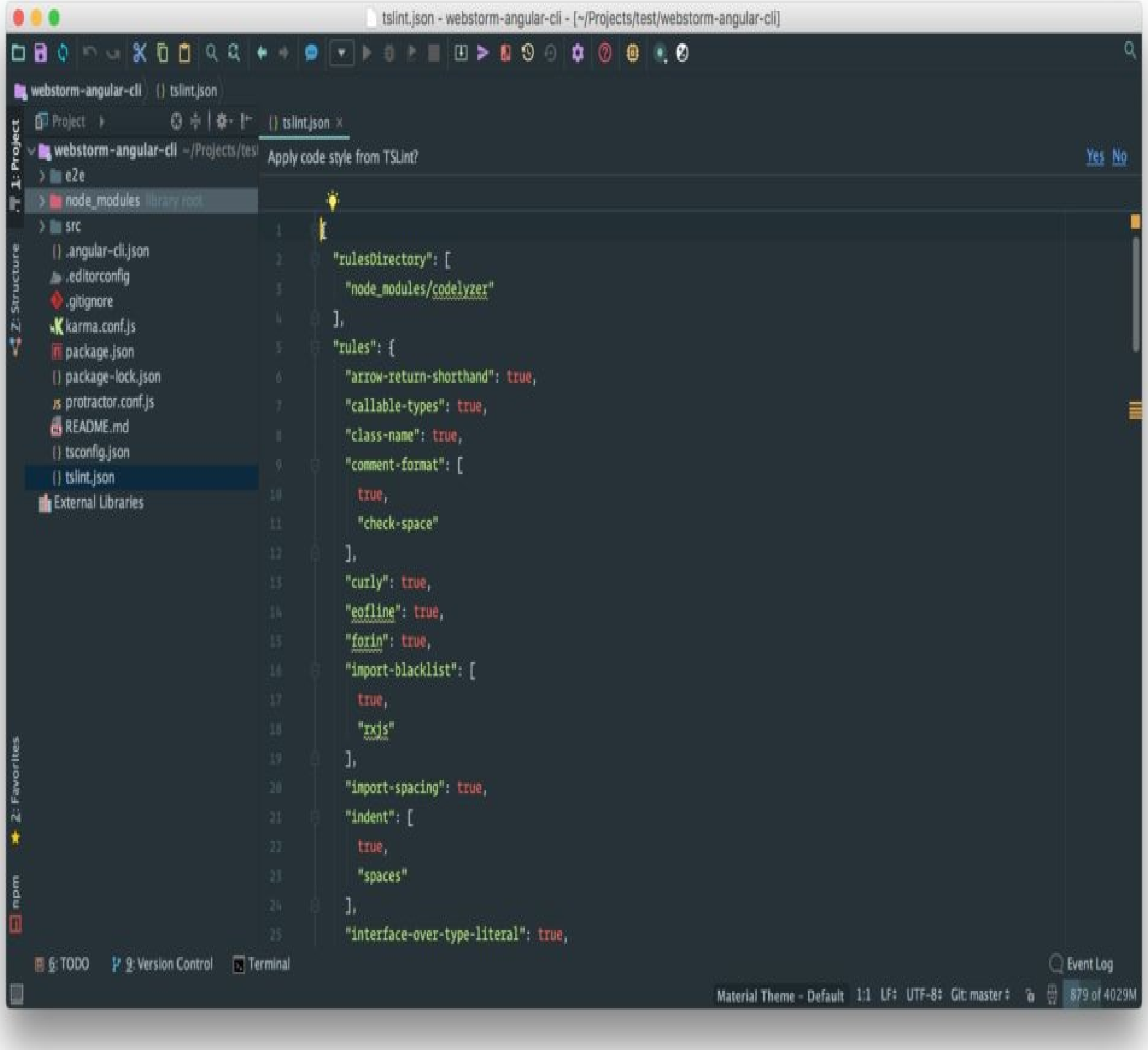
Catching errors without having to compile your project saves you incredible amounts of time.

WebStorm and Angular Tip 4: Auto-format Your Code

Don’t worry about formatting your code manually. WebStorm has you covered.

Whether your’re in a template, a script, a stylesheet or even a JSON file, just hit `cmd` + `option` + `B` and WebStorm will automatically format all code for you.

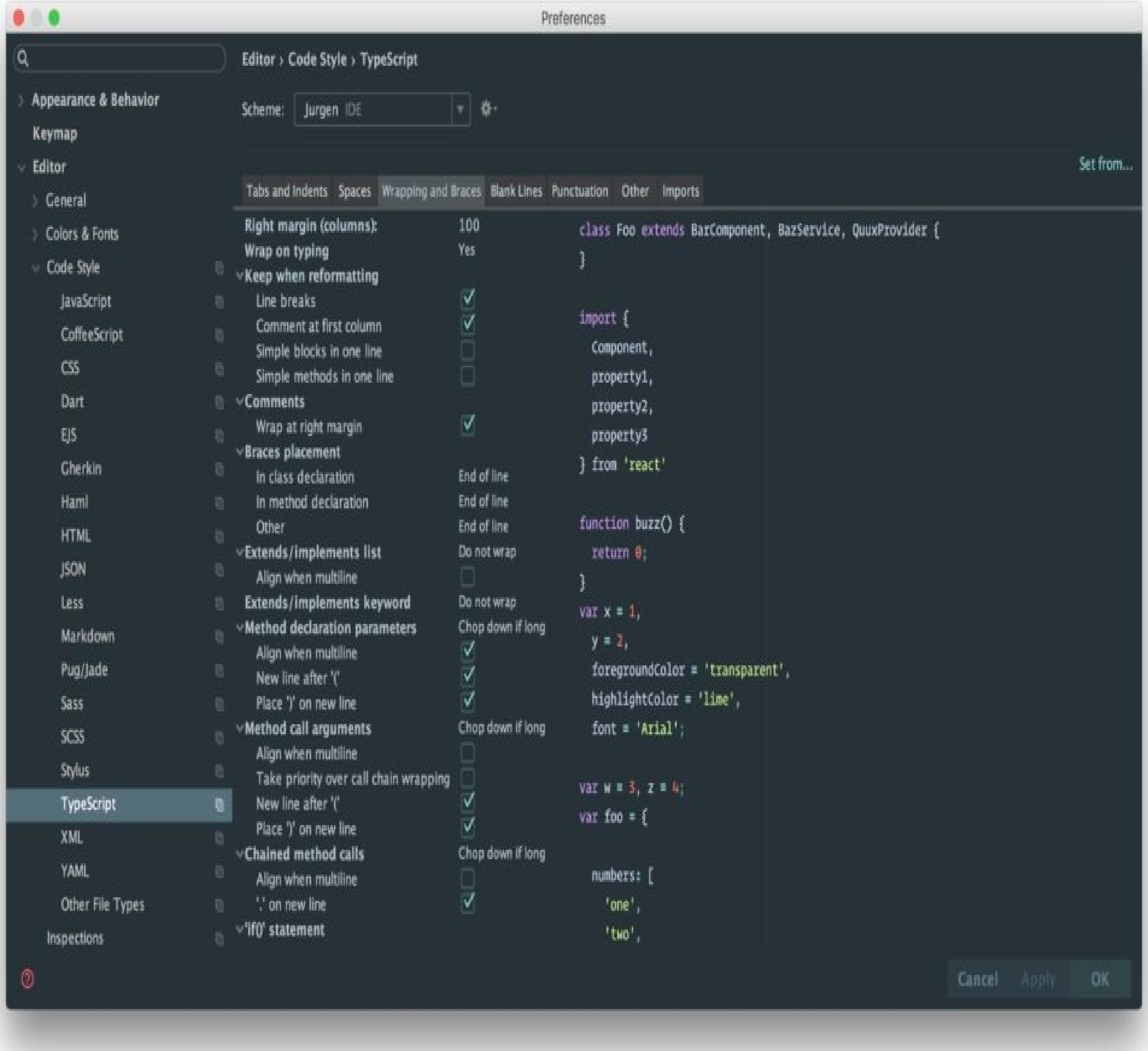
If your project has a `tslint.json` file, just open it up and WebStorm will ask you whether you want to apply the code style from TSLint to your project:



