Introduction to Angular 2

Amin Meyghani

Contents

1

1   Notes

- The book assumes that you are working in a Unix-like environment. If you are on Windows you can use Cygwin so that you can follow along with the bash terminal commands.

- All the project files for the book are hosted on github: https://github.com/st32lth/angular2-intro. You can clone the repository and check out the project files. Throughout the book, you will see references to the project files. Those refer to this repository. For example, `angular2-intro/project-files/hello-angular` refers to the `hello-angular` folder inside the `project-files` folder.

- Make sure you have `git` installed on your machine. That is, make sure you get an output for `git --version`.

- The book assumes that you have a working knowledge of JavaScript and Angular 1.x

- Node is heavily used throughout the book. Make sure that you follow the "Node" chapter to install Node and set permissions correctly.

- All the keyboard shortcuts are mac-based. But if you are using a non-mac machine, you can almost always replace `command` with `ctrl` and you should be good. For example, if you a see a shortcut like `command + shift + b`, you can use `ctrl + shift + b` where `ctrl` is obviously the `control` key.

2   Installing Node

You can use [nvm](#) to install and manage Node on your machine. Copy the install script and run it:

```
1  curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.30.1/install.sh | bash
```

After that, make a new terminal window and make sure that it is installed, by running:

```
1  nvm --help
```

Now you can use `nvm` to install Node `0.12.9` by running:

```
1  nvm install 0.12.9
```

After that, nvm is going to load version 0.12.9 automatically. If it doesn't, you can load it in the current shell, with:

```
1   nvm use 0.12.9
```

Note that you can load any node version in the current shell with `nvm use 0.x.y` after installing that version.

Also note that if you want to make `0.12.9` the default Node version on your machine, you can do so by running the following:

```
1   nvm alias default 0.12.9
```

Then you can verify that it is the default version by making a new terminal window and typing `node -v`.

2.1   Permissions

Never use `sudo` to install packages, never do `sudo npm install <package>`. If you get permission errors while installing without `sudo`, you can own the folders instead. So for example, if you get an error like:

```
1   Error: EACCES, mkdir '/usr/local'
```

you can own the folder with:

```
1   sudo chown -R `whoami` /usr/local
```

You can own folders until Node doesn't complain.

2.2   Installing `live-server`

Install a package to verify that node is installed and everything is wired up correctly. We are going to use `live-server` through the book. So let's install that with:

```
1  npm i -g live-server
```

Then, you should be able to run `live-server` in any folder to serve the content of that folder:

```
1  mdkir ~/Desktop/sample && cd $_
2  live-server .
```

3   Visual Studio Code

Visual Studio Code is a good IDE for developing web apps. In this chapter we will look at installing and configuring VSCode.

3.1   Visual Studio Code Basics

- Install Visual Studio Code from: https://code.visualstudio.com/

- You can open new projects by going to the `File > Open` tag, to etierh open a folder containing your project or a single file

- Some useful keyboard shortcuts are:

    - `command + b`: to close/open the file navigator
    - `command + shift + p`: to open the prompt

- To install extensions open the prompt with `command + shift + p` and type:

    ```
    > install extension
    ```

- You can change the keyboard shortcuts settings from `Preferences >
  Keyboard Shortcuts`. Open the settings and then you can add your own
  shortcuts:

```
// Place your key bindings in this file to overwrite the defaults
[
  {
    "key": "cmd+t",
    "command": "workbench.action.quickOpen"
  },
  {
    "key": "shift+cmd+r",
    "command": "editor.action.format",
    "when": "editorTextFocus"
  }
]
```

3.2   Setting up VSCode for TypeScript

In this section we are going to set up Visual Studio Code for TypeScript. The
project files for this chapter are in **angular2-intro/project-files/vscode-demo**.
You can either follow along or check out the folder to see the final result.

