01 Getting Started with Angular 2

The following describes how to create a project in Angular 2 using:

- SystemJS module loader
- JSPM package manager
- TypeScript language

This tutorial assumes you have npm installed. If not, download and install node.js.

Follow these steps to learn how to build an Angular 2 app with TypeScript and JSPM:

Initial Setup

We will be using JSPM package manager to install dependencies and manage their versions. Start by installing JSPM globally:

```
npm install jspm -g
```

Now that we have JSPM installed we can start setting up the project. Create the folder for your project: "angular2-getting-started".

Go into the angular2-getting-started folder and initialise JSPM with the default configuration using TypeScript.

You should accept most default settings (by hitting enter), ***except*** for this question:

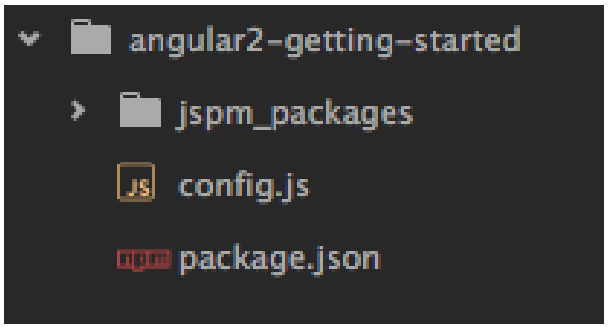
> Q: "Which ES6 transpiler would you like to use, Babel, TypeScript or Traceur?"
>
> A: TypeScript

```
cd angular2-getting-started
jspm init
```

```
➜ angular2-getting-started  jspm init
Package.json file does not exist, create it? [yes]:
Would you like jspm to prefix the jspm package.json properties under jspm? [yes]:
Enter server baseURL (public folder path) [./]:
Enter jspm packages folder [./jspm_packages]:
Enter config file path [./config.js]:
Configuration file config.js doesn't exist, create it? [yes]:
Enter client baseURL (public folder URL) [/]:
Do you wish to use a transpiler? [yes]:
Which ES6 transpiler would you like to use, Babel, TypeScript or Traceur? [babel]:Typ
ok   Verified package.json at package.json
     Verified config file at config.js
     Looking up loader files...
       system.js
       system-csp-production.js
       system.src.js
       system.js.map
       system-polyfills.js
       system-csp-production.js.map
       system-polyfills.src.js
       system-csp-production.src.js
       system-polyfills.js.map

     Using loader versions:
       systemjs@0.19.5
     Looking up npm:typescript
     Updating registry cache...
ok   Installed typescript as npm:typescript@^1.6.2 (1.6.2)
ok   Loader files downloaded successfully
```

You can now open your IDE at this directory:



This init process created a SystemJS config.js file, npm package.json file with a single dependency on the TypeScript npm module, for transpiling T
JavaScript, and a folder called "jspm_packages" where all JSPM dependencies would reside.

From command line add the following dependencies, required for creating an Angular 2 application (specific versions below are required during An

```
jspm install angular2 reflect-metadata zone.js@0.5.10 rxjs@5.0.0-beta.0
```

Look at the config.js and package.json to notice the changes JSPM made to create these dependencies. All the imported library dependencies are flattened. JSPM imported all required files into the jspm_packages folder.

Angular 2 actually ships with some of these required dependencies, and need them because it assumes running in an ES6 environment:

```
es6-shim.js
angular2/bundles/angular2-polyfills.js
  ├─ es6-promise.js
  ├─ reflect-metadata.js
  └─ zone.js
```

Basic Application

Now we're ready to start creating some skeleton for our project. Let's create the index.html:

index.html

```
<html>
<head>

    <meta charset="UTF-8">
    <title>Angular 2 Getting Started</title>

</head>
<body>

<h1>Angular Getting Started</h1>

<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
    System.import('app');
</script>

</body>
</html>
```

Create a folder named app and put the following main.ts TypeScript file in there. The TypeScript syntax is intentionally there to ensure it loads and transpiler properly when we test it.