Apply code style from TSLint? Yes No

```
{
  "rulesDirectory": [
    "node_modules/codelyzer"
  ],
  "rules": {
    "arrow-return-shorthand": true,
    "callable-types": true,
    "class-name": true,
    "comment-format": [
      true,
      "check-space"
    ],
    "curly": true,
    "eofline": true,
    "forin": true,
    "import-blacklist": [
      true,
      "rxjs"
    ],
    "import-spacing": true,
    "indent": [
      true,
      "spaces"
    ],
    "interface-over-type-literal": true,
```

If you’re not happy with the style of the auto-formatted code, you can fine tune the format settings for every supported language separately in *Webstorm > Preferences > Editor > Code Style*:



Scheme: Jurgen IDE

Right margin (columns):	100
Wrap on typing	Yes
Keep when reformatting	
Line breaks	☑
Comment at first column	☑
Simple blocks in one line	□
Simple methods in one line	□
Comments	
Wrap at right margin	☑
Braces placement	
In class declaration	End of line
In method declaration	End of line
Other	End of line
Extends/implements list	Do not wrap
Align when multiline	□
Extends/implements keyword	Do not wrap
Method declaration parameters	Chop down if long
Align when multiline	☑
New line after '('	☑
Place ')' on new line	☑
Method call arguments	Chop down if long
Align when multiline	□
Take priority over call chain wrapping	□
New line after '('	☑
Place ')' on new line	☑
Chained method calls	Chop down if long
Align when multiline	□
'.' on new line	☑
'if()' statement	

```
class Foo extends BarComponent, BazService, QuuxProvider {
}


import {
  Component,
  property1,
  property2,
  property3
} from 'react'


function buzz() {
  return 0;
}
var x = 1,
  y = 2,
  foregroundColor = 'transparent',
  highlightColor = 'lime',
  font = 'Arial';

var w = 3, z = 4;
var foo = {


  numbers: [
    'one',
    'two',
```

WebStorm’s code formatting feature ensures that your code is formatted properly according to your project settings, so that your code linting checks pass successfully and you can focus on writing code.

WebStorm and Angular Tip 5: Optimize Your Imports

When working on an Angular script, you may find that certain imports are no longer used.

If you don't remove the unused imports, your bundle size may grow larger than needed. However, removing unused imports can be a real chore. Not with WebStorm!

Hit `ctrl` + `alt O` to optimize your imports instantly. Alternatively, you can hit `cmd` + `shift` + `A` to open up the *Find actions* panel, type `optim` to find the *Optimize imports* action and hit the `enter` key to run the action.

When optimizing imports, WebStorm will do the following for you:

- merge imports from the same module in the same `import` statement
- remove unused imports
- reformat import statements so they fit within your desired line length.

In the following example, *Optimize imports* is run twice. The first time, it merges all imports from `@angular/core` into one import statement.

Then the `OnInit`, `OnChanges` and `AfterViewInit` interfaces are removed from the code and `ctrl` + `alt` + `O` is pressed again.

This time, *Optimize imports* automatically removes the unused interfaces from the import statement, because they're no longer used in the code. Don't ever worry about your import statements again. WebStorm is smart enough to handle them for you!

Using WebStorm and Angular Together: Summary

Let’s recap Jurgen’s personal tips for working with WebStorm and Angular to increase Angular development productivity:

1. *Use Angular CLI from within WebStorm* to quickly generate new Angular projects and features
2. *Navigate like a pro* to instantly jump to code definitions and easily locate code or files you are looking for
3. *Take advantage of Angular Language Service* to get even better code completion and error checking without compiling your Angular project
4. *Auto-format your code* to let WebStorm format all your code according to your project settings
5. *Optimize your imports* to ensure all unused imports are removed and your generated bundle size remains optimal.

In the next part, Todd Motto shares his favorite tips for working with WebStorm and Angular too.

Chapter 7: Top 12 Productivity Tips for WebStorm and Angular: Part 2

by Todd Motto

This chapter on WebStorm and Angular was sponsored by JetBrains. Thank you for supporting the partners who make SitePoint possible.

In this two-part series on WebStorm and Angular, Google Developer Experts Jurgen Van de Moere and Todd Motto share their 12 favorite productivity tips for developing Angular applications using WebStorm.

In this second part, Todd shares his personal top seven WebStorm features that allow him to increase his WebStorm and Angular productivity on a daily basis:

- Use Import Path Calculation
- Live Templates
- Run Tests within the IDE
- Travel through Time
- Use TypeScript Parameter Hints
- Navigate using Breadcrumbs
- And using WebStorm to look up Angular Documentation

Each WebStorm and Angular tip will power up your productivity while developing Angular applications in WebStorm. Let’s explore these tips.

Ahem, Excuse Me ...

Before we get started: when making changes to settings, remember that WebStorm allows you to change Settings/Preferences at an IDE scope and at a project scope separately.

WebStorm and Angular Tip 6: Import Path Calculation

By default, WebStorm will resolve your import paths relative to the file. This setting will satisfy most projects and avoid unnecessary path editing. It’s also the method used for projects generated with the Angular CLI.

What’s fantastic about WebStorm is that you don’t have to type these import statements yourself! When you need to use a construct that would usually be imported, just type it where you need it. WebStorm will either suggest the construct to import through the AutoComplete context menu, or highlight the construct and give you the option to import it by pressing `option` + `enter`. WebStorm will create a new import statement at the top of the document for you, or add the construct to an existing import group that’s using the same source library.

WebStorm gives you other specialized options to handle your imports. For projects that require it, you can instruct WebStorm to calculate import paths relative to the `tsconfig.json` file location. If you decide to roll up your exports using a barrel `index.ts` file to import your components (read more about the Barrel technique) then you can use *use directory import (Node-style module resolution)*. This will use the Node.js module resolution strategy, instead of TypeScript’s classic module resolution strategy.