3.2.1   Installing TypeScript

Before anything, we need to install the TypeScript compiler. You can install the
TypeScript compiler with npm:

```
1  npm i typescript -g
```

Then to verify that it is installed, run `tsc -v` to see the version of the compiler.
You will get an output like this:

```
message TS6029: Version 1.7.5
```

In addition to the compiler, we also need to install the TypeScript Definition manager for DefinitelyTyped (tsd). You can install tsd with:

```
1   npm i tsd -g
```

Using TSD, you can search and install TypeScript definition files directly from the community driven DefinitelyTyped repository. To verify that tsd is installed, run tsd with the `version` flag:

```
1   tsd --version
```

You should get an output like this:

```
>> tsd 0.6.5
```

After `tsd` and `tsc` are installed, we can compile a hello world program:

make a file called `hello.ts` on your desktop:

```
1   touch ~/Desktop/hello.ts
```

Then, put some TypeScript code in the file:

```
1   echo "const adder = (a: number, b: number): number => a + b;" > ~/Desktop/hello.ts
```

Then you can compile the file to JavaScript:

```
1   tsc ~/Desktop/hello.ts
```

It should output a file in `Desktop/hello.js`:

```
1   var adder = function (a, b) { return a + b; };
```

Now that your TypeScript compiler setup, we can move on to configuring Visual Studio Code.

3.2.2   Add VSCode Configurations

- First download and install Visual Studio Code from the VSCode Website

- After installing VSCode, open it and then make a new window: `File > New Window`

- Then, make a folder on your desktop for a new project: `mkdir ~/Desktop/vscode-demo`

- After that, open the folder in VSCode: `File > open` and select the `vscode-demo` folder on your desktop.

- Now we need to make three configuration files:

  1. `tsconfig.json`: configuration for the TypeScript compiler
  2. `tasks.json`: Task configuration for VSCode to watch and compile files
  3. `launch.json`: Configuration for the debugger

The `tsconfig.json` file should be in the root of the project. Let's make the file and put the following in it:

```
 1   {
 2     "compilerOptions": {
 3       "experimentalDecorators": true,
 4       "emitDecoratorMetadata": true,
 5       "module": "commonjs",
 6       "target": "es5",
 7       "sourceMap": true,
 8       "outDir": "output",
 9       "watch": true
10     }
11   }
```

Now to make the `tasks.json` file. Open the prompt with `command + shift + p` and type:

```
> configure task runner
```

Then put the following in the file and save the file:

```
1  {
2     "version": "0.1.0",
3     "command": "tsc",
4     "showOutput": "silent",
5     "isShellCommand": true,
6     "problemMatcher": "$tsc"
7  }
```

The last thing that we need to set up is the debugger, i.e. the `launch.json` file. Right click on the `.vscode` folder in the file navigator and make a new file called `launch.json` and put in the following:

```
1  {
2     "version": "0.1.0",
3     "configurations": [
4       {
5          "name": "TS Debugger",
6          "type": "node",
7          "program": "main.ts",
8          "stopOnEntry": false,
9          "sourceMaps": true,
10         "outDir": "output"
11      }
12    ]
13 }
```

After you save the file, you should be able to see the debugger in the debugger dropdown options.

Now, we are ready to make the `main.ts` file in the root of the project:

main.ts

```
1  const sum = (a: number, b: number): number => a + b;
2  const r = sum(1,2);
3  console.log(r);
```

Now you can start the task to watch the files and compile as you work. Open the prompt with `command + shift + p` and type:

```
> run build tasks
```

you can also use the `command + shift + b` keyboard shortcut instead. This will start the debugger and watch the files. After making a change to `main.ts`, you should be able to see the output in the `output` folder.

After the build task is running, we can put a breakpoint anywhere in our Type-Script code. Let's put a breakpoint on line 2 by clicking on the margin. Then start the debugger by going to the debugger tab and clicking the green play icon.

Now you should see that the program will stop at the breakpoint and you should be able to step over or into your program.

To stop the task you can terminate it. Open the prompt and type:

```
> terminate running task
```

3.3   Running VSCode from the Terminal

If you want to run VSCode from the terminal, you can follow the guide on VS-Code's website. Below is the summary of the guide:

MAC

Add the following to your "bash" file:

```
function code () { VSCODE_CWD="$PWD" open -n -b "com.microsoft.VSCode" --args $*; }
```

Linux

```
sudo ln -s /path/to/vscode/Code /usr/local/bin/code
```

Windows

You might need to log off after the installation for the change to the PATH environmental variable to take effect.

3.4   Debugging App from VSCode

The "vscode-chrome-debug" extension allows you to attach VSCode to a running instance of chrome. This makes it very convenient because you can put breakpoints in your TypeScript code and run the debugger to debug your app. Let's get started.

