Daniel Schmitz

2016

Practical Angular2

+ Node/npm
+ SystemJS
+ MongoDB
+ TypeScript
+ Visual Studio Code
+ Restfull APIs with Express
+ Json Web Token Authentication

Practical Angular 2 Book

With Node/npm, Typescript, SystemJS and Visual Studio Code

Daniel Schmitz and Daniel Pedrinha Georgii

This book is for sale at http://leanpub.com/practical-angular-2

This version was published on 2016-01-31

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

1. Introduction

This work will present you the new, totally rewritten, version of the Angular framework. Almost all concepts from version 1.0 are now obsolete and new techniques are being used in Angular 2.0 to provide a more modern and dynamic framework.

In this context, there are two possible situation for the readers. The first is for those that already know and use *Angular* since its 1.0 version, and that are looking for a way to continue developing with this framework. The second is for those who does not know Angular and want to learn it.

For those that never worked with the *Angular* framework, learning the version 2.0 will be a simple task, since the old concepts will not be in the way of the new ones. For those that already know *Angular 1.0*, it will require some patience to understand the large changes to the framework. It is best if the developer think about Angular 2 as a new framework.

1.1 Prerequisites

Before the we approach the Angular 2 it is necessary to review some vital technologies to the development of any library using HTML/Javascript. We will extensively use the **Node**, which is used to run Javascript on the server. The use of Node will be vital to work with the libraries in our project, that will be installed with Node's package manager **npm**.

1.1.1 Node

If you use Windows, install Node via the link http://www.nodejs.org. Download and install it and leave the `npm` option selected. . Besides Node, it is useful to install GIT too, available here: https://git-scm.com/.

In Linux, we can use the package manager `apt-get` to install everything that we need with the following line:

```
$ sudo apt-get install git npm
```

After installing all the necessary packages, it is necessary to creat a link to the `node` word as the following line:

```
$ sudo ln -s /usr/bin/nodejs /usr/bin/node
```

1.1.2 Web server

As the web server we will use the `live-server` package. This server, written in Node, updates the page automatically when there are changes to the source code, which is helpful to the learning. To install it, open a shell and type the following command:

```
$ npm install live-server -g
```

In this command, the `-g` parameter indicates that the package will be installed as a global variable. This way you can use it anywhere on the file system.

> To open a shell in Windows, go to *Start*, *Execute* and type **cmd**. On Linux, to install the package globally , use the **sudo** command.

1.1.3 The package.json file

In every project created we will define a set of installation rules and execution that will be stored on the *package.json* file. This is a default configuration file on Node projects.

As a test, create the “test” directory and execute the following code on the shell:

```
$ npm init
```

Many questions will be made about the project. We can, for now, leave the default value and press `enter` until it finishes. In the end a new file will be created on the directory with the following text:

```
{
  "name": "test",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

As a test, we will add the `angular2` library. If we were not using **npm** it would be necessary to download the zip file from the site angular.io, but since we are using `npm` the installation will be done with the following command:

```
$ npm i angular2 --S
```

The parameter `i` stands for `install` and `--S` for `--save` that will add the installed library to the `package.json` file. The log for this command is similar to the following image:

Review the `package.json` file and see that a new entry called `dependencies` was added, as following:

```
{
  "name": "test",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "angular2": "^2.0.0-beta.0",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.13",
    "reflect-metadata": "^0.1.2",
    "rxjs": "^5.0.0-beta.0",
    "zone.js": "^0.5.10"
  }
}
```

Besides the file changes, a new directory called `node_modules` was created. It contains the project libraries and its dependencies.

The `package.json` file has many other functionalities. For instance, in the `scripts` entry there is an item `test`, that at first shows a simple error message. We could add a new entry called `start` and start the web server from it, as the following example:

```
{
  "name": "test",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "live-server --port=12345"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "angular2": "^2.0.0-beta.0",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.13",
    "reflect-metadata": "^0.1.2",
    "rxjs": "^5.0.0-beta.0",
    "zone.js": "^0.5.10"
  }
}
```

See that in `scripts` we added the `start` entry with the command `live-server --port=12345`. On the shell, just execute the command `npm start` to automaticaly execute the *live-server* with the set parameters, as the following image:

To learn more about the script types available on the *package.json* file, access this link.

1.1.4 Text editors and IDEs

There are dozens of text editors and IDEs on the market. Among them we suggest the Sublime Text, Atom and Visual Studio Code. The Visual Studio Code is the best options because it has some features that we will use later, like good compatibility with the TypeScript language and easy access to Git and Debug.

1.2 Besides Javascript

Since the announcement of Angular 2 there were stablished great changes in relation to Angular 2, including its own name. The library before known as *AngularJS* was renamed just to *Angular*. Besides that, the Angular that was kept by Google got supported by Microsoft, mostly for the compatibility with TypeScript.

The Angular 2 now supports TypeScript and Dart, that can be chosen according to ones preference. And obviously the JavaScript language can also be used. Here are three examples in each language:

TypeScript has the following structure:

```
import {Component} from 'angular2/core';

@Component({
     selector: 'my-app',
     template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

This is a simple example of a component in Angular 2. Even without knowing much about it, notice the `import` command to import Angular and the `@Component` to add settings to the `AppComponent` class, created with the `export class` command.

The same component in Dart would be as follows:

```
import 'package:angular2/angular2.dart';
import 'package:angular2/bootstrap.dart';
@Component(selector: 'my-app', template: '<h1>My First Angular 2 App</h1>')
class AppComponent {}
main() {
  bootstrap(AppComponent);
}
```

Notice some differences in how to import the Angular 2, in the component notation and in the `AppComponent` class.

```
(function(app) {
  app.AppComponent = ng.core
    .Component({
      selector: 'my-app',
      template: '<h1>My First Angular 2 App</h1>'
    })
    .Class({
      constructor: function() {}
    });
})(window.app || (window.app = {}));
```

In JavaScript we do not have the class creation, import or the use of anonym functions to create components.

1.3 TypeScript

In this work we will use the TypeScript due to its many functionalities including being typed. Basically, TypeScript is a language that supports the ES6 (ECMAScript 2015, new JavaScript specification), that supports *annotations* and support to types (defining of variable types, parameters and return methods).

We will be talking about TypeScript on the chapter 2 to detail all its basic features. It is necessary to know the basic about TypeScript before using Angular 2.

1.4 Source code

All the source code from this book is on GitHub on the following address:

https://github.com/danielschmitz/angular2-codes

2. TypeScript

Before starting with Angular 2 it's necessary to talk about the TypeScript language, that will be extensively used in this book. TypeScript can be considered a programming language that contains the following features:

- Compatible with the new ECMAScript 2015 (ES6) specification
- Typed language
- Implements annotations
- Implements generics

We can say that TypeScript turns JavaScript similar to languages derived from C as PHP, C#, Java, etc.

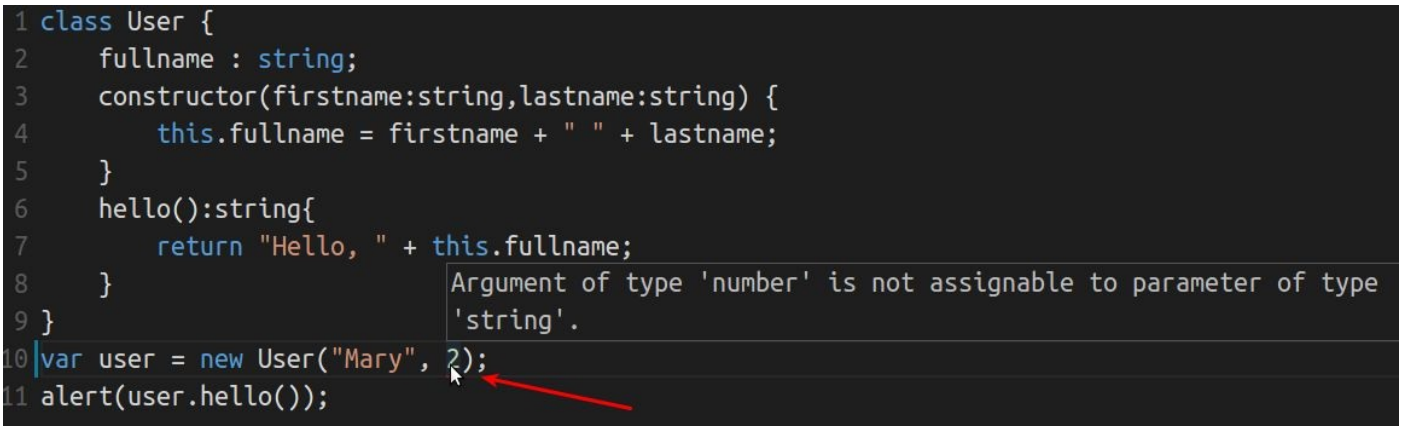
TypeScript is a new language and it will not be executed on the browser. After writing the TypeScript code with some functionality, it is needed to convert this code to JavaScript. In the following example we create a class called `User`, a constructor and a `hello()` methods. After creating this class we instantiate the `user` variable passing the two parameters from constructor and call the `hello()`method.

```
class User {
     fullname : string;
     constructor(firstname:string,lastname:string) {
          this.fullname = firstname + " " + lastname;
     }
          hello():string{
                    return "Hello, " + this.fullname;
          }
}
var user = new User("Mary", "Jane");
alert(user.hello());
```

This code, when translated to JavaScript, becomes something like the next code:

```
var User = (function () {
     function User(firstname, lastname) {
          this.fullname = firstname + " " + lastname;
     }
     User.prototype.hello = function () {
          return "Hello, " + this.fullname;
     };
     return User;
})();
var user = new User("Mary", "Jane");
alert(user.hello());
```

Coding in TypeScript helps creating codes with better structure and defining types helps avoiding bugs. Also text editors can help showing sintax errors as on the following example on the *Visual Studio Code* editor:



2.1 Installing TypeScript

The TypeScript code must be "translated" (we call this process *transpiler*) to JavaScript. It can be done in many ways, but the easiest is via the command line. First we use the **npm** to install the `typescript` library with the following command:

```
$ npm i typescript -g
```

After the TypeScript is installed globally we can use the **tsc** (TypeScript Compiler) to compile any file with the `ts` extension to a `js` file. For example, create the `user.ts` file and add the following code:

```
class User {
     fullname : string;
     constructor(firstname:string,lastname:string) {
          this.fullname = firstname + " " + lastname;
     }
          hello():string{
                    return "Hello, " + this.fullname;
          }
}
var user = new User("Mary", "Jane");
alert(user.hello());
```

And then execute the following command:

```
$ tsc user.ts
```

The `user.js` file will be created with the following JavaScript code:

```
var User = (function () {
     function User(firstname, lastname) {
          this.fullname = firstname + " " + lastname;
     }
     User.prototype.hello = function () {
          return "Hello, " + this.fullname;
     };
     return User;
})();
var user = new User("Mary", "Jane");
alert(user.hello());
```

2.2 Using Visual Studio Code

Any text editor can be used to work with Angular 2. Here we will use the Visual Studio Code for being one of the free editors with a better compatibility with TypeScript. Let's create a small project called `HelloWorldVSCode` demonstrating how the Visual Studio Code can speed up the development in Angular 2.

VScode can be installed from this link. VSCode is an editor of type "folder based", which means that when it opens a directory it tries to find a project file (package.json, project.json, tsconfig.json) to add the features according to the project.

This means that before opening VSCode we must, at least, create a directory and a project file. This is easily done with *npm* as on the following image:

```
daniel@debian: ~/HelloWorldVsCode

daniel@debian:~$ mkdir HelloWorldVsCode
daniel@debian:~$ cd HelloWorldVsCode/
daniel@debian:~/HelloWorldVsCode$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

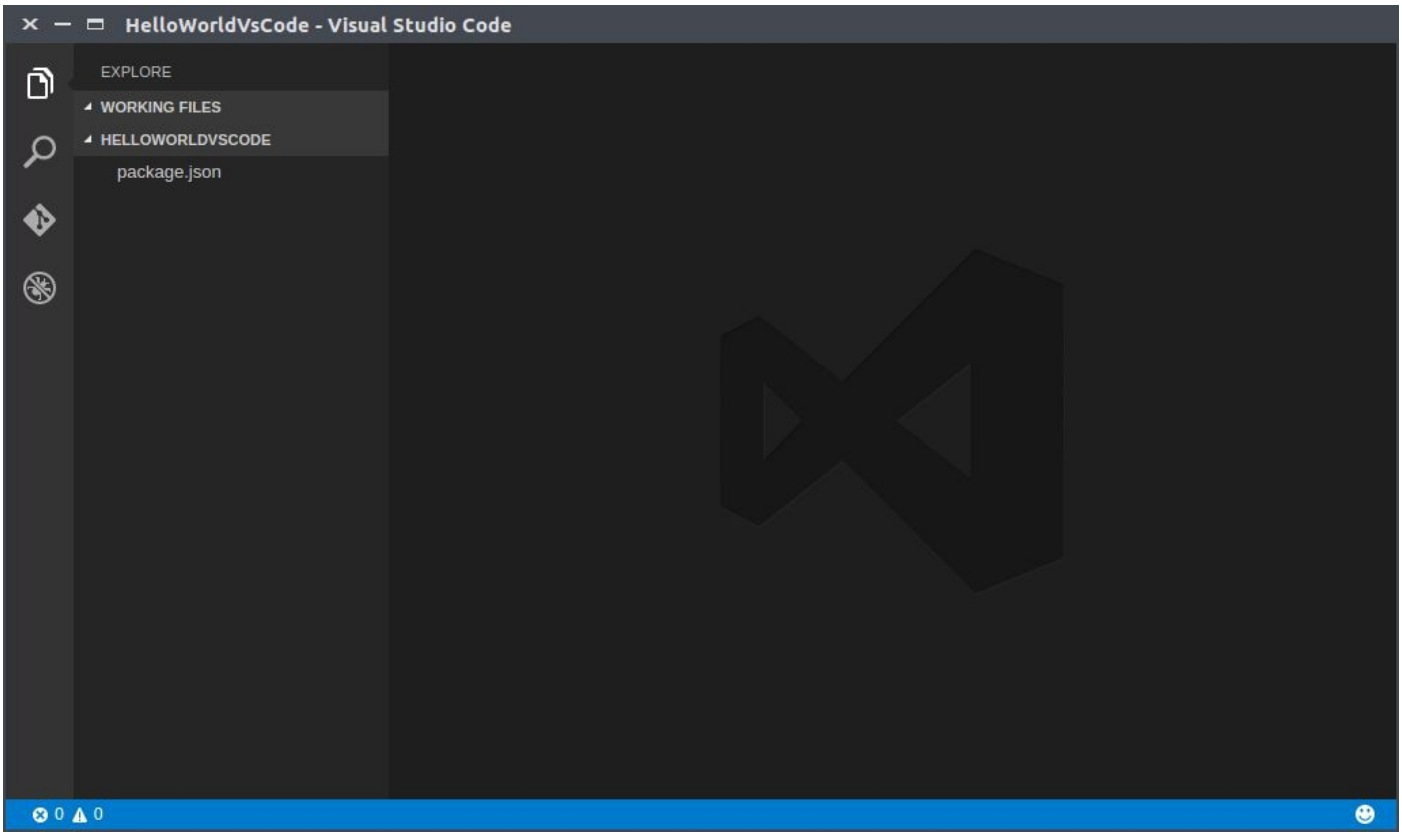
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (HelloWorldVsCode)
version: (0.0.0) 0.0.1
description: A simple hello world project
entry point: (index.js)
test command:
git repository:
keywords:
author: Daniel
license: (ISC)
About to write to /home/daniel/HelloWorldVsCode/package.json:

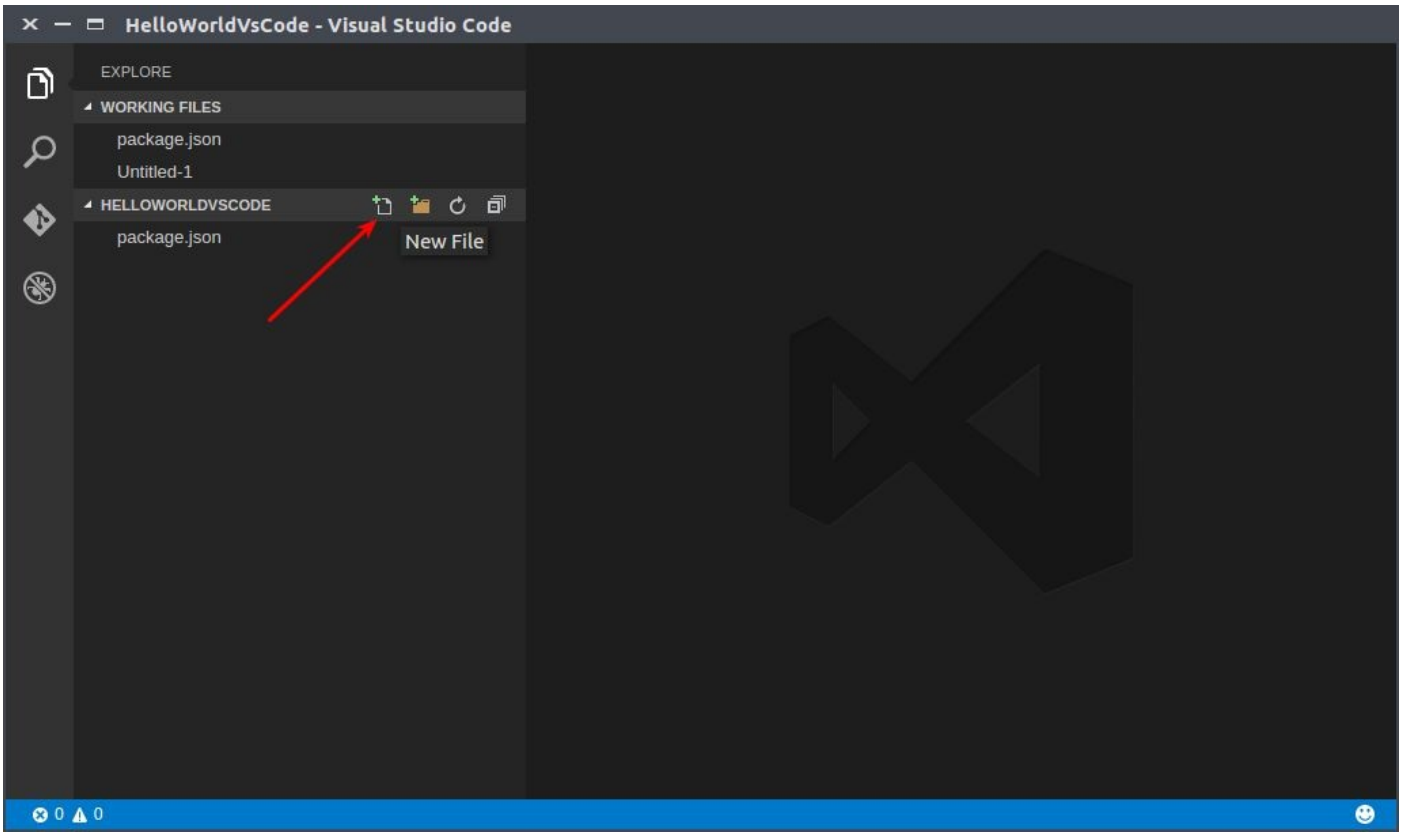
{
  "name": "HelloWorldVsCode",
  "version": "0.0.1",
  "description": "A simple hello world project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Daniel",
  "license": "ISC"
}


Is this ok? (yes)
daniel@debian:~/HelloWorldVsCode$
```

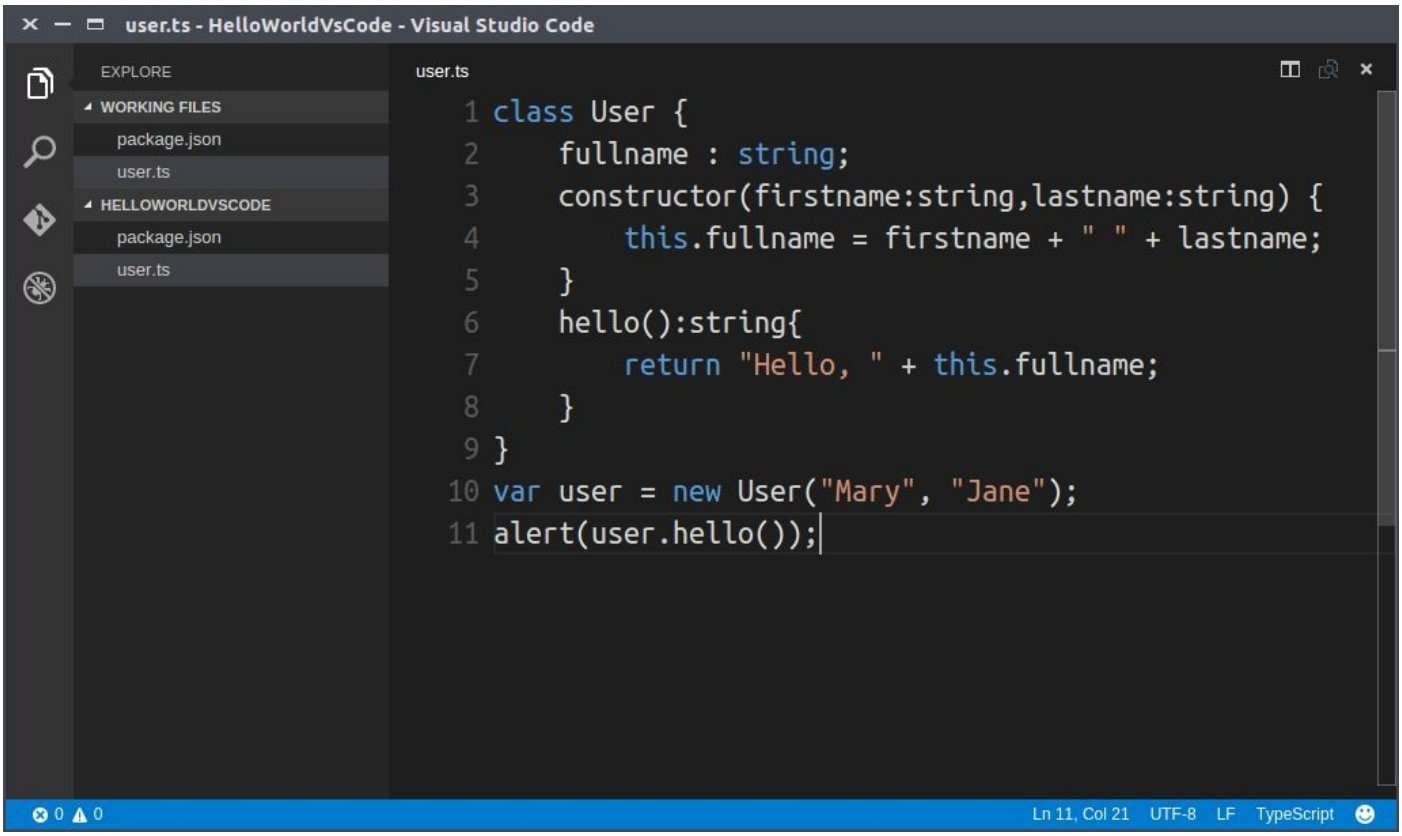
After the directory creation, open Visual Studio Code, go to `File > Open Folder` and choose the created directory. VSCode will look like the following image:



Let's create again the `user.ts` file inside `HelloWorldVSCode` folder. On VSCode it is possible via `File > New File` or clicking on the icon shown on the following image:



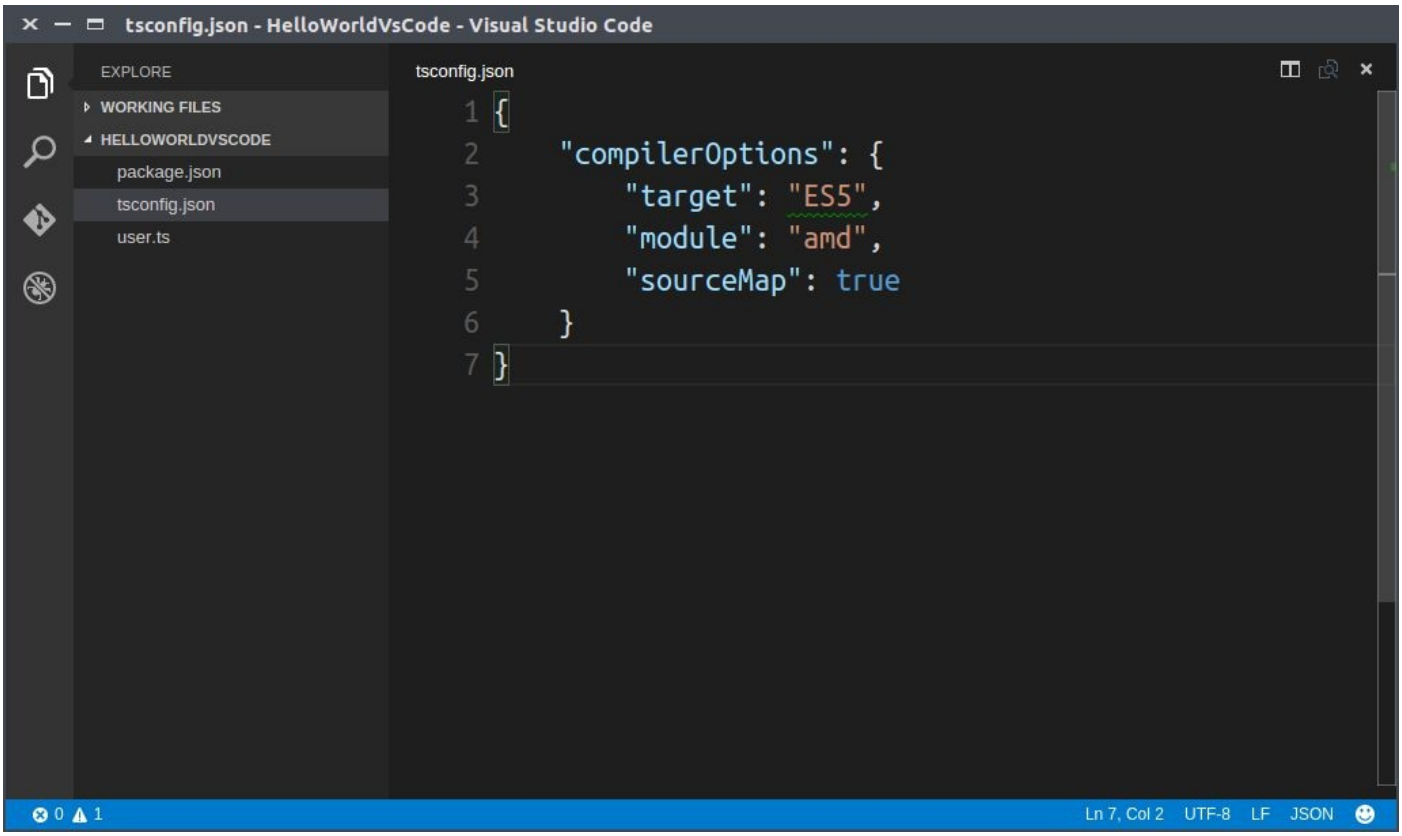
Create the `user.ts` file with the same TypeScript code before as on the following image:



Now we need to convert the TypeScript file to JavaScript. Instead of using the command line, we can set up VScode to do it. First create the `tsconfig.json` file with the following code:

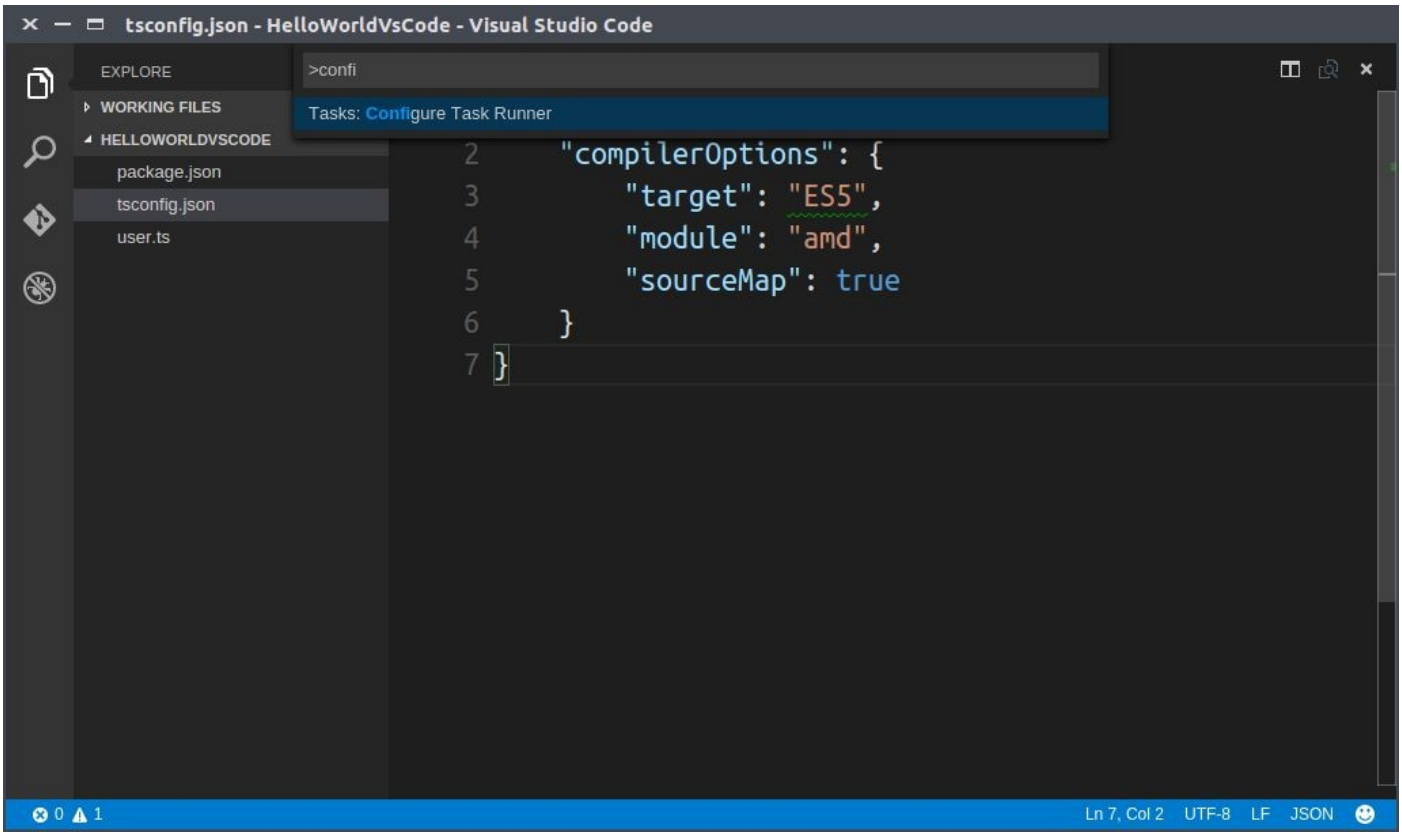
/HelloWorldVSCode/tsconfig.json

```
{
    "compilerOptions": {
        "target": "ES5",
        "module": "amd",
        "sourceMap": true
    }
}
```

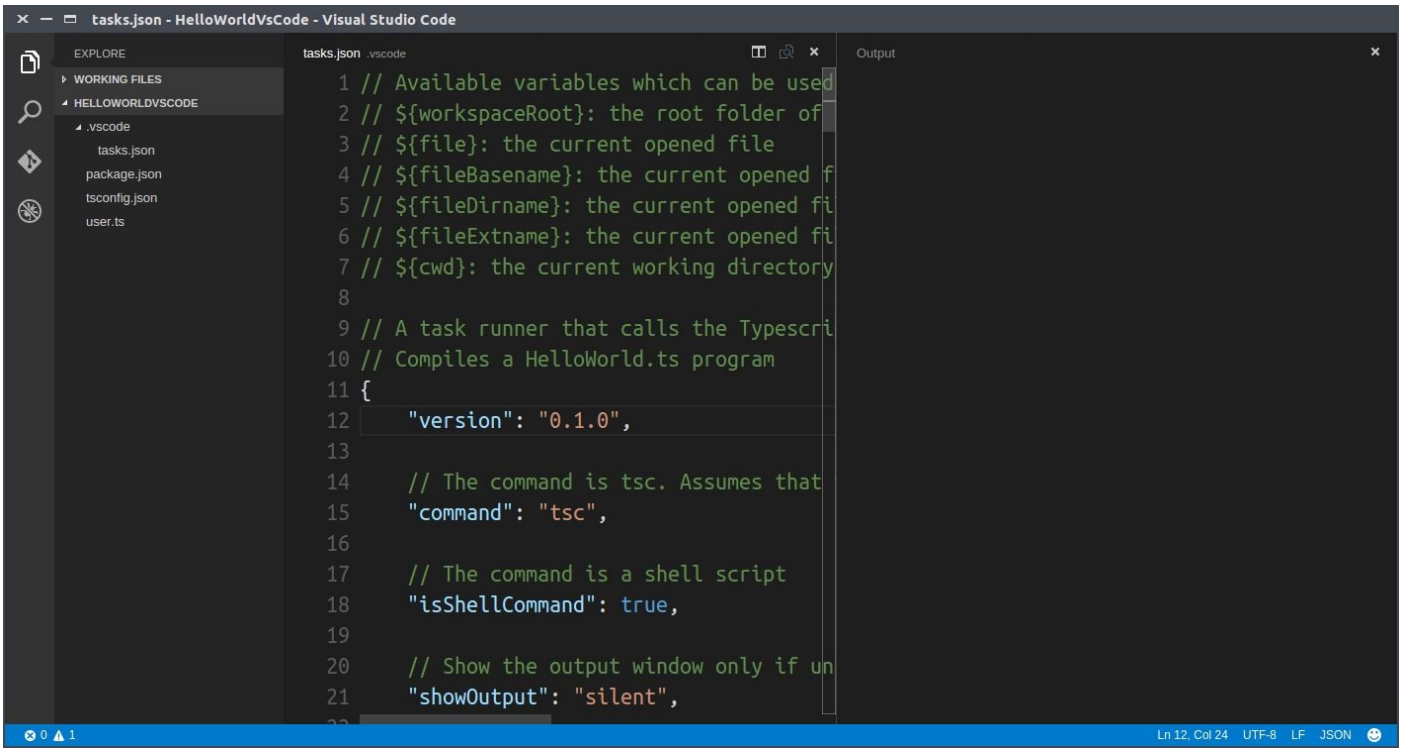


The `tsconfig.json` file will be identified by VSCode and its compile options will be used. This options (target, module, sourceMap) will be useful in a more complex project, but are not needed for this one. For example, the `module` indicates how the modules will be loaded when working with separate files. The `sourceMap` indicates if the "map" file will be created to help the TypeScript debug. And `target` sets up the ECMAScript version being used.