When importing submodules that don’t require the entire module to be imported, add that module to the *Do not import exactly from* list. WebStorm will skip the specified path during the automatic import. For example, instead of having:

```
import {Observable} from 'rxjs'
```

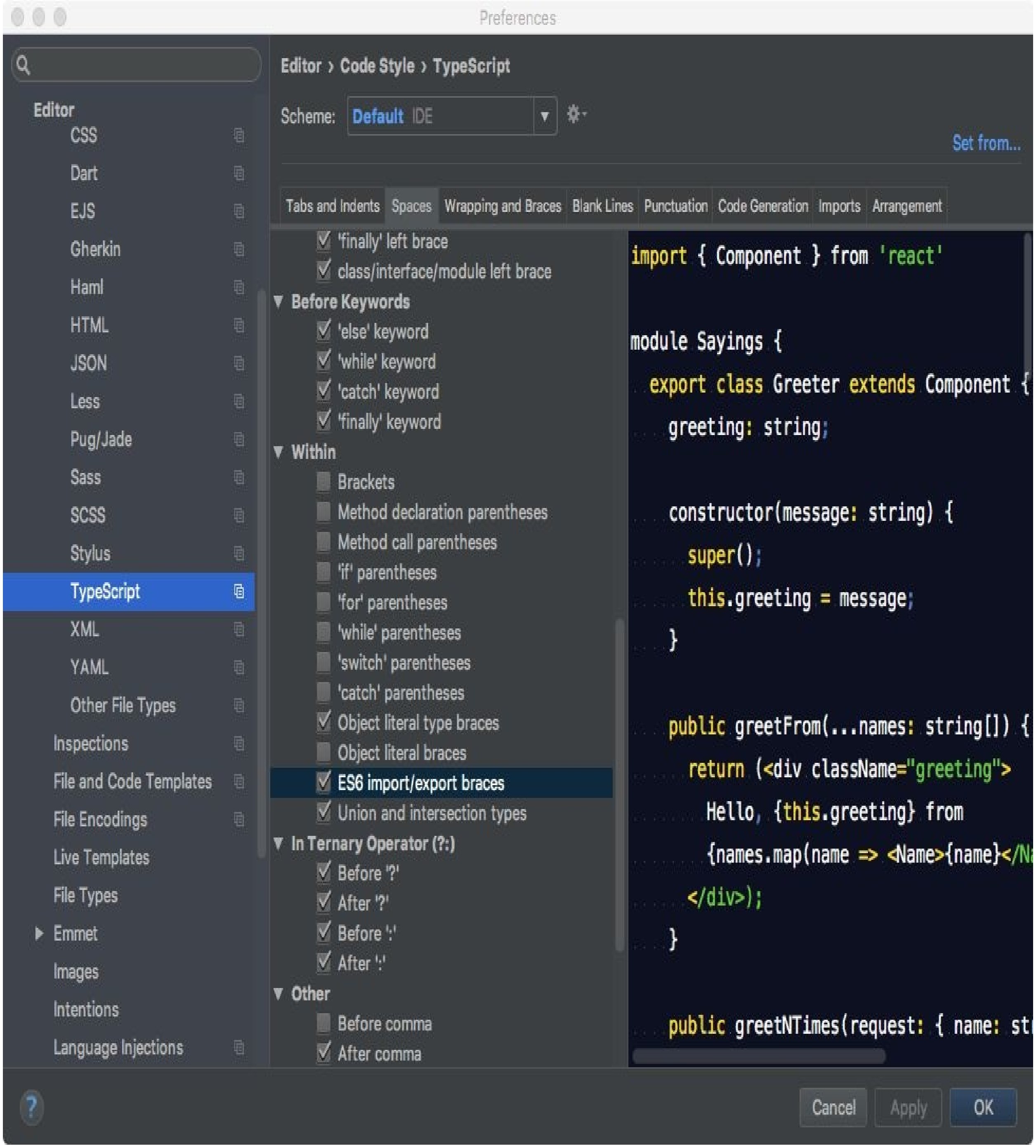
… adding **rxjs** to the list yields:

```
import {Observable} from 'rxjs/Observable'
```

WebStorm skips the RxJS module and imports the Observable submodule automatically for you!

Extra Tip!

Format input to use space inside curly braces in *Preferences > Editor > Code style > TypeScript – Spaces – Within - ES6 import/export braces*



☑ 'finally' left brace
☑ class/interface/module left brace

▼ Before Keywords
☑ 'else' keyword
☑ 'while' keyword
☑ 'catch' keyword
☑ 'finally' keyword

▼ Within
□ Brackets
□ Method declaration parentheses
□ Method call parentheses
□ 'if' parentheses
□ 'for' parentheses
□ 'while' parentheses
□ 'switch' parentheses
□ 'catch' parentheses
☑ Object literal type braces
□ Object literal braces
☑ ES6 import/export braces
☑ Union and intersection types

▼ In Ternary Operator (?:)
☑ Before '?'
☑ After '?'
☑ Before ':'
☑ After ':'

▼ Other
□ Before comma
☑ After comma

```
import { Component } from 'react'

module Sayings {
  export class Greeter extends Component {
    greeting: string;

    constructor(message: string) {
      super();
      this.greeting = message;
    }

    public greetFrom(...names: string[]) {
      return (<div className="greeting">
        Hello, {this.greeting} from
        {names.map(name => <Name>{name}</Na
      </div>);
    }

    public greetNTimes(request: { name: st
```

WebStorm and Angular Tip 7: Use Live Templates

When you find yourself writing certain patterns of code repeatedly, create a Live Template to quickly scaffold the block of code. WebStorm already comes with some predefined Live Templates that you can modify to fit your development style.

To create a Live Template, navigate to the following locations:

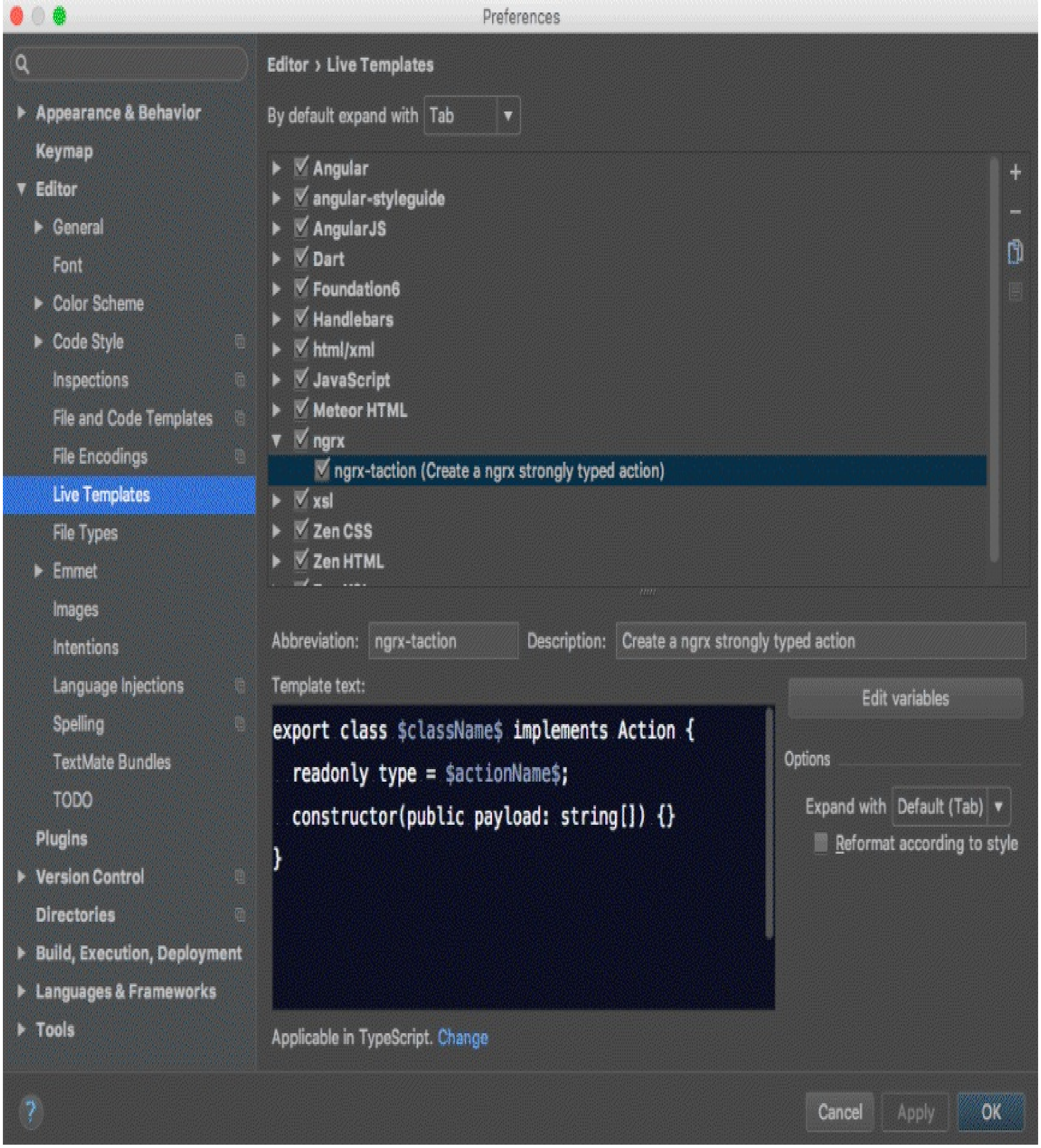
- [macOS] *WebStorm* > *Preferences* > *Editor* > *Live Templates*
- [Windows / Linux] *File* > *Settings* > *Editor* > *Live Templates*

You'll see that WebStorm has already bundled the predefined Templates into categories. I created a category to bundle my ngrx Live Templates by clicking on the + sign and choosing *Template Group*. I then created a new Live template within it by clicking on the + sign again, but choosing Live Template this time.

Let me walk you briefly through the elements that make a Live Template a productivity success:

- **Abbreviation:** the shortcut you'll type into the editor to invoke your template.
- **Description:** tells you what the template does when invoked.
- **Template text:** this is the code fragment to be scaffolded upon invocation. Take advantage of the powerful Live Template Variables that allow you to replace them with your desired text upon scaffolding.
- **Context:** choose in which language or pieces of code WebStorm should be sensitive to the Template.
- **Options:** define which key will allow you to expand the template and reformat it, according to the style settings defined on *WebStorm* > *Preferences* > *Editor* > *Code Style*.

You're ready to try out your template. Open a file that honors the context you defined and type your shortcut, press the defined expansion key and watch your template appear for you! If you defined any variables, the cursor will be placed where the first variable should be entered. If there are other variables defined, you can use tab to navigate to them — no need to click.



By default expand with: Tab

Abbreviation: ngrx-taction

Description: Create a ngrx strongly typed action

Template text:

```
export class $className$ implements Action {
  readonly type = $actionName$;
  constructor(public payload: string[]) {}
}
```

Expand with: Default (Tab)

□ Reformat according to style

Applicable in TypeScript. Change

WebStorm and Angular Tip 8: Running Tests

WebStorm is an excellent testing tool. You can run a variety of JavaScript tests right from the IDE, as long as you have the Node.js runtime environment installed on your computer and the NodeJS plugin enabled. Here are some productivity tips when running tests.

You can run single Karma tests as opposed to running them all. Click the icon next to the test in the editor and select *Run* or *Debug*. The icon will show the test status.

Run and debug tests with Protractor right from the IDE. Make sure that Protractor is installed globally. Test results will be presented in a tool window. You can filter the test results and opt to only display failing tests.

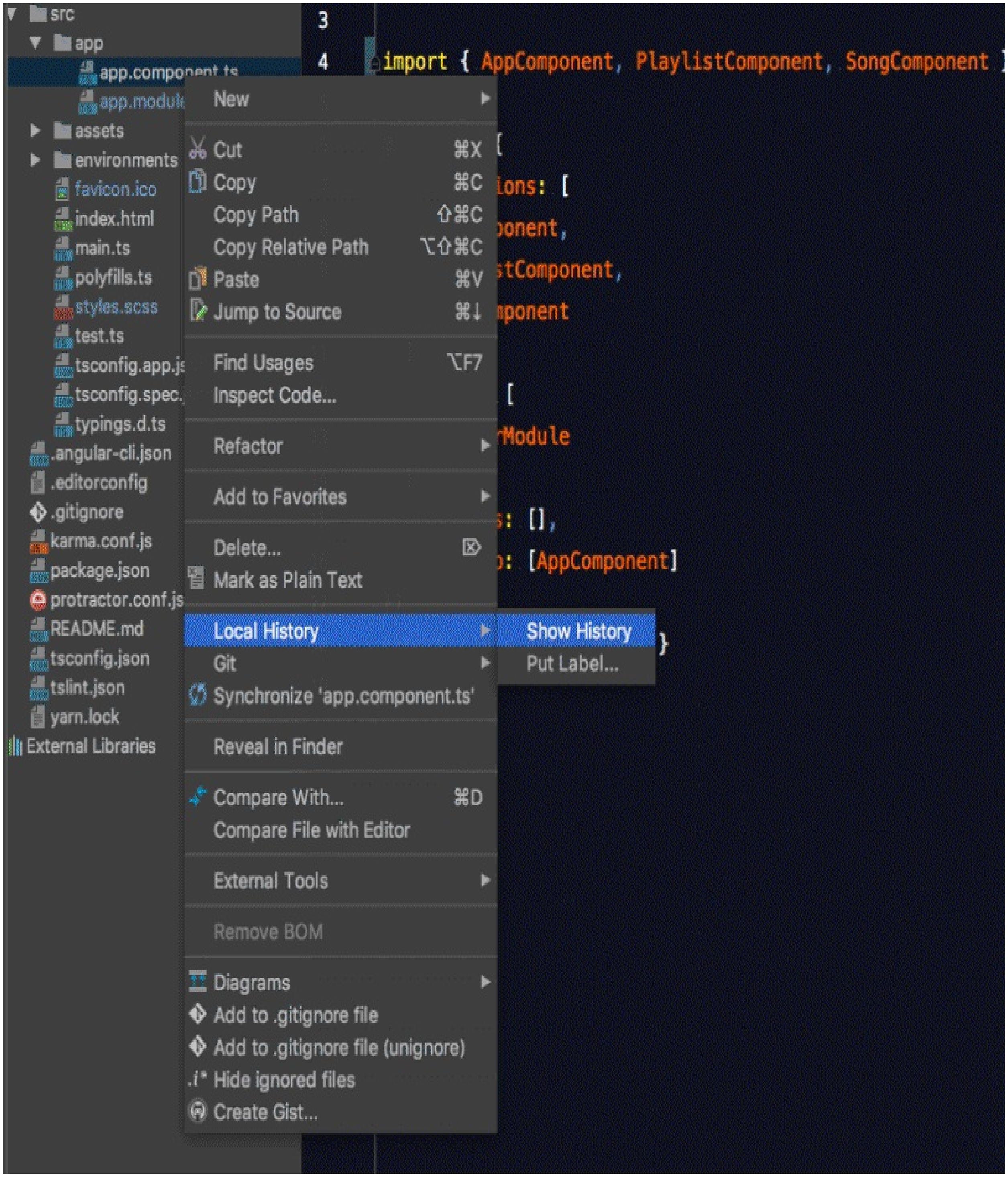
As an extra tip, you can test RESTful services right from the IDE! Go to *Tools > Test RESTful Web Service*.