In order to install the extension open the prompt in VSCode with `command + shift + p` and type:

```
> install extension
```

hit enter and then type:

```
debugger for chrome
```

Then just click on the result to install the extension. Restart VSCode when you are prompted.

After installing the extension, we need to update or create a `launch.json` file for debugging. You can create one in the `.vscode` folder. After you created the file, put in the following:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "Launch Chrome Debugger",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:8080",
      "sourceMaps": true,
      "webRoot": ".",
      "runtimeExecutable": "/Applications/Google Chrome.app/Contents/MacOS/Google Chrom
      "runtimeArgs": ["--remote-debugging-port=9222", "--incognito"]
    }
  ]
}
```

11

Notes:

- Depending on your platform you need to change the `runtimeExecutable` path to Chrome's executable path. After configuring the debugger you need to have a server running serving the app. You can change the `url` value accordingly. Also make sure that the `webRoot` path is set to the root of your web server.

- After that it is a good idea to close all the instances of chrome. Then, put a breakpoint in your code and run the debugger. If everything is set up correctly, you should see an instance of chrome running in incognito mode. To trigger the breakpoint, just reload the page and you should be able to see the debugger paused at the breakpoint.

- Also make sure that you have the compiler running so that you can use the JavaScript output and the sourcemaps to use the debugger. See the TypeScript and VSCode set up for more details.

4 TypeScript Crash-course

In this chapter we will quickly go through the most important concepts in Type-Script so that you can have a better understanding of Angular code that you will write. Knowing TypeScript definitely helps to understand Angular, but again it is not a requirement.

4.1 Types and the Basics

There are 7 types in TypeScript:

- boolean: `var isDone: boolean = false;`
- number: `var height: number = 6;`
- string: `var name: string = "bob";`
- array: `var list:number[] = [1, 2, 3];` also `var list:Array<number> = [1, 2, 3];`
- enum: `enum Color {Red, Green, Blue};`
- any: `var notSure: any = 4;`
- void: `function hello(): void { console.log('hello'); }`

4.2   Interface

- An Interface is defined using the `interface` keyword
- Interfaces are used only during compilation time to check types
- By convention, interface definitions start with an `I`, e.g. : `IPoint`
- Interfaces are used in classical object oriented programming as a design tool
- Interfaces don't contain implementations
- They provide definitions only
- When an object implements an interface, it must adhere to the contract defined by the interface
- An interface defines what properties and methods an object must implement
- If an object implements an interface, it must adhere to the contract. If it doesn't the compiler will let us know.
- Interfaces also define custom types

4.2.1   Basic Interface

Below is an example of an Interface that defines two properties and three methods that implementers should provide implementations for:

```
1  interface IMyInterface {
2    // some properties
3    id: number;
4    name: string;
5
6    // some methods
7    method(): void;
8    methodWithReturnVal():number;
9    sum(nums: number[]):number;
10  }
```

Using the interface above we can create an object that adheres to the interface:

```
1  let myObj: IMyInterface = {
2    id: 2,
3    name: 'some name',
```

```
4
5    method() { console.log('hello'); },
6    methodWithReturnVal () { return 2; },
7    sum(numbers) {
8      return numbers.reduce( (a,b) => { return a + b } );
9    }
10   };
```

Notice that we had to provide values to **all** the properties defined by the Interface, and the implementations for **all** the methods defined by the Interface.

And then of course you can use your object methods to perform operations:

```
1    let sum = myObj.sum([1,2,3,4,5]); // -> 15
```

4.2.2   Classes as Interfaces

Because classes define types as well, they can also be used as interfaces. If you have an interface you can extend it with a class for example:

```
1    class Point {
2      x: number;
3      y: number;
4    }
5    interface Point3d extends Point {
6      z: number;
7    }
8    const point3d: Point3d = {x: 1, y: 2, z: 3};
9    console.log(point3d.x); // -> 1
```

First we are defining a class called `Point` that defines two fields. Then we define an interface called `Point3d` that extends the `Point` by adding a third field. An then we create a point of type `point3d` and assign a value to it. We read the value and it outputs 1.

4.3   Classes

- Classes are heavily used in classical object oriented programming
- It defines what an object is and what it can do
- A class is defined using the `class` keyword followed by a name
- By convention, the name of the class start with an uppercase letter
- A class can be used to create multiple objects (instances) of the same class
- An object is created from a class using the `new` keyword
- A class can have a `constructor` which is called when an object is made from the class
- Properties of a class are called instance variables and its functions are called the class methods
- Access modifiers can be used to make them public or private
- The instance variables are attached to the instance itself but not the prototype
- Methods however are attached to the prototype object as opposed to the instance itself
- Classes can inherit functionality from other classes, but you should favor composition over inheritance or make sure you know when to use it
- Classes can implement interfaces

Let's make a class definition for a car and incrementally add more things to it.

4.3.1   Adding an Instance Variable

The `Car` class definition can be very simple and can define only a single instance variable that all cars can have:

```
1  class Car {
2    distance: number;
3  }
```

- `Car` is the name of the class, which also defines the custom type `Car`
- `distance` is a property that tracks the distance that car has traveled
- Distance is of type `number` and only accepts `number` type.

Now that we have the definition for a car, we can create a car from the definition:

```
1  let myCar:Car = new Car();
2  myCar.distance = 0;
```

- myCar:Car means that myCar is of type Car
- new Car() creates an instance from the Car definition.
- myCar.distance = 0 sets the initial value of the distance to 0 for the newly created car

4.3.2   Adding a Method

So far our car doesn't have any definitions for any actions. Let's define a move method that all the cars can have:

```
1  class Car {
2    distance: number;
3    move():void {
4      this.distance += 1;
5    }
6  }
```

- move():void means that move is a method that does not return any value, hence void.
- The body of the method is defined in { }
- this refers to the instance, therefore this.distance points to the distance property defined on the car instance.
- Now you can call the move method on the car instance to increment the distance value by 1:

```
1  myCar.move();
2  console.log(myCar.distance) // -> 1
```

4.3.3   Using Access Modifiers

If you wanted to tell the compiler that the distance variable is private and can only be used by the object itself, you can use the private modifier before the name of the property:

```
1  class Car {
2    private distance: number;
3    constructor () {
4      ...
5    }
6    ...
7  }
```

- There are 3 main access modifiers in TypeScript: `private`, `public`, and `protected`:

- `private` modifier means that the property or the method is only defined inside the class only.

- `protected` modifier means that the property or the method is only accessible inside the class and the classes derived from the class.

- `public` is the default modifier which means the property or the method is the accessible everywhere and can be accessed by anyone.

4.3.4   Adding a constructor

A `constructor` is a special method that gets called when an instance is created from a class. A class may contain at most one constructor declaration. If a class contains no constructor declaration, an automatic constructor is provided.

Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the cars that are crated from this class, will have their `distance` set to 0 automatically:

```
1  class Car {
2    distance: number;
3    constructor () {
4      this.distance = 0;
5    }
6    move():void {
7      this.distance += 1;
8    }
9  }
```

- `constructor()` is called automatically when a new car is created
- Parameters are passed to the constructor in the `()`
- The body of the constructor is defined in the `{ }`

Now, let's customize the car's constructor to accept `distance` as a parameter:

```
1  class Car {
2    private distance: number;
3    constructor (distance) {
4      this.distance = distance;
5    }
6  }
```

- On line 3 we are passing distance as a parameter. This means that when a new instance is created, a value should be passed in to set the distance of the car.
- On line 4 we are assigning the value of distance to the value that is passed in

This pattern is so common that TypeScript has a shorthand for it:

```
1  class Car {
2    constructor (private distance) {
3    }
4  }
```

Note that the only thing that we had to do was to add `private distance` in the constructor parameter and remove the `this.distance` and `distance: number`. TypeScript will automatically generate that. Below is the JavaScript outputed by TypeScript:

```
1  var Car = (function () {
2    function Car(distance) {
3      this.distance = distance;
4    }
5    return Car;
6  })();
```

18

Now that our car expects a `distance` we have to always supply a value for the distance when creating a car. You can define default values if you want so that the car is instantiated with a default value for the distance if none is given:

```
1  class Car {
2    constructor (private distance = 0) {
3    }
4    getDistance():number { return this.distance; }
5  }
```

Now if I forget to pass a value for the `distance`, it is going to be set to zero by default:

```
1  const mycar = new Car();
2  console.log(mycar.getDistance()); //-> 0
```

Note that if you pass a value, it will override the default value:

```
1  const mycar = new Car(5);
2  console.log(mycar.getDistance()); //-> 5
```

4.3.5   Setters and Getters (Accessors)

It is a very common pattern to have setters and getters for properties of a class. TypeScript provides a very simple syntax to achieve that. Let's take our example above and add a setter and getter for the distance property. But before that we are going to rename `distance` to `_distance` to make it explicit that it is private. It is not required but it is a common pattern to prefix private properties with an underscore.