With the `tsconfig` file created, let us create a task to compile the `ts` file. For that, press F1 and type "configure". The `Configure Task Runner` option will appear as the following image:



When we select `Configure Task Runner`, a folder `.vscode` is created with the `tasks.json` file as on the following image:

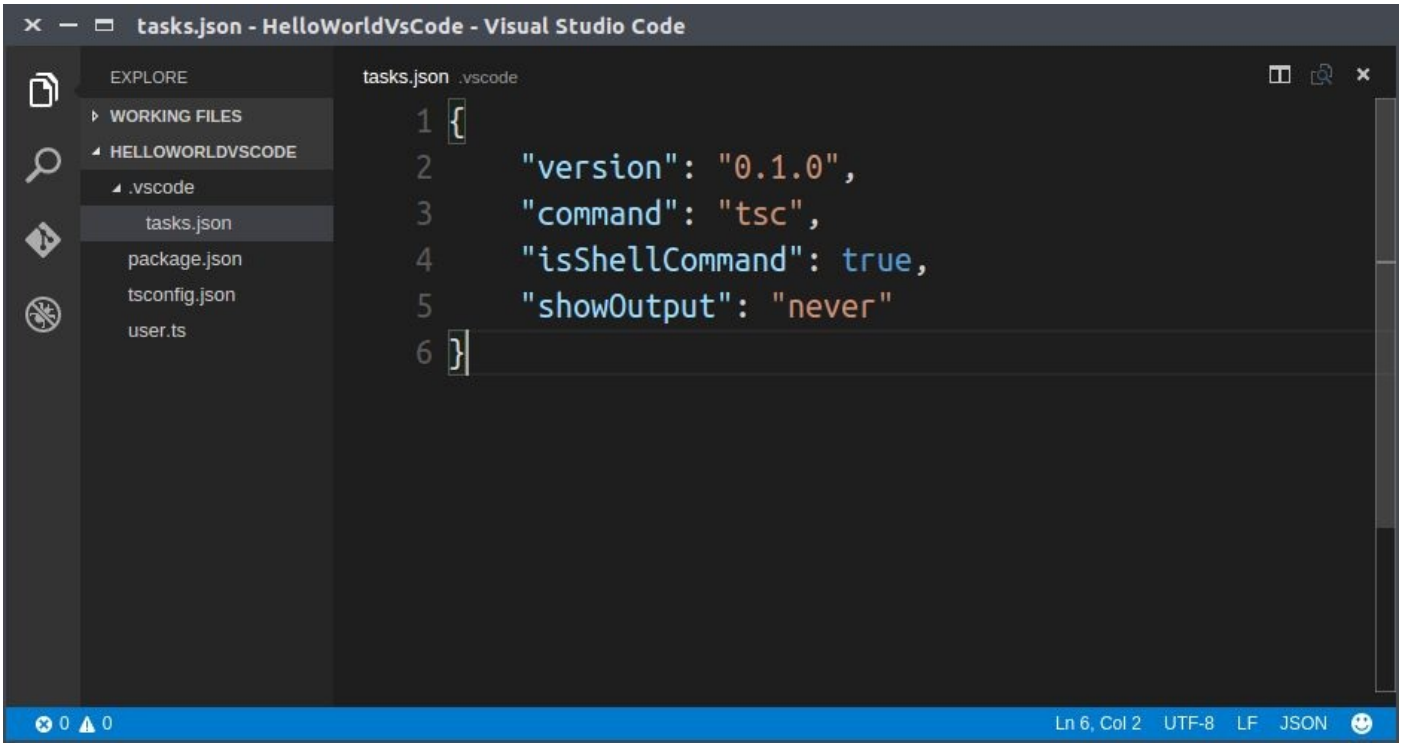


The `tasks.json` file has miscellaneous setups for various kinds of tasks, but let us add just the basic to compile the TypeScript. Delete all the file content and add the following code:

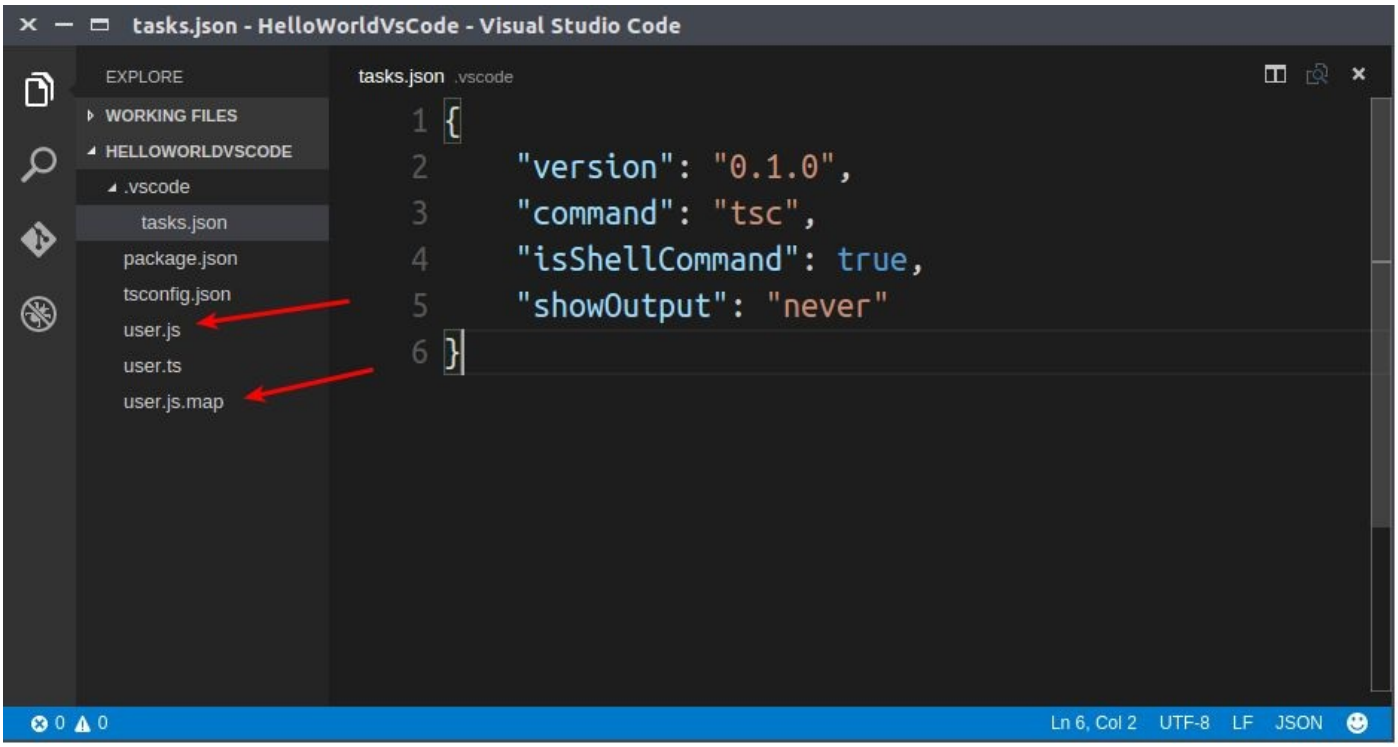
/HelloWorldVSCode/.vscode/tasks.json

```
{
    "version": "0.1.0",
    "command": "tsc",
```

```
    "isShellCommand": true,
    "showOutput": "never"
}
```

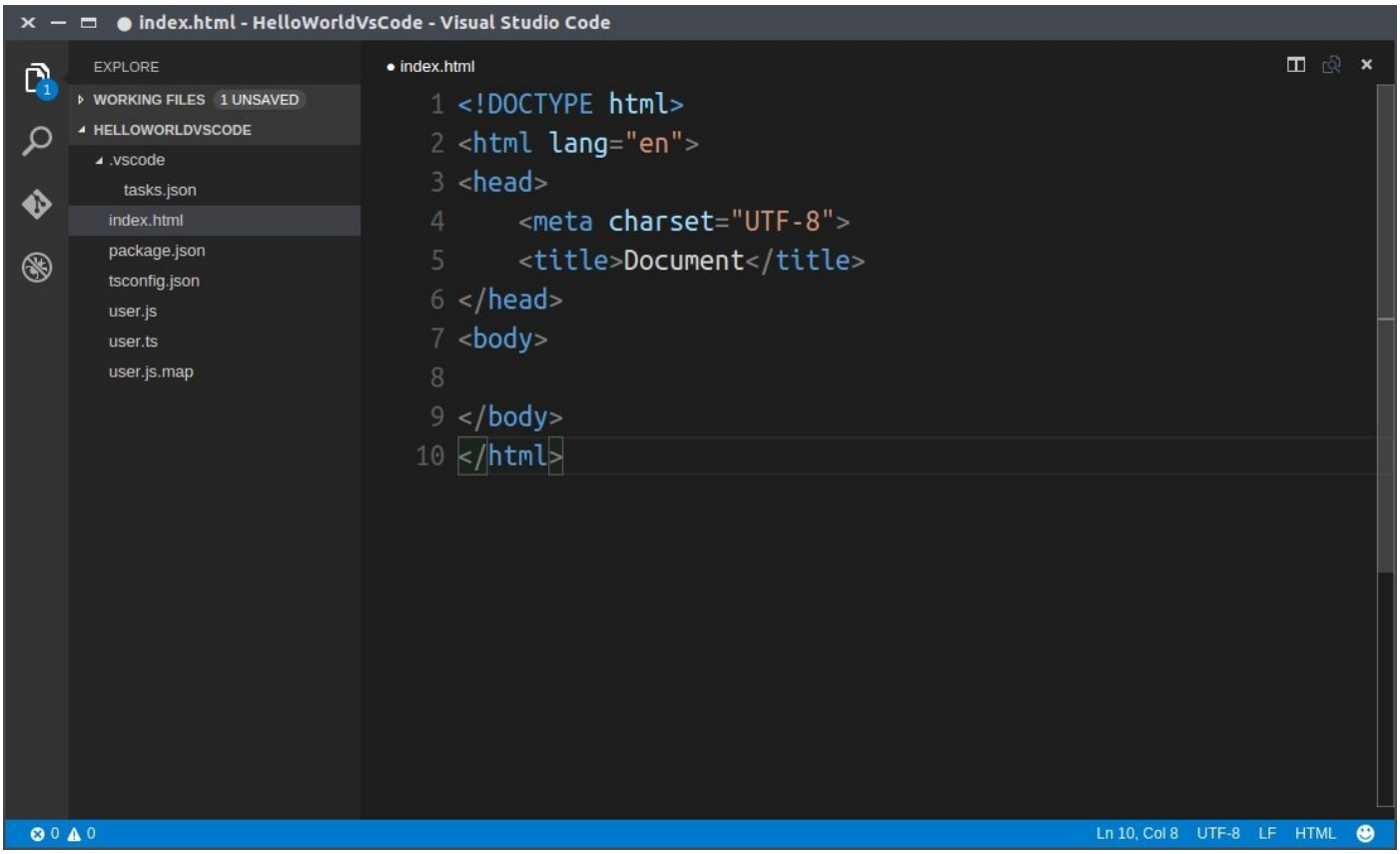


We just created the `tsc`command task. To execute this task in VSCode, press CTRL+SHIFT+B or press F1 and type “run task build”. When `tsc` is executed the `tsconfig.json` file will be read and the `js` and `map`files created as follows:



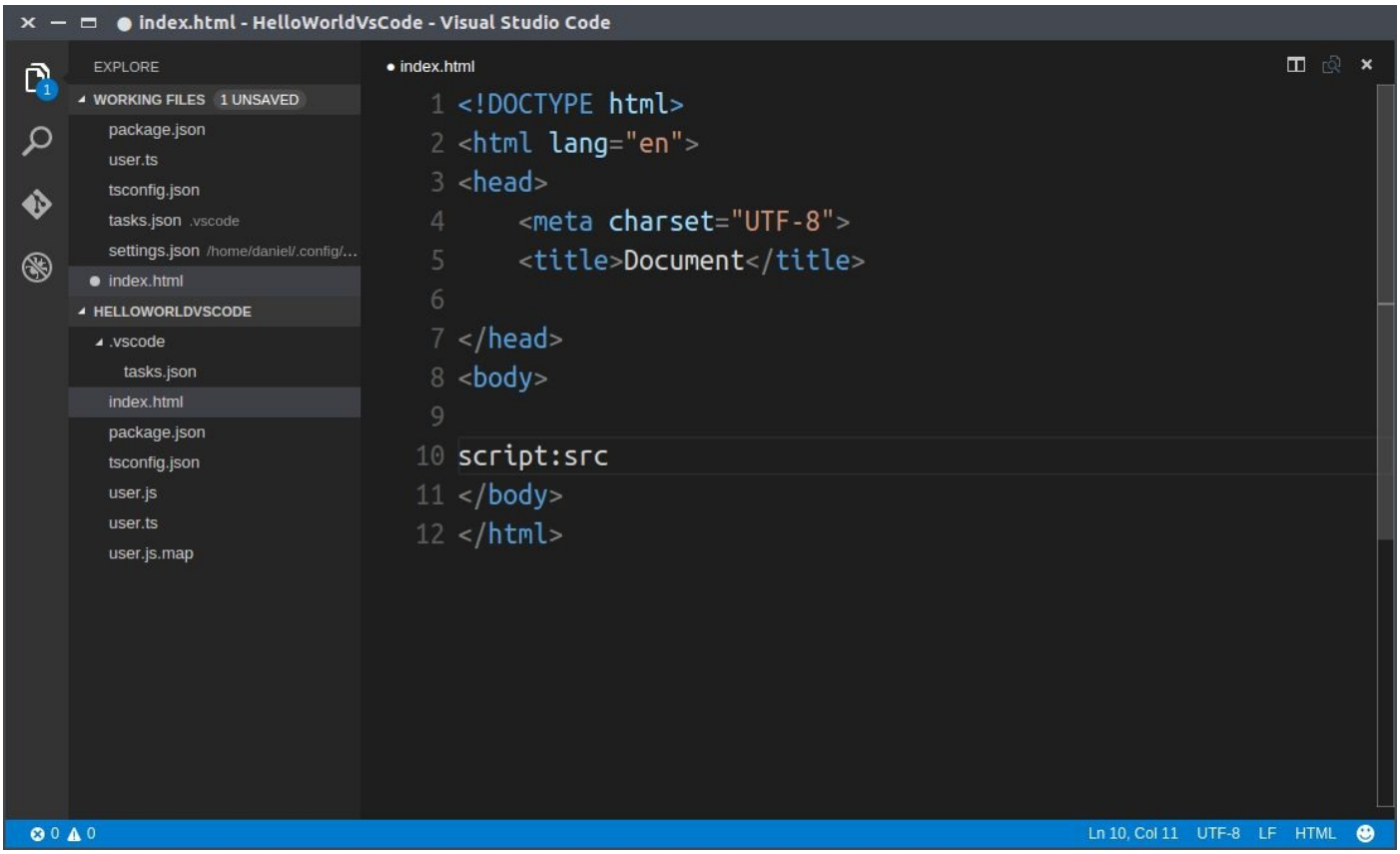
If the files are not created, change the attribute `showOutput` to `always` and check the error. Also check if the `tsc` command is working on the command line. This command was added to the system when we executed `npm i typescript -g`.

To test the `user.js` file on the browser, create the `index.html` file on the `HelloWorldVSCode` directory. With the cursor on the beginning of the `index.html` file type `html:5` and press tab. The code will be converted to a complete html document as follows:

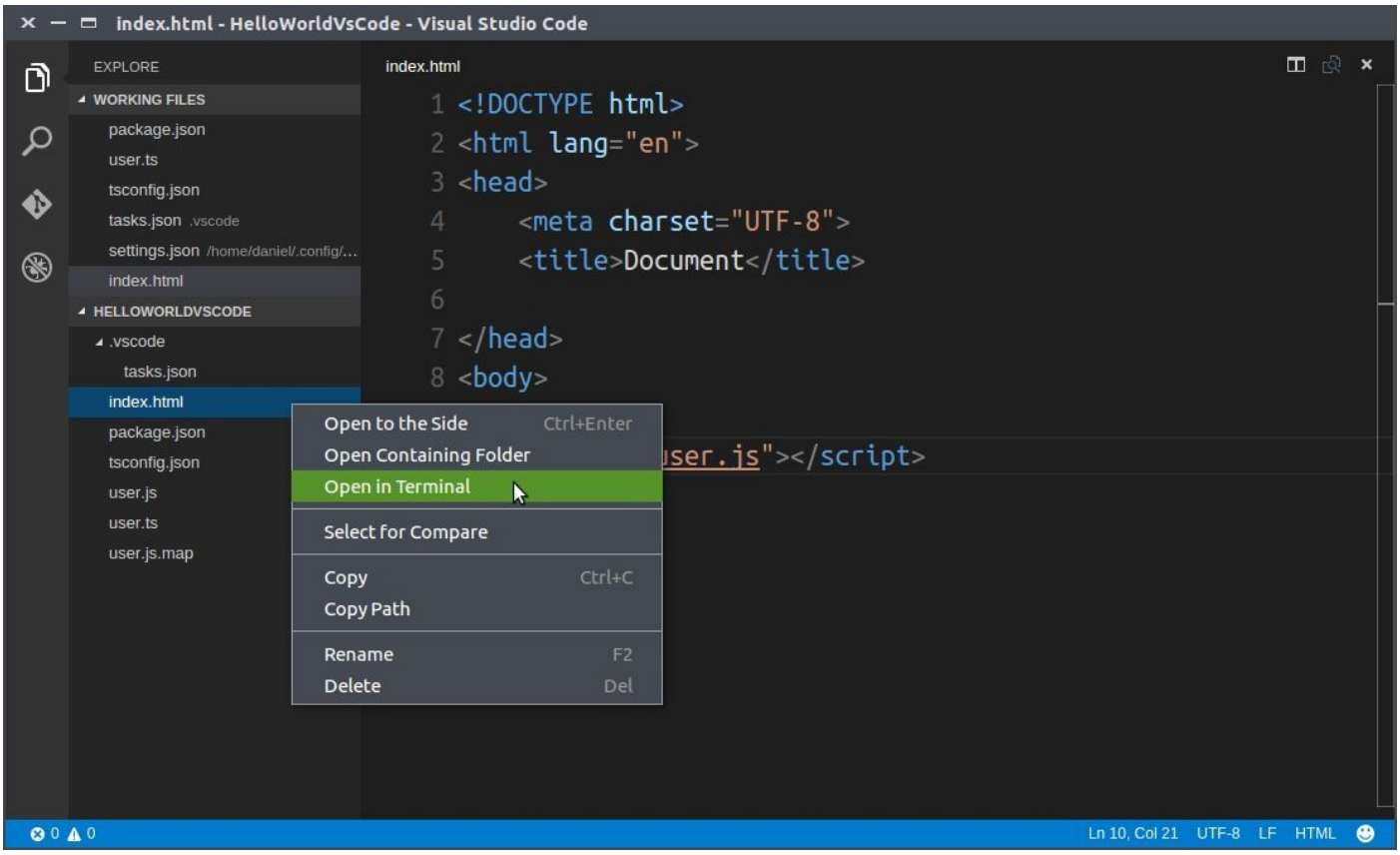


This code generation is made by the integration between Visual Studio Code and *emmet*, that you can learn about here.

Now put the cursor before `</body>` and type `script:src`:



Press tab and notice that the generated code was `<script src=""></script>` where we can add the `user.js` to the `src` attribute. After adding it we can finally test the application on the browser. Select the `index.html` file on the file list, right-click it and chose “Open in Command Prompt”:



With the command prompt open on the `HelloWorldVSCode` directory type `live-server` and wait for the page to load. When the page loads an alert message will be shown. It is not necessary to close the browser or restart the live-server to change the TypeScript code. All that is needed is to compile the application again. For example on the `user.ts` file change the `alert(user.hello());` code to `document.body.innerHTML = user.hello();` and press `ctrl+shift+b` to recompile and change the message on the screen.

With that initial setup the use of TypeScript becomes more dynamic taken that it is not needed to use the command prompt to compile every time the file is modified.

2.2.1 Detecting changes

It is possible to set up VSCode to compile automatically after code changes. This way, after saving the `.ts` file the changes will be detected and the JavaScript file will be generated again.

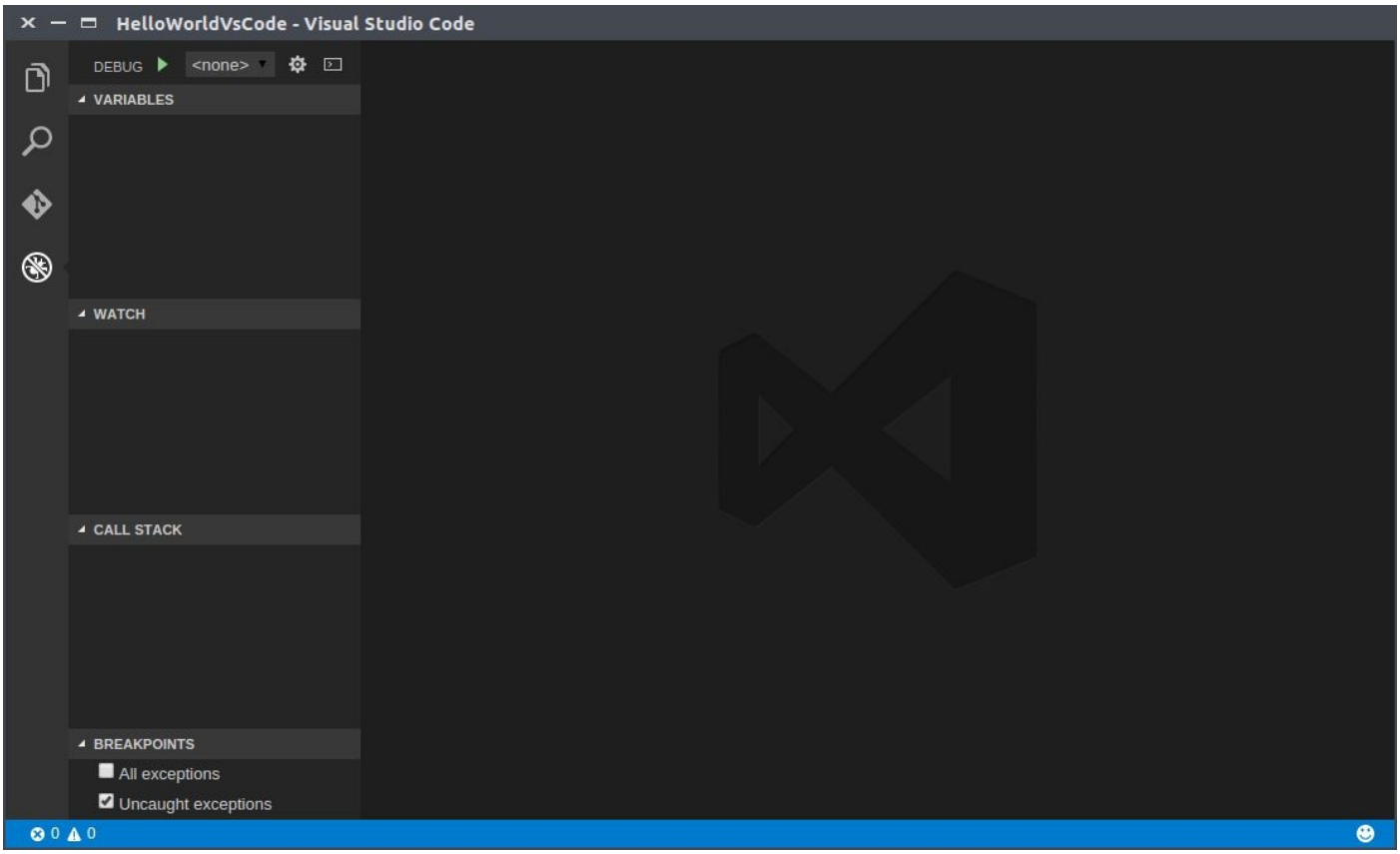
For that, change the tsconfig.json file adding the `watch:true` item and activating the output to verify if the file was compiled as following:

```
{
     "compilerOptions": {
          "target": "es5",
          "module": "amd",
          "sourceMap": true,
          "watch": true
     }
}
```

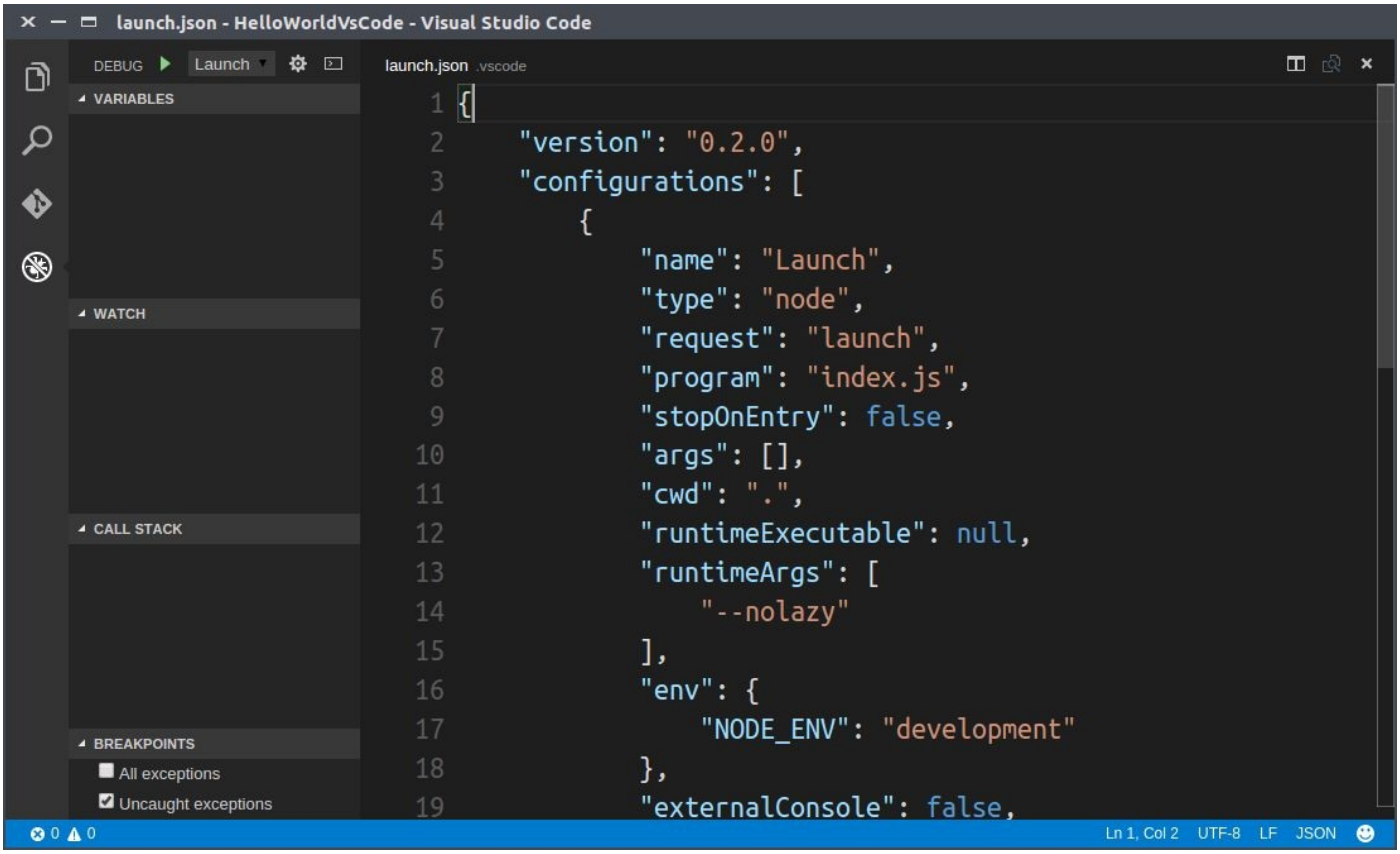
After the first compilation (ctrl+shift+b) the message “Watching for file changes” will appear on the output window. Change the `user.ts` file and after saving it will compile again without the need to press `ctrl+shift+b`. To end the process press F1, type `task` and select `Terminate running task`.

2.2.2 Debugging Visual Studio Code

To debug code in Virtual Studio Code click on the debug icon or press `ctrl+shift+d`. The next screen will show up, at first, without debug settings:

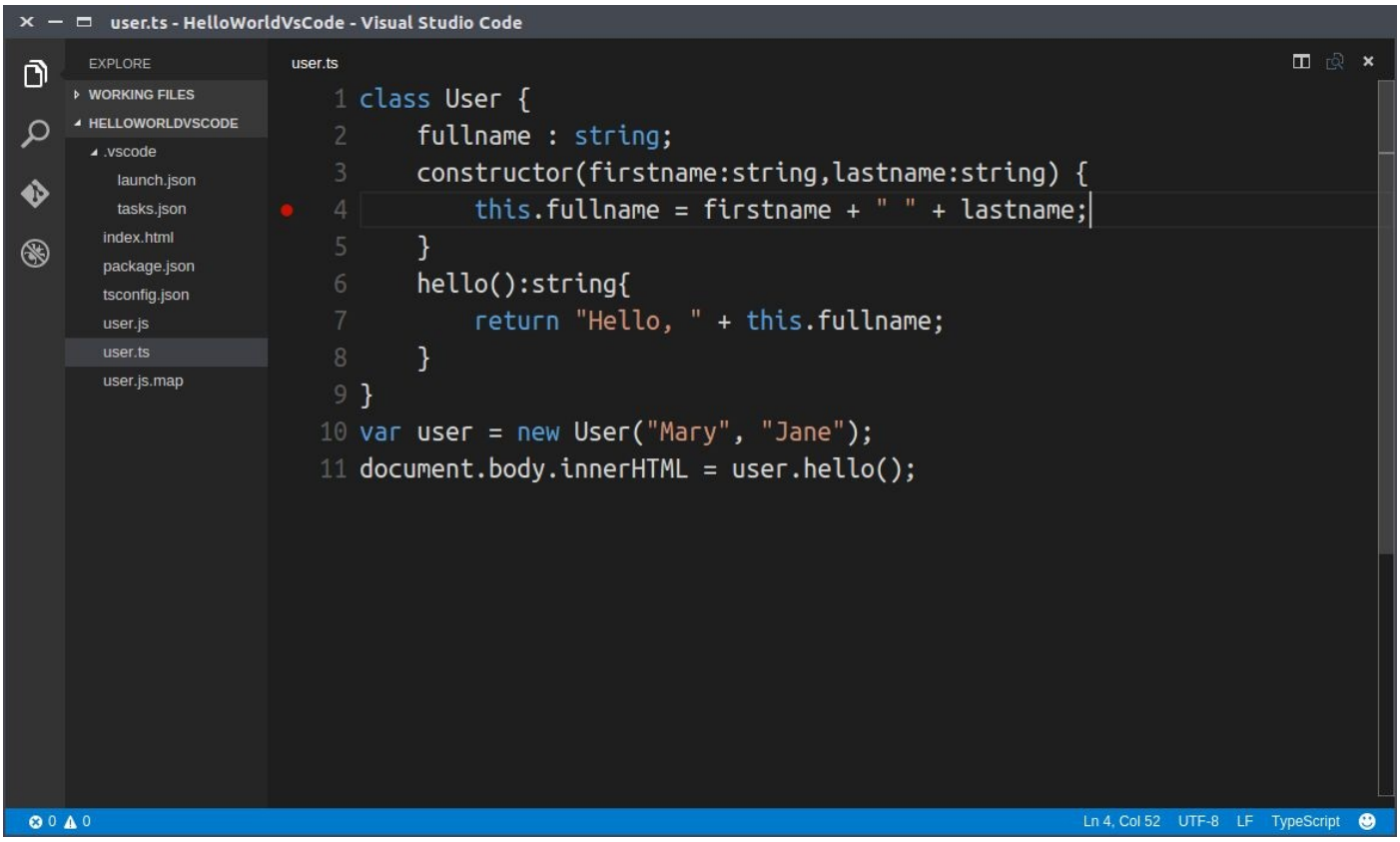


Click on the green arrow to start the first debug set up. The `.vscode/launch.json` file will be created:



First change the `program` parameter to `user.js` and change the `sourceMaps` parameter to `true`. Notice that to debug TypeScript code it must have the appropriate *sourceMap*.

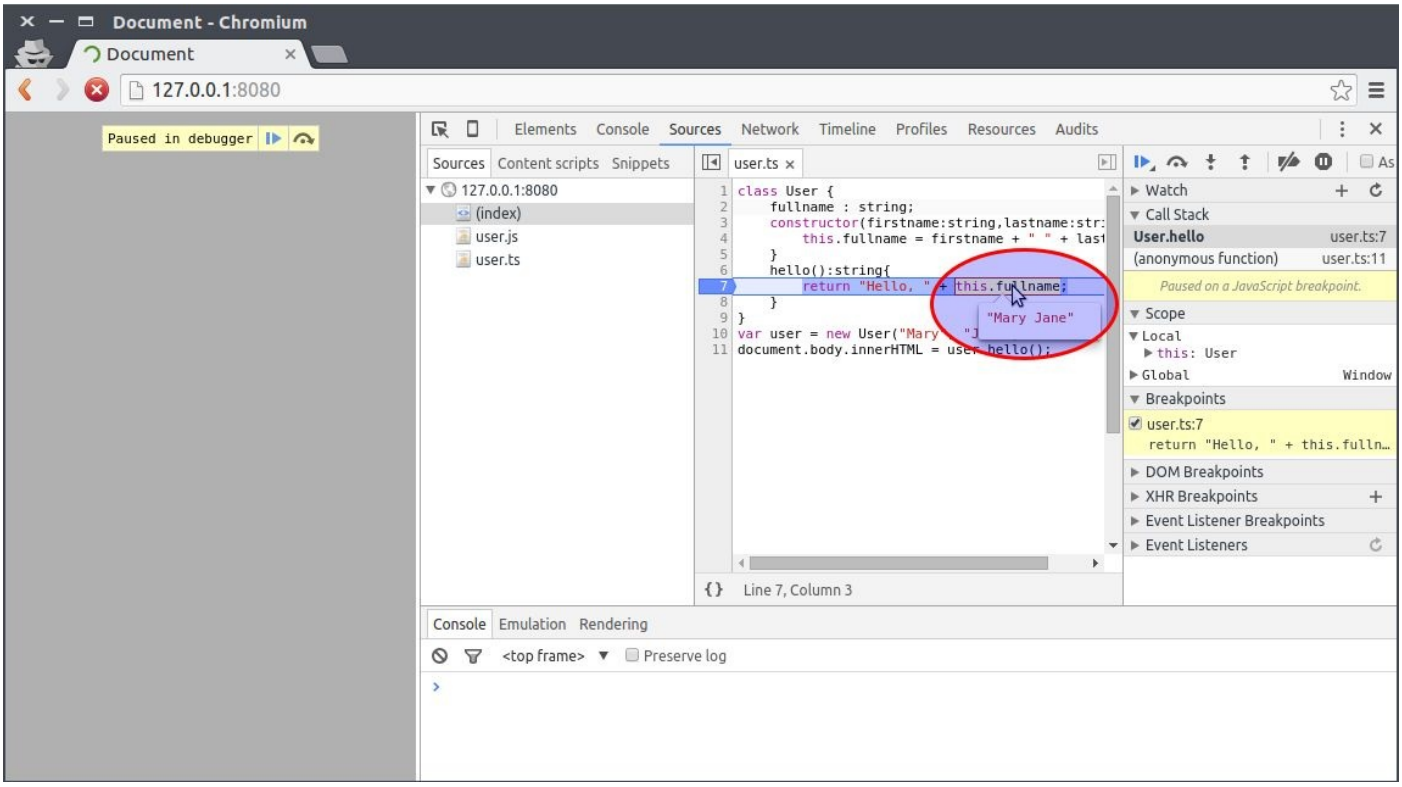
Access the `user.ts` file and add the break point to the constructor method in line 4 as follows:



2.2.3 Debug on the browser

It is always important to have the code debug on the browser active. It is fundamental to generate the ".map" files via "–sourceMap" in `.vscode/tasks.json` or execute the `tsc` command using the `--sourceMap` parameter. In this project the `user.ts` file has its `user.js` and `user.js.map` compiled allowing it to be debuged.

In Google Chrome, press F12 and go to the `sources` tab. Find the `user.ts` file and add a break point to the `hello()` method. Reload the page to notice that the code is stopped on the break point line as the following image:



2.3 Types

Now that we have a workspace capable of compiling the TypeScript files into JavaScript we can start learning the language. It is fundamental to learn it to have success with Angular 2. Fortunately TypeScript is similar to other languages like C, Java or C#. And differently from JavaScript we can create classes and interfaces, define variable and return types.

2.3.1 Basic types

To create a variable in TypeScript, the following syntax is used:

```
var VARIABLE_NAME : TYPE = VALUE
```

The basic types are:

- boolean: Can store `true` or `false` as values.
- number: Can store any number as integer or float.
- string: Text type, can be assigned with single or double quotes.

2.3.2 Arrays

The Arrays in TypeScript can be created in two different ways:

```
var list:number[] = [1, 2, 3];
```

The second way is known as “generics” and uses <> to define the type:

```
var list:Array<number> = [1,2,3];
```

It is possible to create complex arrays, as on the next example where the `Point` class is a TypeScript previously created:

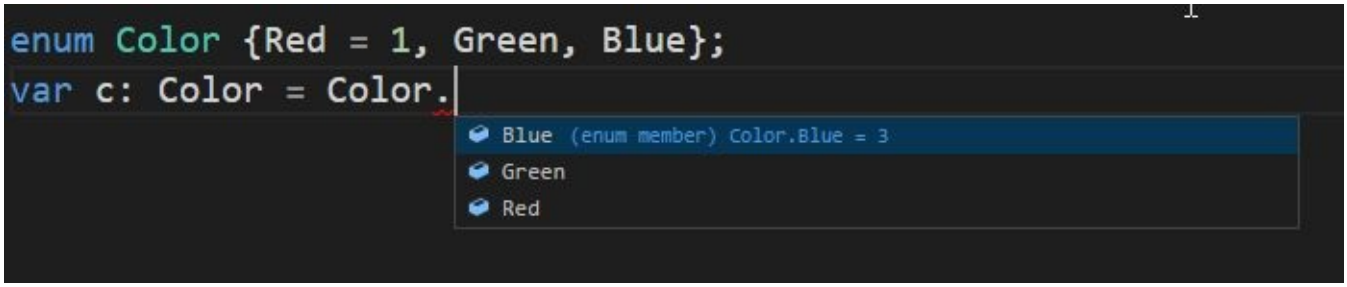
```
var arr:Array<Point> = [
                new Point(10,10),
                new Point(20,30),
                new Point(30,10)
                ]
```

2.3.3 Enum

We can also create Enums, that are enumerations that can define a status or a set of values:

```
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
```

Enums are easily changed with VSCode as on the next image, using the code completion (ctrl+space):



2.3.4 Any

The type any assumes any type (string, number, boolean, etc).

2.3.5 Void

Void is used to determine that a method does not return a value:

```
function warnUser(): void {
     alert("This is my warning message");
}
```

2.4 Classes

The classes concept on TypeScript is the same as any class in a object oriented language. The TypeScript classes follow the ECMAScript 6 pattern. The class has a very similar syntax as C#:

```
class Greeter {
     greeting: string;
     constructor(message: string) {
          this.greeting = message;
     }
     greet() {
          return "Hello, " + this.greeting;
     }
}
var greeter = new Greeter("world");
```

2.4.1 Constructor

The constructor is defined by the `constructor` word. Methods do not need the `function` word.

2.4.2 Method and properties visibility

Notice that on the example we did not define the visibility of the class properties or the return type of the `greet` method. But we can define those parameter as follows:

```
class Greeter {
     private greeting: string;
     constructor(message: string) {
          this.greeting = message;
     }
     public greet() : string {
          return "Hello, " + this.greeting;
     }
}

var greeter:Greeter = new Greeter("world");
```

Methods and properties of a class can have the private, public and protected visibility.

- private: They are visible only inside the own class.
- public: They are visible to all classes.
- protected: They are visible to the class and its child classes.