WebStorm and Angular Tip 9: Travel through Time

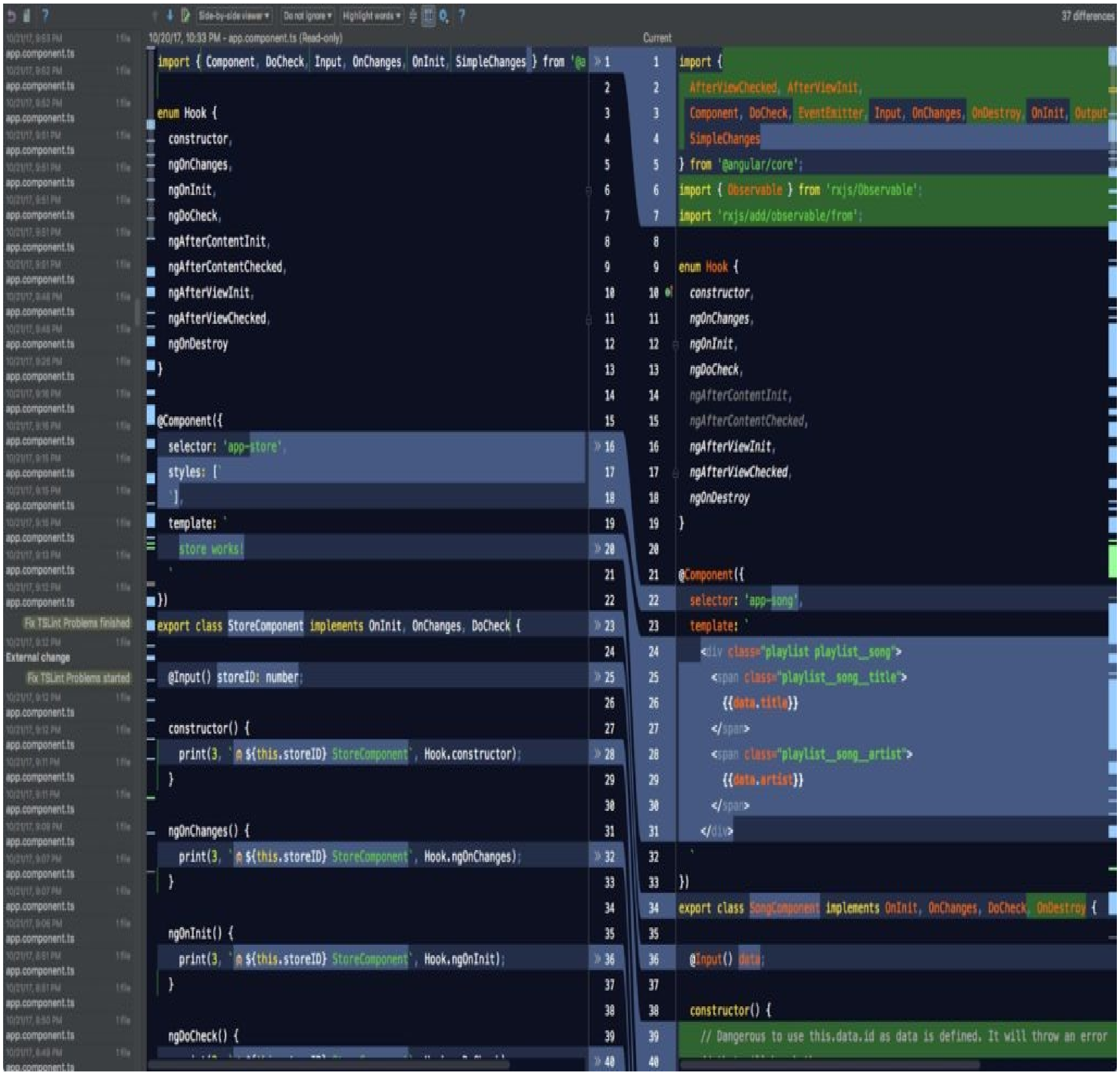
This is an area where WebStorm can save the day. You may have coded many new lines non-stop and forgotten to commit your code to version control periodically — or to initiate it at all. It happens. Whatever the context, there are situations where we need to go back in time.

We can use “Undo”, but WebStorm has a default limit of 100 undos. This limit can be increased, but there’s a better way to restore a previous state: using Local History.