```
1  class Car {
2    constructor (private _distance = 0) {}
3    getDistance():number { return this._distance; }
4  }
```

In order to create the getter method, we are going to use the `get` keyword and the name for the property followed by ():

19

```
1  class Car {
2    constructor (private _distance = 0) {}
3    get distance() { return this._distance; }
4  }
```

Now we can get the value of `distance`:

```
1  const car2 = new Car(5);
2  console.log(car2.distance) //-> 5
```

Note on line 2 that we didn't call a function. Behind the scenes, TypeScript creates a property for us, that's why it is not a method. Below is the relevant generated JavaScript:

```
1  Object.defineProperty(Car.prototype, "distance", {
2    get: function () { return this._distance; },
3    enumerable: true,
4    configurable: true
5  });
```

JavaScript behind the scenes calls the get function for you to get the value, and that's why we simply did `car2.distance` as opposed to `car2.distance()`. For more information about `Object.defineProperty` checkout the MDN docs.

Similar to the getter, we can define a setter as well:

```
1  class Car {
2    constructor (private _distance = 0) {}
3    get distance() { return this._distance; }
4    set distance(newDistance: number) { this._distance = newDistance; }
5  }
```

Now we can both get and set the distance value:

```
1  const coolCar = new Car();
2  console.log(coolCar.distance); // -> 0
3
4  coolCar.distance = 55;
5  console.log(coolCar.distance); // -> 55
```

20

Note that if we take out the setter, we won't be able to assign a new value to `distance`.

4.3.6   Static Methods and Properties

Static methods and properties belong to the class but not the instances. For example, the `Array.isArray` method is only accessible through the `Array` but not an instance of an array:

```
1  var x = [];
2  x.isArray // -> undefined
3  Array.isArray(x) //-> true
```

- On line 2 we are trying to access the `isArray` method, but obviously it is not defined because `isArray` is a static method.
- On line three we are calling the static `isArray` method from `Array` and we can check if `x` is an array.

If you look at the Array documentation you can see that methods and properties are either defined on the `Array.prototype` or `Array`:

- `Array.prototype.x`: makes `x` available to all the instances of `Array`
- `Array.x`: `x` is static and only available through `Array`.

Now that we have some context, let's see how you can define static methods and properties in TypeScript. Consider the code below:

```
1  class Car {
2    static controls: {isAuto: boolean } = {
3      isAuto: true
4    };
5    static isAuto():boolean {
6      return Car.controls.isAuto;
7    }
8    constructor (private _distance = 0) {}
9    get distance() { return this._distance; }
```

```
10  }
11
12  console.log(Car.controls); // -> { isAuto: true }
13  console.log(Car.isAuto()); // -> true
```

- On line 2 we are defining a static property called `controls` using the `static` modifier. Then we specify the form and then assign a value for it.
- On line 5 we are defining a static method called `isAuto` using the the `static` modifier. This method simply returns the value of `isAuto` from the static `control` object. Not that we get access to the class using the name of the class as opposed to using `this`. i.e. `return Car.controls.isAuto`

4.3.7   Implementing an Interface

Classes can implement one or multiple interfaces. We can make the `Car` class implement two interfaces:

```
1  interface ICarProps {
2    distance: number;
3  }
4  interface ICarMethods {
5    move():void;
6  }
```

Making the `Car` class implement the interfaces:

```
1  class Car implements ICarProps, ICarMethods {
2    distance: number;
3    constructor () {
4      this.distance = 5;
5    };
6    move():void {
7      this.distance += 1;
8    };
9  }
```

The above example is silly, but it shows the point that a class can implement one or more interfaces. Now if the class does not provide implementations for any of the interfaces, the compiler will complain. For example, if we leave out the `distance` instance variable, the compiler will print out the following error:

> error TS2420: Class 'Car' incorrectly implements interface 'ICarProps'. Property 'distance' is missing in type 'Car'.

4.4   Modules

- In TypeScript you can use modules to organize your code, and expose functionalities for others to use.
- Modules can be internal and external
- They can be split into different files and references from other files.
- A module is define using the `module` keyword. The module definition goes inside curly braces: `module MyModule { ... }`

5   Angular Basics

This chapter will walk you through the basics of Angular2. We will start by looking at the basics of components, and then we move onto pipes, events and other concepts. By the end of the chapter you should have a basic understanding of the new concepts in Angular2.

The goal of this chapter is to get your feet wet without scaring you with a lot of details. Don't worry, there will be a lot coming in the later chapters.

5.1   Components

Components are at the heart of Angular. The main idea is that you break down your application into different cohesive components and let the components handle the rest. Every component has a controller defined by a class and a template defined by html. In addition, a component's job is to enable the user experience and delegate everything non-trivial to services.

In this section we are going to write a simple `HelloAngular` component, compile it and run it in the browser. In addition, we will configure VSCode to build the TypeScript files as we go.

Note that there is a lot to talk about components. We are going dive into components a lot more in later chapters, but for now let's just keep things simple.

5.1.1   Project Files

The project files for this chapter are in **angular2-intro/project-files/basic-component**

You can either follow along or just look at the final result. As always, the `node_modules` folder is not included. You would have to install it with `npm i` in the project folder:

```
cd angular2-intro/project-files/basic-component && npm i
```

5.1.2   Getting Started

Make a folder on your desktop called `hello-angular` and navigate to it:

```
1   mkdir ~/Desktop/hello-angular && cd $_
```

Start npm in this folder with `npm init` and accept all the defaults.

After that, install the dependencies with:

```
1   npm i angular2 rxjs -S
```

Then install the "devDependencies":

```
1   npm i systemjs -D
```

After all the dependencies are installed, start VSCode in this folder with `code .`

Then create a `index.html` file in the root of the project and put in the following:

index.html

```
1   <html>
2   <head>
3     <title>Hello Angular</title>
4
5     <script src="/node_modules/angular2/bundles/angular2-polyfills.js"></script>
6     <script src="/node_modules/systemjs/dist/system.src.js"></script>
7     <script src="/node_modules/rxjs/bundles/Rx.js"></script>
8     <script src="/node_modules/angular2/bundles/angular2.dev.js"></script>
9
10    <!-- add systemjs settings later -->
11
12  </head>
13
14  <body>
15    <!-- add app stuff later -->
16  </body>
17
18  </html>
```

This loads all the necessary scripts that we need to run Angular in the browser.

Note

If you need to support older browsers, you need to include the `es6-shims` before everything else:

```
1   <script src="/node_modules/es6-shim/es6-shim.js"></script>
```

5.1.3   Making the Component

Let's start by making the `main.ts` file in the root of the project. In this file we are going to define the main component called `HelloAngular` and then bootstrap the app with it:

main.ts

```
1   import {Component, OnInit } from 'angular2/core';
2   import {bootstrap} from 'angular2/platform/browser';
```

```
3
4   @Component({
5     selector: 'app',
6     template: `<h1> hello angular </h1> `
7   });
8
9   class HelloAngular implements OnInit  {
10    constructor() { console.log('constructor called'); }
11    ngOnInit() { console.log('component initialized'); }
12  }
13
14  bootstrap(HelloAngular, []);
```

- On line 1 we are importing the `component` meta data (annotation) and the `onInit` interface.
- On line 2 we are loading the `bootstrap` method that bootstraps the app given a component.
- On line 4, we are defining a component using the `component` annotation. The `@component` is technically a class decorator because it precedes the `HelloAngular` class definition.
- On line 5, we are telling angular to look out for the `app` tag. So when Angular looks at the html and comes across the `<app></app>` tag, it is going to load the template (on line 6) and instantiates the class for it (defined on line 9).
- On line 9, we are defining a class called `HelloAngular` that defines the logic of the component. And for fun, we are implementing the `OnInit` interface to log something to the console when the component is ready with its data. We will learn more about the lifeCycle hooks later.
- Last but not least, we call the `bootstrap` method with the `HelloAngular` class as the first argument to bootstrap the app with the `HelloAngular` component.

5.1.4   Compiling the Component

Now we need to compile the file to JavaScript. We can do it from the terminal, but let's stick to VSCode. In order to that, we need to make two config files:

1. First is the standard `tsconfig.json` file

2. And the `tasks.json` file for VSCode to do the compiling

Create the `tsconfig.json` file in the root of the project and put in the following:

tsconfig.json