2.5 Heritage

The heritage between one class and another is defined by the `extends` word. It is possible to use the `super` word to call methods from the parent class as the following example:

```
class Animal {
     name:string;
     constructor(theName: string) { this.name = theName; }
     move(meters: number = 0) {
          alert(this.name + " moved " + meters + "m.");
     }
}

class Snake extends Animal {
     constructor(name: string) { super(name); }
     move(meters = 5) {
          alert("Slithering…");
          super.move(meters);
     }
}

class Horse extends Animal {
     constructor(name: string) { super(name); }
     move(meters = 45) {
          alert("Galloping…");
          super.move(meters);
     }
}
```

```
var sam = new Snake("Sammy the Python");
var tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

In this example we used the `Snake` super to call the constructor method of the `Animal` class. If this is not clear to you, it is advised to learn Object Orientation.

2.6 Accessors (get/set)

The Accessors are used to protect properties of a class. The accessors in TypeScript are used with the `get` and `set` words:

```
class Person {
     private _password: string;

     get password(): string {
          return this._password;
     }

     set password(p : string) {
          if (p != "123456") {
               this._password = p;
          }
          else {
               alert("Hey, the password can not be: 123456");
          }
     }
}

var p = new Person();
p.password = "123456"; //shows the error
```

2.7 Static Methods

It is possible to create static methods with the `static` word before the method. There are dozens of applications for static methods, being one of them to be able to avoid instantiating a class as follows:

```
class SystemAlert{

          static alert(message:string):void{
                    alert(message);
          }

          static warm (message:string):void{
                    alert("Atention: " + message);
          }

          static error(message:string):void{
                    alert("Error: " + message);
          }

}
```

```
SystemAlert.alert("Hi");
SystemAlert.error("It was not possible to connect to the database");
```

2.8 Interfaces

An interface defines a contract for the classes. An interface is created on the following way:

```
interface Point{
 x: number;
 y: number;
 x: number;
}
```

To implement the interface we use `implements`:

```
class point3d implements Point{
    //...
}
```

2.9 Functions

Let us exemplify some particularities of a function in TypeScript. The function can be created outside or inside a class.

It is not necessary to use the `function` word in a class to define a function. But it is necessary when outside a class.

2.9.1 Default value

It is possible to define a default value to a function parameter as following:

```
function buildName(firstName: string, lastName : string = "Smith") {
}
// or
class Foo{
  buildName(firstName: string, lastName : string = "Smith") {
  }
}
```

2.9.2 Optional parameter

Use the `?` char to define an optional parameter

```
class Foo{
  buildName(firstName: string, lastName? : string) {
      if (lastName){
             // may the force be with you
      }
  }
}
```

2.10 Rest parameters

It is possible to pass an array of values straight to a parameter, but only on the last parameter of the function as follows:

```
class Foo{
 static alertName(firstName: string, ...restOfName: string[]) {
          alert(firstName + " " + restOfName.join(" "));
 }
}
Foo.alertName("Mr","John","Doe");
```

2.11 Parameters in JSON format

One of the biggest advantages of JavaScript is to use parameters in JSON format. With TypeScript it is possible to use the same feature:

```
class Point{

          private _x : number = 0;
          private _y : number = 0;
          private _z : number = 0;

          constructor( p: {x:number;y:number;z?:number;}){
                    this._x = p.x;
                    this._y = p.y;
                    if (p.z)
                              this._z = p.z;
          }

          is3d():boolean{
                    return this._z!=0;
          }

}

var p1 = new Point({x:10,y:20});

alert(p1.is3d()); //false
```

2.12 Modules

Learning Angular 2 it is possible to write all the code in one file, creating the classes on the same code block, but for serious development each application class should be a different TypeScript file.

The way to load modules, classes and files on JavaScript depends on many factors, including the choice of using a library for it. Here we will use the **systemjs** since it is the most used way in AngularJS.

2.12.1 Example with Systemjs

Let us creat a new directory called `testModules` and use the VSCode to open it. Creating at first two TypeScript files called `index.ts` (the main application file) and `module.ts` (generic module).

module.ts

```
export class foo{
    getHelloWorldFromModule():string{
        return "Hello World modules world";
    }
}
```

The `foo` class has a modifier called `export` before the `class`. This modifier tells `module.ts` that the class can be exported to other application classes. With the module ready, let's creat the `index.ts` file with the following code:

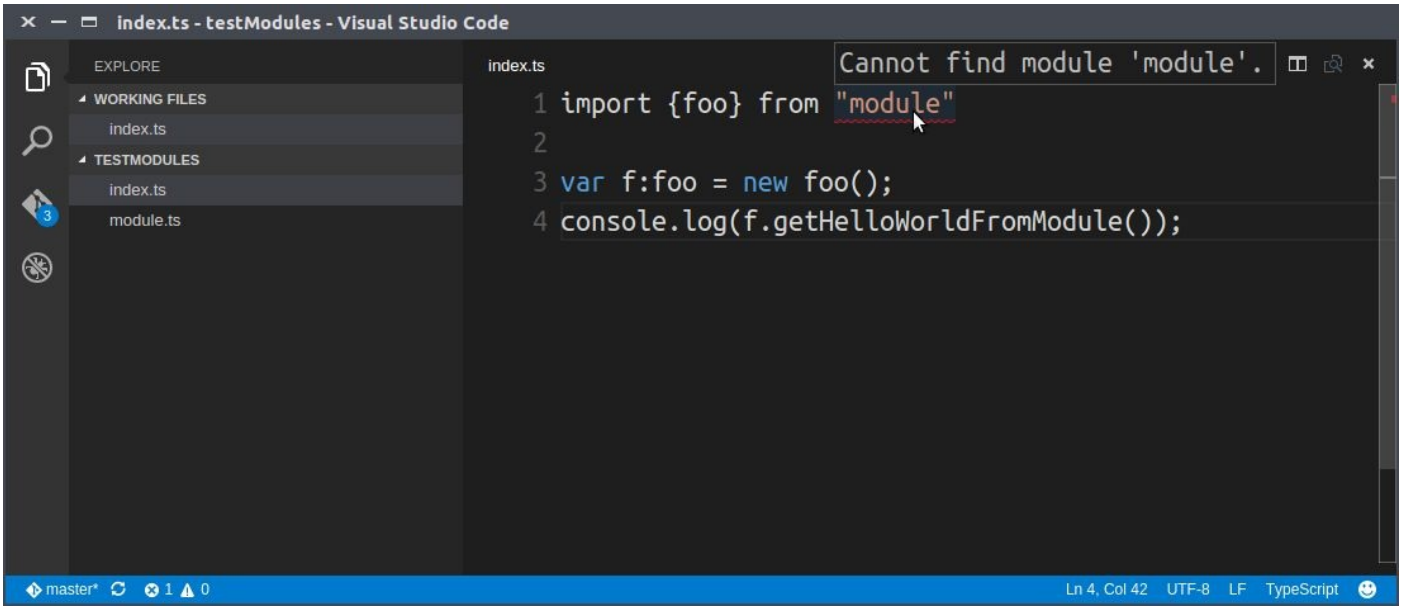
index.js

```
import {foo} from "module"

var f:foo = new foo();
console.log(f.getHelloWorldFromModule());
```

On the first line we imported the `foo` class from `module` (file name).

Until now the VSCode is not ready to understand the TypeScript code presented and your screen should be similar to the following:



Notice the error in `module`, because the VSCode editor was unable to find the module. In fact VSCode does not know yet how to load modules. As seen before, it is necessary to create the `tsconfig.json` file according to the following code:

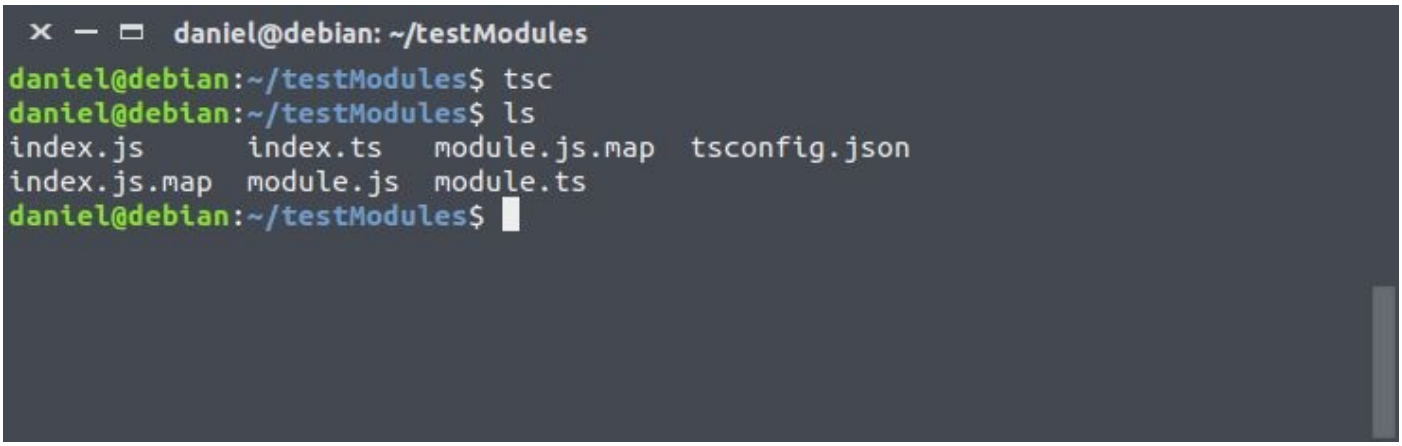
tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "system",
        "sourceMap": true
    },
    "files": [
        "index.ts"
```

```
    ]
}
```

The TypeScript setup file indicates, via the `module` property, which will be the file loading strategy. In this case `system` is used. SourceMap is also set to generate the `.map` files for debug. We also included the `files` property that sets up which files should be compiled to JavaScript. If we remove the `files` property all the files from `testModules` will be compiled. As we provided only the `index.ts` file, it will be compiled first and the others will be compiled if, and only if, imported.

Remember the importance of `tsconfig.json` file, it helps the `tsc` command that was installed via *npm*. It means that if we execute `tsc` on the command line, inside the `testModules` directory, everything will be compiled as the following image:



But it is not our intention to open a terminal and execute the `tsc` command every time we want to compile the project. For that, we will set up a task in VSCode. On VSCode press F1 and type `Configure Task Runner`. The file `.vscode/tasks.json` will be created with many options. Change the file to the following json setup:

.vscode/tasks.json

```
{
        "version": "0.1.0",
        "command": "tsc",
        "isShellCommand": true,
        "showOutput": "silent",
        "problemMatcher": "$tsc"
}
```

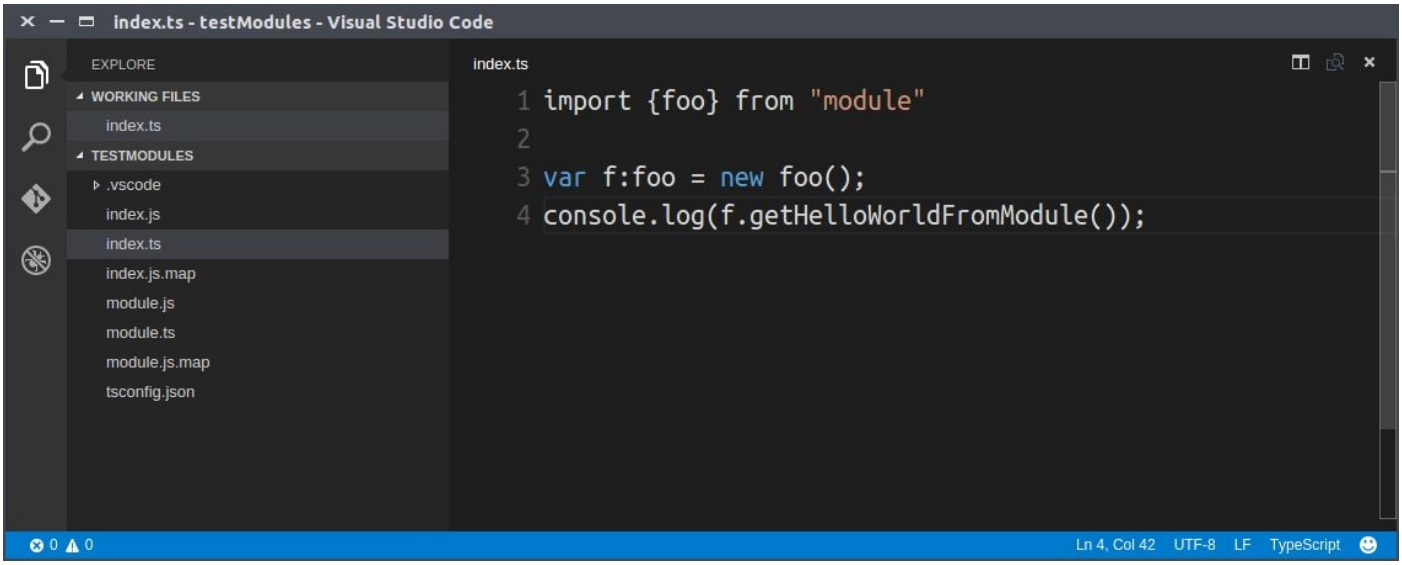
This settings will execute the `tsc` command with the `config.json` file options.

Do not include the `tasks.json` file to the `args` property or the `tsconfig.json` file will be ignored.

With the task ready, press `ctrl+shift+b` to execute it. The `.js` and `.map` files will be created without the module error from before.

2.12.2 Hiding js e map files on VSCode

After the initial compilation the VSCode interface should be similar to the next image:



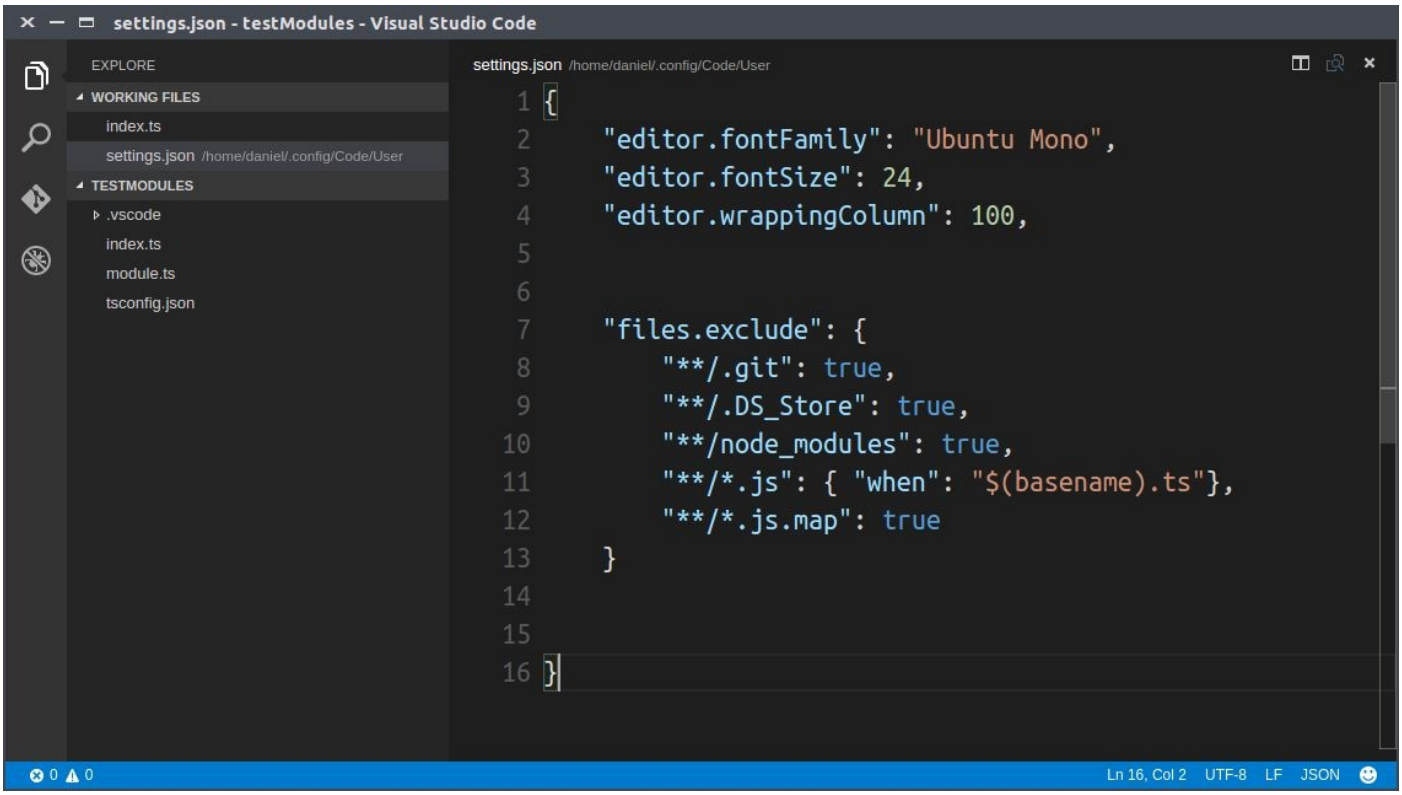
Notice that the file list is a little confused due to the files with the same name as `index.js`, `index.ts` e `index.js.map`. The `js` and `map` files are compiled files and don't need to be edited manually, therefore do not need to show on the editor.

To fix it, access `File > Preferences > User Settings` where the global user settings file will be opened. To hide the files on the editor, we must set up the `files.exclude` property as the following:

```
"files.exclude": {
        "**/.git": true,
        "**/.DS_Store": true,
        "**/node_modules": true,
        "**/*.js": { "when": "$(basename).ts"},
        "**/*.js.map": true
    }
```

In this example the `.git`, `.DS_Store`, `node_modules` will be hidden. With the `"**/*.js": { "when": "$(basename).ts"}` setting we are hiding all the `.js` files that have an equal `.ts` file and with `"**/*.js.map": true` we are hiding all the `.js.map` files.

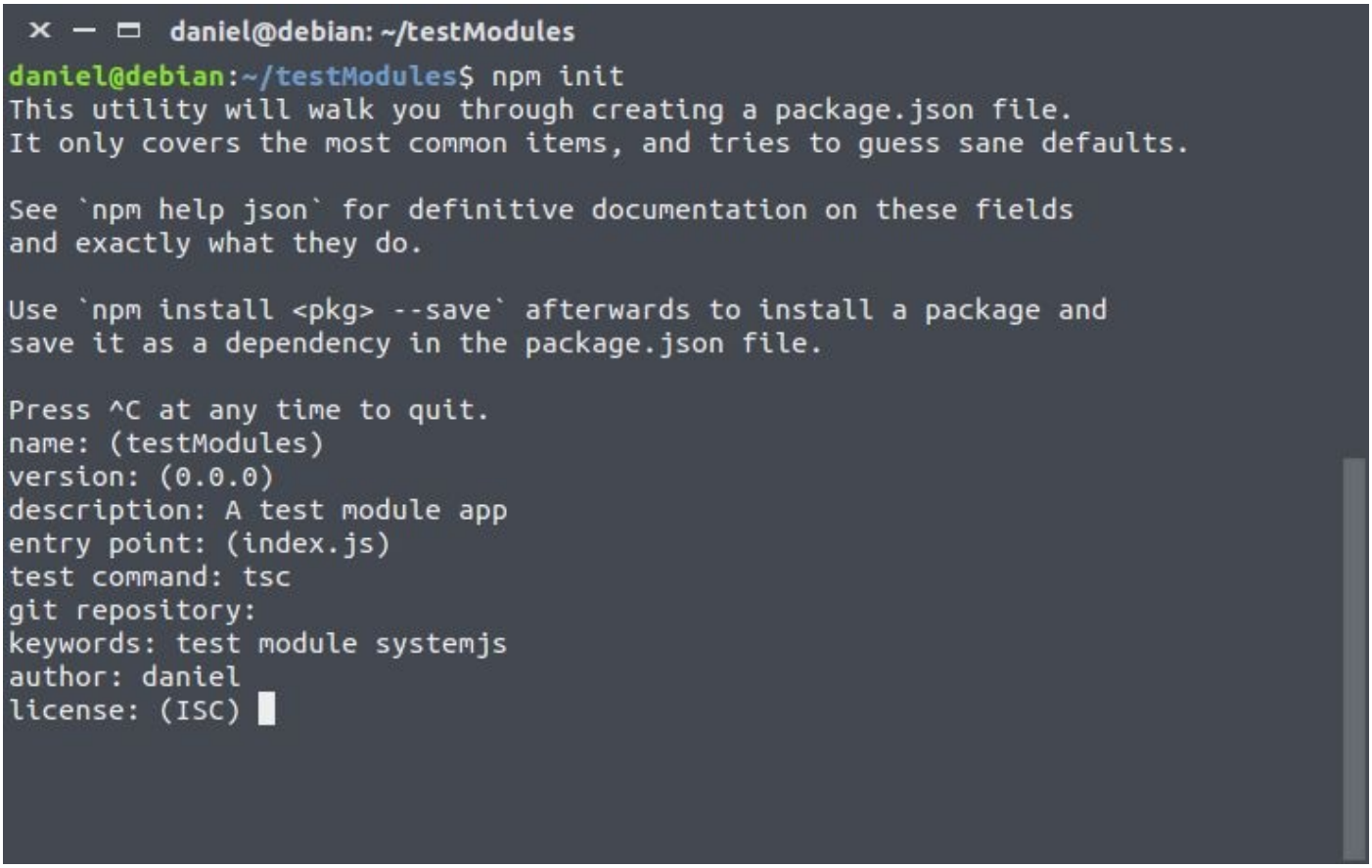
Now VScode should look like following:



2.12.3 Using SystemJS

Now that we created and compiled the TypeScript files, it is needed to test them on the browser. For that, it is needed to create the `index.html` file and insert the `SystemJS` library obtained from `npm`.

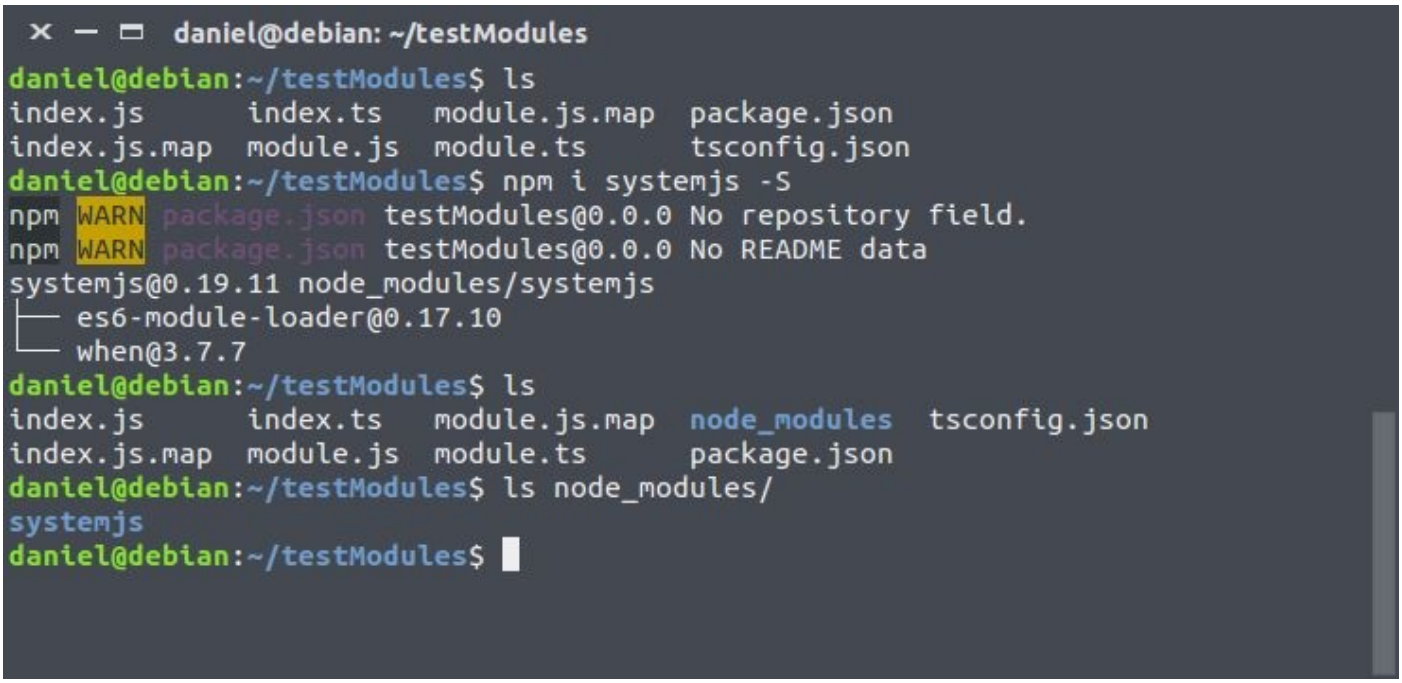
First start the *npm* on the directory with the `npm init` command as the following image:



With the `package.json` file ready it is possible to add the `systemjs` library with the following command:

```
$ npm i systemjs -S
```

As expected, the `node_modules` directory will be created and the `systemjs` added to it as the following image:



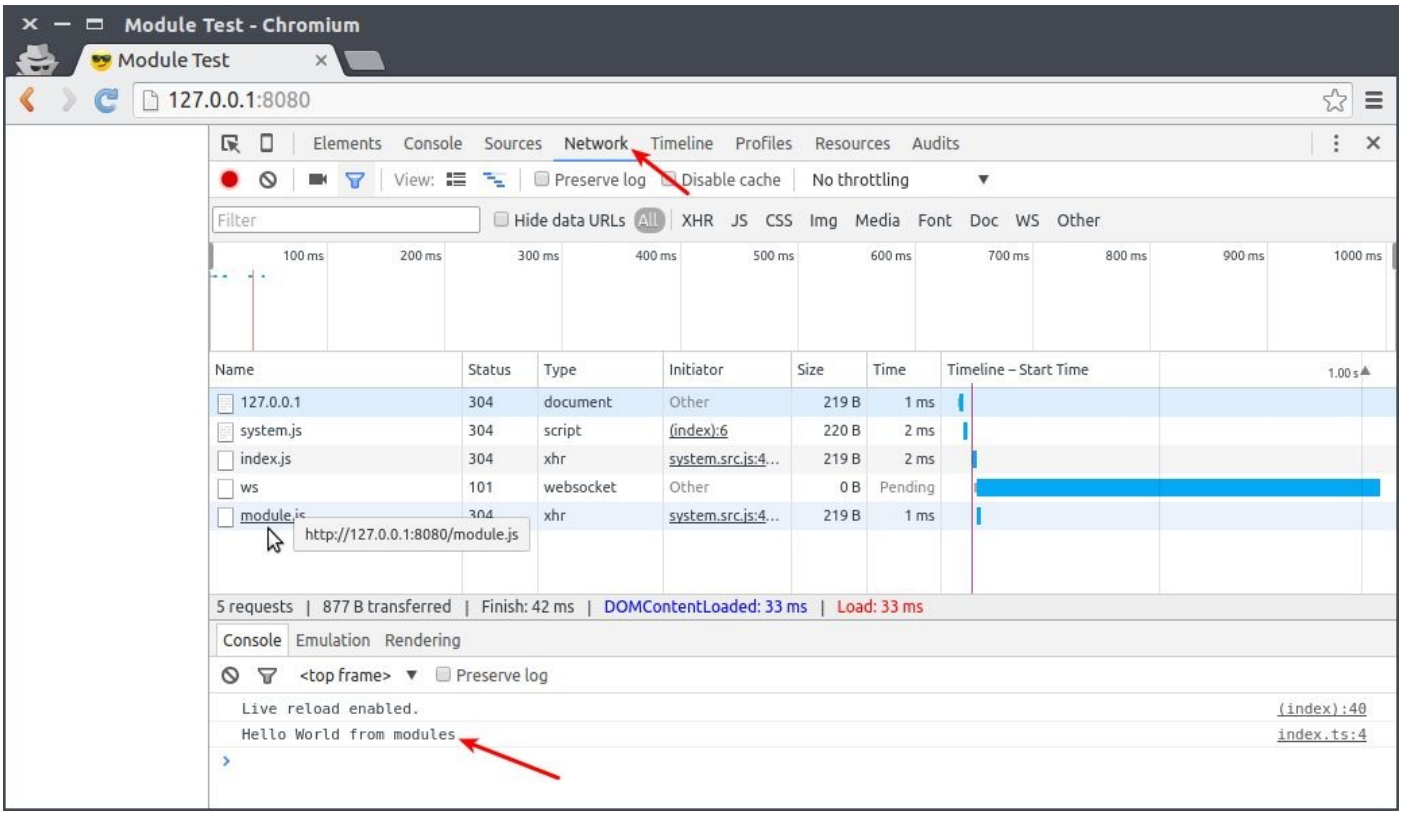
With the `systemjs` library ready we can create the `index.html` file with the following code:

index.html

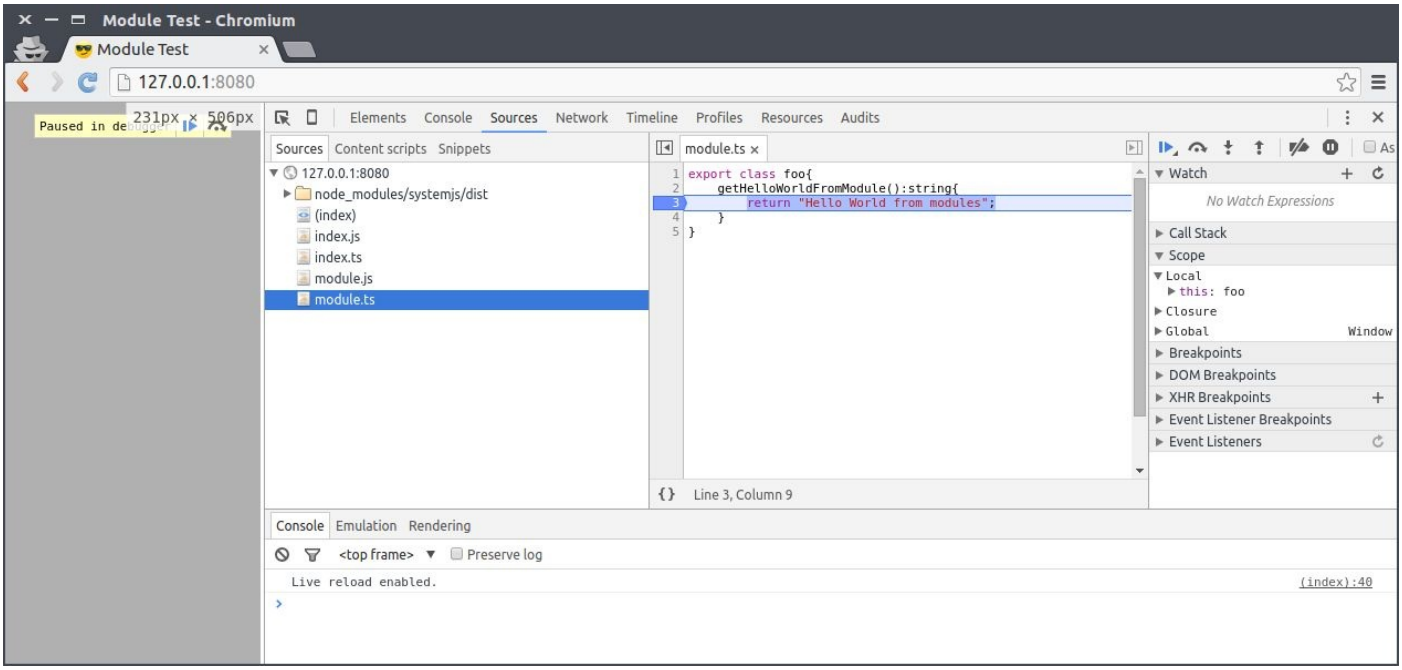
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Module Test</title>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script>
        System.defaultJSExtensions = true;
        System.import('index');
    </script>
</head>
<body>
</body>
</html>
```

The `index.html` file contains the `system.js` library, previously installed with *npm*. After including `system.js` to the html document, we added the `System.defaultJSExtensions = true` to enable the automatic extension of the files, and `System.import('index')` to import the `index.js` file.

The `index.js` file imports the `modules.js`. This process can be observed in Google Chrome (press F12 after loading the page), on the `Network` tab as on the following image:



To enable Debug on Google Chrome go to the `Sources` tab, select the TypeScript to be debugged and add a break point as the following image:



Too access the `index.html` file on the browser use the `live-server` previously installed. Access the `testModules` directory and type `live-server` as the following image:

```
I>  ![](images/969.png)
```

2.13 Decorators (or annotation)

Decorators are used in many programming languages. Its main job is to set up a class with properties. In some languages the *decorators* are called *annotations*.

For example, the Hibernate Java library use decorators to set up a class that represents a table, as on the following example:

```
import javax.persistence.*;

@Table(name = "EMPLOYEE")
public class Employee {
    @Id @GeneratedValue
    @Column(name = "id")
    private int id;
}
```

On the previous example we have the @Table decorator with the `name=="EMPLOYEE"` property. The `id` field also has the `@id` e `@Column` decorators, configuring the table field. The same can be used in C# as the following example:

```
public class Product
{
    [Display(Name="Product Number")]
    [Range(0, 5000)]
     public int ProductID { get; set; }
 }
```

And finally the same can be applied to TypeScript and is largely used in Angular 2. Here is an example:

```
import {Component} from 'angular2/core';

@Component({
     selector: 'my-app',
     template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

The @Component before the `AppComponent` class is an Angular 2 decorator. In this example it transforms the class in an Angular component with two distinct properties: `selector` and `template`.

It is important to know about the TypeScript decorators that the `tsconfig.json` file must have two new properties: `emitDecoratorMetadata` and `experimentalDecorators`. This will be explained in detail on the next chapter.

2.14 Conclusion

In this chapter we had a wide approach to TypeScript, which is the favorite language for Angular 2. The TypeScript is getting into the marked for offering a more solid alternative to the JavaScript development.

We also learned how powerful is Visual Studio Code to the JavaScript and Angular 2 development.

3. Let's practice

In this chapter we will introduce a few concepts about Angular 2.

3.1 AngularBase Project

The first task will be to create a basic project that we can be as a base project to new ones and copy/paste code blocks.

3.1.1 Setting up the project

Create the directory `AngularBase` and in the command line set up the `package.json` file with npm:

```
$ npm init
...(set up the options)...
```

Add the libraries `angular2` and `systemjs` to the project:

```
$ npm i angular2 systemjs -S
```

3.1.2 Setting up the TypeScript compilation

Open the `AngularBase` directory in `Visual Studio Code` and create the `tsconfig.json` file. As seen before, the `tsconfig.json` file contains the information on how to compile the TypeScript to JavaSCript. This file should have the following code:

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "sourceMap": true,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "exclude": [
    "node_modules"
  ]
}
```

This setup has a few extra settings compared to the previous ones. In `moduleResolution` it is defined the module loading strategy, in this case `node`. This way we can load the modules from `node_modules` directory. The `emitDecoratorMetadata` and `experimentalDecorators` settings enable the use of *decorators*. In `noImplicitAny` we

disable the error message for any method or property without type. And `exclude` will exclude the compilation of any TypeScript found in `node_modules` directory.

To set up the *task* to compile the TypeScript press `ctrl+shift+b` (in *Visual Studio Code*). If it doesn't exist yet, create it clicking in `Configure Task Runner`. The `.vscode/tasks.json` file should have the following settings:

.vscode/tasks.json

```
{
          "version": "0.1.0",
          "command": "tsc",
          "isShellCommand": true,
          "showOutput": "silent",
          "problemMatcher": "$tsc"
}
```

If you using another editor, instead of creating a task go to the command line and type `tsc`.

3.1.3 Creating the first Angular 2 component

Up to now no TypeScript file was created, so there is nothing to compile. Angular 2 is focused in component creation, so instead of adding HTML or JavaScript code to create an application we will focus on the creation and edition of components.

Let's create the `app` directory to store the components. Each component will have the `.component.ts` postfix and the first component to create is the one that defines the application. That said, let's create the `app/app.component.ts` file with the following code:

app/app.component.ts

```
import {Component} from 'angular2/core';

@Component({
     selector: 'my-app',
     template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

On the first line we imported the `Component` class straight from Angular 2. :Notice that `angular2/core` is defined in `node_modules/angular2/core.js`. After the *import* it is possible to use the `@Component` *decorator* with two parameters: `selector` and `template`. The `selector` parameter defines the tag to be used for the instance of the component on the *html* page. The `my-app` value states that the component will be loaded on the `<my-app>` `</my-app>` tag. And `template` defines the *html* code that will be used as the template for the component.

After the `@Component` *decorator* we can create the `AppComponent` class that uses `export` to state that the class is public to the module.

3.1.4 Creating the bootstrap

The `bootstrap` is responsible to start the Angular 2 providing may settings (that will be explained on the next chapters). Create the `app/boot.ts` with the following code:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'

bootstrap(AppComponent);
```

In the first line we import the `bootstrap` method from `angular2/platform/browser`. On the second line the `AppComponent` was imported. Notice that the path used in `from` is `./app.component`, where the `./` indicates the file root, in this case `app`. After the *imports* the `bootstrap(AppComponent);` method was called to start the Angular 2 application.

3.1.5 Creating an HTML file

To finish the base project it's necessary to create the HTML file of the application. Create the `index.html` file inside the `AngularBase` directory with the following code:

index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('app/boot')
            .then(null, console.error.bind(console));
    </script>
  </head>
  <body>
    <my-app>Loading…</my-app>
  </body>
</html>
```

Attention On the PDF file you will see some codes with back slash "\" on the end of line as on the following image:

index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script>
      System.config({
        packages: {
```

This indicates that the line continues on the next line. If you copy and paste the code, remove the back slash. In this example, instead of `</scri\pt>` it should be `</script>`.

This HTML file contains 3 blocks. The first contains the default AngularJS libraries, as on the previous chapter. Then we set up the SystemJS library to define the format of the application with the `format` property as `register`. In this case, the `register` is the same as `system.register`. We also used the `defaultExtension` property with the `js` value, which means that every file requested to SystemJS will have the post-fix added. On the `<body>` tag we have the `<my-app></my-app>` tag where the application will be rendered.

The `System.import` command will import the *bootstrap* `app/boot` previously created.

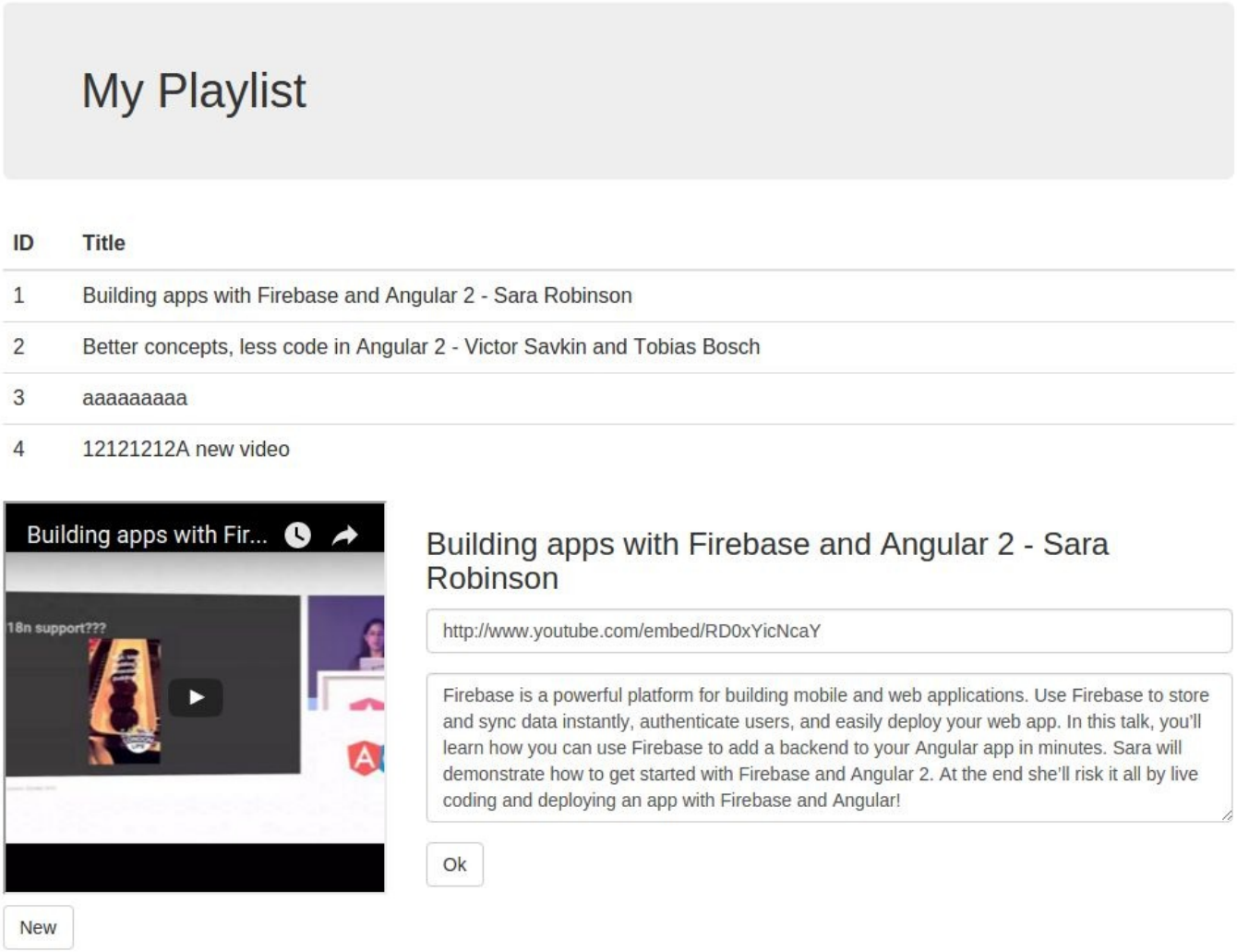
To test the project, compile the code with `ctrl+shit+b` in Visual Studio Code (VSCode) (or type tsc on the command line in the `AngularBase` directory) and use the `live-server` to start the web service.

With the project ready it is possible to use it to copy and paste for new projects (Copy the entire directory and paste in another project directory).

3.2 Creating a playlist

The objective here is to learn Angular 2 with examples, in practice. So instead of showing the theory of the different aspects of Angular 2, let's create a working application and explain the theory on the next chapter.

The next application is a *playlist* of videos. At first the *playlist* shows just the titles and when an item is selected the URL and description is shown. It is also possible to create and edit new videos, but if the page is refreshed the new data will be lost since, for now, we will not be using a server with a database.



3.2.1 Initial file structure

Copy the `AngularBase` directory and paste as `AngularPlaylist`. Check if the `node_modules` directory contains the libraries from `angular2` and `systemjs`. If not, execute the `npm install` to update the libraries.

Open the new `AngularPlaylist` directory in VSCode. The first thing to change is to add the `watch:true` property to the `tsconfig.json` file. This way the project will be compiled every time a file is changed. Press `ctrl+shift+b` to compile the project (and let the *watcher* compile the next changes) and execute the `live-server` on the command line to open the `index.html` file. The page will be opened with the `My First Angular 2 App` text.

To test if the TypeScript is working with the `live-server` open the `app/app.component.ts` file and change the component *template* to:

app/app.component.ts

```
import {Component} from 'angular2/core';

@Component({
     selector: 'my-app',
     template: '<h1>My Playlist</h1>'
})
export class AppComponent { }
```

The `live-server` will reload the page with the new *template* information.

3.2.2 Creating the application settings file

First it will be explained how to pass static variables between classes of the application. In a real project some of the application settings are stored in a settings file. Here we have the concept of *}Service** which is a class with the purpose of serving another class. In this case, we will create the `app/config.service.ts` file with the following code:

app/config.service.ts

```
export class Config{
     /**
     * Application page title
     */
    static TITLE_PAGE : string = "My Playlist";
}
```

The `Config` class contains at first the `TITLE_PAGE` static variable with the value `My Playlist`. To use this variable on the App component, first it's necessary to change the `app.component.ts` file to:

app/app.component.ts