With Local History, you can navigate through snapshots of your code to visually find the codebase state that you want to reach. It’s similar to version control, contrasting current state with previous state side by side. (A Unified Viewer is also available.) However, Local History is independent from version control: you can use it even if you haven’t initiated git, for example. The snapshots will be easy to navigate, as they’re sorted by time — from newest to oldest.



```
import { AppComponent, PlaylistComponent, SongComponent 

                   [
                   ions: [
                   ponent,
                   stComponent,
                   nponent

                    [
                   rModule

                   s: [],
                   p: [AppComponent]
                          }
```



10/20/17, 10:33 PM - app.component.ts (Read-only)

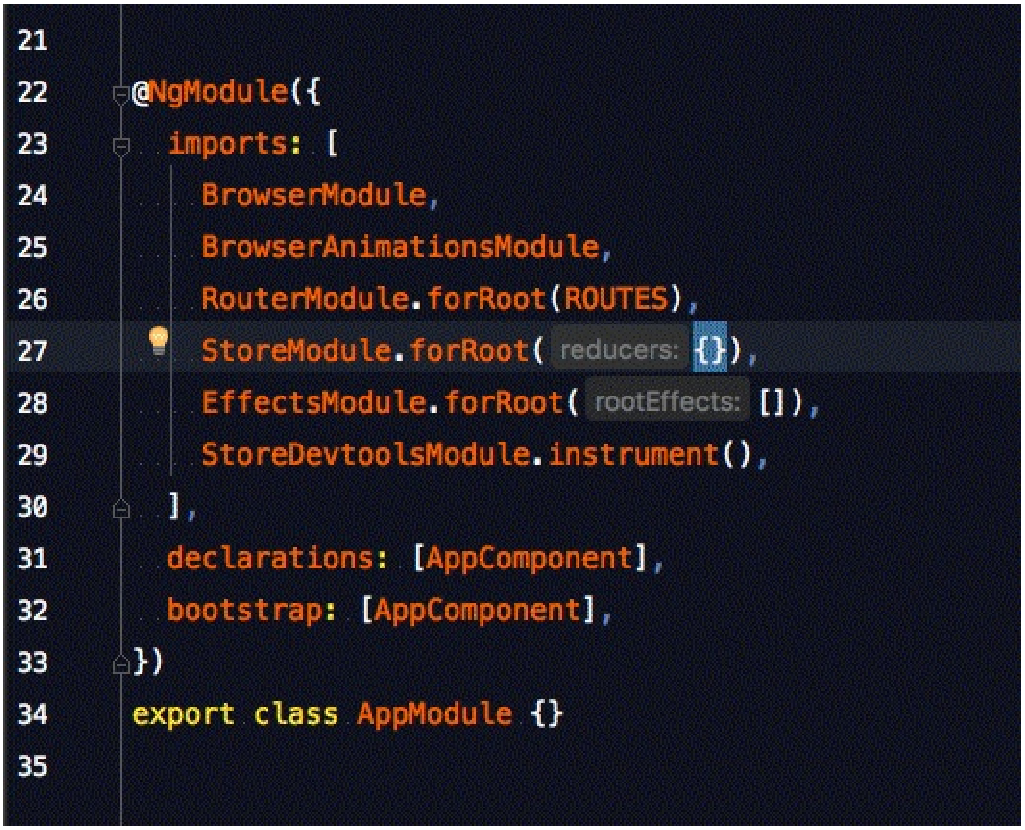
```
import { Component, DoCheck, Input, OnChanges, OnInit, SimpleChanges } from '@a

enum Hook {
  constructor,
  ngOnChanges,
  ngOnInit,
  ngDoCheck,
  ngAfterContentInit,
  ngAfterContentChecked,
  ngAfterViewInit,
  ngAfterViewChecked,
  ngOnDestroy
}

@Component({
  selector: 'app-store',
  styles: [`
  `],
  template: `
    store works!
  `
})
export class StoreComponent implements OnInit, OnChanges, DoCheck {

  @Input() storeID: number;

  constructor() {
    print(3, `@ ${this.storeID} StoreComponent`, Hook.constructor);
  }

  ngOnChanges() {
    print(3, `@ ${this.storeID} StoreComponent`, Hook.ngOnChanges);
  }

  ngOnInit() {
    print(3, `@ ${this.storeID} StoreComponent`, Hook.ngOnInit);
  }

  ngDoCheck() {
```

Current

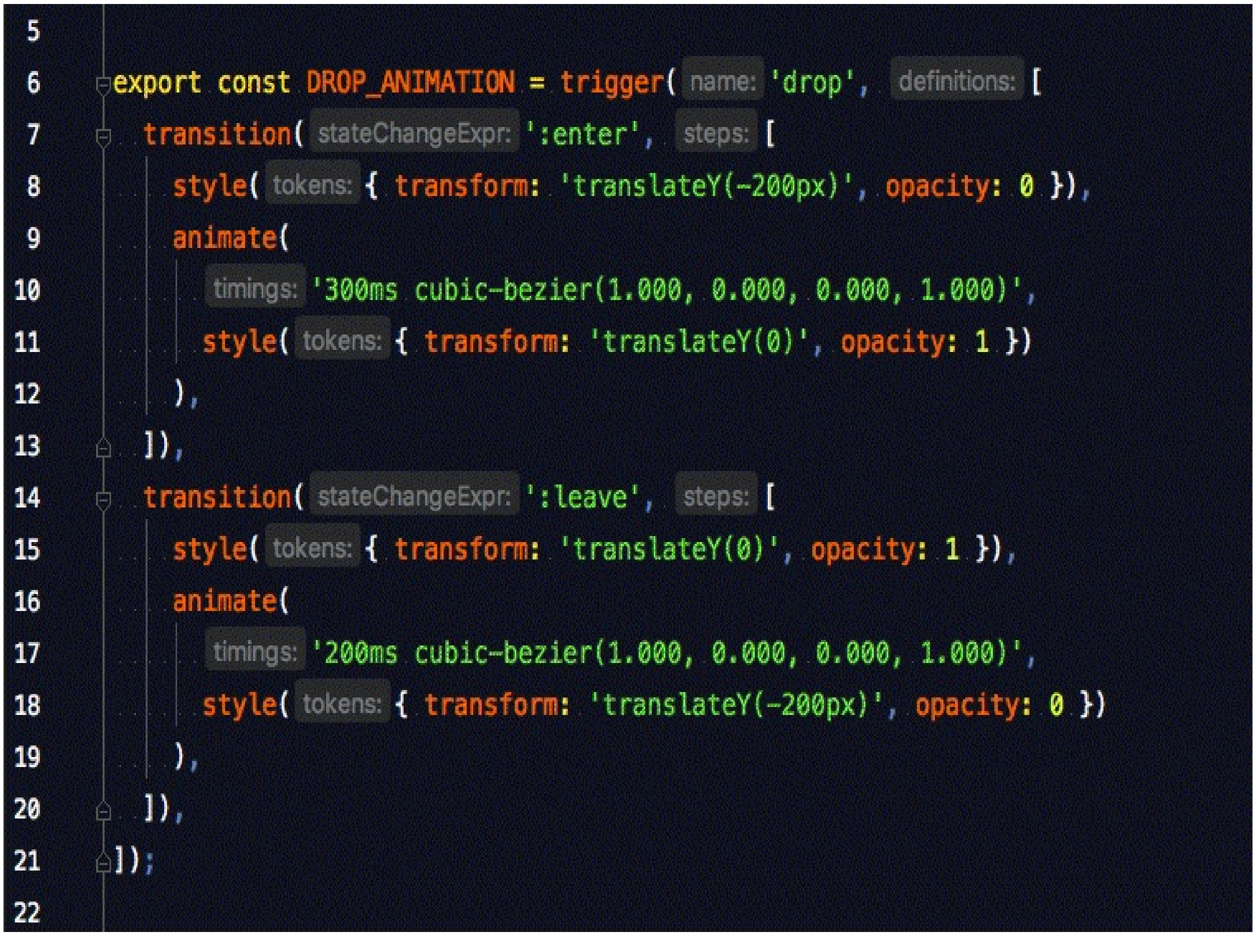
```
import {
  AfterViewChecked, AfterViewInit,
  Component, DoCheck, EventEmitter, Input, OnChanges, OnDestroy, OnInit, Output
  SimpleChanges
} from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/from';

enum Hook {
  constructor,
  ngOnChanges,
  ngOnInit,
  ngDoCheck,
  ngAfterContentInit,
  ngAfterContentChecked,
  ngAfterViewInit,
  ngAfterViewChecked,
  ngOnDestroy
}

@Component({
  selector: 'app-song',
  template: `
    <div class="playlist playlist__song">
      <span class="playlist__song__title">
        {{data.title}}
      </span>
      <span class="playlist__song__artist">
        {{data.artist}}
      </span>
    </div>
  `
})
export class SongComponent implements OnInit, OnChanges, DoCheck, OnDestroy {

  @Input() data;

  constructor() {
    // Dangerous to use this.data.id as data is defined. It will throw an error
```