```
1   {
2     "compilerOptions": {
3       "target": "es5",
4       "module": "system",
5       "moduleResolution": "node",
6       "sourceMap": true,
7       "emitDecoratorMetadata": true,
8       "experimentalDecorators": true,
9       "removeComments": false,
10      "noImplicitAny": false,
11      "outDir": "output",
12      "watch": true
13    },
14    "exclude": [
15      "node_modules"
16    ]
17  }
```

Then create the `tasks.json` in the `.vscode` folder in the root of the project and put in the following:

.vscode/tasks.json

```
1   {
2     "version": "0.1.0",
3     "command": "tsc",
4     "showOutput": "silent",
5     "isShellCommand": true,
6     "problemMatcher": "$tsc"
7   }
```

- Now we can build the TypeScript files as we work. We just need to start the build task with `command + shift + b` or using the prompt. If you want to use the prompt do the following:

- – Use `command + shift + p` to open the prompt

  – Then, type `> run build task` and hit enter to start the build task.

- After you run the build task, you should see an `output` file generated with `main.js` and the source maps in it.

- The task is watching the files and compiling as you go. To stop the task, open the prompt and type:

  ```
  > terminate running task
  ```

5.1.5   Loading the Component

After compiling the component, we need to load it to the `index.html` file with `Systemjs`. Open the `index.html` file and replace `<!-- add systemjs settings later -->` with the following:

```
1   <script>
2     System.config({
3       packages: {
4         output: {
5           format: 'register',
6           defaultExtension: 'js'
7         }
8       }
9     });
10    System.import('output/main')
11    .then(null, console.error.bind(console));
12  </script>
```

Now we can use our component in the body of the html:

```
1   <body>
2     <app>Loading ...</app>
3   </body>
```

It is finally time to serve the app. You can serve the app in the current directory using the `live-server`:

```
1   live-server .
```

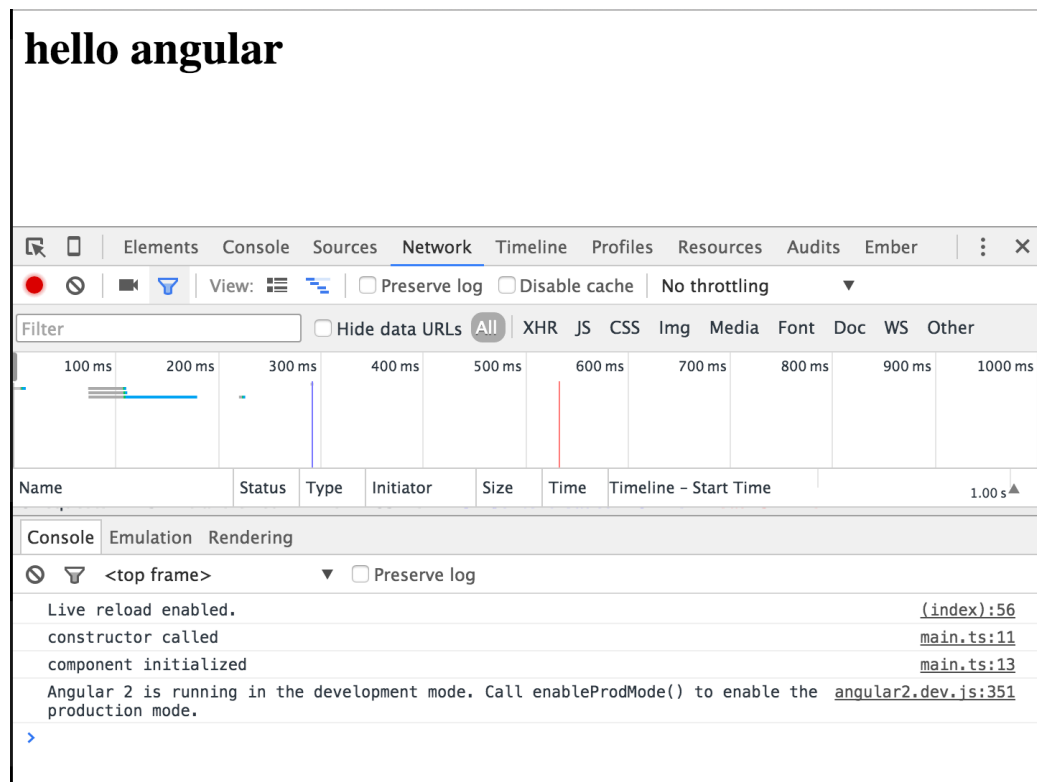If everything is wired up correctly, you should be able to see the following:

Figure 1: Running a basic component in the browser

5.2   Data Architecture

- Angular is flexible and doesn't prescribe a recipe for managing data in your apps
- Since observables are integrated into Angular, you can take advantage of observables to manage data and state

- You ca use services to manage streams that emit models
- Components can subscribe to the streams maintained by services and render accordingly.

    - For example, you can have a service for a Todo app that contains a stream of todos and a `ListComponent` can listen for todos and render when a new task is added.
    - You may have another component that listens for the user that has been assigned to a task provided by a service.

- The steps for creating different parts of an app can be summarized in three steps:

    - Defining a Model using a class
    - Defining the service
    - Defining the component

5.3   Dependency Injection

Dependency Injection is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself. In order to achieve Dependency Injection we need a Dependency InjectionFramework to handle the dependencies for us. Using a DI framework, you simply ask for a class from the injector instead of worrying about the dependencies inside the class itself.

Angular has a standalone module that handles Dependency Injection. This framework can also be used in non-Angular applications to handle Dependency Injection.

6   Meta Data

- Angular uses Meta Data or annotations in a lot of places.

- The most notable Meta Data is the `@component` meta data.

- Below is a list of Angular's core meta data classes:

    - Attribute

- Component
- ContentChild
- ContentChildren
- Dependency
- Directive
- HostBinding
- HostListener
- Host
- Inject
- Injectable
- Input
- Optional
- Output
- Pipe
- Query
- Self
- SkipSelf
- ViewChild
- ViewChildren
- View
- ViewQuery

6.1 Observables

- Angular embraces observables using the RxJS library.

- Observables emit events and observers observe observables.

- An observer *subscribes* to events emitted from an observable.

- RxJS has an object called *subject* that can be used both as an observer or an observable. *Subject* can be imported from `RxJS` very easily:

```
1  import {Subject} from 'rxjs/Subject';
```

- A subscription can be canceled by calling the `unsubscribe` method.

6.2   Services and Providers

- A service is nothing more than a class in Angular 2. It remains nothing more than a class until we register it with the Angular injector.

- When you bootstrap your app, Angular creates an injector on the fly that can inject services and other dependencies throughout the app.

- You can register the service or the dependencies during when bootstrapping the app or when defining a component.

- If you have a class called `MyService`, you can register it with the Injector and then you can inject it everywhere:

```
1  bootstrap(App, [MyService]); // second param is an array of providers
```

- Providers is a way to specify what services are available inside the component in a hierarchical fashion.

- A provider can be a class, a value or a factory.

- Providers create the instances of the things that we ask the injector to inject.

- `[SomeService];` is short for `[provide(SomeService, {useClass:SomeService})];` where the first param is the token, and the second is the definition object.

- A simple object can be passed to the Injector to create a Value Provider:

```
1  beforeEachProviders(() => {
2    let someService = { getData: () => [] };
3    // using `useValue` instead of `useClass`
4    return [ provide(SomeSvc, {useValue: someService}) ];
5  });
```

- You can also use a factory as a provider.

- You can use a factory function that creates a properly configured Service:

```
1  let myServiceFactory = (dx: DepX, dy: DepY) => {
2    return new MyService(dx, dy.value);
3  }
```

```
4
5   // provider definition object.
6   let myServiceDefinition = {
7      useFactory: myServiceFactory,
8      deps: [DepX, DepY]
9   };
10
11  // create provider and bootstrap
12  let myServiceProvider = provide(MyService, myServiceDefinition);
13  bootstrap(AppComponent, [myServiceProvider, DepX, DepY]);
```

- Defining object dependencies is simple. You can make a plain JavaScript object available for injection using a string-based token and the `@Inject` decorator:

```
1   var myObj = {};
2
3   bootstrap(AppComponent, [
4     provide('coolObjToken', {useValue: myObj})
5   ]);
6
7   // and you can inject it to a component
8
9   import {Inject} from 'angular2/core'
10  constructor(dx: DepX, @Inject('coolObjToken') config)
```