```
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: '<h1>{{title}}</h1>'
})
export class AppComponent {
    title = "My Playlist";
}
```

In other words, we edited the *template* using a resource called **Databind**. And with `{{ }}` we can reference the class variable. The value of the `title` variable from `AppComponent` class will be assigned to the `{{title}}` in the template. There are many ways of **Databind** that we will see on the next lessons.

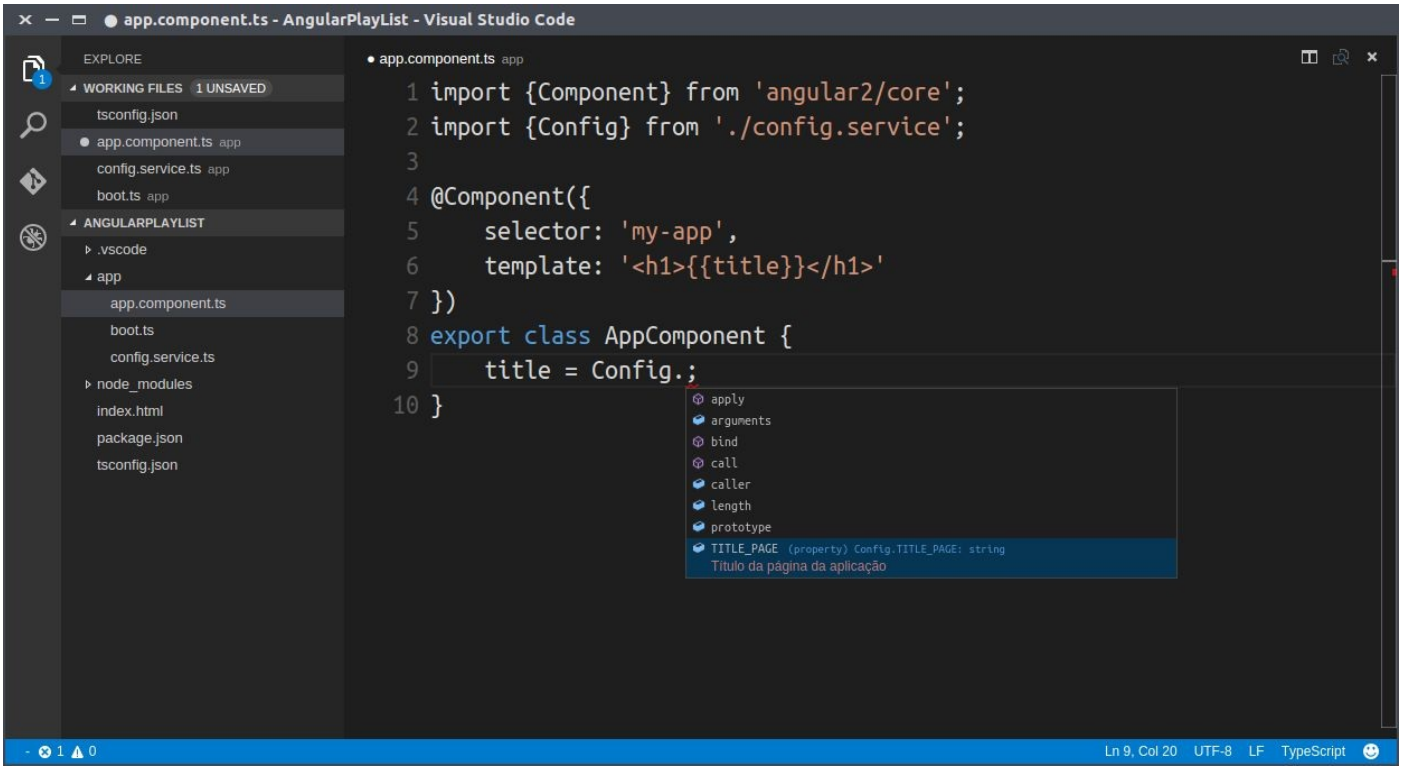
Now we need to bind the `title` in the `AppComponent` class to the `TITLE_PAGE` variable in the `Config` class. For that it is necessary to import the class and use it. As the variable is static, the page title is accessed by the `AppComponent` class without the need to instantiate the `Config` class.

app/app.component.ts

```
import {Component} from 'angular2/core';
import {Config} from './config.service';

@Component({
    selector: 'my-app',
    template: '<h1>{{title}}</h1>'
})
export class AppComponent {
    title = Config.TITLE_PAGE;
}
```

Notice that VSCode can help you with the code auto-complete:



3.2.3 Adding the bootstrap

To give a nice look to the application it is necessary to use CSS styles. One way to do it is using *bootstrap*. On the command line, add the bootstrap library with the following code:

```
$ npm i bootstrap -S
```

The `node_modules/bootstrap` directory will be added and we can include the style sheet on the `index.html` file:

index.html

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\
in.css">
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
```

```
        System.import('app/boot')
                .then(null, console.error.bind(console));
    </script>

  </head>
  <body>
      <div class="container">
          <my-app>Loading…</my-app>
      </div>
  </body>
</html>
```

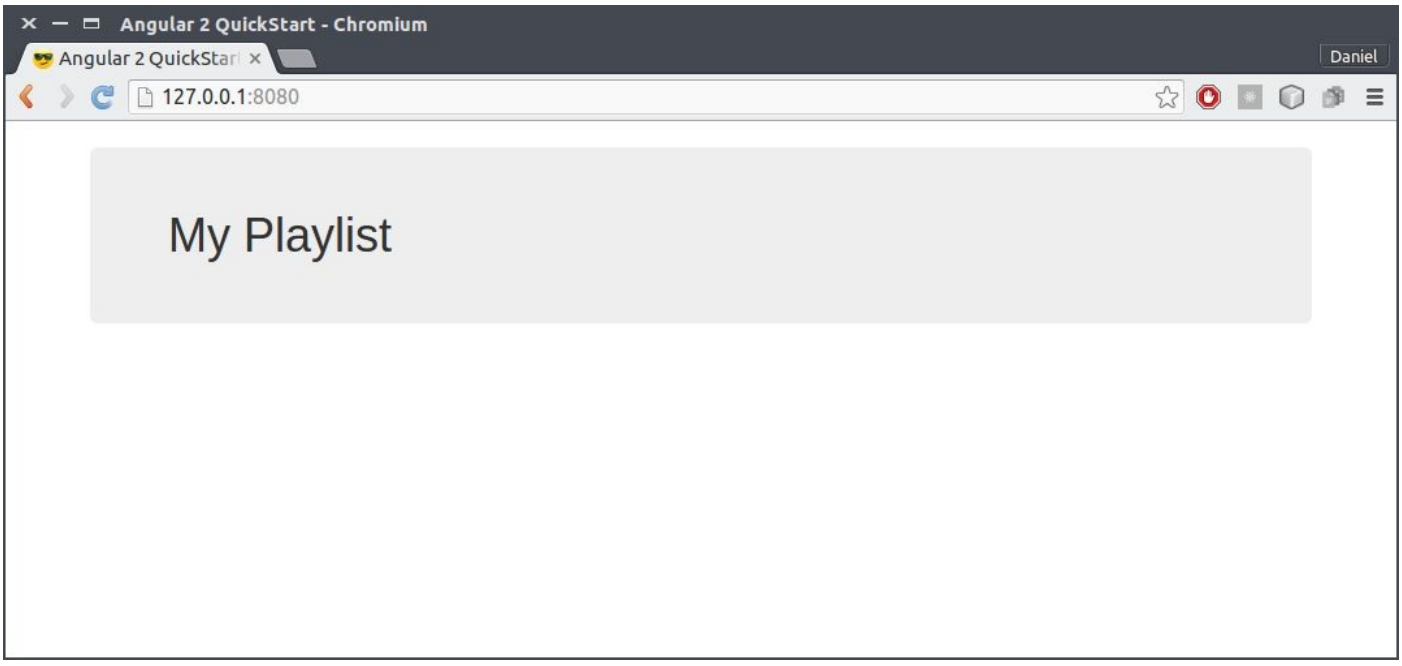
Changing the `index.html` the `bootstrap.min.css` style sheet was added and a `div` was created with the `container` class adding a margin to the application. Also in the `app.component.ts` file a css class should be added to the title:

app/app.component.ts

```
import {Component} from 'angular2/core';
import {Config} from './config.service';

@Component({
     selector: 'my-app',
     template: '<h1 class="jumbotron">{{title}}</h1>'
})
export class AppComponent {
     title = Config.TITLE_PAGE;
}
```

By adding `class="jumbotron"` to the `<h1>` of the component the application should be like the following:



3.2.4 Creating the Video class

For a video Playlist we need a class to represent the video containing the title, URL and description fields. This class will be called `video.ts` with the following code:

app/video.ts

```
export class Video{
     id:number;
     title:string;
     url:string;
     desc:string;
     constructor(id:number,title:string,url:string,desc?:string){
          this.id=id;
          this.title=title;
          this.url=url;
          this.desc=desc;
     }
}
```

3.2.5 Creating a simple video list

With the `Video` class created it is possible to create the video list. This list has an *array* with the data, a class and a *template*. First the data *array* is created with static data since we still don't have a server for the database.

```
import {Component} from 'angular2/core';
import {Config} from './config.service';
import {Video} from './video';

@Component({
     selector: 'my-app',
     template: '<h1 class="jumbotron">{{title}}</h1>'
})
export class AppComponent {
     title = Config.TITLE_PAGE;
     videos : Array<Video>;

     constructor(){
          this.videos = [
               new Video(1,"Test","www.test.com","Test Description"),
               new Video(2,"Test 2","www.test2.com")
          ]
     }
}
```

Notice that the `Video` class and the `videos` variable were added to the `app.component` file. And in its constructor we fed the *array* with two video items.

3.2.6 Creating child components

To display the video list we can add `<ul>` and `<li>` to the AppComponent *template* (where <h1> was created) but this is not a good practice, since we should always split the application into components.

Splitting the application into smaller components is a good practice in Angular 2, respecting the principle of Object Orientation. Access this link for more details.

For a list of videos we can create the `VideoListComponent` component. Create the `app/videolist.component.ts` file with the following code:

```
import {Component} from 'angular2/core';
@Component({
     selector: 'video-list',
     templateUrl: 'app/videolist.component.html'
})
export class VideoListComponent {

}
```

In this class we used the `templateUrl` instead of `template` just to exemplify that we can have a HTML template file separate apart from the component, which is a good practice in Angular 2. The path to the *template* must be full from the application base directory, and that is why the `app/` was added. But you can create a "templates" directory wherever you want.

Another good practice is to split each component into directories, mainly in projects with lots of components. As an example, the `VideoList` component would be in a directory called `app/components/video`. And this directory would have the related components, including the templates in a path similar to `app/components/video/templates`.

Create the `VideoList` component with a simple message:

app/videolist.component.html

```
<b> Video List Component </b>
```

After creating the component and its template, it is possible to add it to the main AppComponent:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
@Component({
 selector: 'my-app',
 template: '<h1 class="jumbotron">{{title}}</h1><video-list></video-list>',
})
export class AppComponent {
     ...
```

Notice that the `<video-list>` tag itself is not enough to load the `VideoListComponent` component. It's necessary to inform Angular to load the component with the **directives** property to be added in the `@Component`:

app/app.component.ts

```
1 import {Component} from 'angular2/core'
2 import {Config} from './config.service'
3 import {Video} from './video'
```

```
4 import {VideoListComponent} from './videolist.component'
5
6 @Component({
7   selector: 'my-app',
8   template: '<h1 class="jumbotron">{{title}}</h1><video-list></video-list>',
9   directives: [VideoListComponent]
10 })
11 export class AppComponent {
12     ...
```

The `directives` property was added to the line 9 with the `VideoListComponent` property with its related `import` on the line 4.

3.2.7 Formatting the template

The `AppComponent` template is described in a single line, which can be confusing since the HTML code can be long. There are two ways to fix this problem. The first is to use multiple lines on the template using back-ticks instead of apostrophe as delimiter:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
     selector: 'my-app',
     template: `
               <h1 class="jumbotron">
                    {{title}}
               </h1>
               <video-list></video-list>
               `,
     directives: [VideoListComponent]
})
export class AppComponent {
    ...
```

It is also possible to use the `templateUrl` property instead of `template`,, thus passing the HTML file URL:

app/app.component.ts

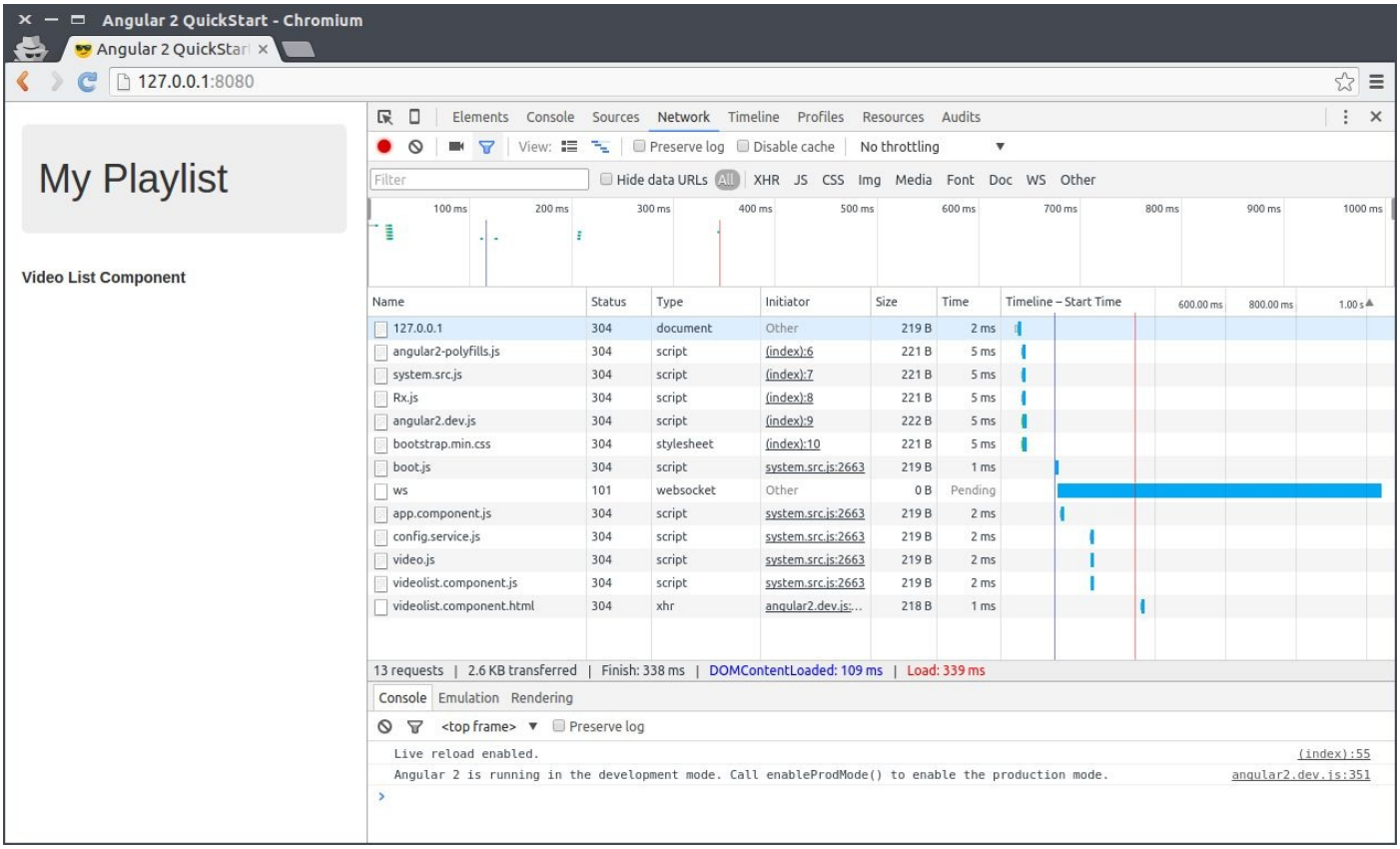
```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html',
     directives: [VideoListComponent]
})
export class AppComponent {
    ...
```

app/app.component.html

```
<h1 class="jumbotron">
    {{title}}
</h1>
<video-list></video-list>
```

After creating the template `AppComponent`, we can test the application, which must have a similar interface to the following image:



3.2.8 Passing values between components

To pass the array to the `VideoList` component we use the concept of *binding*. To pass a value from one component to another it is necessary to create a property on the component selector with brackets as following:

app/app.component.html

```
<h1 class="jumbotron">
    {{title}}
</h1>
<video-list [videos]="videos"></video-list>
```

By adding `[videos]="videos"` the compiler binds the `[videos]` variable from `VideoListComponent` to the `videos` variable in AppComponent. To create the `videos` variable on the `VideoListComponent` it is used the `inputs` property on the @Component setting:

app/app.component.html

```
import {Component} from 'angular2/core';
@Component({
     selector: 'video-list',
     templateUrl: 'app/videolist.component.html',
     inputs: ['videos']
})
export class VideoListComponent {

}
```

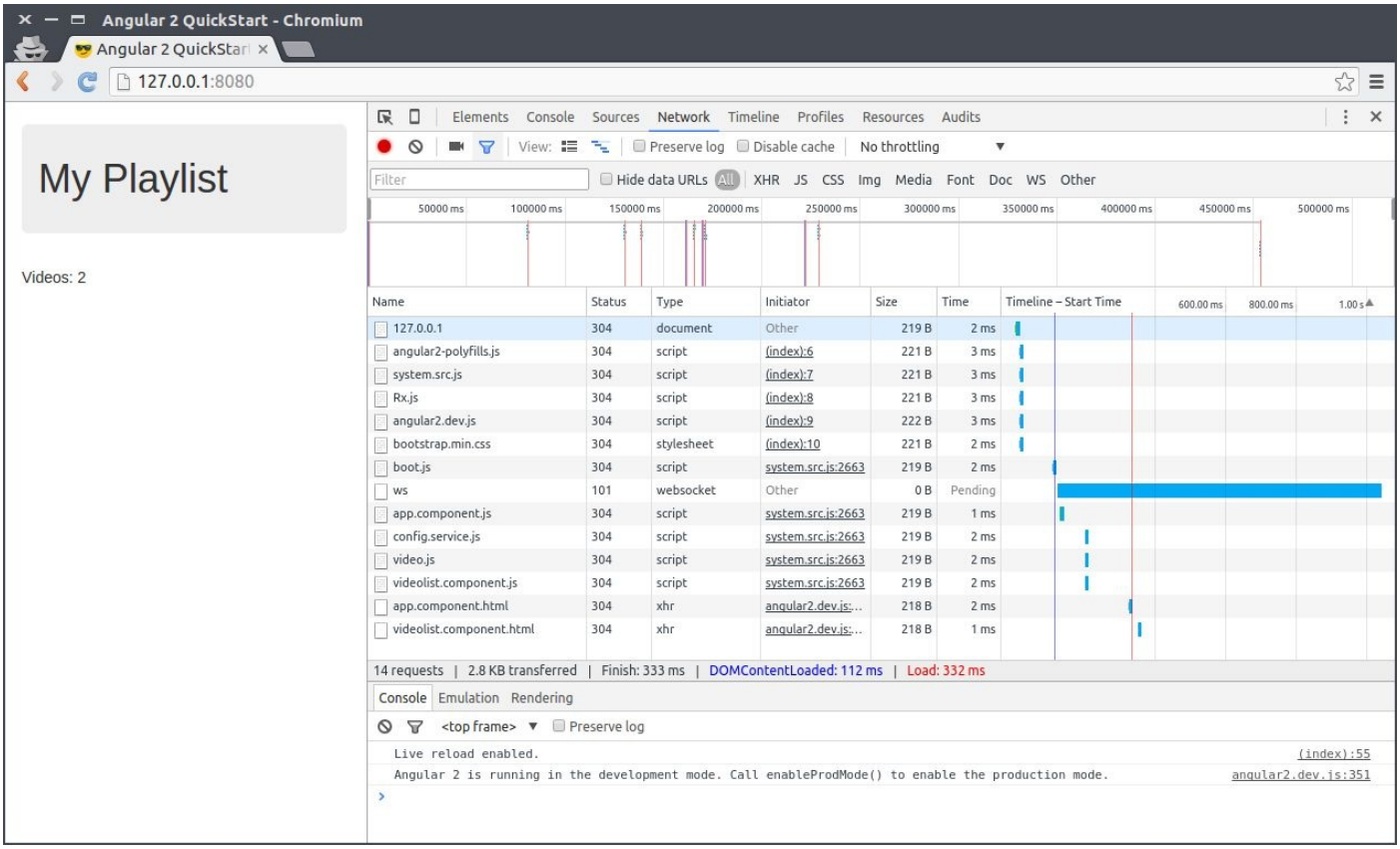
Another way is to create the `videos` variable in the class and se the `@input` metadata:

```
import {Component} from 'angular2/core';
@Component({
     selector: 'video-list',
     templateUrl: 'app/videolist.component.html'
})
export class VideoListComponent {
     @input() videos;
}
```

Since the `VideoListComponent` has the `videos` property, it is possible to change the template to use this variable. At first, we can change the template like this:

```
Videos: {{videos.length}}
```

Using the databind `{{videos.length}}` to return the amount of items in the `videos` variable. Since on the `AppComponent` two items were created, the value will be two, as on the following image:

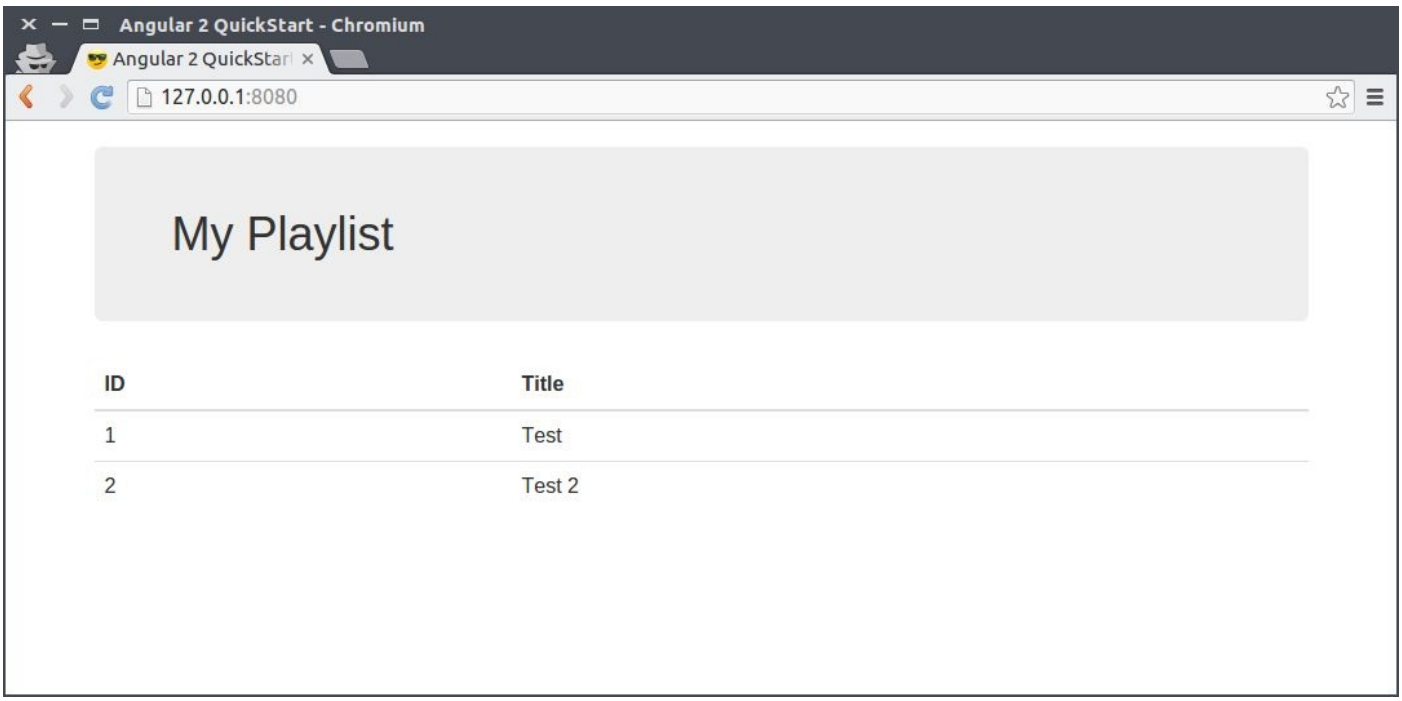


Instead of showing the amount of items in the array, let's change the template to draw a table, with the help of bootstrap:

```
<table class="table">
     <thead>
          <tr>
               <th>ID</th>
               <th>Title</th>
          </tr>
     </thead>
     <tbody>
          <tr *ngFor="#v of videos">
               <td>{{v.id}}</td>
               <td>{{v.title}}</td>
          </tr>
     </tbody>
</table>
```

To show the video list we used the `*ngFor` directive, in which the asterisk indicates that the `li` tag is part of a master/detail element. In other words, it will repeat itself according to the amount of items. The loop is done according to the `#v of videos` which means that each item in the array will be loaded into a `v` variable.

The result with the new template:



3.2.9 Selecting a video

Let's add the option to select a video from the list. When a line on the table is created, the `VideoComponent` will be shown with a call to the video and three text fields: title, URL and description.

3.2.10 Events

To add an event to the generated `<li>` we use the `(click)` directive:

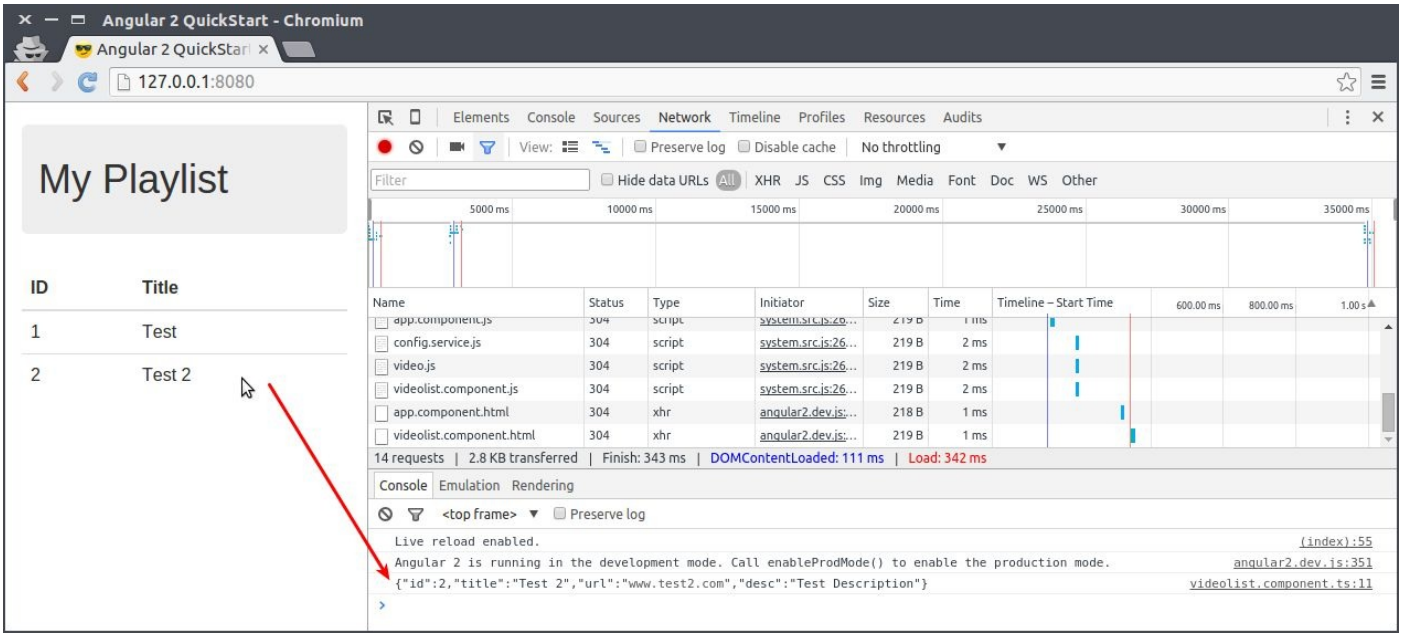
```
<table class="table table-hover">
     <thead>
          <tr>
               <th>ID</th>
               <th>Title</th>
          </tr>
     </thead>
     <tbody>
          <tr *ngFor="#v of videos" (click)="onSelect(v)">
               <td>{{v.id}}</td>
               <td>{{v.title}}</td>
          </tr>
     </tbody>
</table>
```

The `onSelect` method will be executed on the `VideoListComponent`, in which at first we can only send an `alert` to test:

```
import {Component} from 'angular2/core';
import {Video} from './video'

@Component({
     selector: 'video-list',
     templateUrl: 'app/videolist.component.html',
     inputs: ['videos']
})
export class VideoListComponent {
     onSelect(vid: Video) {
          console.log(JSON.stringify(vid));
     }
}
```

In the `VideoListComponent` class we created the `onSelect` method with the `vid` property, which is of type `Video` (include the `import` too). For now, the method only uses the `alert` to show the video data of the selected item:



3.2.11 Event Bubbling

When the user clicks on a video, it is necessary to inform the `AppComponent` about this event. The `VideoListComponent` can not directly call a method from `AppComponent`, this is a bad programming practice. The `VideoListComponent` must dispatch an event that will propagate to the `AppComponent`.

First let's create the event inside the `VideoListComponent`:

```
import {Component, EventEmitter} from 'angular2/core';
import {Video} from './video'

@Component({
     selector: 'video-list',
     templateUrl: 'app/videolist.component.html',
     inputs: ['videos'],
     outputs: ['selectVideo']
})
export class VideoListComponent {
     selectVideo = new EventEmitter();

     onSelect(vid: Video) {
           this.selectVideo.next(vid);
     }
}
```

Instead of `outputs: ['selectVideo']` in the `@Component` it is possible to use `@output() selectVideo = new EventEmitter()`.

Notice that we added `EventEmitter` class to the imports. We also added the new directive `outputs`, indicating that `SelectVideo` is an event to be sent out of the component. On the `onSelect` method, instead of the `console.log` we used the `selectVideo` variable to dispatch the event, with the `next` method, passing the selected video.

With the `VideoListComponent` dispatching the event, we can go back to the `AppComponent` to catch it. It will be don in two steps, first on the template:

```
<h1 class="jumbotron">
     {{title}}
</h1>
<video-list [videos]="videos"
     (selectVideo)="onSelectVideo($event)">
     </video-list>
```

On the template we used the `(selectVideo)` calling back the `onSelectVideo` and passing `$event`. Now we need to create the `onSelectVideo` method on the `AppComponent class:

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html',
```

```
    directives: [VideoListComponent]
})
export class AppComponent {
    title = Config.TITLE_PAGE;
    videos : Array<Video>;

    constructor(){
        this.videos = [
            new Video(1,"Test","www.test.com","Test Description"),
            new Video(2,"Test 2","www.test2.com","Test Description")
        ]
    }

    onSelectVideo(video){
        console.log(JSON.stringify(video));
    }
}
```

The interface result so far is the same, but now we have the selected video in the `AppComponent` which is the class that controls the entire application. The `VideoListComponent` should only show the video list and dispatch events.

On the templates we use [ ] to pass variables to the component and ( ) to indicate events.

3.2.12 Showing the video details

Let's create a new component called `VideoDetailComponent` showing the title of the video:

app/videodetail.component.ts

```
import {Component} from 'angular2/core'
import {Video} from './video'

@Component({
    selector:'video-detail',
    templateUrl: 'app/videodetail.component.html',
    inputs: ['video']
})
export class VideoDetailComponent{

}
```

The `VideoDetailComponent` class has the `video-detail` selector, the `videodetail.component.html` selector and `inputs:['video']` that will be the property to represent the selected video.

At this moment, the template has only the book title information:

app/videodetail.component.html

```
<h2>{{video.title}}</h2>
```

Now let's change the main application where we will create a property called `selectedVideo` to control the selected video by the `VideoListComponent` event.

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html',
     directives: [VideoListComponent]
})
export class AppComponent {
     title = Config.TITLE_PAGE;
     videos : Array<Video>;
     selectedVideo: Video;

     constructor(){
          this.videos = [
               new Video(1,"Test","www.test.com","Test Description"),
               new Video(2,"Test 2","www.test2.com","Test Description")
          ]
     }

     onSelectVideo(video){
          //console.log(JSON.stringify(video));
          this.selectedVideo = video;
     }
}
```

With the `selectedVideo` property ready we can control the visualization of the `VideoDetailComponent` component via the application's template:

app/app.component.html

```
<h1 class="jumbotron">
     {{title}}
</h1>

<video-list [videos]="videos"
     (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
     *ngIf="selectedVideo"
     [video]="selectedVideo">
</video-detail>
```

In this template we added the `<video-detail>` tag with the `*ngIf` directive to control if the component is shown. The asterisk is necessary to indicate to Angular that this control will change the DOM some how. Both `ngIf` and `ngFor` have this quality.

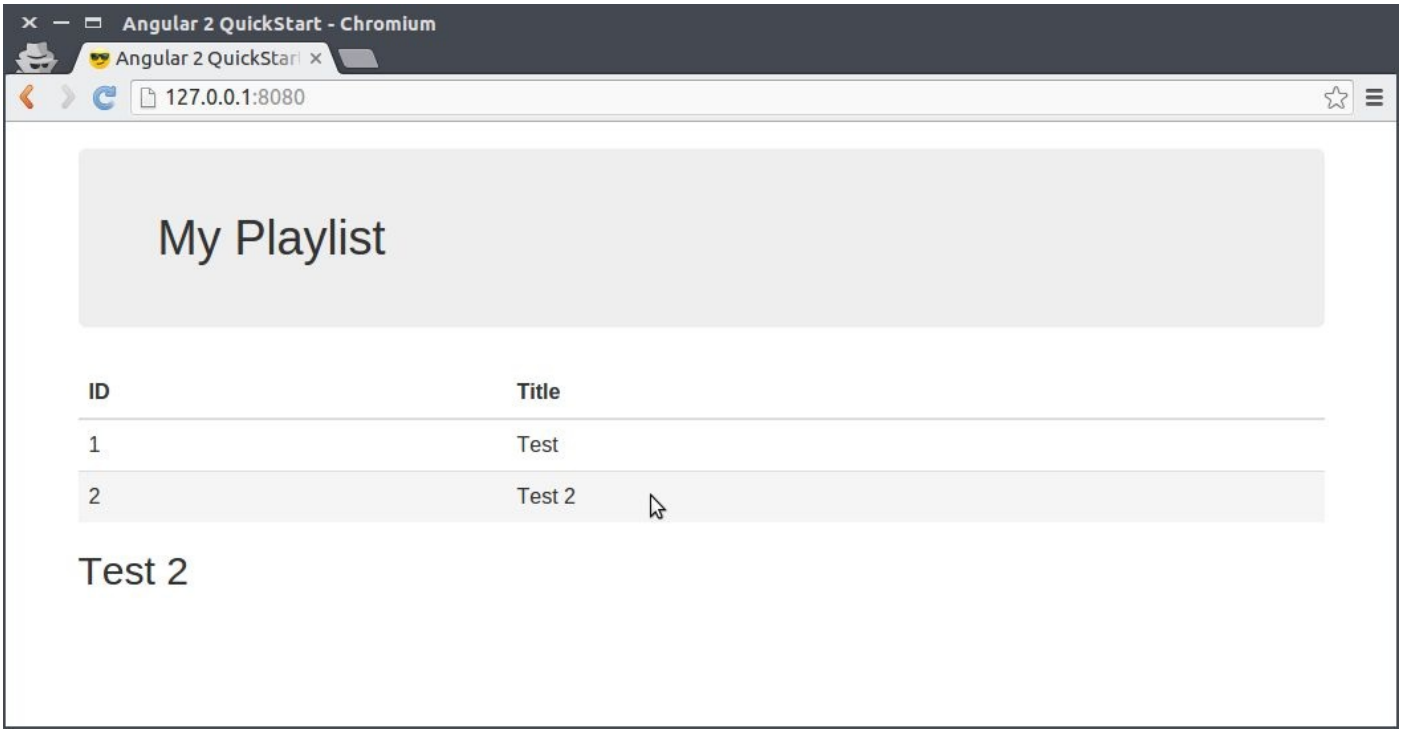
In `[video]="selectedVideo"` we have the `video` property from `VideoComponentDetail` receiving the value from `selectedVideo` in `AppComponent`.