WebStorm and Angular Tip 10: TypeScript Parameter Hints

TypeScript parameter hints show the names of parameters in methods and functions to make your code easier to read. By default, only certain parameter hints are shown based on their type, and some hints for common methods are hidden.



```

@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    RouterModule.forRoot(ROUTES),
    StoreModule.forRoot( reducers: {}),
    EffectsModule.forRoot( rootEffects: []),
    StoreDevtoolsModule.instrument(),
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
})
export class AppModule {}

```



```
export const DROP_ANIMATION = trigger( name: 'drop', definitions: [
  transition( stateChangeExpr: ':enter', steps: [
    style( tokens: { transform: 'translateY(-200px)', opacity: 0 }),
    animate(
      timings: '300ms cubic-bezier(1.000, 0.000, 0.000, 1.000)',
      style( tokens: { transform: 'translateY(0)', opacity: 1 })
    ),
  ]),
  transition( stateChangeExpr: ':leave', steps: [
    style( tokens: { transform: 'translateY(0)', opacity: 1 }),
    animate(
      timings: '200ms cubic-bezier(1.000, 0.000, 0.000, 1.000)',
      style( tokens: { transform: 'translateY(-200px)', opacity: 0 })
    ),
  ]),
]);
```

To avoid distractions, you can blacklist parameter hints so that they aren't shown for that method again in the project. You can also enable all parameters or disable them completely. To show parameter hints for all arguments:

- open the *Appearance* page in *WebStorm* > *Preferences* > *Editor* > *General* > *Appearance*.
- Click *Configure* next to the checkbox labeled "Show parameter name hints".
- In the *Options* box, you can select "Show name for all arguments".*

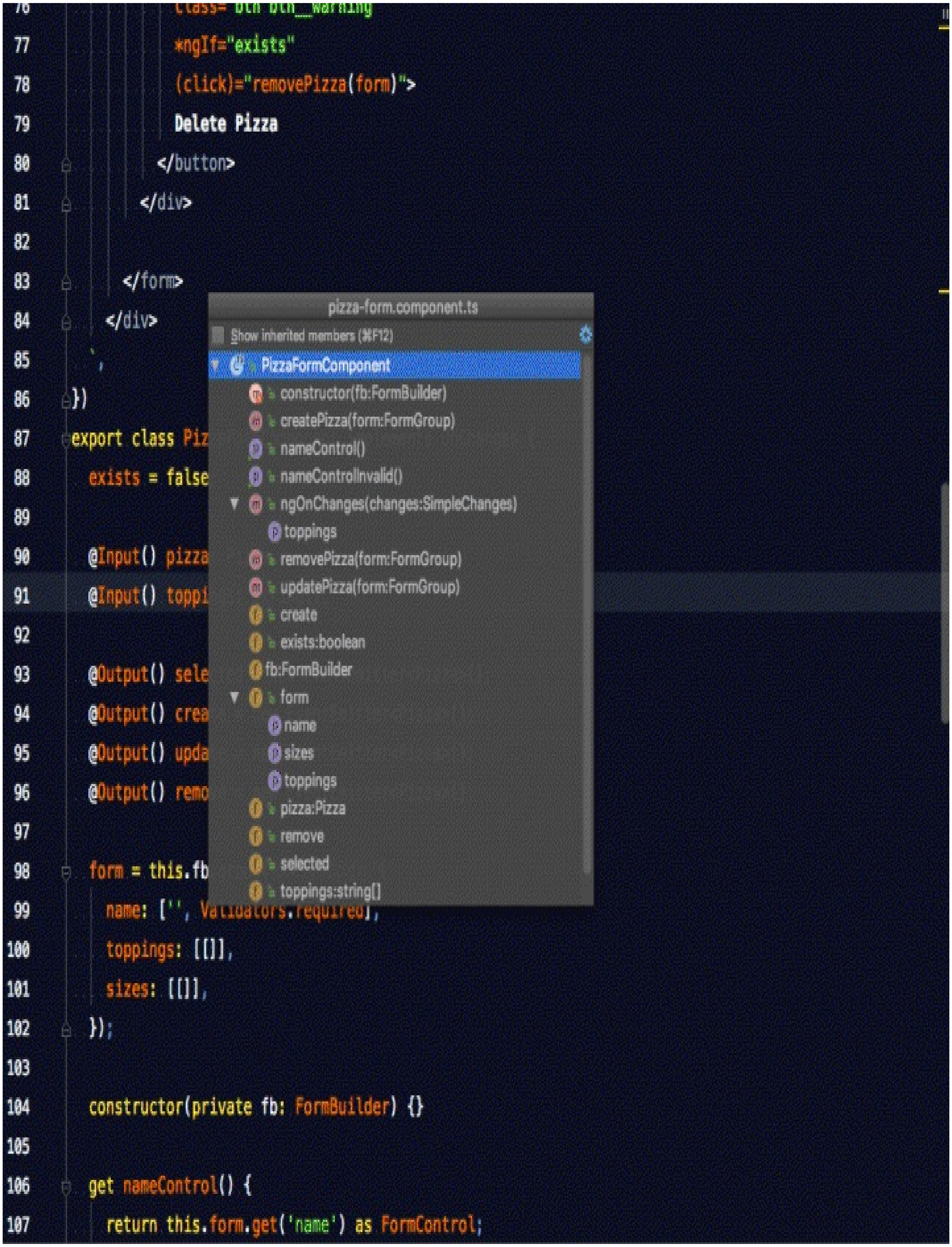
Also Available

In this same dialog box, you can modify the parameter hints blacklist.

WebStorm and Angular Tip 11: Navigate using Breadcrumbs

Looking at the bottom of your `.ts` file, you can see your location in the current file with breadcrumbs. Breadcrumbs show the names of classes, variables, functions and methods. Click on the name of a breadcrumb to jump to the parent object. You can configure them to show at the top of the editor, or not at all, by right-clicking on a breadcrumb and then clicking *Breadcrumbs > Top* or *Breadcrumbs > Don't show*.