Before testing the application it is necessary to get back to the `app.component.ts` and insert the `VideoDetailComponent` into the `directives` of the `@Component`, since this control is being added by the template and the Angular needs the class to be loaded:

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'
import {VideoDetailComponent} from './videodetail.component'

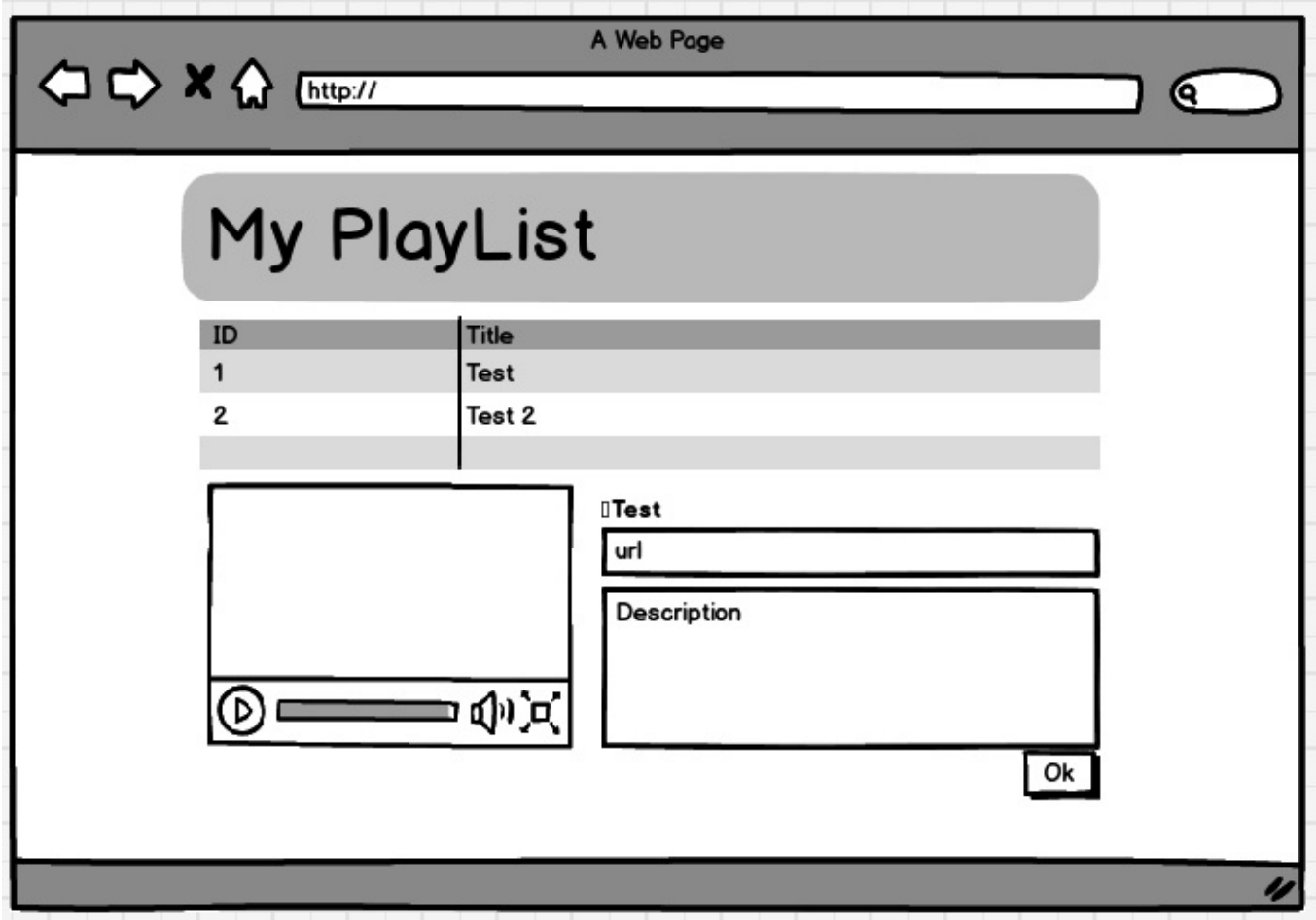
@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html',
      directives: [VideoListComponent,VideoDetailComponent]
})
export class AppComponent {
    ....
```

With the `VideoDetailComponent` added to `directives`, we can test the application and click on the video table lines to show the `VideoDetail` component as on the following image:



3.2.13 Editing the selected video data

Now let's improve the `VideoDetailtComponent` template. The following image illustrates how the screen should be drawn.



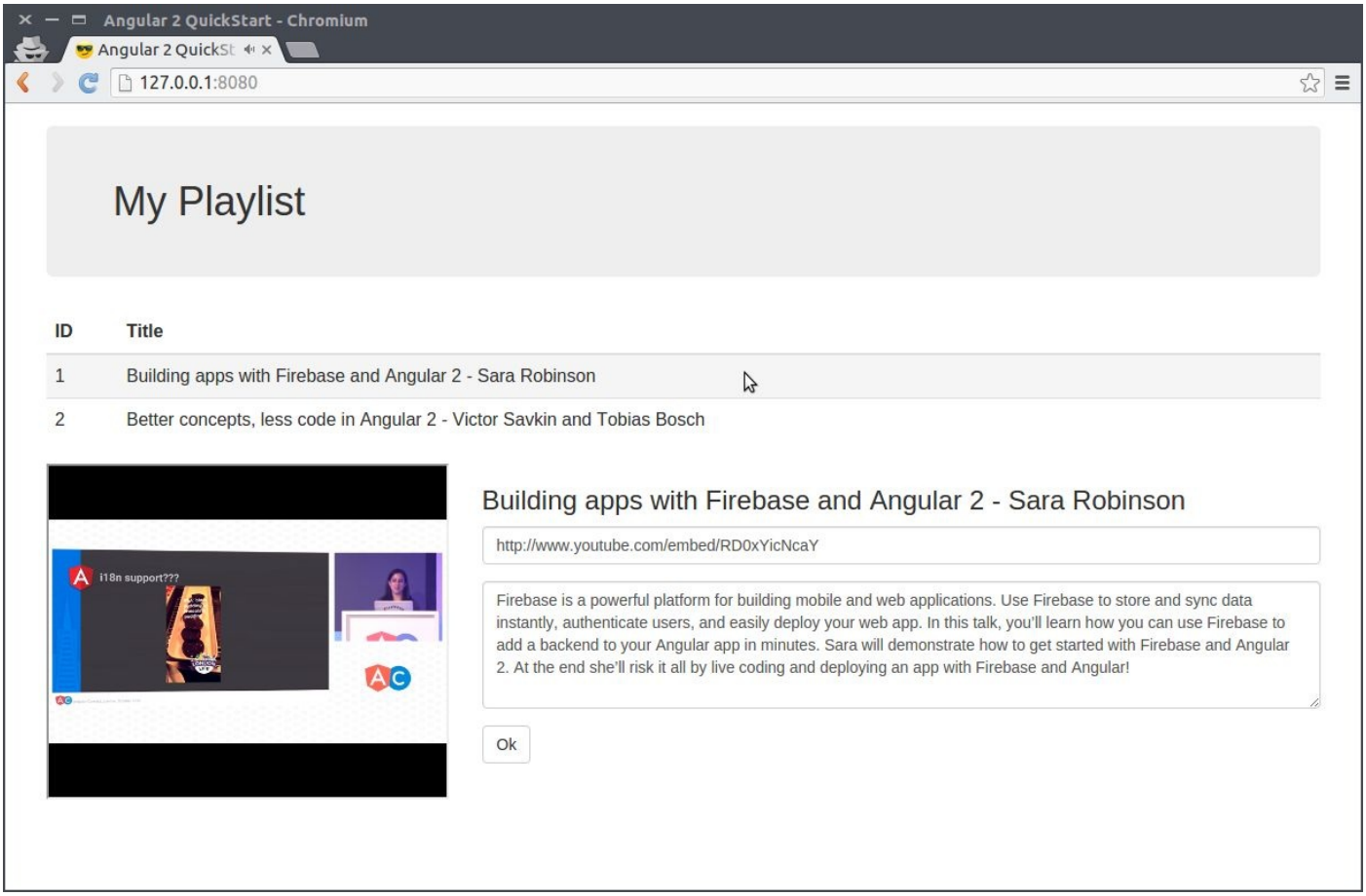
To implement this screen we will use some Bootstrap. We will also use the `[(model)]="field"` directive that implements the *Databind* between the text field and the property value.

veideodetail.component.html

```
<div class="row">
     <div class="col-md-4">
          <iframe width="100%" height="300" src="{{video.url}}">
          </iframe>
     </div>
     <div class="col-md-8">
          <form>
               <h3>{{video.title}}</h3>
               <div class="form-group">
                    <input type="input"
                         class="form-control"
                         id="url"
                         required
                         placeholder="url"
                         [(ngModel)]="video.url"
                          >
               </div>
               <div class="form-group">
               <textarea class="form-control" rows="5" [(ngModel)]="video.desc">
               </textarea>
               </div>
               <button type="button"
                    class="btn btn-default"
                    (click)="onButtonOkClick()"
                    >Ok</button>
```

```
        </form>
    </div>
</div>
```

In this form the `url` and `desc` fields are bound to the model via the `[()]`. This resource is called `Two Way DataBind` because if the text box value is changed, the `selectedVideo` property and array values are also changed.



3.2.14 Editing the title

A simple resource that we can use is the `(click)` event on the form title to change it from `<h3>` to `<input>`. To control this effect it is necessary to create the title field and add a variable to the `VideoDetailComponent`, on the following way:

veideodetail.component.html

```
<div class="row">
     <div class="col-md-4">
          <iframe width="100%" height="300" src="{{video.url}}">
          </iframe>
     </div>
     <div class="col-md-8">
          <form>
               <h3 *ngIf="!editTitle" (click)="onTitleClick()">
                    {{video.title}}</h3>
               <div *ngIf="editTitle" class="form-group">
                    <input type="input"
                         class="form-control"
                         id="title"
                         required
```

```
                placeholder="title"
                [(ngModel)]="video.title"
                 >
        </div>
        <div class="form-group">
             <input type="input"
                class="form-control"
                id="url"
                required
                placeholder="url"
                [(ngModel)]="video.url"
                 >
        </div>
        ....
        ....
        ....
```

The `editTitle` variable will control which element should be visible. By default the `<h3>` shows first, and if the user clicks on the element, the `onTitleClick` event will be dispatched. On the component we have:

videodetail.component.ts

```
import {Component} from 'angular2/core'
import {Video} from './video'

@Component({
     selector:'video-detail',
     templateUrl: 'app/videodetail.component.html',
     inputs: ['video']
})
export class VideoDetailComponent{
     private editTitle:boolean = false;
     onTitleClick(){
          this.editTitle=true;
     }
     onButtonOkClick(){
          //todo
     }
     ngOnChanges(){
          this.editTitle=false;
     }
}
```

On the component, when `onTitleClick` is executed, we change the value of the `editTable` property, thus hiding the `<h3>` and showing the `<input>`. We also added the `ngOnChanges` event that is executed every time the component data are changed. In other words, when the `video` property changes, the event will be dispatched and the `editTitle` variable returns to false.

The last implementation of the component is the `Ok` button to close the form dispatching the `closeForm` event.

videodetail.component.ts

```
import {Component, EventEmitter} from 'angular2/core'
import {Video} from './video'
```

```
@Component({
     selector:'video-detail',
     templateUrl: 'app/videodetail.component.html',
     inputs: ['video'],
     outputs: ['closeForm']
})
export class VideoDetailComponent{
     private closeForm = new EventEmitter();
     private editTitle:boolean = false;
     onTitleClick(){
          this.editTitle=true;
     }
     onButtonOkClick(){
          this.closeForm.next({});
     }
     ngOnChanges(){
          this.editTitle=false;
     }
}
```

To added the event, we import the `EventEmitter` class and created the `closeForm` event. The event is dispatched on the `onButtonOkClick` method.

In the `AppComponent` we add the event to the template:

```
<video-detail
     *ngIf="selectedVideo"
     [video]="selectedVideo"
     (closeForm)="onCloseDetailForm($event)"
     >
     </video-detail>
```

And set up the `onCloseDetailForm` method as follows:

app/app.component.ts

```
imports….
export class AppComponent {
     ....
      onCloseDetailForm(event){
          this.selectedVideo = null;
     }
     ....
}
```

By changing the value of the `selectedVideo` variable to `null`, the `VidelDetailComponent` will become invisible, thanks to `*ngIf="selectedVideo"`.

3.2.15 Creating a new item

To finish the application, it is necessary to create a button to add a new video.

app/app.component.html

```
<h1 class="jumbotron">
     {{title}}
</h1>
```

```
<video-list [videos]="videos"
     (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
     *ngIf="selectedVideo"
     [video]="selectedVideo"
     (closeForm)="onCloseDetailForm($event)"
     >
</video-detail>

<button type="button"
     class="btn btn-default"
     (click)="newVideo()">New</button>
```

And now create the `newVideo` method in the `AppComponent` class:

```
imports….
export class AppComponent{
     ...
     newVideo(){
          var v : Video = new Video(this.videos.length+1,"A new video");
          this.videos.push(v);
          this.selectedVideo = v;
     }
     ...
}
```

3.2.16 Some considerations

This little application use the simplest concepts in Angular 2. For example, to fill the data on the video list, we used a simple array populated in the `AppComponent`. In real applications the data should be requested from a server with a `service` for this task. We will talk about it on a further chapter.

There are also many other features in Angular 2 that helps when working with forms, that we will also see in a further chapter.

3.3 Creating Components

One of the advantages of working with components in Angular 2 is the possibility to reuse them. In other words, when we create a component, we can use it anywhere on our application. One of the main features of componentization is the creation of components inside components.

For the next example, copy the `AngularBase` and paste it as `AngularPanel` and add the bootstrap with the command:

```
$ npm i bootstrap -S
```

Add the `bootstrap` library to the `index.html` file:

index.html

```
<html>

  <head>
    <title>Angular 2 Quick Start</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\
in.css">

    <!-- 2. Configure SystemJS -->
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('app/boot')
              .then(null, console.error.bind(console));
    </script>

  </head>

  <!-- 3. Display the application -->
  <body>
    <div class="container">
        <my-app>Loading…</my-app>
    </div>
  </body>

</html>
```

Let's change the `AppComponent` template as the following:

```
import {Component} from 'angular2/core';

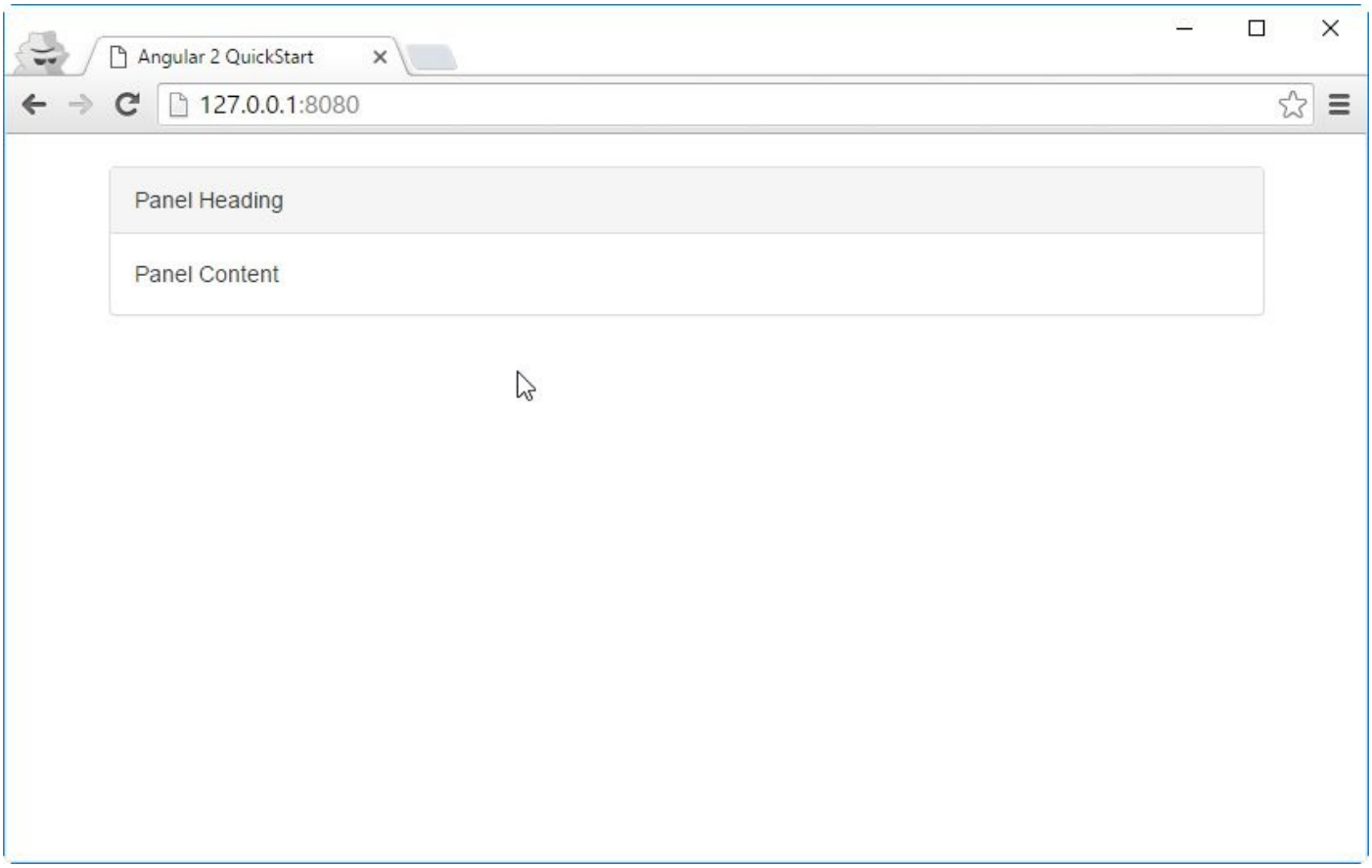
@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html'
})
export class AppComponent { }
```

And add the following template to the `app.component.html`:

```
<br/>

<div class="panel panel-default">
  <div class="panel-heading">Panel Heading</div>
  <div class="panel-body">Panel Content</div>
</div>
```

By now, we have a Panel from bootstrap as this:



Now let's create the `Panel` component that will be a *library* inside the `container` directory.

app/container/panel.ts

```
import {Component} from 'angular2/core';

@Component({
     selector: 'panel',
     templateUrl: 'app/container/panel.html'
})
export class Panel { }
```

The template to the Panel is, for now, the same HTML code of a bootstrap Panel:

```
<div class="panel panel-default">
  <div class="panel-heading">Panel Heading</div>
  <div class="panel-body">Panel Content</div>
</div>
```

With the component created, we can add it to the `AppComponent` changing the template:

app/app.component.html

```
<br/>
<panel></panel>
```

And also add the `Panel` directive:

app/app.component.ts

```
import {Component} from 'angular2/core';
import {Panel} from './container'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html',
     directives: [Panel] // <<<<<< Add panel to directives
})
export class AppComponent { }
```

Notice how we imported the `Panel`:

```
import {Panel} from './container'
```

This means that we have a library set up on the `container.ts` file:

app/container.ts

```
export * from './container/panel'
```

By refreshing the application (remember to use the `tsc` command and the `live-server`) we will see the same screen, but now we have the `Panel` component ready. Now we can create the `title` property like this:

app/container/panel.ts

```
import {Component} from 'angular2/core';

@Component({
     selector: 'panel',
     templateUrl: 'app/container/panel.html',
     inputs: ['title']
})
export class Panel { }
```

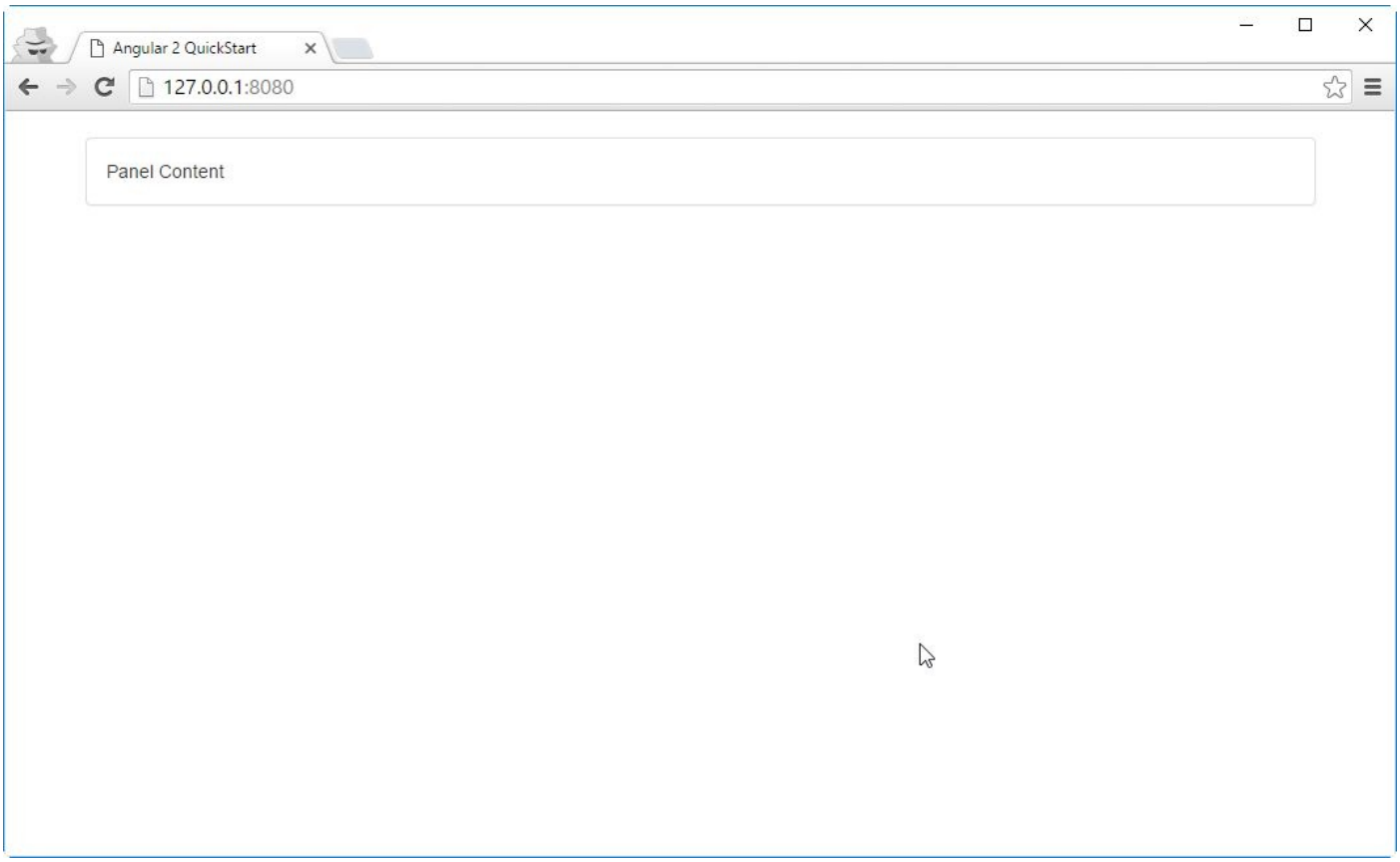
Notice that we added the `title` value into the `inputs` directive of the `@Component` setting up and entry variable to the component. We can use it on the template like following:

app/container/panel.html

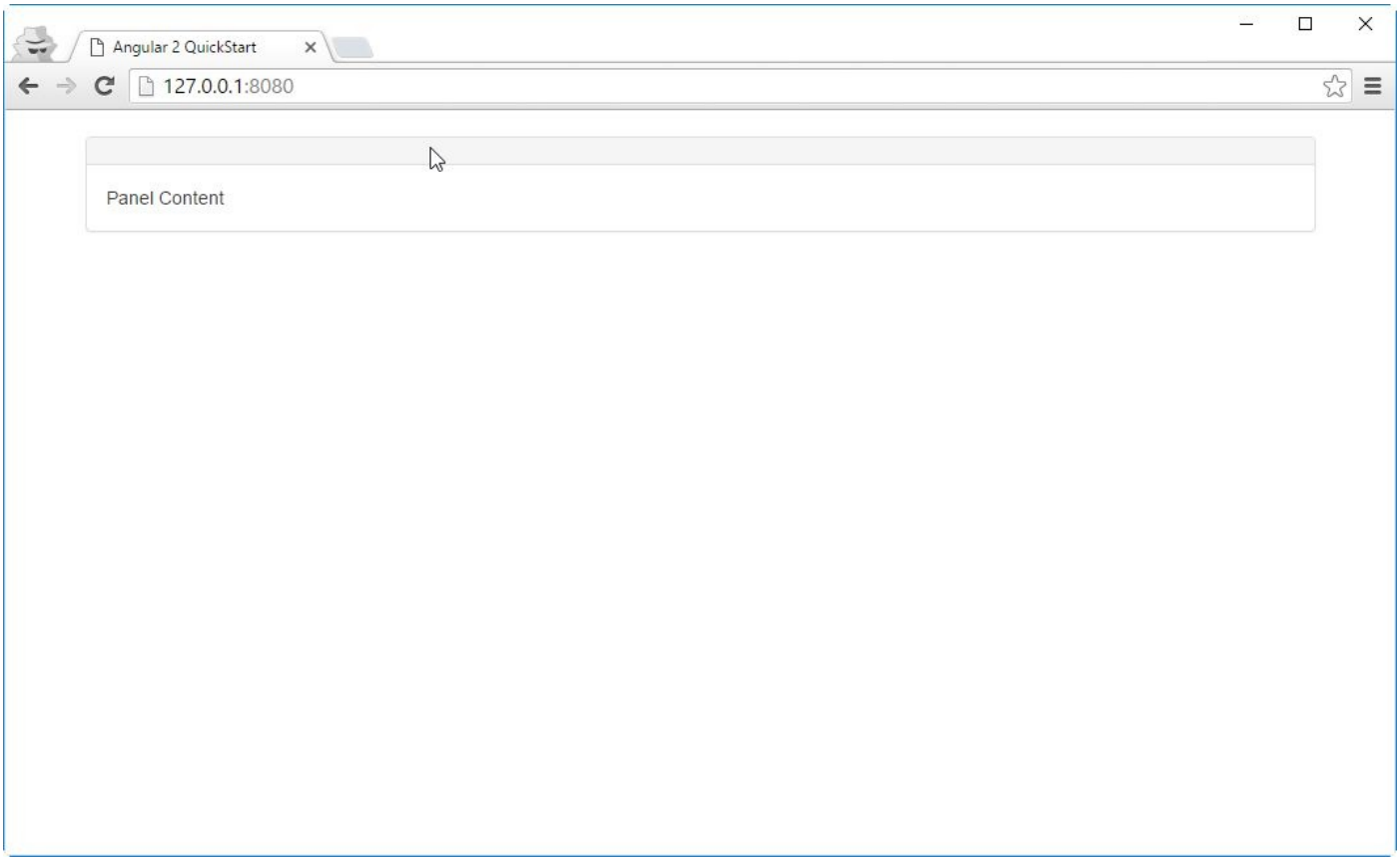
```
<div class="panel panel-default">
  <div class="panel-heading" *ngIf="title">{{title}}</div>
  <div class="panel-body">Panel Content</div>
</div>
```

Here we used the `*ngIf="title"` directive setting up that the `<div>` will only be inserted if there is a value for the `title` variable. In this div we add the variable with Databind `{{title}}`.

By compiling the application we have the following interface:



If, for example, the `*ngIf` were not in the `div` we would have the following output:

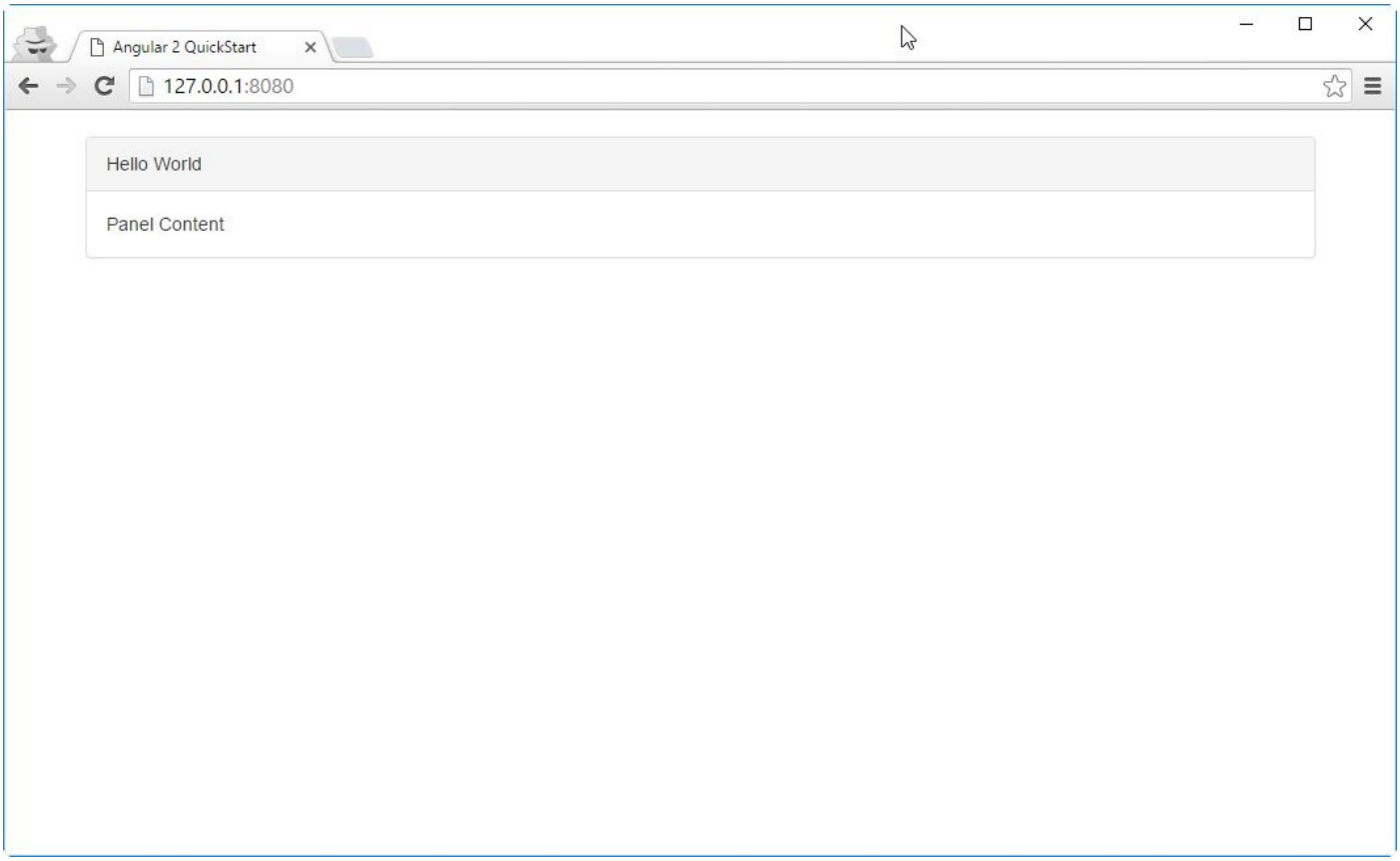


Now change the `AppComponent` template:

app/app.component.html

```
<br/>
<panel title="Hello World"></panel>
```

And the interface should be as follows:



3.4 Hierarchical components

The Panel component has an area where we can add a text or another component. This area is defined by the `<ng-content></ng-content>` element added to the `Panel`:

app/container/panel.html

```
<div class="panel panel-default">
  <div class="panel-heading" >{{title}}</div>
  <div class="panel-body"><ng-content></ng-content></div>
</div>
```

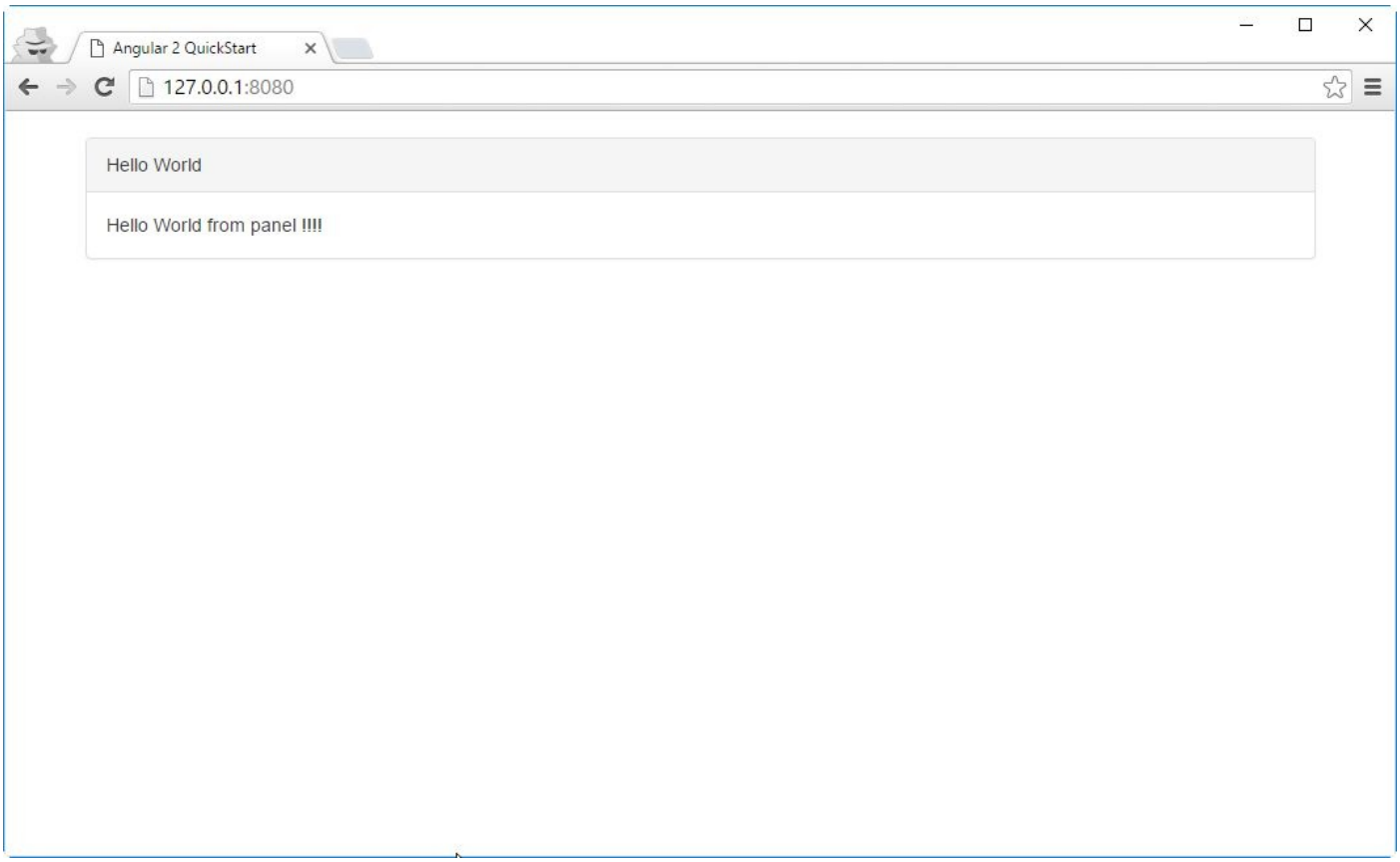
On the main component, we have the following changes:

app/app.component.html

```
<br/>

<panel title="Hello World">
    Hello World from panel !!!!
</panel>
```

After compiling the components we have the following screen:



It is also possible to add components inside components, which allows the creation of screens reusing HTML code:

app/app.component.html

```
<br/>

<panel title="Hello World">
        <panel title="Step 1">
            Open a terminal
        </panel>
        <panel title="Step 2">
            Say hello world!
        </panel>
</panel>
```

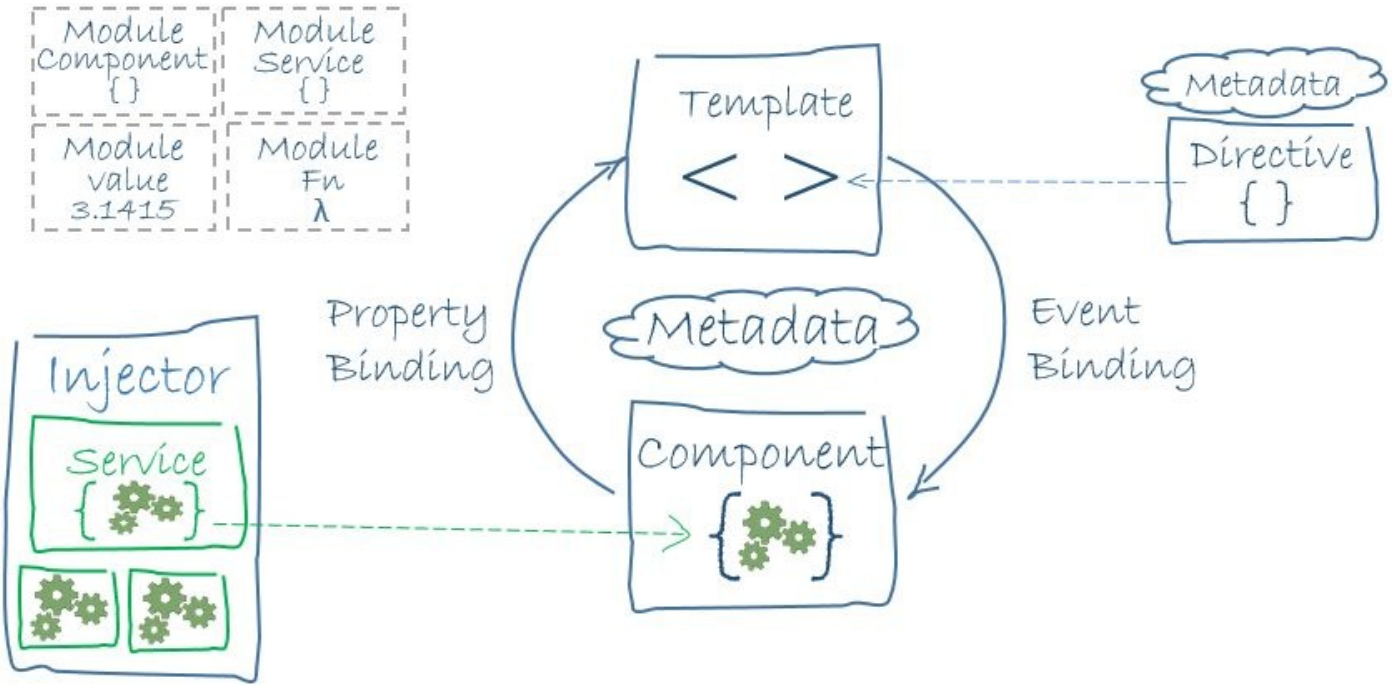
This template has the following result:

4. Some theory

In this chapter we will see some theory about Angular 2. We had a first impression with the previous chapters so now we should learn deeper the most important concepts of the framework.

4.1 Overview

On the official documentation of the framework we have an image that shows everything that you should know about Angular 2:



From that image we can highlight some key points to master the framework. They are:

- Module
- Component
- Template
- Metadata
- Service
- Directives
- Dependency injection

4.2 Module

Every Angular application can be composed by many modules that are loaded when the components are loaded. A module name is the same as its file name.

In the previous chapter example (*AngularPlayList*) we have examples of modules with the same name of the class, like:

app.component.ts

```
export class AppComponent{

}
```

and

config.service.ts

```
export class Config{

}
```

The `export` word sets up that the class will be **public** to the module. That way it is possible to import it in anywhere in the application.

```
import {Config} from './config.service'
```

The Angular 2 framework also works with modules. For instance:

```
import {Component} from 'angular2/core'
```

The difference between importing an Angular module or a module from the application is in the directory definition. When we use `angular2/core` we are referencing the Angular module that is in the `node_modules` directory.

4.2.1 Library Module

When many classes are created for just one module, it is called a Library. A library may contain components, directives, services, etc. The use of libraries doesn't mean that many classes should be created on the same file, but that we can group its calls on the same place.

Suppose an Angular application in which the model classes (that represents an object) are grouped on the model directory as on the following examples:

app/model/video.ts

```
export class Video{
     id:number;
     title:string;
     url:string;
     desc:string;
}
```

and

app/model/video.ts

```
import {Video} from './video'
export class Playlist{
     videos:Array<Video>;
```

```
    count():number{
        return this.videos.length;
    }
}
```

On the main application class, to use those model classes they should be imported on the following way:

app/app.component.ts

```
import {Video} from './model/video'
import {Playlist} from './model/playlist'
```

Until now our code is 100% right, but suppose that you want to group these classes in a library. You can create the `model.ts` file and use the following code:

app/model.ts

```
export * from './model/video'
export * from './model/playlist'
```

With the model library created, importing the `Video` and `Playlist` classes should be this way:

app/app.component.ts

```
import {Video,Playlist} from './model'
```

4.3 Component

A component in Angular is any visual part of your application. In most times a component has the `@Component` *decorator* that contains information as the component's directive, template, css style, etc. All the properties are defined by its *api*, available here.

A component in Angular 2 has the concept of relative design, in other words, the visual changes on the component are not defined accessing the DOM directly, but changing the component states. In the previous chapter, when the user clicked on the video list, a new component appeared on the screen showing the video details. To make the component visible, we never accessed the DOM of the HTML document to change the `display` property. We used the `*ngIf` directive:

```
<video-detail
    *ngIf="selectedVideo"
    >
</video-detail>
```

Components are the base of the applications in Angular 2. Every time we create an application it is necessary to think in how to split it into components in a way that they can exchange information.

4.4 Template

An important part of the component is the template, which defines how it will be drawn on the application. A template has HTML code, directives, event calls and other templates. A full example of a template was on the previous chapter:

```
<h1 class="jumbotron">
     {{title}}
</h1>

<video-list [videos]="videos"
     (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
     *ngIf="selectedVideo"
     [video]="selectedVideo"
     (closeForm)="onCloseDetailForm($event)"
     >
</video-detail>

<button type="button"
     class="btn btn-default"
     (click)="newVideo()">New</button>
```

Notice that there are notations on the template that define specific behaviors. Let's see each one of them:

4.4.1 Interpolation (Using {{ }})

The `{{title}}` is a *databind* for the template. It will use the value of the `title` variable from the class in the template. It is possible to use this *databind* in *tags* and *html* attributes, as on the following example:

```
<h3>
   {{title}}
  <img src="{{heroImageUrl}}" style="height:{{App.Style.Height}}">
</h3>
```

4.4.2 Template Expressions

Beside inserting values, expressions can be used to get results. It is possible to do `{{1+1}}` to get the value 2, for example. Or use `{{{meuarray.length}}` to get the amount of items in an array.

4.5 Property Bind

A property in a component may be bound to an event or method of the component. See the following example:

```
<video-detail
     *ngIf="selectedVideo"
     [video]="selectedVideo"
     (closeForm)="onCloseDetailForm($event)"
     >
</video-detail>
```

There are three examples of the *property bind* that can be used for three different situations:

- `*ngIf` will set if the component should be visible or not on the application. The use of the asterisk indicates to Angular that this property may change the page DOM. This will change the way Angular treats this component so it can be optimized.
- The use of `[video]=selectedVideo` indicates that the `selectedVideo` value will be passed to the `video` variable of the `VideoDetailComponent`.
- `(closeForm)="onCloseDetailForm($event)"` indicates an event that will happen on the `VideoDetailComponent` component and that will execute the `onCloseDetailForm` method. The `$event` property must be always present.

There is also the TwoWay DataBind, that is indicated with `[(target)]` as on the following example:

```
<input type="input"
        [(ngModel)]="video.title">
```

4.5.1 Loops

Another important *template expression* is the loop, used with the directive `*ngFor`. Notice the use of the asterisk again indicating that this directive changes the application DOM. The directive can be added to the element that will be repeated. On the previous chapter example we used:

```
<table class="table table-hover">
     <thead>
          <tr>
               <th>ID</th>
               <th>Title</th>
          </tr>
     </thead>
     <tbody>
          <tr *ngFor="#v of videos" (click)="onSelect(v)">
               <td>{{v.id}}</td>
               <td>{{v.title}}</td>
          </tr>
     </tbody>
</table>
```

On the `<tr>` example we created the loop and used the expression `#v of videos`, which means: take each element from `videos`, load it in a `v` variable and render `<tr>`. If you want to use the array index, change the expression to: `*ngFor="#v of videos, #i=index"`.

4.5.2 Pipes (Operator |)

A pipe adds a transformation to the expressions. The following example illustrates this behavior:

```
<div>{{ title | uppercase }}</div>
```

It is even possible to format a variable with *json*:

```
<div>{{currentHero | json}}</div>
```

The output would be:

```
{ "firstName": "Hercules",
  "lastName": "Son of Zeus",
  "birthdate": "1970-02-25T08:00:00.000Z",
   "url": "http://www.imdb.com/title/tt0065832/",
   "rate": 325, "id": 1 }
```

To format a date it is used `{{ birthday | date:"MM/dd/yy" }}` where `birthday` is an object of type Date.

4.6 Metadata (annotation)

The metadata are used to provide information to a class. Angular uses *decorators* as on the following example:

```
@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html',
})
export class AppComponent { }
```

In this example we have the `selector` that defines the selector to be used on the HTML document, in other words, by inserting `<my-app></my-app>` on the HTML document, the `AppComponent` will be rendered.

The `templateUrl` property defines an `url` with the path to the component's template. It is also possible to use the `template` as a metadata:

```
@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

4.7 Service

A `service` class is supposed to provide data, from a server or not. In fact the `service` class is an ordinary TypeScript class, but with the concept of data manipulation. On the chapter about Ajax we will see more details on how these classes works.

4.8 Dependency injection

The concept of dependency injection doesn't come from Angular 2, but from Object Orientation. Angular uses this concept to make easier the instantiation of classes by the components, providing the class instance automatically.

An "injected" class is globally available to the applications. Which means that if we change the property of an injected class, this change will still be active if the class is injected somewhere else.

To inject a class into another, it is necessary to inform it on the application's `bootstrap`:

```
bootstrap(AppComponent, [ConfigService]);
```

Or define the class on the `@Component` decorator:

```
@Component({
  providers:   [ConfigService]
})
export class AppComponent { ... }
```

After the setup it is possible to use the following syntax to inject an instance of the class:

```
export class AppComponent{
    constructor(private _config: ConfigService){ }
}
```

On the previous chapter we used a settings class called `Config` in which we used a static property called `TITLE_PAGE`. Let's change this behavior injecting the `Config` class into the `AppComponent`.

First, remove the `static` from the variable `TITLE_PAGE`:

app/config.service.ts

```
export class Config{
    /**
     * Title of the application page
     */
    TITLE_PAGE : string = "My Playlist";
}
```

And then change the `bootstrap` indicating that the `Config` class may be injected in any application class:

boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {Config} from './config.service'
import {AppComponent} from './app.component'

bootstrap(AppComponent, [Config]);
```

Now we change the `AppComponent` to reference the title in its constructor, using the `Config` class injected.

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'
import {VideoDetailComponent} from './videodetail.component'

@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html',
    directives: [VideoListComponent,VideoDetailComponent]
})
```

```
export class AppComponent {

     title:string;
     videos : Array<Video>;
     selectedVideo: Video;

     constructor(_config:Config){
          this.title = _config.TITLE_PAGE;
          ....
     }

     ...
}
```

4.8.1 Using the @Injectable()

In some classes where there are no *metadata* (@Component, for example) it is necessary to add the `@Injectable()` *metadata*, mostly if this class uses dependency injection. On the following example, the `TestService` class generates an execution error, because there is no *metadata* related to it, and it uses dependency injection with the `Http` class:

```
import {Http} from 'angular2/http'

export class TestService{
     constructor(http:Http){
          http.get(.......);
     }
}
```

To fix this problem, it is necessary to insert the `@Injectable()` metadata:

```
import {Http} from 'angular2/http'
import {Injectable} from 'angular2/core';
@Injectable()
export class TestService{
     constructor(http:Http){
          http.get(.......);
     }
}
```

```
{bump-link-number}

# Forms

In this chapter we will work on the features that Angular 2 provides for the\
 web forms development. Forms are used for the entry of data and is necessar\
y to control how this entry is done and if the data provided by the user is \
valid.

Angular 2 provides the following features:

- Databind between objects and form fields
- Capturing the form changes
- Validation of input data
```

```
- Event capturing
- Show error messages


## Creating the base project

Copy the base project `AngularBase` as `AngularForms` and add the bootstrap:

```bash
$ npm i bootstrap --D
```

Change the `index.html` to:

index.html

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\
in.css">

    <!-- 2. Configure SystemJS -->
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('app/boot')
            .then(null, console.error.bind(console));
    </script>

  </head>

  <!-- 3. Display the application -->
  <body>
    <div class="container">
        <my-app>Loading…</my-app>
    </div>
  </body>

</html>
```

Before creating the first form, let's create the `Person` class available in the `model` library. Create the `app/model/person.ts` and `app/model.ts` files like the following:

app/model/person.ts
```

```
export class Person {
  constructor(
    public id: number,
    public name: string,
    public email: string,
    public birthdate?:string
  ) {  }
}
```

This is a simple class that defines four fields, being the `birthdate` optional. Make the `model.ts` class exports the `Person` class to create a library:

app/model.ts

```
export * from './model/Person'
```

Now just create a simple service called `Mock` to contain the information shown on the form:

app/mock.ts

```
import {Person} from './model'
export class Mock{
     public mike = new Person(1,"Mike","mike@gmail");
}
```

Preparing the `Mock` class to be injected into other classes with bootstrap:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Mock} from './mock'

bootstrap(AppComponent,[Mock]);
```

With the `Mock` service ready it is possible get edit the AppComponent as follows:

app/app.component.ts

```
import {Component} from 'angular2/core';
import {Mock} from './mock'
import {Person} from './model'

@Component({
     selector: 'my-app',
     template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent {
     user:Person;

     constructor(_mock:Mock){
          this.user = _mock.mike;
     }
 }
```

With this setup it is possible to create a simple form and add it to the `AppComponent` component, changing the `template` property to `templateUrl` with the following code:

app/app.component.html

```
<form>
     <input type="hidden" id="id" name="id"/>
     <div class="form-group">
     <label for="name">Name</label>
     <input type="text" class="form-control" required>
     </div>
     <div class="form-group">
     <label for="email">Email</label>
     <input type="text" class="form-control" required>
     </div>
     <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Until now the Mock data were not used because we haven't set up the *TwoWay Databind* yet. To do it we must use the following syntax: `[(ngModel)]="model.name"`:

app/app.component.html

```
<form>
     <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
     <div class="form-group">
     <label for="name">Name</label>
     <input type="text" class="form-control"
                         required [(ngModel)]="user.name">
     </div>
     <div class="form-group">
     <label for="email">Email</label>
     <input type="text" class="form-control"
                         required [(ngModel)]="user.email">
     </div>
     <button type="submit" class="btn btn-default">Submit</button>
</form>
```

With that modification we already have data being shown on the form:

A simple way to check the `user` variable of the `AppComponent` class is to use the following template above the form:

app/app.component.html

```
{{user | json}}
<form>
     <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
     <div class="form-group">
     <label for="name">Name</label>
     <input type="text" class="form-control"
                              required [(ngModel)]="user.name">
     </div>
     <div class="form-group">
     <label for="email">Email</label>
     <input type="text" class="form-control"
                              required [(ngModel)]="user.email">
     </div>
     <button type="submit" class="btn btn-default">Submit</button>
</form>
```

This template will show the `user` variable and the `|` will format it into `json`. The result should be similar to the following:

Notice that by changing the value of the text box, the value of the `user` object is changed automatically.

4.9 Using the ngControl

Allowing the databind of the control is just one of the features that Angular provides. It is also possible to track the state of each component, allowing to know when a component was changed, invalidated, etc. Besides checking the actual state of the control it is possible to add *css* styles to it, highlighting the control states, for example, an error state.

To enable this validation it is necessary to use the `ngControl` directive passing the property to be tracked:

app/app.component.html

```
{{user | json}}
<form>
     <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
     <div class="form-group">
     <label for="name">Name</label>
     <input type="text" class="form-control"
                    required [(ngModel)]="user.name"
                    ngControl="name"
```

```
                >
    </div>
    <div class="form-group">
    <label for="email">Email</label>
    <input type="text" class="form-control"
                required [(ngModel)]="user.email"
                ngControl="email"
                >
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
</form>
```

By adding the `ngControl` it will observe the control and control the CSS styles of the field according to the behavior of the control. For instance, the `name` field has the `required` instruction, that defines the field as mandatory. If this field is filled, the `ng-invalid` css class will be added to it. We can then use some css to show an information to the user. First create the `styles.css` file with the following code:

styles.css

```
.ng-invalid {
  border-left: 5px solid #a94442;
}
```

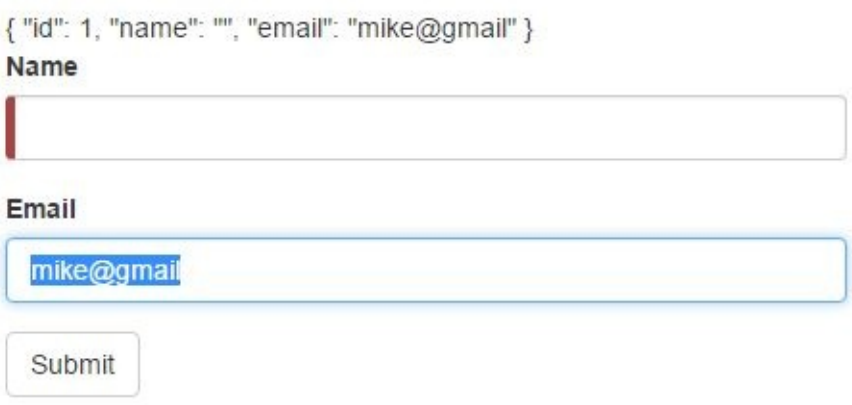
Depois, adicione o arquivo `styles.css` no `<head>` do `index.html`:

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    ....
    <link rel="stylesheet" href="styles.css">

    <!-- 2. Configure SystemJS -->
    ....
```

After adding the style, compile the application and access the form. Delete the text from the `Name` field and `Email` and notice that the style was applied, with a red border as the following image:



Fill again the fields and notice that the style is removed.

4.10 Showing an error message

To show an error message it is necessary to create a whose visibility can be controlled fromo a control variable. To create this variable, use the # followed by the name of the variable and its value. In the case of the `name` field we have:

```
<div class="form-group">
     <label for="name">Name</label>
     <input type="text" class="form-control"
                    required [(ngModel)]="user.name"
                    ngControl="name"
                    #name="ngForm"
                    >
          <div [hidden]="name.valid" class="alert alert-danger">
                Name is required
          </div>
     </div>
```

Notice that we created `#name="ngForm"` which means we created the `name` variable in the input field pointing to `ngForm`, which is the form itself. Then we created a and used the `[hidden]` directive to control its visibility:

4.11 Disabling the submit button of the form

It is possible to disable the *Submit* button from the form in case of a form not ready to be submitted. For that it is necessary to create a variable that represents the `ngForm`. It is done with the `<form>` tag:

```
<form #f="ngForm">
```

On the `submit` button it is possible to create the `[disabled]` directive with the `#f` variable created:

```
<button [disabled]="!f.valid"
     type="submit"
     class="btn btn-default">Submit</button>
```

When the form is invalid, the `f.valid` will be `false` and `!f.valid` will be `true`, disabling the button as on the following image:

4.12 Submitting the form

In Angular 2 it is recommended to use Ajax to send the data to the server, using the `(ngSubmit)` directive passing a method that will be executed on the component, as on the following example:

app/app.component.html

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
```

In the component we have:

app/app.component.ts

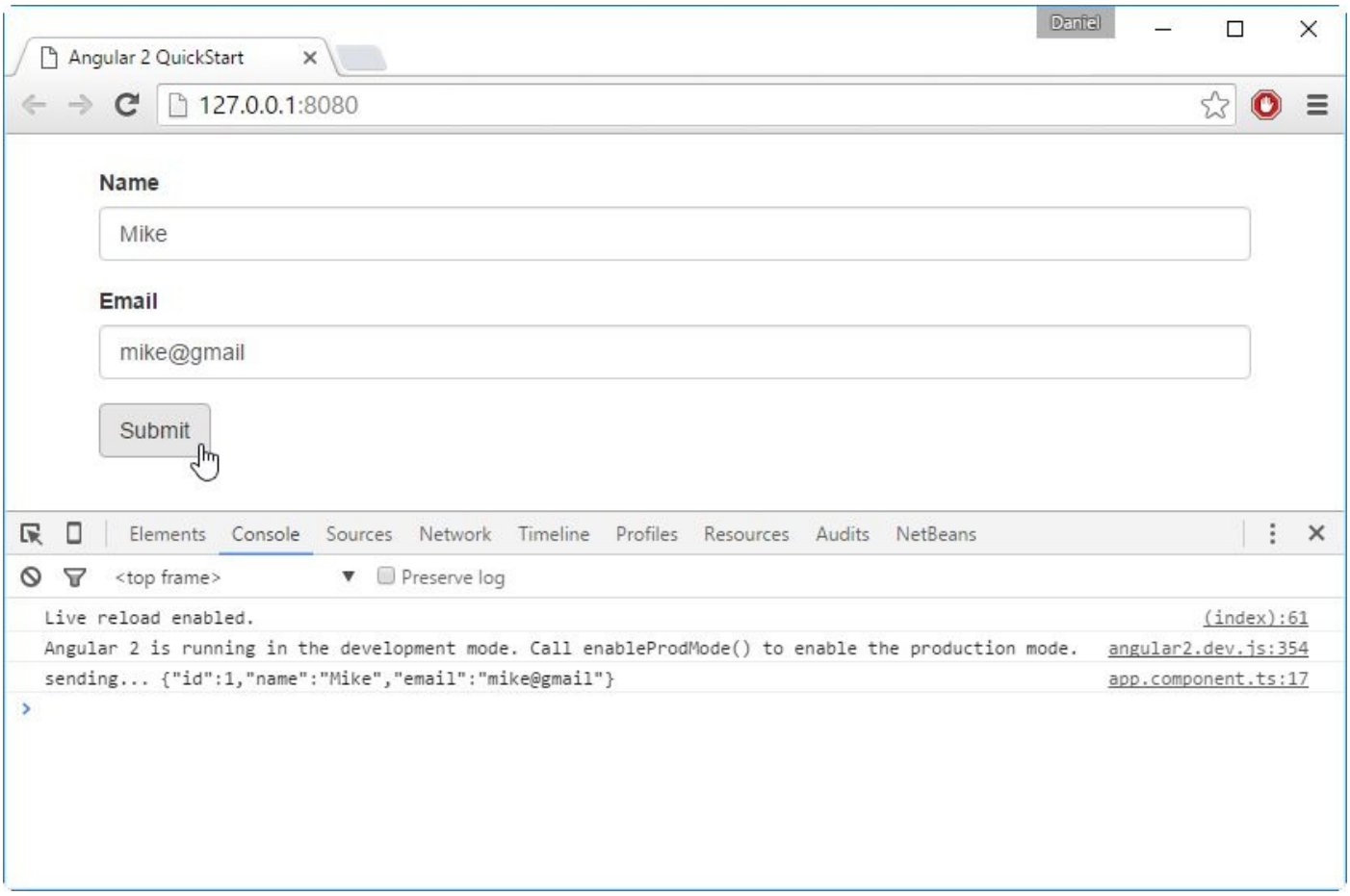
```
import {Component} from 'angular2/core';
import {Mock} from './mock'
import {Person} from './model'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html'
})
export class AppComponent {
     user:Person;

     constructor(_mock:Mock){
          this.user = _mock.mike;
     }

     onSubmit(f){
          console.log("sending…" + JSON.stringfy(user));
     }
 }
```

By clicking on the `submit` button we get a response similar to the following image:



4.13 Controlling the form visibility

A good practice that helps the user to get a response from the form submission is to create a control variable that indicates if the form was submitted. With this variable it is possible to use `[hidden]` to define what will be visible:

app/app.component.html

```
<div [hidden]="submitted">
     <form>
          ....
      </form>
</div>
<div [hidden]="!submitted">
     Sending… {{user|json}}
</div>
```

The `submitted` variable is controlled on the `AppComponent` component:

app/app.component.ts

```
import {Component} from 'angular2/core';
import {Mock} from './mock'
import {Person} from './model'

@Component({
     selector: 'my-app',
     templateUrl: 'app/app.component.html'
})
export class AppComponent {
     user:Person;
```

```
    submitted:boolean;

    constructor(_mock:Mock){
         this.submitted = false;
         this.user = _mock.mike;
    }

    onSubmit(f){
         this.submitted = true;
         console.log("sending… " + JSON.stringify(this.user));
    }
}
```

5. Connecting to a server

Almost every real application needs to persist data in a data base, and on the web development it is used an architecture on the style client/server, in other words, the Angular application must access a server via an URL, that will respond back to the client application.

The web server can be made in any language (java, php, .Net, etc), but the communication between a server and the client application must follow a pattern. This pattern, in this case, is the *json*, a data text format that can represent objects or array of objects.

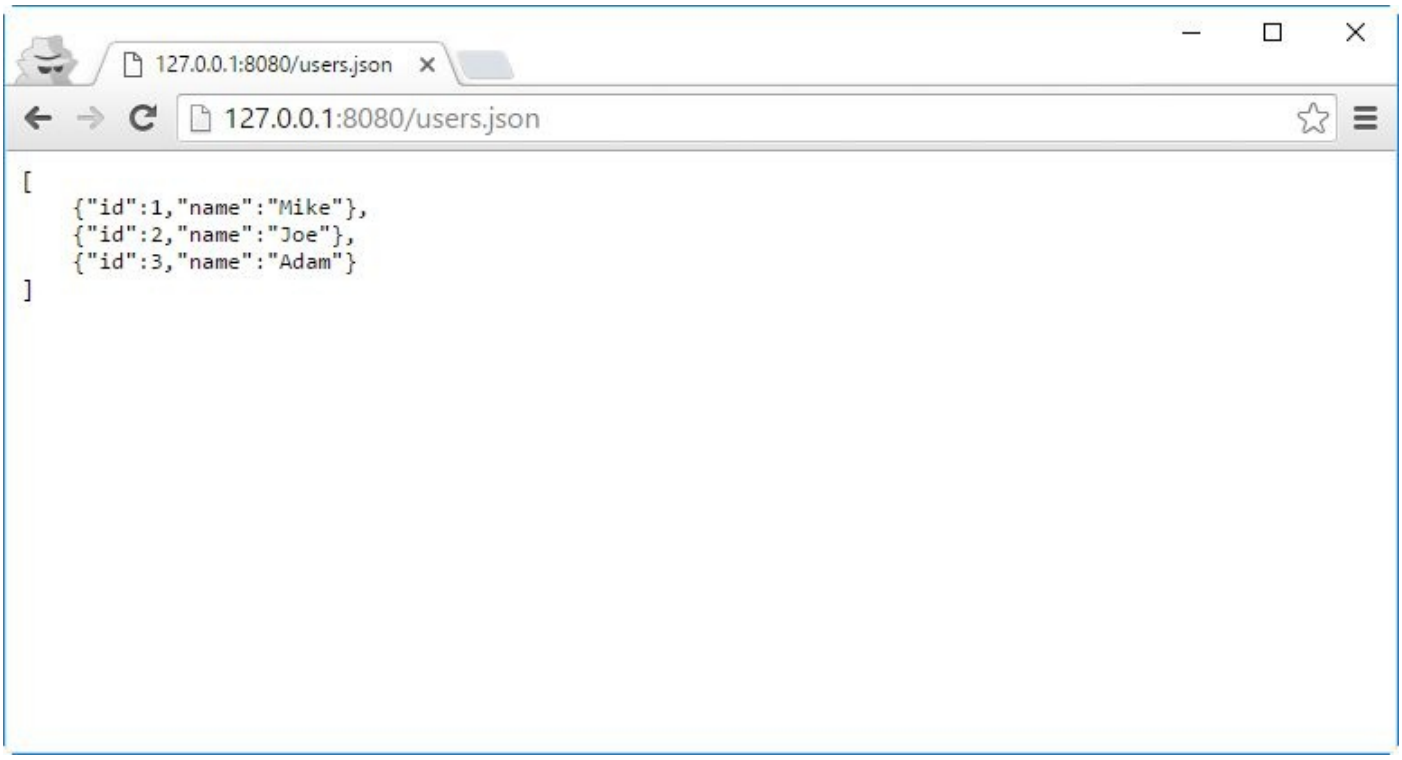
5.1 Creating the project

Copy the `AngularBase` directory as `AngularHttp`. The first test we will do is accessing an URL to get *json* data. For that, create the following file on the project:

users.json

```
[
     {"id":1,"name":"Mike"},
     {"id":2,"name":"Joe"},
     {"id":3,"name":"Adam"}
]
```

Execute the `live-server` and access the following URL: 'http://localhost:8080/users.json'. You should get the following response:



5.2 Using the Http class

To use the `Http` class to access an URL via Ajax it is necessary to execute the following procedures:

- Add the `http.dev.js` file to the script list of the `index.html` file:

```
<!-- 1. Load libraries -->
<script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
<script src="node_modules/rxjs/bundles/Rx.js"></script>
<script src="node_modules/angular2/bundles/angular2.dev.js"></script>
<script src="node_modules/angular2/bundles/http.dev.js"></script>
```

- Add the following *imports* to the component that uses the Http:

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import 'rxjs/add/operator/map'
```

Notice that we also imported the `rxjs/add/operator/map` which is necessary for the last Angular version (beta).

- Add the property `HTTP_PROVIDERS` to the `@Component`, imported previously, that defines may Angular classes that can be injected into the component.

```
@Component({
    ...
    providers: [HTTP_PROVIDERS],
    ...
})
```

As we learned previously, the `providers` property can be replaced on the bootstrap of the application, in the file `boot.ts`, making the `Http` class available to any class of the application.

- Inject the Http class into the component:

```
export class AppComponent {

    constructor(private _http:Http){

    }
}
```

It is even possible to use a class variable, as on the following example:

```
export class AppComponent {
    private _http;
    constructor(_http:Http){
        this._http = _http;
    }
}
```

or just:

```
export class AppComponent {
    constructor(private _http:Http){
        this._http = _http;
    }
}
```

On the first example, we will use the `_http` only on the constructor.

With the class already injected, we will use it to access the server via Ajax. As we used the `live-server`, using the path `./users.json` will access an URL similar to: `http://localhost:8080/users.json`.

To access this address via GET we use:

```
    constructor(_http:Http){
        _http.get("./users.json").................
    }
```

We used the `_http.get(url)` to access the URL, and when the server returns the data, we call two methods with distinct jobs. The first is the `map` method that will format the return text from the server, and the second is the `subscribe` that will attribute the formatted text to a variable.

```
export class AppComponent {
    users:Array<any>;
    constructor(_http:Http){
        _http.get("./users.json")
            .map(res => res.json())
            .subscribe(users => this.users = users);
    }
}
```

In this example, the `map` method formats the `res` response in `json`. This is possible because the server returns a text in json format. The `subscribe` method attributed the response already formatted in json to the variable `this.users`.

It is possible instead of attributing the value to a variable, call a method on the class, for example: `.subscribe(users => this.onGetUsers(users));`.

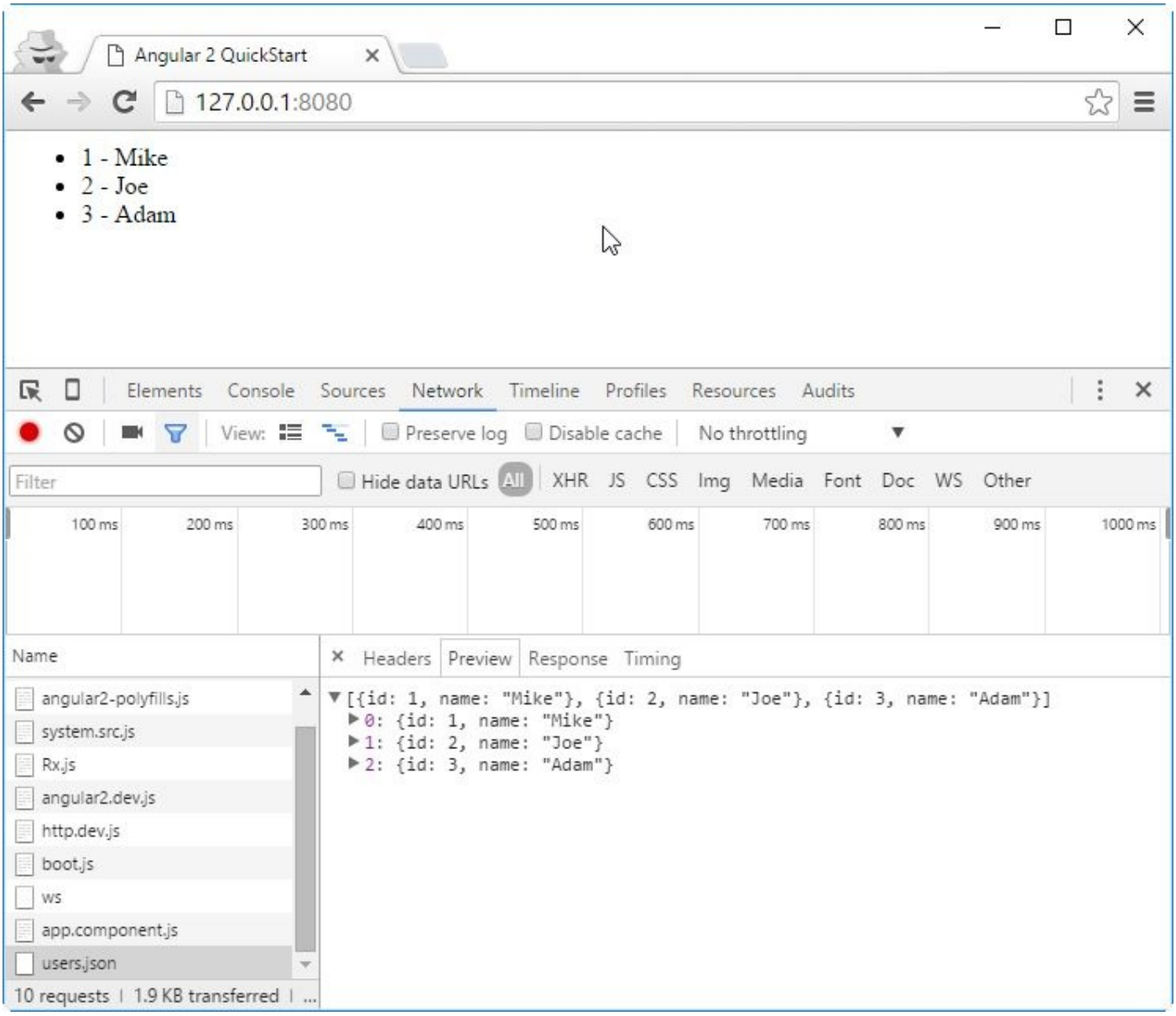
With the `this.users` variable set, we can use the template to show the data. On the following example, we have the `AppComponent` class ready:

```
import {Component} from 'angular2/core'
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import 'rxjs/add/operator/map'

@Component({
    selector: 'my-app',
    providers: [HTTP_PROVIDERS],
    template: `
```

```
      <ul>
            <li *ngFor="#u of users">
                  {{u.id}} - {{u.name}}
            </li>
      </ul>
      `
})
export class AppComponent {
      users:Array<any>;
      constructor(_http:Http){
            _http.get("./users.json")
                  .map(res => res.json())
                  .subscribe(users => this.users = users);
      }
}
```

The news here is the template, that does a `ngFor` loop on the `<li>` from the variable `users`, that is filled by the `Http`. The result of this component is as follows:



5.3 Using services

On the previous example we used the `Http` class withing the component, accessing the server and getting the `users` array. Let's refactor the project to put each functionality in its right place.

A **service** class is responsible to provide data to the application, so it is natural to create the `UserService` class for that purpose.

5.4 Organizing the project

We have three options to better organize the project classes:

1. Create all the application classes inside the `app` directory.
2. Create the `services` directory with the `user.ts` file inside. And create the `services` library inside the `app` directory.
3. Create the `user` directory with the `user.service.ts` file inside. And create the `services` library inside the `app` directory.

The worst option is the first, that should be used only when learning or testing a small project. An Angular 2 project divides itself into many files and store them in only one directory is not a good practice.

The second option is good for small projects, because the `services/classes/components/templates/estyles` are organized in each place. For example:

```
app
 |-components
   |- user.component.ts
   |- product.component.ts
   |- employee.component.ts
 |-
 |-templates
   |- user.html
   |- product.html
   |- employee.html
 |-
 |-services
   |- user.ts
   |- product.ts
   |- employee.ts
 |-
 |-styles
   |- app.css
   |- user.css
   |- product.css
   |- employee.css
 |-services.ts
 |-app.component.ts
 |-boot.ts

   ....
```

Notice that this organizes the files that represent entities of the application are mixed. In other words, to work with `products` you will need to open many files in many directories.

It becomes not effective when there are too many files in the same directory. This option is better than the first, but works only for small projects.

The third option separates the entities of the application in distinct directories like the following:

```
app
 |-user
   |- user.service.ts
   |- user.component.ts
   |- user.template.html
   |- user.style.css
 |-product
   |- product.service.ts
   |- product.component.ts
   |- product.template.html
   |- product.style.css
 |-employee
   |- employee.service.ts
   |- employee.component.ts
   |- employee.template.html
   |- employee.style.css
 |-services.ts
 |-app.component.ts
 |-boot.ts
```

This option separates each entity in a directory, adding related files to that entity. This way when the project becomes big there will e many directories, but each one represents an unique peace of the application.

5.5 User model

Before we create the service class, notice that in the `app.component` the `users` variable is of type `Array<any>`, which means an array of objects of any type. Let's improve this code creating an object to represent the `user`. We will call it User model.

app/user/user.ts

```
export class User{
    constructor(
        public id:number,
        public name:string
    ){}
}
```

In this class we used `public id` inside the `constructor` creating a public variable. In bigger projects it is possible to encapsulate those variables. By creating the `User` model, we can add it to the model library like the following:

app/model.ts

```
export * from './user/user'
```

5.6 User service

We can create the `UserService` class on the following way:

app/user/user.service.ts

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {User} from '../model'

@Injectable()
export class UserService {
     constructor(private http: Http) { }

     public getUsers() {
          return this.http
               .get('./users.json')
               .map(res => res.json());
     }
}
```

In this class we have the `imports` from the beginning of the chapter, but also the `User` import with a different path: `../model`. The use of `../` goes one directory level up to find the `model.ts` class.

We used `@Injectable()` to set up the dependency injection into the class. I makes possible to inject the `Http` class into the `UserService` class. When we created the constructor, we inject the instance of the `Http` class into the `http` variable, that can be used in other method of the class. To be able to inject the `Http` class, we must use the `providers` metadata or change the `bootstrap` of the application. As the `Http` is a class that will be injected into other classes, let's add it into the `bootstrap` of the application:

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Http, HTTP_PROVIDERS} from 'angular2/http'


bootstrap(AppComponent,[HTTP_PROVIDERS]);
```

Back to the `UserService` class we have the `getUsers()` method:

```
public getUsers() {
     return this.http
          .get('./users.json')
          .map(res => res.json());
}
```

This method uses the `Http` class to make the `GET` request to the `./users.json` url. And on the server response, the `map` method is called, where the `.json()` method formats the response text into JSON.

Notice that we are returning the `this.http.get(...).map(...)` to who called the `getUsers()` method. The code that calls `getUsers()` that is responsible to manage the return. This is realized on the `AppComponent`.

To create a services library, create the following file:

app/service.ts

```
export * from './user/user.service'
```

5.7 Changing the AppComponent component

Now the `AppComponent` will use the `UserService` class on the following way:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {User} from './model'
import {UserService} from './service'

@Component({
     selector: 'my-app',
     providers: [UserService],
     template: `
     <ul>
          <li *ngFor="#u of users">
               {{u.id}} - {{u.name}}
          </li>
     </ul>
     `
})
export class AppComponent {
     public users:Array<User>;
     constructor(userService:UserService){
           userService.getUsers()
               .subscribe(users => this.users = users);
     }
 }
```

In the `AppComponent` we imported the `UserService` class and used it as a *provider* in `@Component`. This means that we can use the dependency injection inside the constructor. The constructor will call the method `userService.getUsers()` that will return an instance of the `Observable` class, in which we can use the `subscribe` method to fill the value of the variable `this.users`.

5.8 Submitting data

To submit data from the client application to the server we will do a POST request:

```
@Injectable()
export class UserService {
     ...
     public addUser(u:User){
          return this.http
               .post('./addUser',JSON.stringify(u))
               .map(res => res.json());
     }
     ...
}
```

Instead of the `http.get` we used `http.post` and in the second parameter of `.post` we passed the data that should be sent to the server. As we are still not dealing with data on

the server, we will see more examples of data changes on the next chapters.

6. Routes

The definition of routes in Angular 2 follows the concept of splitting the code in smaller parts that can be loaded via Ajax by the framework itself.

6.1 Applying AngularRoutes

Copy the `AngularBase` project as `AngularRoutes`. First, add the `router.dev.js` file to the `index.html`:

AngularRoutes/index.html

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js">
    </script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script src="node_modules/angular2/bundles/router.dev.js"></script>
    <base href="/">

    <!-- 2. Configure SystemJS -->
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('app/boot')
            .then(null, console.error.bind(console));
    </script>

  </head>

  <!-- 3. Display the application -->
  <body>
    <my-app>Loading…</my-app>
  </body>

</html>
```

Notice that there are files with with the extension `.dev.js` and `.min.js`. When the application is still under development we use the `.dev` so that programming errors are easily found. When the application is already in production we use the `.min` to have a better performance.

Important: Include also `<base href="/">` inside the `<head>` tag of the HTML document.