As an alternative, you can navigate a file by using the File Structure popup window. It can be accessed through *Navigate > File Structure* or by pressing `CMD` + `F12` on macOS and `Ctrl` + `F12` on Windows/Linux.



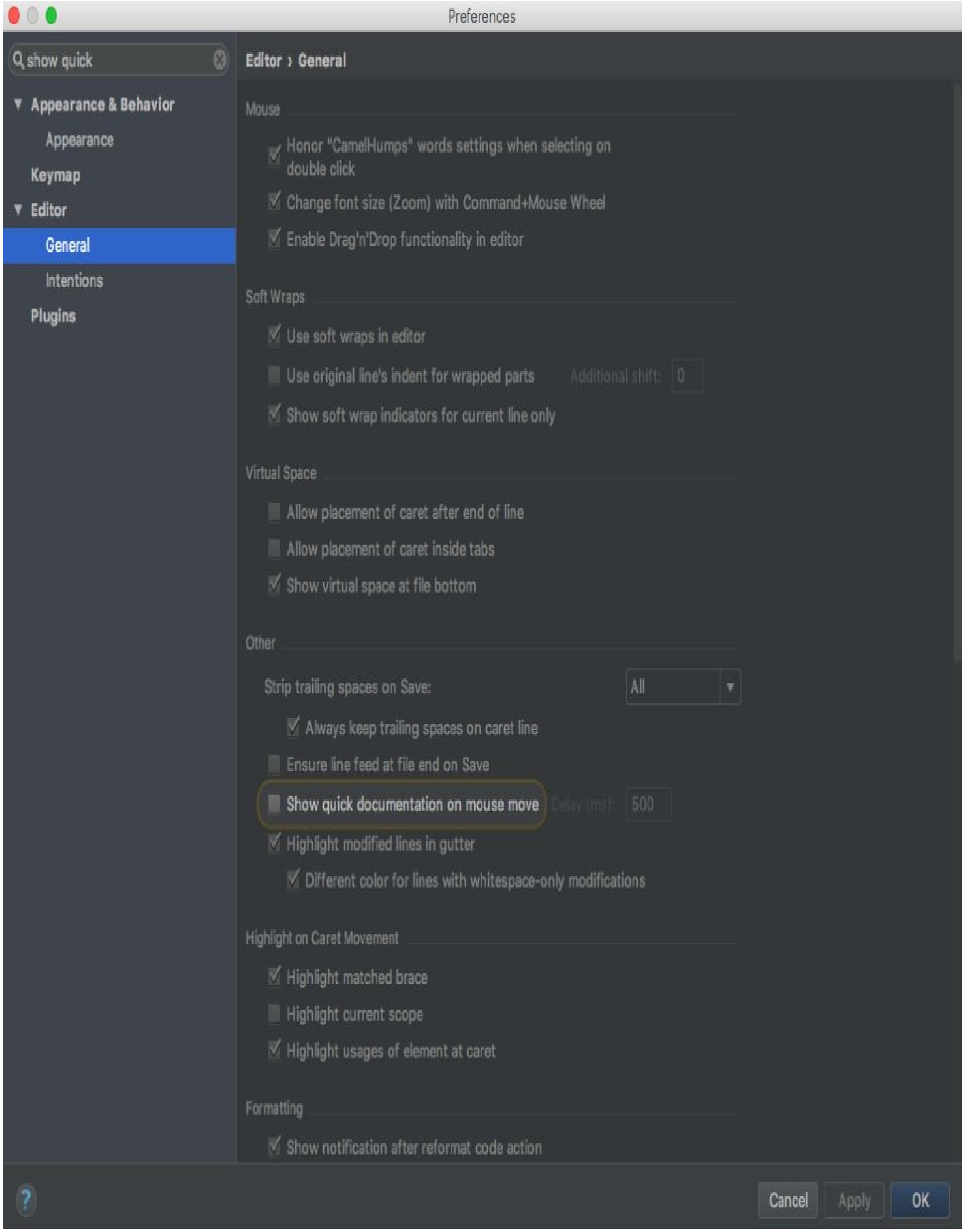
```
          class="btn btn__warning"
          *ngIf="exists"
          (click)="removePizza(form)">
          Delete Pizza
        </button>
      </div>

    </form>
  </div>
  `,
})
export class Piz
  exists = false

  @Input() pizza
  @Input() toppi

  @Output() sele
  @Output() crea
  @Output() upda
  @Output() remo

  form = this.fb
    name: ['', Validators.required],
    toppings: [[]],
    sizes: [[]],
  });

  constructor(private fb: FormBuilder) {}

  get nameControl() {
    return this.form.get('name') as FormControl;
```

WebStorm and Angular Extra Tip 12: Documentation Look Up

There’s no need to leave the IDE to get deeper information on what Angular is doing.

As long as WebStorm has documentation for the Angular construct in question, you can place the cursor on an Angular method or function, and press `F1` to quickly view documentation for it. This also works for modules and other Angular constructs.

Another quick way to invoke documentation is to rest the mouse over a construct. In *Preferences* > *Editor* > *General*, check the “Show quick documentation on mouse move” box. You can adjust the time delay before the quick documentation is invoked.



Editor › General

Mouse

☑ Honor "CamelHumps" words settings when selecting on double click

☑ Change font size (Zoom) with Command+Mouse Wheel

☑ Enable Drag'n'Drop functionality in editor

Soft Wraps

☑ Use soft wraps in editor

□ Use original line's indent for wrapped parts Additional shift: 0

☑ Show soft wrap indicators for current line only

Virtual Space

□ Allow placement of caret after end of line

□ Allow placement of caret inside tabs

☑ Show virtual space at file bottom

Other

Strip trailing spaces on Save: All ▾

☑ Always keep trailing spaces on caret line

□ Ensure line feed at file end on Save

□ Show quick documentation on mouse move Delay (ms): 500

☑ Highlight modified lines in gutter

☑ Different color for lines with whitespace-only modifications

Highlight on Caret Movement

☑ Highlight matched brace

□ Highlight current scope

☑ Highlight usages of element at caret

Formatting

☑ Show notification after reformat code action

As a last word: don't just follow these tips! Measure their results. WebStorm allows you to get a Productivity Report in *Help* > *Productivity Guide*. You can see how much typing code completion has saved you!

Summary

Let's recap Todd’s personal tips for increasing Angular development productivity in WebStorm:

- use efficient import techniques
- use Live Templates to scaffold code patterns quickly
- run and monitor a wide variety of tests within the IDE
- use Local History as an independent, personal, real-time version control system
- improve the readability of your code by using TypeScript parameter hints
- navigate to a file quickly by using Breadcrumbs or the File Structure tool
- look up Angular documentation right from the editor.

This concludes our WebStorm and Angular productivity tips for developing Angular applications with WebStorm by Google Developer Experts. You’re now empowered to streamline your development workflow and make the most of your time at the keyboard.