After including the Router it is necessary to add the `ROUTER_PROVIDERS` variable into the application `bootstrap`:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {ROUTER_PROVIDERS} from 'angular2/router'

bootstrap(AppComponent,[ROUTER_PROVIDERS]);
```

With those two settings the Router is enabled:

6.2 Splitting the application in smaller parts

An application may be divided into many parts, for example: home, login and dashboard. Each part has its own component and template, that are set up via the Router.

First create the following components:

- HomeComponent
- LoginComponent
- DashBoardComponent

```
import {Component} from 'angular2/core';
@Component({
     templateUrl: 'app/home/home.html'
})
export class HomeComponent { }

import {Component} from 'angular2/core';
@Component({
     templateUrl: 'app/login/login.html'
})
export class LoginComponent { }

import {Component} from 'angular2/core';
@Component({
     templateUrl: 'app/dashboard/dashboard.html'
})
export class Dashboard { }
```

Create the templates for each component only with the name of the components.

6.3 Creating the area where the components will be created

It is necessary to define an area for the components to be loaded. This is done with the `<router-outlet>` tag:

app/app.component.ts

```
import {Component} from 'angular2/core';
import {ROUTER_DIRECTIVES} from 'angular2/router'

@Component({
     selector: 'my-app',
     directives: [ROUTER_DIRECTIVES],
     template: `
               <h1>My First Angular 2 App</h1>
               <div><router-outlet></router-outlet></div>
               `
})
export class AppComponent { }
```

6.4 Setting up the Router

In the `app.component` component we will set up the routing with the `@RouteConfig` metadata. Many settings should be done in the `app.component`:

```
import {Component} from 'angular2/core';
import {ROUTER_DIRECTIVES, RouteConfig} from 'angular2/router'
import {HomeComponent} from './home/home.component'
import {DashboardComponent} from './dashboard/dashboard.component'
import {LoginComponent} from './login/login.component'

@Component({
     selector: 'my-app',
     directives: [ROUTER_DIRECTIVES],
     template: `
               <h1>My First Angular 2 App</h1>
               <div><router-outlet></router-outlet></div>
               `
})
@RouteConfig([
     { path: '/home', name: 'Home', component: HomeComponent, useAsDefault: t\
rue },
     { path: '/login', name: 'Login', component: LoginComponent },
     { path: '/dashboard', name: 'Dashboard', component: DashboardComponent }
])
export class AppComponent { }
```

The *imports* must include `ROUTER_DIRECTIVES` and `RouteConfig`, besides the already created components: `HomeComponent`, `DashboardComponent` and `LoginComponent`.

In `@RouteConfig` we define routes, where each one has at least three parameters:

- path: The URL path.
- name: The name of the route.
- component: The component to be loaded.

There is also the `useAsDefault:true` property that defines the default route.

To test the routes, we will create a small menu that points to the three already created components.

6.5 Creating route links

Let's create a small menu that has the links to the routes. At first, this menu will be on the `AppComponent` template:

app/app.component.ts

```
@Component({
     selector: 'my-app',
     directives: [ROUTER_DIRECTIVES],
     template: `
               <h1>My First Angular 2 App</h1>
               <ul>
                    <li><a [routerLink]="['/Home']">Home</a></li>
                    <li><a [routerLink]="['/Login']">Login</a></li>
                    <li><a [routerLink]="['/Dashboard']">Dashboard</a></li>
               </ul>
               <div><router-outlet></router-outlet></div>
               `
})
```

We used `[routerLink]` to define a link to a route with the name of the route.

6.6 Passing parameters

It is possible to pass parameters between routes using the `[routerLink]` on the following way:

```
<a [routerLink]="['User', {id:1}]">Princess Crisis</a>
```

Or with TypeScript (with a router injected in the class):

```
this._router.navigate( ['User', { id: user.id }] );
```

To get the passed parameter it is necessary to use the RouteParams class:

```
import {RouteParams, Router} from 'angular2/router';

@Component({.....})
export class SomeClass{
     constructor(
     private _router: Router,
     private _routeParams: RouteParams
     ) { }
     ngOnInit() {
          var id = this._routeParams.get('id');
     }
}
```

7. Final example - Server

After reviewing all the relevant concepts of Angular 2, we will create a working example of how to integrate Angular 2 to a server API. The goal of this application is to create a simple blog with `Posts` and `Users` and a login interface.

7.1 Creating the RESTful server

We will use a 100% Node.js project with the following technologies:

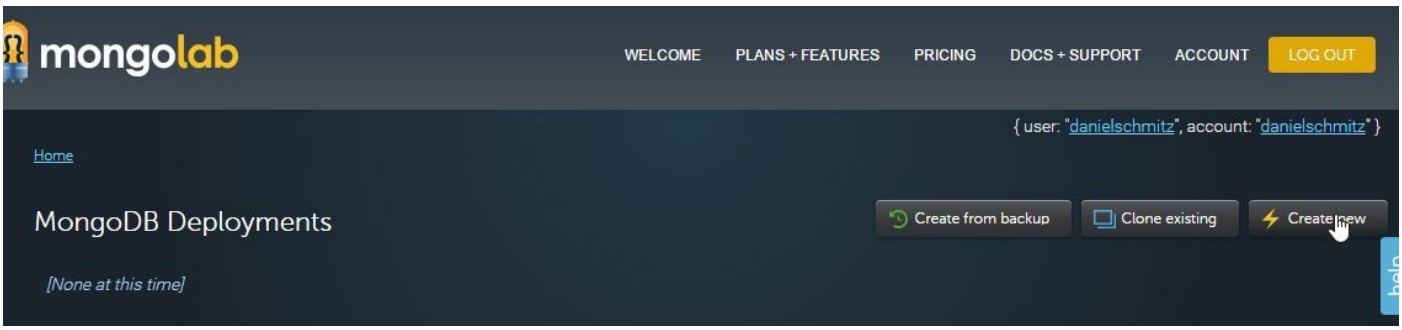
- **express**: It is the web server responsible responsible for the API. We will not use the `live-server` but the **express** has the auto-loading feature with the **nodemon** library.
- **body-parser**: It is a library for the JSON data POST requests.
- **mongoose**: It is an adapter for the MongoDB data base that is a NoSql data base with features almost as good as a relational data base.
- **jsonwebtoken**: It is a node library used for authentication via web token. We will use it for the user login.

All those technologies can be installed with `npm` as follows.

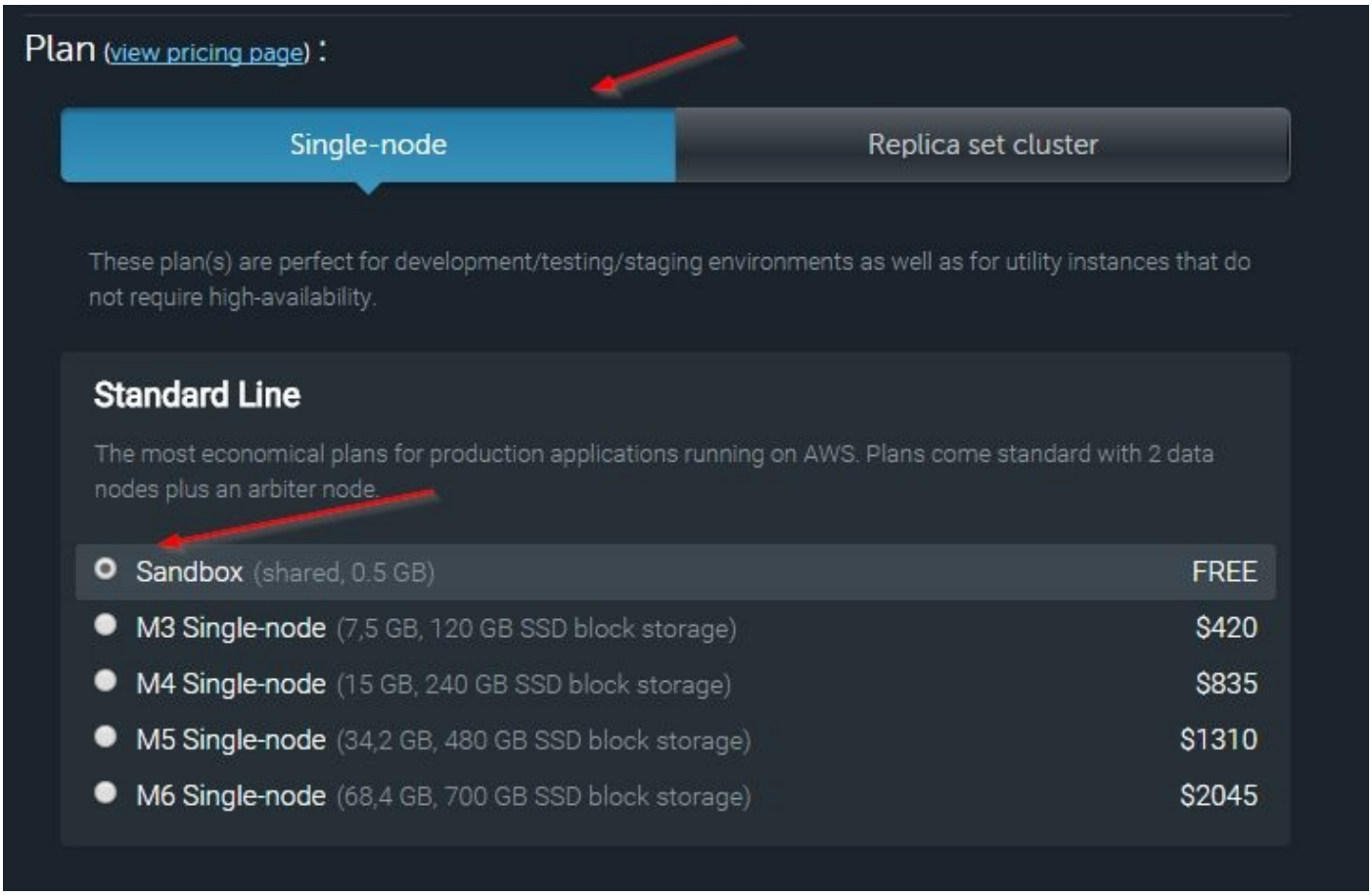
7.2 The MongoDB data base

The MongoDB data base has a premise different from relational data bases (SQL data bases), storing data in JSON format. It can be installed in your computer and used, but we will use the https://mongolab.com/ service that has a public account for public data bases (for testing).

Access the link https://mongolab.com/welcome/ and sign up. After logging in on the administration screen, create a new instance as the following image:

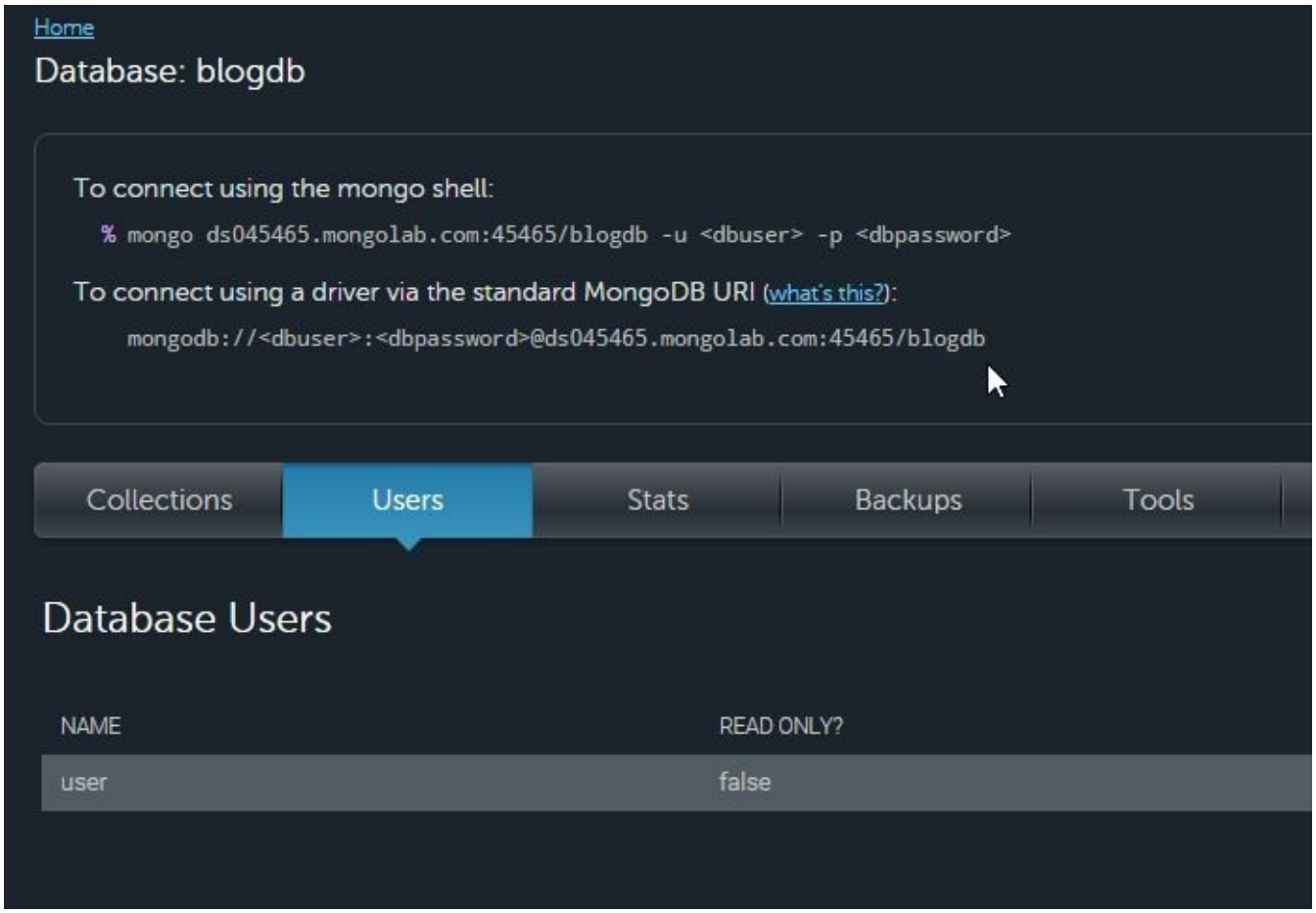


On the next screen choose the `Single-node` tab and the `Sandbox` free plan:

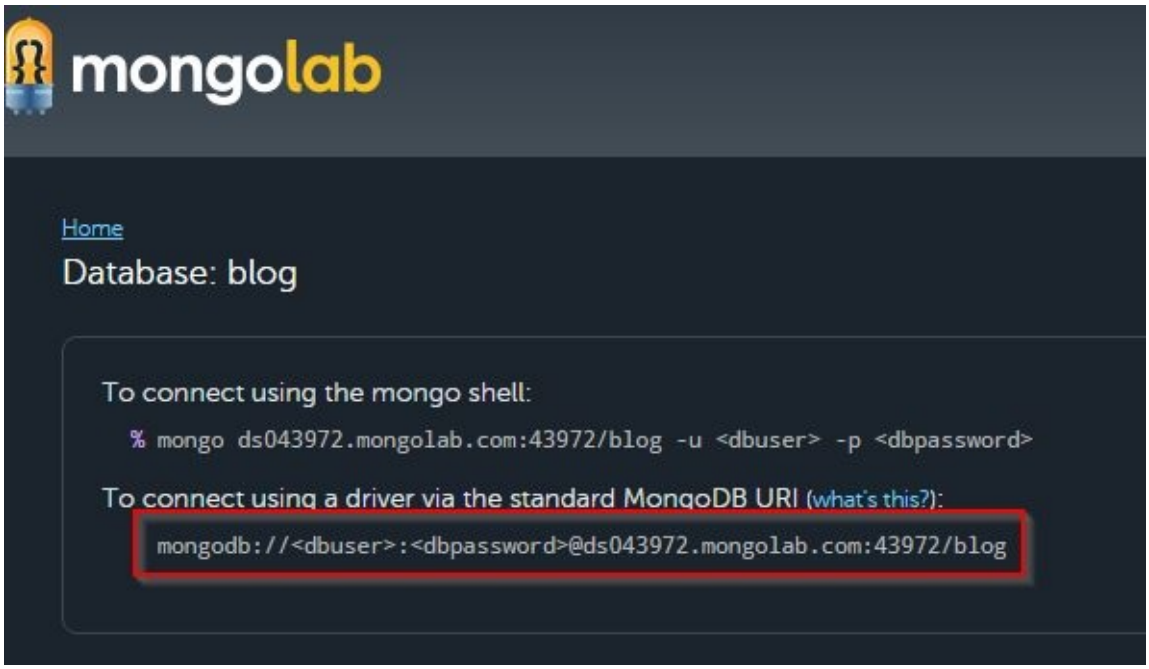


Choose a name, line `blog` and click on `Create new MongoDB deployment`. On the next screen, access the data base and check if you get the following message: “A database user is required….”

Click on the link and add any user (login and password) to access the data base:



After creating the user we will use the connection URI as indicated on your administration screen:



7.3 Creating the project

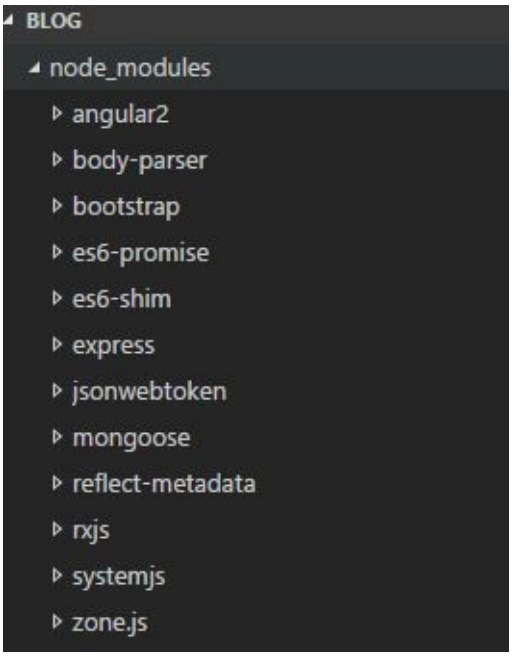
Create the `blog` directory and start the project settings file:

```
\blog $ npm init
```

After inserting the information about the project, the `package.json` file is created. Let's use `npm` to install all the necessary libraries:

```
$ npm i express body-parser jsonwebtoken mongoose angular2 systemjs bootstra\
p --S
```

After installing all the libraries, the `node_modules` directory should have the following projects:



7.4 The project structure

The project structure will rely on the following architecture:

```
blog
|-.vscode (Visual Studio Code settings)
|-node_modules
|-app   (Aplication - TypeScript files)
|-model (Model files of the data base)
|-public (Public directory with the static files of the project)
    |-- index.html (Main HTML page)
|-api (Virtual directory with the application API)
|-package.json
|-tsconfig.json (TypeScript conpilation settings)
|-server.js (Web Express server)
```

The `app` directory will contain the Angular 2 TypeScript code, that will be compiled to the `public` directory. This means that the files with the `.ts` extension will be in the `app` directory and the `.js` and `.js.map` in the `public` directory.

7.5 Setting up the MondoDB models

The `server.js` file has every thing the application needs to work as a web application. Each step will be explained, but first let's work on the files that represent the MongoDB model.

/model/user.js

```
var mongoose      = require('mongoose');
var Schema        = mongoose.Schema;

var userSchema = new Schema({
  name:  String,
  login: String,
  password:   String
});

module.exports = mongoose.model('User', userSchema);
```

The `User` model is created with the `mongoose` library help. With the `Schema` we created a model (similar to a data base table in other data bases) called `User`, that has the `name`, `login` and `password` fields.

And here is the `Post` model:

/model/post.js

```
var mongoose      = require('mongoose');
var Schema        = mongoose.Schema;

var postSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  user: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
  date: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Post', postSchema);
```

The `Post` model is very similar to the `User`, except for the relationship between `Post` and `User`, where `Post` has a reference to `User`.

7.6 Setting up the Express server

Create the `server.js` file to be able to add the necessary code for the `api`. Let's see step-by-step how the server is created:

server.js

```
1 var express = require('express');
2 var app = express();
3 var bodyParser = require('body-parser');
4 var jwt = require('jsonwebtoken');
```

First we added the libraries to be used and the `app` variable to hold the instance of the *express* server.

server.js

```
5 //secret key (use any big text)
6 var secretKey = "MySuperSecretKey";
```

At line 6 we created a variable called `secretKey` to be used with the `jsonwebtoken` module, to be able to generate an access token for the user. In a production server you must change this key to any other text.

server.js

```
 7 //Database in the cloud
 8 var mongoose = require('mongoose');
 9 mongoose.connect('mongodb://USER:PASSWOD@___URL___/blog', function (err) {
10     if (err) { console.error("error! " + err) }
11 });
```

We imported the `mongoose` library and used the `connect` command to connect to the data base from the `mongolab` service. Remember to change the connection address with the one you created before.

server.js

```
12 //bodyparser to read json post data
13 app.use(bodyParser.urlencoded({ extended: true }));
14 app.use(bodyParser.json());
```

At line 13 and 14 we set up the `bodyParser` that will get the JSON request and format it to be used on the application.

server.js

```
15 //Load mongodb model schema
16 var Post = require('./model/post');
17 var User = require('./model/user');
```

At line 16 and 17 we loaded the `models` previously created into variables.

server.js

```
15 //create a REST ROUTER
16 var router = express.Router();
```

The 16th line creates the `router` which prepares the `express` to behave as an API. A router is responsible to get requests and execute a code depending of the format of the request. Usually we have four types of requests:

- GET: Used to get data. Can be used via the browser with an URL.
- POST: Used to insert data, usually from a form.
- DELETE: Used to delete data.
- PUT: Can be used to edit data. We will not use PUT in this project, but you can use it if desired.

Besides the type of request we also have the `url` and the parameters that we will see further up.

server.js

```
17 //Static files
18 app.use('/', express.static('public'));
```

```
19 app.use('/libs', express.static('node_modules/bootstrap/dist'));
20 app.use('/libs', express.static('node_modules/systemjs/dist'));
21 app.use('/libs', express.static('node_modules/rxjs/bundles/'));
22 app.use('/libs', express.static('node_modules/angular2/bundles'));
```

At line 18 we set up the `public` directory as static, which means all the content of this directory will be treated as a single file. This concept is similar to the `webroot` directory from other web servers. See that in `express` the directories that don't belong to `express.static` won't be able to be accessed by the web browser. This makes the server more safe hiding the files that should be executed only on the server, as the `model/user.js`, for example.

From line 19 to 22 we set up the `/libs` static directory that has the JavaScript libraries from the `node_modules` directory. We added the Bootstrap, SystemJS, rxjs e Angular 2 libraries that will be added later to the `public/index.html` file.

server.js

```
23 //middleware: run in all requests
24 router.use(function (req, res, next) {
25     console.warn(req.method + " " + req.url +
26                         " with " + JSON.stringify(req.body));
27     next();
28 });
```

At line 24 we created the `middleware` which is a peace of code that will be executed in every request the server receives. The `console.warn` sends a notification to the console, showing the type of method, the URL and the JSON parameters. This information should be used only in development environment. The text produced by this line should be similar to the following:

```
POST /login with {"login":"foo","password":"bar"}
```

The method `JSON.stringify` gets a JSON object and returns it to the text format.

At line 27 the `next()` method was used so the request continues its processing.

server.js

```
29 //middleware: auth
30 var auth = function (req, res, next) {
31     var token = req.body.token || req.query.token
32                           || req.headers['x-access-token'];
33     if (token) {
34          jwt.verify(token, secretKey, function (err, decoded) {
35               if (err) {
36                    return res.status(403).send({
37                                         success: false,
38                                         message: 'Access denied'
39                                    });
40               } else {
41                    req.decoded = decoded;
42                    next();
43               }
44          });
45     }
```

```
46      else {
47           return res.status(403).send({
48                success: false,
49                message: 'Access denied'
50           });
51      }
52 }
```

At line 30 we have the `auth middleware` to verify if the request token is valid. When the user logs in it gets a token that will be used in every request.

At line 31 the `token` variable gets the content of the token from the client. In this case we will use the http header with the `x-access-token` variable.

At lnie 34 the `jwt.verify` method is used to verify the token. Notice that the `secredKey` is used there and that the third parameter is a `callback function`.

At line 35 we check if the `callback` function got an error, which means that the token is not valid and returns the 403 *status* code `res.status(403).send()`, that means access denied.

At line 40 if no error is found the decoded object is stored in the `req.decoded` variable and the `next()` assures that the request will continue its way.

At line 46 if no token was sent by the client the 403 status code is sent back.

server.js

```
53 //simple GET / test
54 router.get('/', function (req, res) {
55      res.json({ message: 'hello world!' });
56 });
```

At line 54 we have an example of how the express router works. With the `router.get` method we set up the url “/” that when called will execute the `callback` on the second parameter. This `callback` sets up the router response with the `res.json` method returning the JSON object `{ message: 'hello world!' }`.

server.js

```
56 router.route('/users')
57      .get(auth, function (req, res) {
58           User.find(function (err, users) {
59                if (err)
60                     res.send(err);
61                res.json(users);
62           });
63      })
64      .post(function (req, res) {
65           var user = new User();
66           user.name = req.body.name;
67           user.login = req.body.login;
68           user.password = req.body.password;
69
70           user.save(function (err) {
71                if (err)
72                     res.send(err);
```

```
73                res.json(user);
74           })
75      });
```

At line 56 we started to set up the user routing that will, at first, be accessed via the URL `/users`. The GET request to this URL has the `auth` method *middleware* to check the validity of the token before executing the GET callback. If valid is valid the `callback` is executed and an array with the users is sent back to the client.

At line 64 we set up the `POST /users` method to store an User. Notice that here it is not necessary to be authenticated with a token. At lina 65 We used the properties of the User Schema to save the registry with the client data from `req.body` variable which is filled thanks to the body-parser.

The `user.save` method saves the registry into the daba base and uses `res.json` to return the `user` object to the client.

server.js

```
76 router.route('/login').post(function (req, res) {
77      if (req.body.isNew) {
78           User.findOne({ login: req.body.login }, 'name')
79                .exec(function (err, user) {
80                     if (err) res.send(err);
81                     if (user != null) {
82                          res.status(400).send('Login OK');
83                     }
84                     else {
85                          var newUser = new User();
86                          newUser.name = req.body.name;
87                          newUser.login = req.body.login;
88                          newUser.password = req.body.password;
89                          newUser.save(function (err) {
90                               if (err) res.send(err);
91                               var token = jwt.sign(newUser, secretKey, {
92                                    expiresIn: "1 day"
93                               });
94                               res.json({ user: newUser, token: token });
95                          });
96                     }
97                });
98      } else {
99           User.findOne({ login: req.body.login,
100                     password: req.body.password }, 'name')
101                .exec(function (err, user) {
102                     if (err) res.send(err);
103                     if (user != null) {
104                          var token = jwt.sign(user, secretKey, {
105                               expiresIn: "1 day"
106                          });
107                          res.json({ user: user, token: token });
108                     }else{
109                          res.status(400).send('Wrong Login/`Password');
110                     }
111                });
112     }
113 });
```

Here we have the code for the Login via the URL `/login`.

At line 77 with the property `isNew` we check if the user is trying to log in or sign in.

At line 78 the `findOne` method checks if a user already exists based on the `{login:req.body.login}` filter and if positive, returns the `name` field. The `.exec` method will execute the `findOne` and the callback will be called, where we can return an error, since it is not possible to register the same login more than once.

If `req.body.isNew` is false, the code searches for the login and password on the data base and, if correct, the `jwt.sign` method creates the authentication token for the user. If incorrect the status code 400 is returned.

server.js

```
114 router.route('/posts/:post_id?')
115      .get(function (req, res) {
116           Post
117                .find()
118                .sort([['date', 'descending']])
119                .populate('user', 'name')
120                .exec(function (err, posts) {
121                     if (err)
122                          res.send(err);
123                     res.json(posts);
124                });
125      })
126      .post(auth, function (req, res) {
127           var post = new Post();
128           post.title = req.body.title;
129           post.text = req.body.text;
130           post.user = req.body.user._id;
131           if (post.title==null)
132                res.status(400).send('TÃtulo nÃ£o pode ser nulo');
133           post.save(function (err) {
134                if (err)
135                     res.send(err);
136                res.json(post);
137           });
138      })
139      .delete(auth, function (req, res) {
140          Post.remove({
141                _id: req.params.post_id
142           }, function(err, post) {
143                if (err)
144                     res.send(err);
145                res.json({ message: 'Successfully deleted' });
146           });
147      });
```

At line 114, to get a post from the server we use the URL `/posts/:post_id?` where `post_id` is a variable with the post ID, for example `/posts/5`. The `?` question mark makes the variable optional.

At line 115 we are using the method `GET /posts` to get all the posts from the data base. The `sort` method sorts the posts in the array and the `populate` method adds a reference to the `user` model for the post author.

At line 126 the `POST /posts` method adds a new post checking if it is valid and persisting it with `Post.save()` to the data base.

At line 139 the `DELETE /posts/` deletes a post from the data base with the method `Post.remove` and the post ID from `req.params.post_id`.

server.js

```
148 //register router
149 app.use('/api', router);
150 //start server
151 var port = process.env.PORT || 8080;
152 app.listen(port);
153 console.log('Listen: ' + port);
```

Finally we point the `router` variable to the `/api` URL. That way all the API will be exposed in the `/api` URL. For example, to get all the posts from the data base one can make a GET request to the address `/api/posts`. We also defined in which ports the server express will be listening.

7.7 Testing the server

To test the Web Server we can just execute the following command:

```
$ node server.js
```

It should return `Listen: 8080`. If the `server.js` file is changed it won't reflect on the running server and it will be necessary to restart the server for it to be applied. To avoid it, let's install the `nodemon` library to load the server every time the `server.js` file is edited.

```
$ npm install nodemon -g
```

After the installation, run:

```
$ nodemon server.js

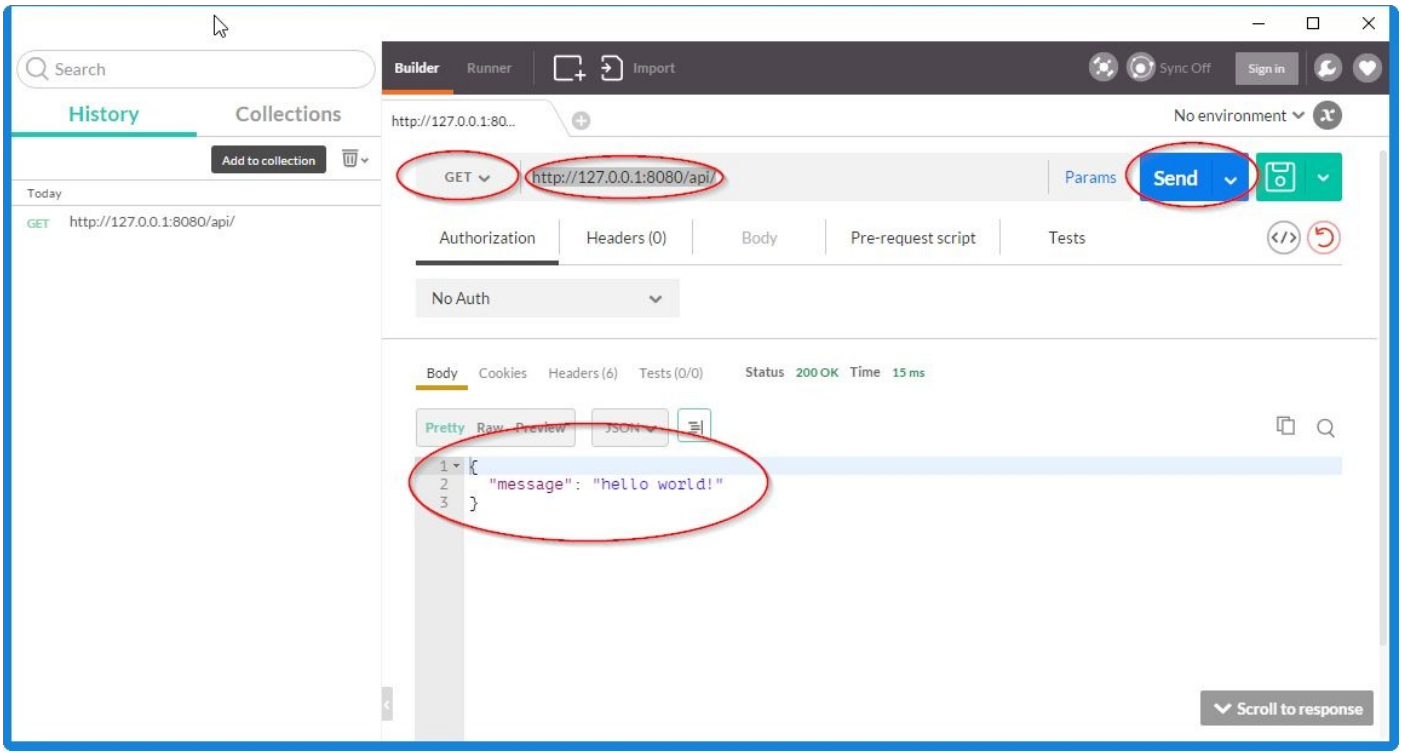
[nodemon] 1.8.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node server.js`
Listen: 8080
```

The response already explains that every time the `server.js` file is updated the `node server.js` will also be executed.

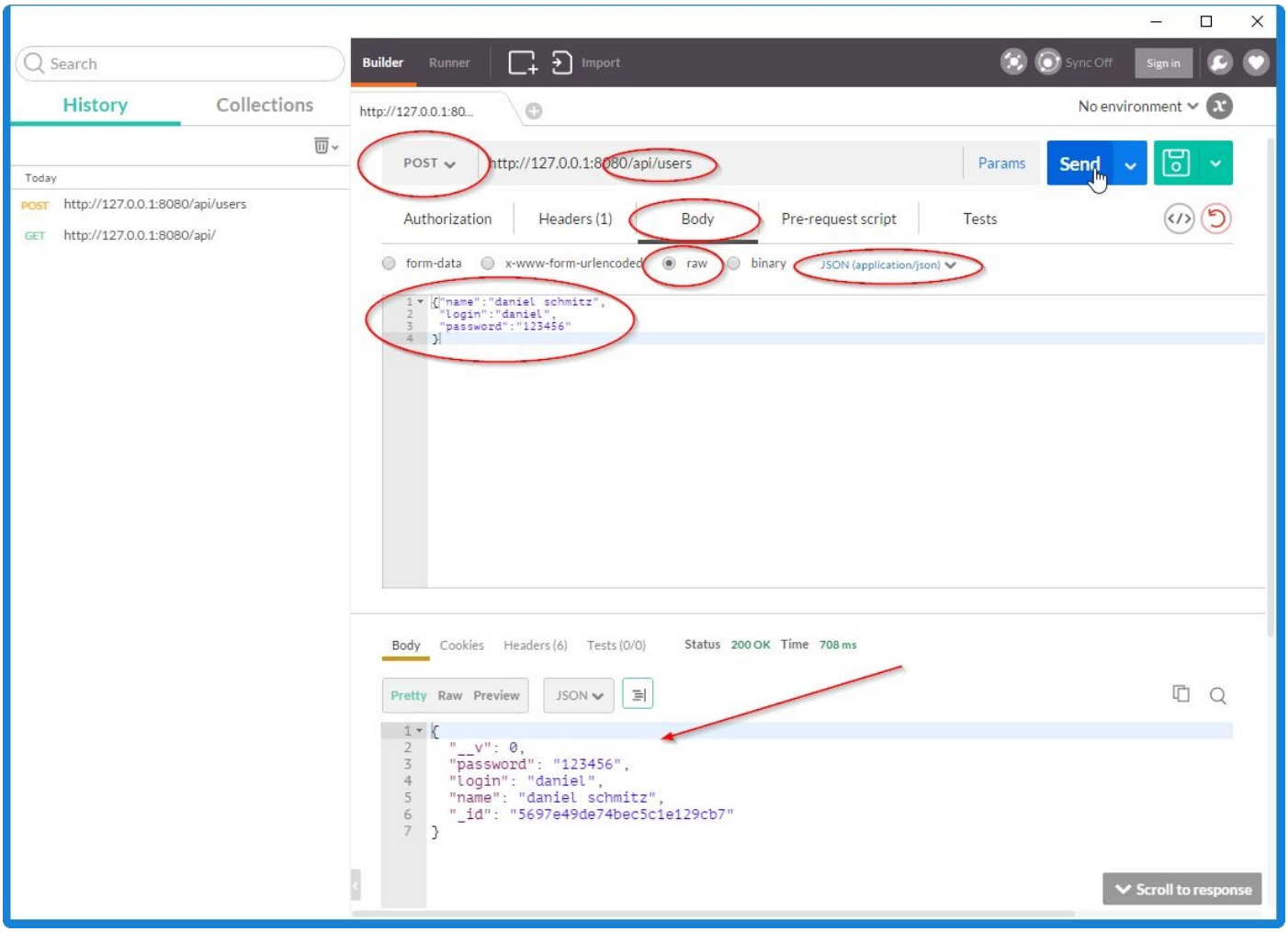
7.8 Testing the API without Angular

It is possible to test the API that we just created sending and receiving data with a program capable of requesting GET/POST calls to the server. One of this programs is **Postman**, that can be installed as a plugin for Google Chrome.

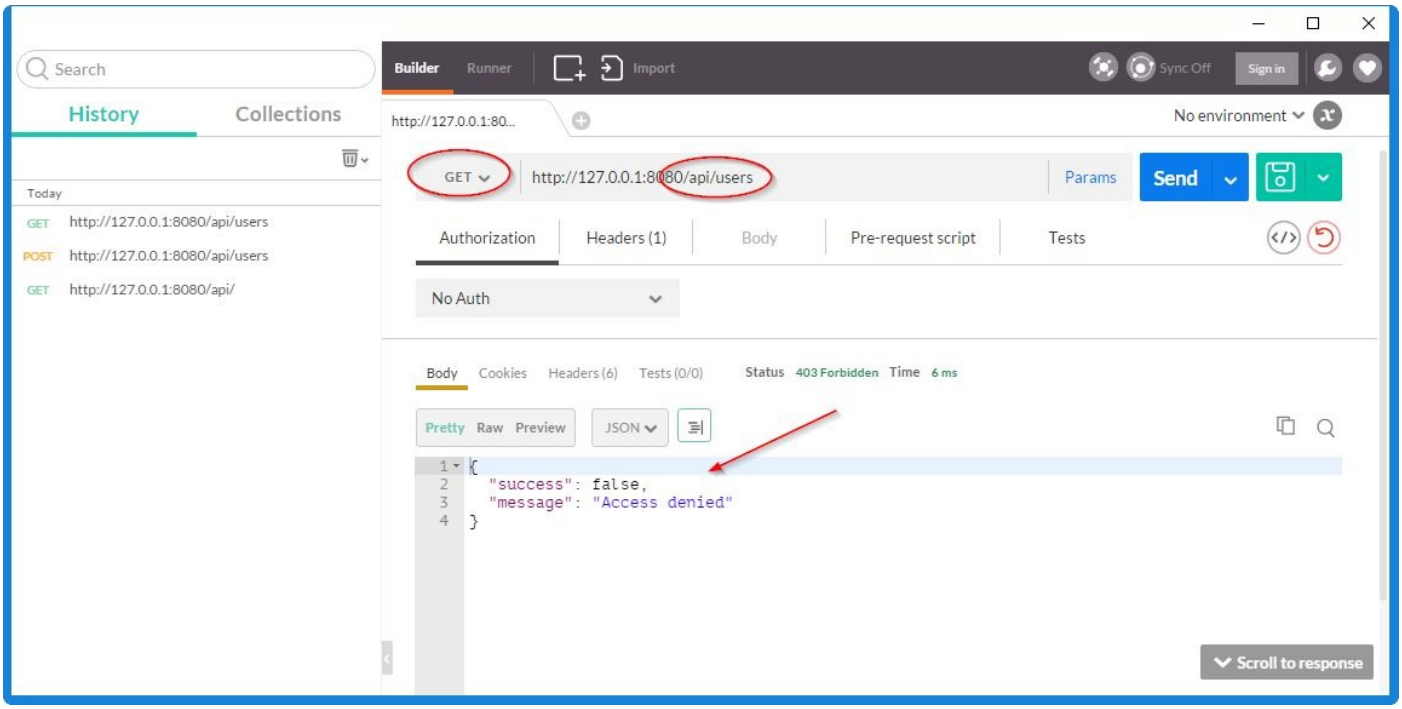
For example, to test the address `http://127.0.0.1:8080/api/`, we set up the Postman like the following:



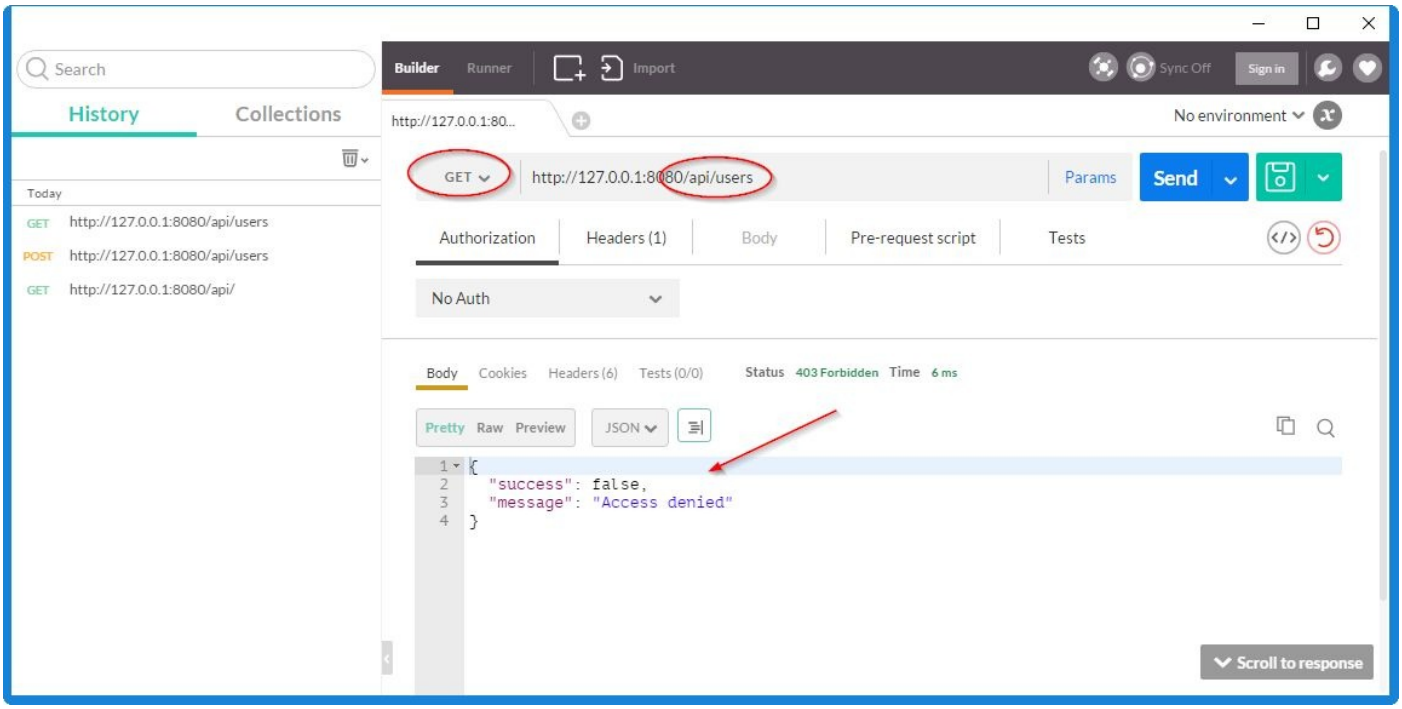
Notice that we got the `hello world` response as set on the server. To create a user, we can do a POST to the `/users` URL with the following data:



To test the login, try to access the URL `/api/users`. Since it will go through the `auth middleware`, the following error will be returned:

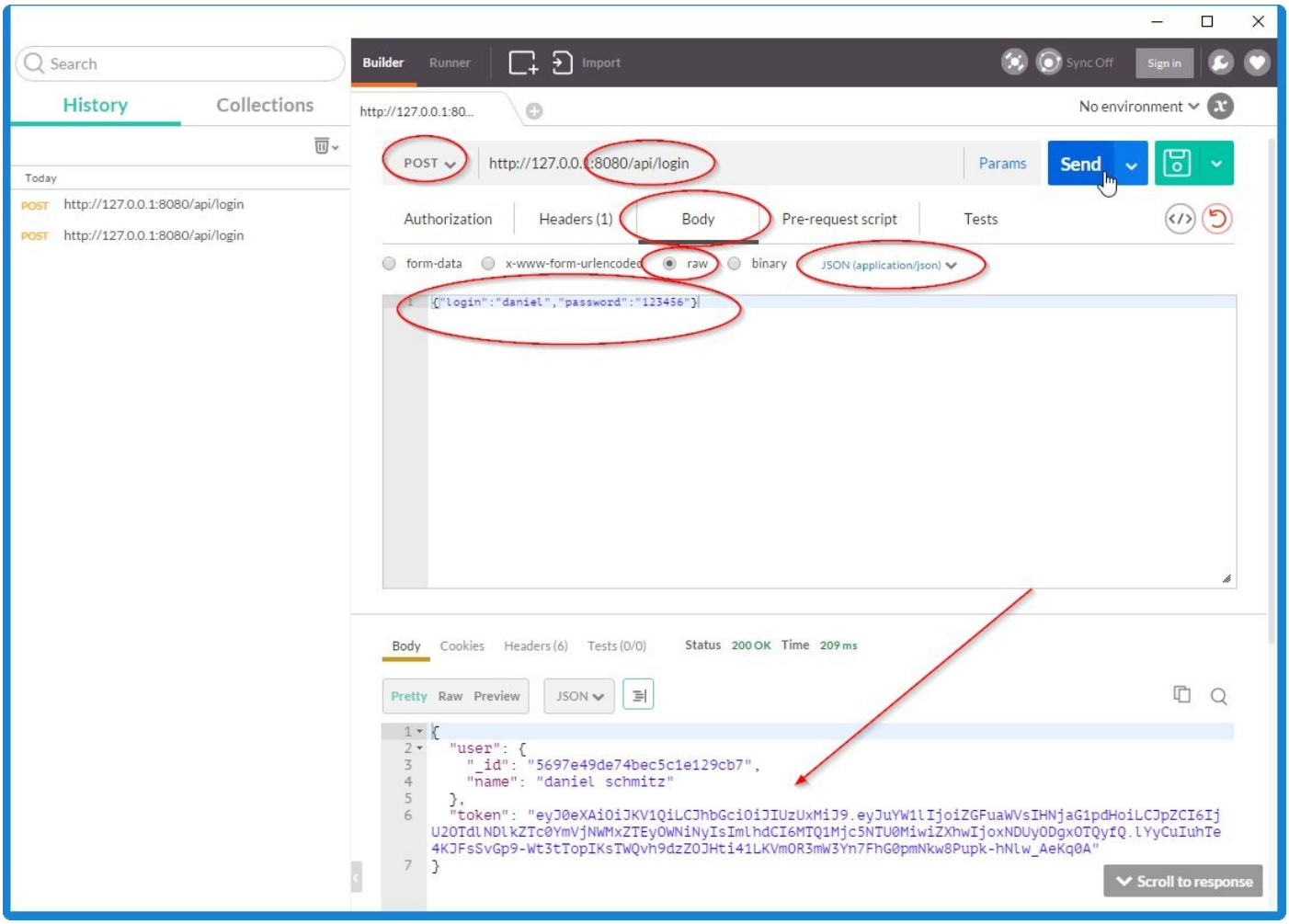


To log in we access the URL on the following way:



Notice that by sending `login` and `password` the token is generated and returned to the *postman*. Copy and paste this token some where so we can use it on the next requests. In Angular, this token will be stored in a variable.

With the token it is possible to request the `GET /users` passing the token within the HTTP request header as on the following image:



Notice that with the token, the user data is returned. If you change the token, you will get the `Failed to authenticate` error. Try it.

8. Final Example - Client application

With the server ready, we can start our Angular 2 structure. This time instead of copying from `AngularBase` let's create each step of the project.

On the `express` server, we set the static JavaScript files like the following:

```
...
app.use('/',express.static('public'));
app.use('/libs',express.static('node_modules/bootstrap/dist'));
app.use('/libs',express.static('node_modules/systemjs/dist'));
app.use('/libs',express.static('node_modules/rxjs/bundles/'));
app.use('/libs',express.static('node_modules/angular2/bundles'));
...
```

8.1 First files

Create the `public/index.html` file initially with the following code:go:

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
     <meta charset="UTF-8">
     <title>Blog</title>
     <meta name="viewport" content="width=device-width, initial-scale=1">
     <base href="/"> <!-- <<<< IMPORTANT --->

     <!-- LIBS -->
     <script src="libs/angular2-polyfills.js"></script>
     <script src="libs/system.src.js"></script>
     <script src="libs/Rx.js"></script>
     <script src="libs/angular2.dev.js"></script>
     <script src="libs/http.dev.js"></script>
     <script src="libs/router.dev.js"></script>
     <link rel="stylesheet" href="libs/css/bootstrap.min.css">

     <!-- BOOT -->
     <script>
       System.config({
         packages: {
           app: {
             format: 'register',
             defaultExtension: 'js'
           }
         }
       });
       System.import('app/boot')
             .then(null, console.error.bind(console));
     </script>
</head>
<body>
     <my-app>Loading…</my-app>
```

```
</body>
</html>
```

The HTML document that has the initial settings of the blog application has a few differences from the previous ones. First, we are adding the libraries using the `/libs` path, that fiscally doesn't exist, but was created virtually on the `express` server. This is important to avoid exposing all the `node_modules` directory to public access.

After setting up the libraries it's time to set up the `System.config` file, which is identical to the previous examples but for a very important detail: As the `index.html` file is in the `public` directory, by executing the `System.import('app/boot')` we are in fact executing the `public/app/boot.js` file that, at first, still doesn't exist. Notice that we can't set the `import` with the `../app/boot` path to access the `blog/app` directory because it is not accessible. This means that in this project, we will have the TypeScript files in the `/blog/app` directory and the `js` and `js.map` files in the `blog/public/app` directory. For this, it is necessary to set up the `tsconfig.json` file like the following:

blog/tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "watch":true,
    "outDir": "./public/app"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Notice that the file is very similar to the previous ones, but with the attribute `outDir` where we indicate a directory where the compiled files should be created. In this context we use the `./public/app` path.

The `app/boot.ts` and `app/component.ts` files can be created, at first, with the same code from the `AngularBase` application:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Http, HTTP_PROVIDERS} from 'angular2/http'

bootstrap(AppComponent,[HTTP_PROVIDERS]);
```

Notice that the `bootstrap` already loads the `HTTP_PROVIDERS`, to be able to inject `Http` into the services.

The AppComponent class can be, at first, like this:

```
import {Component} from 'angular2/core';

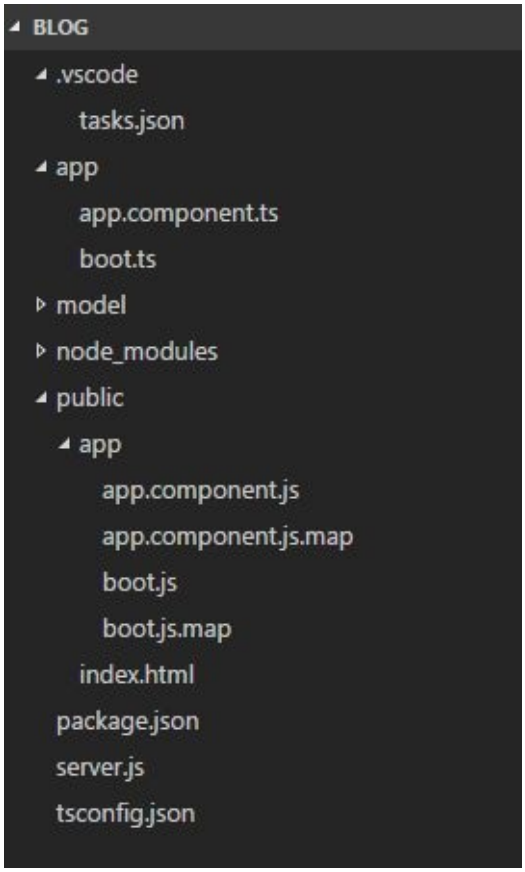
@Component({
     selector: 'my-app',
     template: '<h1>My Blog</h1>'
})
export class AppComponent { }
```

Before compiling the application, if you are using `Visual Studio Code`, set up the `.vscode/tasks.json` file (it will be created the first time you press `ctrl+shift+b`) with the following code:

.vscode/tasks.json

```
{
          "version": "0.1.0",
          "command": "tsc",
          "isShellCommand": true,
          "showOutput": "silent",
          "problemMatcher": "$tsc"
}
```

Compiling the application, pressing `ctrl+shift+b` in `Visual Studio Code` or executing `tsc` in the command line (on the `blog` directory), the project is compiled and the `js` and `map` files are created in `public/app`. This is our directory structure:

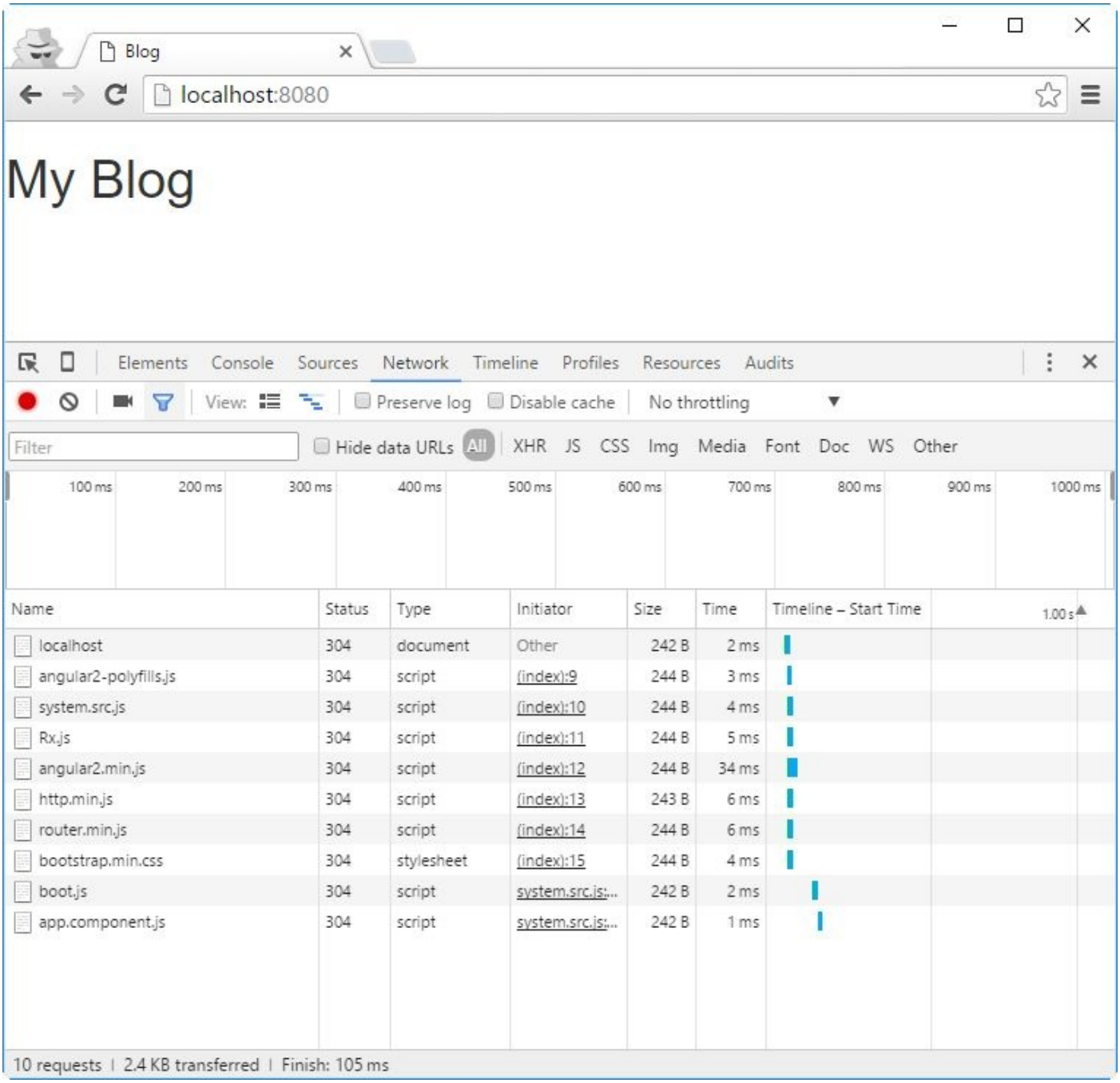


To test it on the browser, go to the command line on the `blog` directory and execute:

```
$ nodemon server.js

[nodemon] 1.8.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node server.js`
Listen: 8080
```

Access `http://localhost:8080/` and you should get a result similar to the following image:



8.2 The base template for the application

The `index.html` file has a call to the `<my-app>` tag and it is there that we set up the application base, that will be instantiated in the `AppComponent` class. Usually all the HTML form the application (including the menu and content area) is added to the `AppComponent`. And for that we will create a template called `appComponent.html` in the

`public` directory. This is necessary because only the public directory is visible to the application. Create the `public/appComponent.html` file with the following code:

public/appComponent.html

```
<h1>My Blog</h1>
```

And change the `AppComponent` to:

```
import {Component} from 'angular2/core';

@Component({
     selector: 'my-app',
     templateUrl: 'appComponent.html'
})
export class AppComponent { }
```

It has the same result, but with the template separated from the component. In this template we can use some css classes from *bootstrap* to style the application. At first, we can use the following HTML code:

public/appComponent.html

```
 <nav class="navbar navbar-inverse navbar-fixed-top">
     <div class="container">
     <div class="navbar-header">
          <button type="button"
                    class="navbar-toggle collapsed"
                    data-toggle="collapse"
                    data-target="#navbar"
                    aria-expanded="false"
                    aria-controls="navbar">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" href="#"> BLOG </a>
     </div>
     <div id="navbar" class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
               <!--     -->
               <!-- MENU -->
               <!--      -->
          </ul>
     </div>
     </div>
</nav>
<div style="padding-top:50px">
     CONTENT
</div>
```

Notice that we separated the application in a menu and a content area.

8.3 Implementing the routing

To implement the application router we need the following:

- Split the application in components.
- Set up the AppComponent.
- Include the menu.
- Set up the HTML including the `<router-outlet>` tag.

8.3.1 Creating the components

Each application view must be a component. In this blog we will have three views:

1. HomeComponent: Shows all the posts
2. LoginComponent: Login form
3. AddPostComponent: Add a post

The components can be created in the `app/components` directory, at first with no information:

app/component/home.ts

```
import {Component} from 'angular2/core';

@Component({
     template: `HomeComponent`
})
export class HomeComponent {

}
```

app/component/login.ts

```
import {Component} from 'angular2/core';

@Component({
     template: `LoginComponent`
})
export class LoginComponent {

}
```

app/component/addpost.ts

```
import {Component} from 'angular2/core';

@Component({
     template: `AddPostComponent`
})
export class AddPostComponent {

}
```

To define a `library` for this components, create the `app/component.ts` file with the following code:

```
export * from './component/home'
export * from './component/login'
export * from './component/addpost'
```

With this library we can reference the component classes like the following:

```
import {HomeComponent,LoginComponent,AddPostComponent} from './component'
```

Which is better than importing this way:

```
import {HomeComponent} from './component/home'
import {HomeComponent} from './component/home'
import {AddPostComponent} from './component/addpost'
```

8.3.2 Setting up the @RouteConfig

With the components ready we can set up the router in the `AppComponent` with the `RouterConfig` directive:

```
import {Component} from 'angular2/core';
import {ROUTER_DIRECTIVES, ROUTER_PROVIDERS, RouteConfig} from 'angular2/rou\
ter'
import {HomeComponent,LoginComponent,AddPostComponent} from './component'

@Component({
     selector: 'my-app',
     providers: [ROUTER_PROVIDERS],
     directives: [ROUTER_DIRECTIVES],
     templateUrl:'appComponent.html',

})
@RouteConfig([
     { path: '/', name: 'Home', component: HomeComponent,
                                                useAsDefault: true },
     { path: '/Login', name: 'Login', component: LoginComponent },
     { path: '/Addpost', name: 'AddPost', component: AddPostComponent }
])
export class AppComponent {

}
```

Notice that we imported `ROUTER_DIRECTIVES` and `ROUTER_PROVIDERS` that must be added to the `directives` and `providers` properties from `@Component`. Then we use the `@RouteConfig` to create three routes linking them to the created components.

8.3.3 Setting up the menu

The application menu must be integrated to the routes. For that, edit the `public\appComponent.html` file with the following code:

public\appComponent.html

```
<nav class="navbar navbar-inverse navbar-fixed-top">
       <div class="container">
         <div class="navbar-header">
           <button type="button"
                   class="navbar-toggle collapsed"
                   data-toggle="collapse"
                   data-target="#navbar"
                   aria-expanded="false"
                   aria-controls="navbar">
```

```
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" href="#">{{title}}</a>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
          <ul class="nav navbar-nav">

           <li><a [routerLink]="['/Home']">Home</a></li>
            <li><a [routerLink]="['/Login']">Login</a></li>
            <li><a [routerLink]="['/AddPost']">Add Post</a></li>

          </ul>
        </div>
      </div>
    </nav>
<div style="padding-top:50px">
    CONTENT
</div>
```

8.3.4 Setting up the router-outlet

To finish the Router creating it is necessary to inform where the components are loaded. This is done with the `<router-outlet>` tag that must be inserted into the template that contains the router settings:

public\appComponent.html

```
<nav class="navbar navbar-inverse navbar-fixed-top">
     <!-- ... MENU…. -->
    </nav>
<div style="padding-top:50px">
    <router-outlet></router-outlet>
</div>
```

To test the application, reload the page and click on the menu items to check the content being loaded.

8.4 Showing Posts

In the `HomeComponent` component we will show the posts from the data base. For that we will do a GET request to the `/api/posts` URL that doesn't need authentication. Instead of programming this component call, we will create a service called `PostService` and the `Post` model with the definition of a Post:

```
export class Post{
    constructor(
        public title:  String,
        public author: String,
        public body:   String,
        public date: Date
    ){}
}
```

And also create the library:

app/model.ts

```
export * from './model/post'
```

The service uses the `Http` class to make requests to the server:

app/service/post.ts

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map' // <<<<<<<

@Injectable()
export class PostService {
     constructor(private http: Http) { }
     public getPosts() {
          return this.http
               .get('./api/posts')
               .map(res => res.json());
     }
}
```

We used the dependency injection of the `Http` class that is set in the bootstrap of the application (app/boot.ts). The `getPosts` method will make an Ajax request to the `./api/posts` URL and return ans instance of the `Observable` class from Angular 2, that will be used in the class that called `getPosts()`.

Since we will show the *posts* on the main page, we must set up the `HomeController` to access this service:

app/component/home.ts

```
import {Component} from 'angular2/core';
import {PostService} from '../service/post'
import {Post} from '../model'

@Component({
     providers: [PostService],
     template: "HomeComponent"
})
export class HomeComponent {
     public posts: Array<Post>;
     constructor(private postService: PostService) {
          postService.getPosts().subscribe(
               p => this.posts = p,
               err => console.log(err)
          );
     }
 }
```

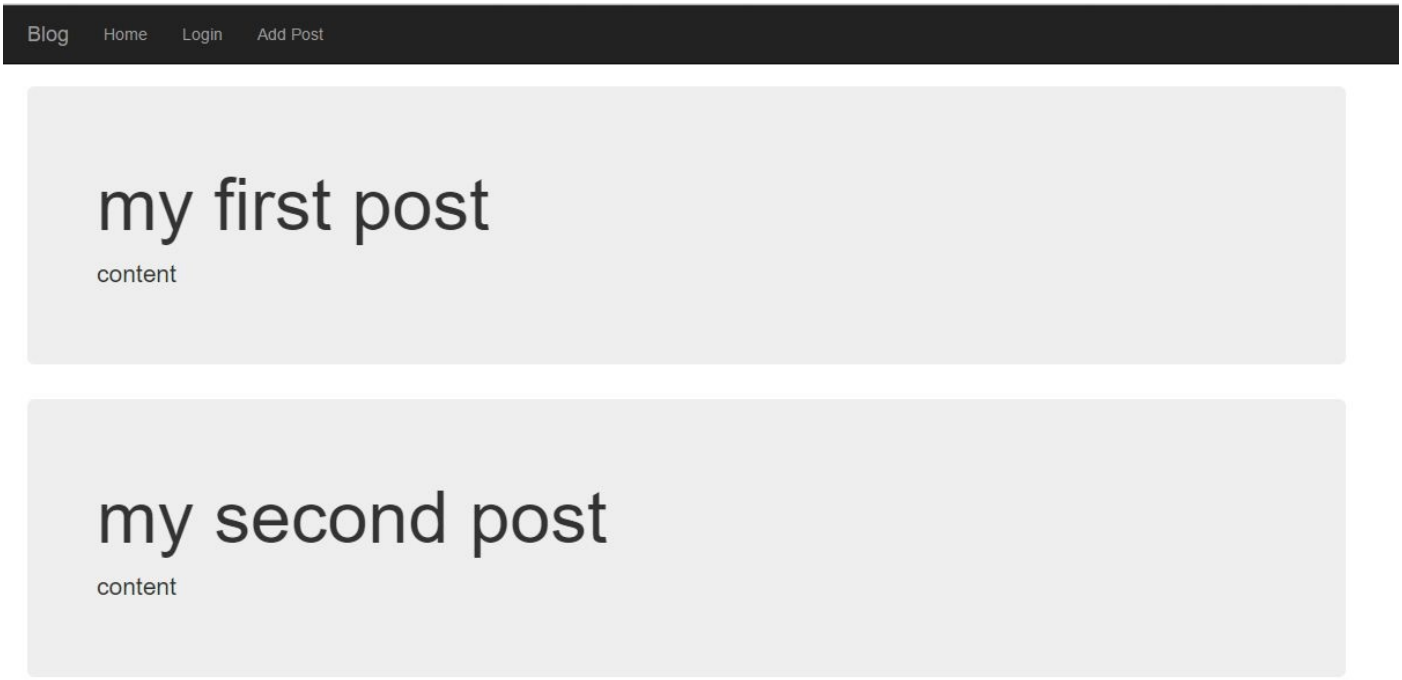
We imported the `PostService` and added it to the `providers` of the component. That way we can inject it into the constructor method. After injecting it we call `postService.getPosts()` that will execute the `subscribe` method to get the posts from

the response or deal with any error. Notice that the posts will be stores in the variable `this.posts` to be able to show them on the page.

And finally we edit the component template to:

```
// imports…
@Component({
     providers: [PostService],
     template: `
     <div class="jumbotron" *ngFor="#p of posts">
          <h1>{{p.title}}</h1>
          <p>content</p>
     </div>
     `
})
export class HomeComponent { ..... }
```

We used `*ngFor` to loop between the `posts` variable items and for now, show the post title. The result should be similar to the following image:



8.5 Login

The login view is defined by the `LoginComponent` that has the following structure:

app/component/login.ts

```
import {Component} from 'angular2/core'
import {User} from '../model'
import {UserService} from '../service/user'
import {LoginService} from '../service/login'
import {Router} from 'angular2/router';

@Component({
     providers: [UserService],
     template: `
               >>TEMPLATE<<
      `
```

```
})
export class LoginComponent {
     private user:User=new User();
     private showLoading:boolean = false;
     private errorMessage:string = null;
     constructor(private userService:UserService,
          private loginService:LoginService,
          private router:Router){

     }
     onClick(event){
          event.preventDefault();
          this.showLoading = true;
          this.errorMessage = null;
          this.userService.insert(this.user).subscribe(
               result => this.onLoginResult(result),
               error => this.onLoginError(error)
          );
     }
     onLoginResult(result){
          console.log(result);
          this.loginService.setLogin(result.user,result.token);
          this.router.navigate( ['Home'] );
     }
     onLoginError(error){
          this.showLoading = false;
          this.errorMessage = error._body;
     }
}
```

Notice that we imported `UserService` and `LoginService` classes to provide data to the component. We will create those files soon.

We also have here two variables to control the application template. For example, `showLoading` will control if a “Loading” message will be shown, as on the following example:

```
<div class="col-md-4 col-md-offset-4" *ngIf="showLoading">
          Loading…
</div>
```

With this template it is possible to set up its visibility just defining the `showLoading` variable value as `true` or `false`.

The full template of the `LoginComponent`:

```
 <div class="col-md-4 col-md-offset-4" *ngIf="!showLoading">

     <div *ngIf="errorMessage" class="alert alert-danger" role="alert">
          {{errorMessage}}
     </div>

     <form ngForm>
          <div class="form-group">
               <label for="login">Login</label>
               <input type="text" class="form-control"
                    id="login"
```

```
                required
                placeholder="Login"
                [(ngModel)]="user.login">
        </div>
        <div class="form-group">
            <label for="password">Password</label>
            <input type="password" class="form-control"
            id="password"
            required
            placeholder="Password"
            [(ngModel)]="user.password">
        </div>
        <div class="checkbox">
            <label>
            <input id="createAccount" type="checkbox"
            [(ngModel)]="user.isNew"> Create Account?
            </label>
        </div>
        <div class="form-group" *ngIf="user.isNew">
            <label for="login">Your Name</label>
            <input type="text" class="form-control"
                id="name"
                placeholder="Your Name"
                [(ngModel)]="user.name">
        </div>

        <button type="submit" class="btn btn-default pull-right"
            (click)="onClick($event)" >Login</button>

    </form>
    </div>

    <div class="col-md-4 col-md-offset-4"
            *ngIf="showLoading">
        Loading…
    </div>
```

8.6 Services

Some classes like `LoginService` and `HeadersService` may be used in many components, so they should be injected into the `boot.ts` file:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {LoginService} from './service/login'
import {HeadersService} from './service/headers'


bootstrap(AppComponent,[HeadersService, HTTP_PROVIDERS, LoginService]);
```

There are some services that are used in just a few classes, as the `UserService`, so it is injected into the class via the `providers` as in the `LoginComponent`.

8.6.1 LoginService

The `LoginService` class provides some information about the user login:

app/service/login.ts

```
import {Injectable} from 'angular2/core'
import {User} from '../model'

@Injectable()
export class LoginService {
     private user:User=null;
     private token:string=null;
     constructor() { }
     setLogin(u:User,t:string){
          this.user = u;
          this.token = t;
     }
     getToken():string{
          return this.token;
     }
     getUser(){
          return this.user;
     }
     isLogged(){
         return this.user!=null && this.token!=null;
     }
     logout(){
          this.user = null;
          this.token = null;
     }
}
```

8.6.2 UserService

The `UserService` class is responsible for adding a new user:

```
import {Http, HTTP_PROVIDERS, Headers} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {User} from '../model'
import {HeadersService} from './headers'

@Injectable()
export class UserService {
     constructor(private _http: Http, private _header:HeadersService) {

      }
     public insert(u:User) {
         return this._http
               .post('./api/login',
                      JSON.stringify(u),
                      this._header.getJsonHeaders())
               .map(res => res.json());
     }
}
```

8.6.3 HeadersService

This class is used to provide *HTTP* header information to the client request to the server:

```
import {Injectable} from 'angular2/core'
import {Headers} from 'angular2/http'

@Injectable()
export class HeadersService {
     constructor(){}

     getJsonHeaders(token?:string){
          var headers = new Headers();
          headers.append('Content-Type', 'application/json');
          if (token)
               headers.append('x-access-token',token)
          return {headers: headers};
     }
```

The `getJsonHeaders` methods returns a header set up to the JSON format, and if the `token` property is sent, it will be added to the header. Every *Http* request from the client to the server has a header in which we can send some information. In this case, we are sending `Content-Type` and `x-access-token`.

8.7 Connecting to the server

Let's understand how the Angular connects to the server to get information. In the LoginComponent we have the Login button with the following action: `(click)="onClick($event)"`. When the button is clicked the method `onClick` from the `LoginComponent` class is called:

app/component/login.ts

```
class LoginService{
onClick(event) {
     event.preventDefault();
     this.showLoading = true;
     this.errorMessage = null;
     this.userService.insert(this.user).subscribe(
          result => this.onLoginResult(result),
          error => this.onLoginError(error)
     );
}
onLoginResult(result) {
     console.log(result);
     this.loginService.setLogin(result.user,result.token);
     this.router.navigate( ['Home'] );
}
onLoginError(error) {
    this.showLoading = false;
    this.errorMessage = error._body;
}
}
```

At first, the `onClick` method calls `event.preventDefault()` forcing that the form submission event is canceled (since we will use Ajax). Then we show the "Loading" message to show the user that the information is being processed. It is also important to hide the form, to avoid the user from clicking again on the button. The `insert` method from the `userService` class is called to connect to the server sending `this.user` in JSON

format. When the service returns the data from the server, the `subscribe()` method is executed. It has two callbacks: `result` and `error`. Each one has its own method to process the data.

The `onLoginResult` method is executed when tue user is authenticated. We use the `LoginService` class to inform this login and it will and it will be accessible anywhere in the application (one of the advantages of using dependency injection). And then we use the `router` to go back to `Home`.

If any error happen the `onLoginError` method will be called to hide the "Loading" message and show the error message. Notice that the `div` will become visible by just setting the `this.errormessage` variable.

8.8 Posts

After the user login he will be able to click on the "AddPost" link and be redirected to the `AddPostComponent`.

app/component/addPost.ts

```
import {Component} from 'angular2/core'
import {Post} from '../model'
import {LoginService} from '../service/login'
import {Router} from 'angular2/router'
import {PostService} from '../service/post'

@Component({
     providers: [PostService],
     template: `
               >>TEMPLATE<<
          `
})
export class AddPostComponent {

     private post:Post = new Post();
     private errorMessage:string = null;
     private showLoading:boolean = false;

     constructor(private _loginService:LoginService,
                                       private _router:Router,
                                       private _postService:PostService) {
          if (!_loginService.isLogged())
               this._router.navigate( ['Login'] );
          this.post.user = this._loginService.getUser();
     }
     onClick(event){
          event.preventDefault();
          this.showLoading = true;
          this.errorMessage = null;
          this._postService.insert(this.post).subscribe(
               result => this.onInsertPostResult(result),
               error => this.onInsertPostError(error)
          );
     }
     onInsertPostResult(result){
          this._router.navigate( ['Home'] );
     }
```

```
    onInsertPostError(error){
        this.showLoading = false;
        this.errorMessage = error._body;

    }
}
```

The `AddPostsComponent` has some similarities with the Login form. The way that the errors and "Loading" messages are shown is the same. But here we used the `LoginService` to check if the user is logged in, and if not, redirect him to the login form.

Another difference is that we used the injected variable names (LoginService, PostService) with the prefix _ (`_postService` and `_loginService`). You can choose however is best for your project.

This view template is the following:

```
<div class="col-md-4 col-md-offset-4" *ngIf="showLoading">
    Loading…
</div>
<div class="col-md-8 col-md-offset-2" *ngIf="!showLoading">
<div *ngIf="errorMessage" class="alert alert-danger" role="alert">
    {{errorMessage}}
</div>
<div class="panel panel-default">
    <div class="panel-heading">Add Post</div>
    <div class="panel-body">
        <form ngForm>
    <div class="form-group">
        <label for="title">Title</label>
        <input type="text" class="form-control"
            id="title" required placeholder="Title"
            [(ngModel)]="post.title">
    </div>
    <div class="form-group">
        <label for="text">Text</label>
        <textarea rows=10 cols=100 class="form-control"
                id="text" [(ngModel)]="post.text"></textarea>
    </div>
    <button type="submit" class="btn btn-default pull-right"
            (click)="onClick($event)">Create</button>
    </form>
  </div>
 </div>
</div>
```

8.8.1 PostService

The `PostService` is responsible to connect to the server for three operations: get posts, include a post and delete posts:

app/service/post.ts

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {Post} from '../model'
import {HeadersService} from '../service/headers'
```

```
import {LoginService} from '../service/login'

@Injectable()
export class PostService {
     constructor(private _http: Http,
                    private _headerService:HeadersService,
                    private _loginService:LoginService) { }
     public getPosts() {
          return this._http
               .get('./api/posts')
               .map(res => res.json());
     }
     public insert(p:Post){
         return this._http
               .post('./api/posts', JSON.stringify(p),
                    this._headerService.
                         getJsonHeaders(this._loginService.getToken()))
               .map(res => res.json());
     }
     public delete(p:Post){
         return this._http
               .delete('./api/posts/' + p._id ,
               this._headerService.
                    getJsonHeaders(this._loginService.getToken()))
               .map(res => res.json());
     }
}
```

The `getPosts` method does a “GET /api/posts” request to the server and the `insert` will execute a “POST /api/posts” request, but there is an important detail here:

```
.post('./api/posts',
               JSON.stringify(p),
               this._headerService.
                    getJsonHeaders(this._loginService.getToken())
               )
```

Notice that we used the `HeaderService` class to call the `getJsonHeaders` method that will inform the request header (HTTP request) as JSON (*application/json*). It is possible to check this behavior analyzing the request on the networking tab from *Google Chrome Developer Tools*. Besides that, we passed the `Token` parameter, because for that request the user must be logged in.

The `delete` method adds the post `id` to the URL together with the Token via the `HeaderService`.

8.9 Refactoring the home screen

To finish this small system let’s get back to the `HomeComponent` template and add more features:

app/component/home.ts

```
import {Component} from 'angular2/core';
import {PostService} from '../service/post'
import {Post} from '../model'
```

```
import {LoginService} from '../service/login'
import {User} from '../model';


@Component({
     providers: [PostService], // OBS: LoginService at boot.ts
     template: `
     <div class="alert alert-info" *ngIf="showLoading">
     Loading...
     </div>
     <div *ngIf="!showLoading">
          <div *ngIf="_loginService.isLogged()"
                         class="alert alert-success">
                 Olรก {{_loginService.getUser().name}}
                 <a href="#" (click)="logout($event)"
                 class="pull-right" >
                 Sair</a>
          </div>
          <div class="jumbotron" *ngFor="#p of posts">
               <h1>{{p.title}}</h1>
               <p>{{p.text}}</p>
               <p>Por: {{p.user?.name}}</p>
               <a href="#" (click)="deletePost(p)"
               *ngIf="checkPost(p)">Delete</a>
          </div>
     </div>
     `
})
export class HomeComponent {

     private posts: Array<Post>;
     private showLoading:boolean=false;

     constructor(private _postService: PostService,
                    private _loginService:LoginService) {
          this.loadAllPosts();
     }
     loadAllPosts(){
          this.showLoading = true;
          this._postService.getPosts().subscribe(
               p => this.onLoadAllPostsResult(p),
               err => console.log(err)
          );
     }
     onLoadAllPostsResult(p){
          this.posts = p;
          this.showLoading = false;
     }
     logout(event){
          this._loginService.logout();
     }
     checkPost(p:Post):boolean{
          try {
               if (p.user == null) return false;
               if (!this._loginService.isLogged()) return false;
          return p.user._id==this._loginService.getUser()._id;
          } catch (error) {
               return false;
          }
```

```
            return false;
        }
        deletePost(p){
            this._postService.delete(p).subscribe(
                result => this.onDeletePostResult(result),
                error => this.onDeletePostError(error)
            )
        }
        onDeletePostResult(result){
            this.loadAllPosts();
        }
        onDeletePostError(error){
            console.log(error);
        }
    }
```

There are two main changes to the component. The first is that we included a bar showing if the user is logged in:

```
<div *ngIf="_loginService.isLogged()"
      class="alert alert-success">
        Olá {{_loginService.getUser().name}}
        <a href="#" (click)="logout($event)"
        class="pull-right" >
            Sair
        </a>
</div>
```

Notice that we used `LoginService` to control both the visibility and the login information.

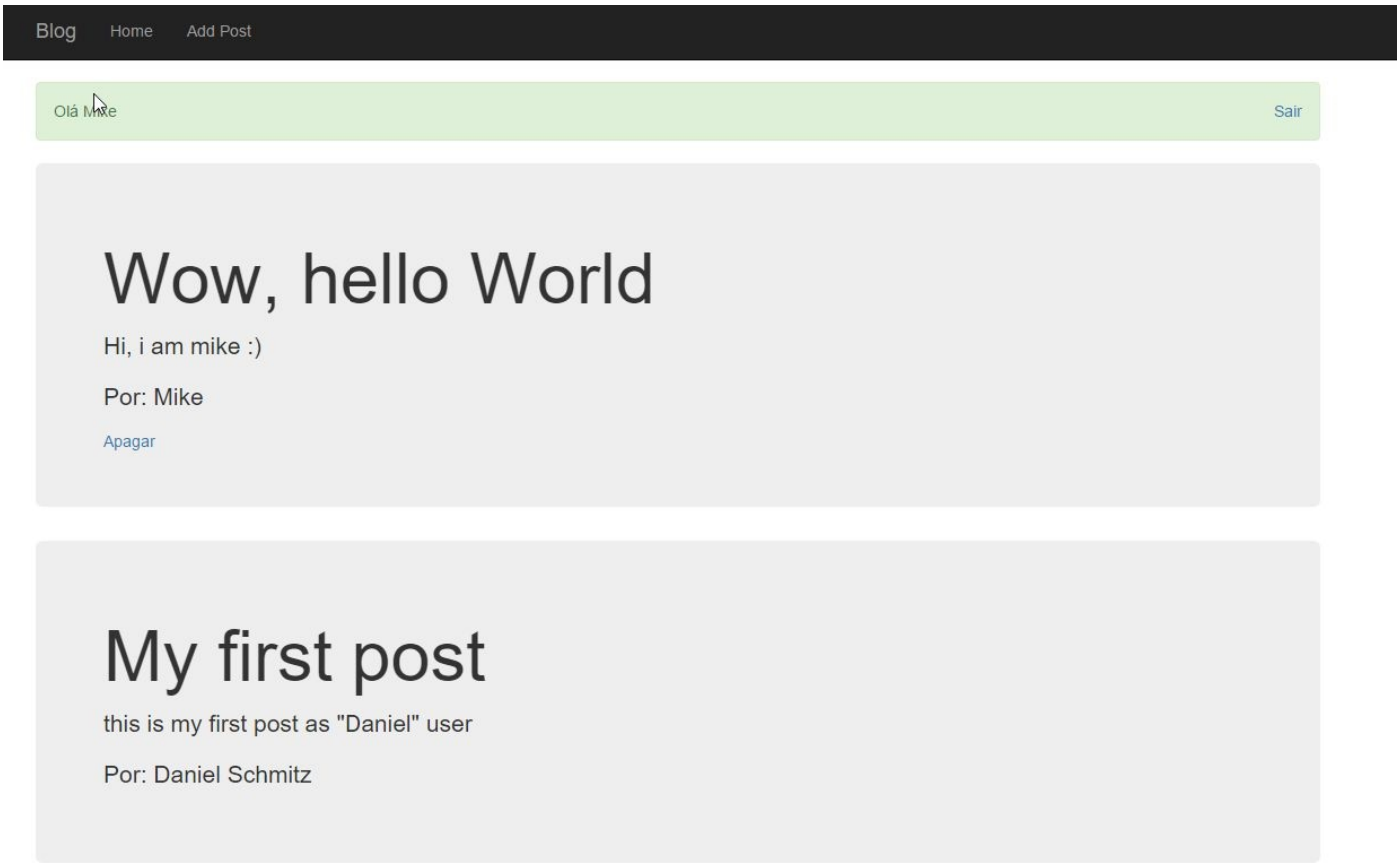
We also created the `logout` method that will call the `this._loginService.logout()` method, updating the entire view again.

Another detail is that we added the “Delete” button in each Post:

```
<a href="#" (click)="deletePost(p)"
             *ngIf="checkPost(p)">Delete</a>
```

The visibility of the link is controlled by the `checkPost` method that returns true if, and only if, the logged in user is the Post owner.

Now the main view of the system should be similar to the following image:



8.10 Conclusion

There are a few more details to be done on the `blog` project, that we will leave as homework for the reader. For example the possibility to edit Posts, change the password, send the password to the user e-mail, comments session, etc. All those features use th same concepts learned here.

As this book is published through Leanpub, it is possible to extend the book with more content, but for that we needd the community feedback. For that, access this book page in https://leanpub.com/livro-angular2 and click on “Discuss this Book”.