app/main.ts

```
import 'zone.js/dist/zone.min.js';
import 'reflect-metadata';

var isDone:boolean = false; // some TypeScript syntax to confirm it is compiled properly
console.log("app script loaded successfully with typescript");
```

To successfully load and transpile TypeScript we need to update SystemJS's configuration with a few options. Add the following snippet to the con as the default file extension when importing project files and configure some TypeScript options:

config.js

```
//...
typescriptOptions: {
  "module": "commonjs",
  "emitDecoratorMetadata": true
},
packages: {
  "app": {
    "main": "main",
    "defaultExtension": "ts"
  }
},
//...
```

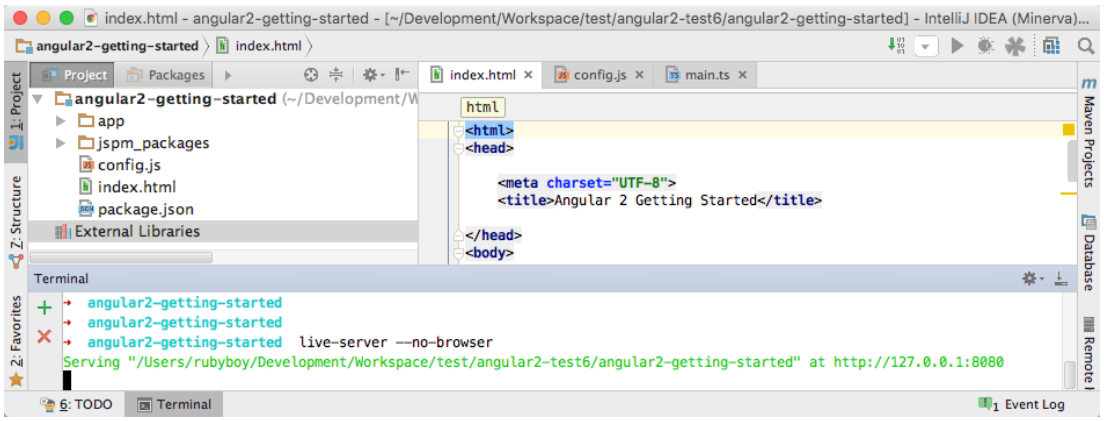
Let's test our basic project. To run the project in a browser we'll be using a utility called live-server. It runs a local web server pointing at the develo refreshes the browser page automatically when it detects changes in files.

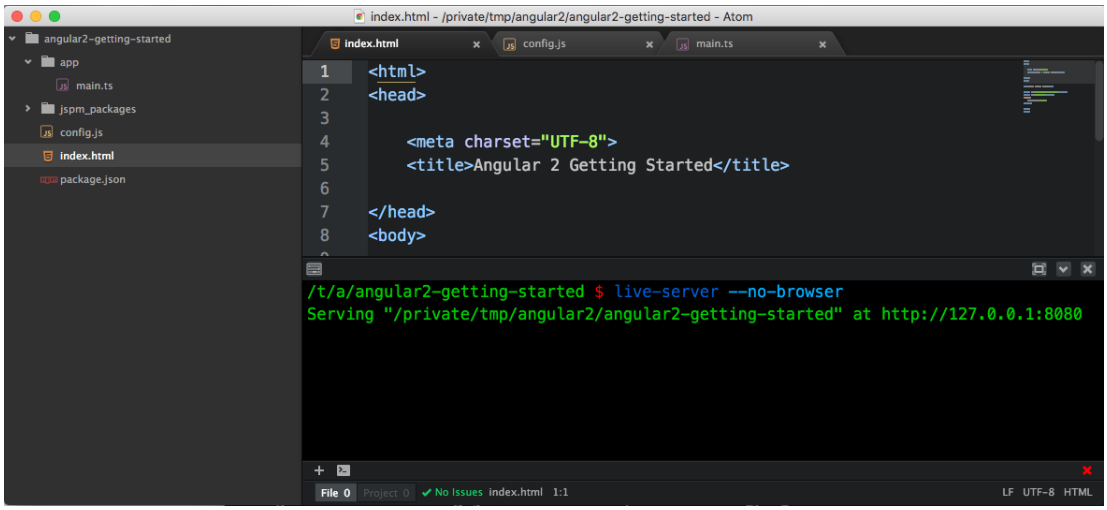
```
npm install -g live-server
```

Then, while in the root folder of our project run:

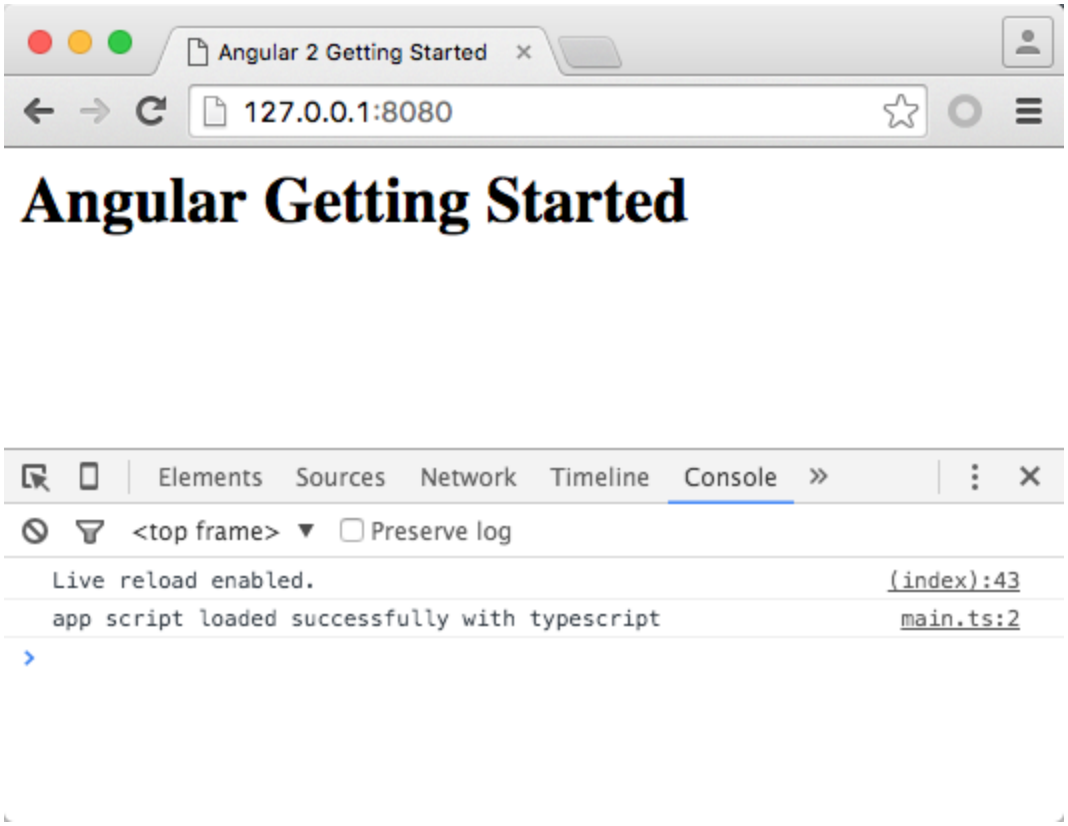
```
live-server
```

If you use IntelliJ or Atom, you can run it from the terminal pane:





If everything went well you should see the following in your browser:



You can copy the commands and code from the following screencast (select from the video - it works!):

Adding Angular 2 Components

Let's add Angular 2 components to the mix. Create a main component for our app in a new file: app-component.ts under the app folder:

app-component.ts

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

@Component({
    selector: 'my-app',
    template: `<h3>{{title}}</h3>`
})
export class AppComponent {
    private title:string = "Angular 2.0 is in the house.";
}
```

The @Component decorator provides configuration options for your component. The selector provides a way to reference this component in HTML configuration the title will be interpolated from the component context attributes, very similar to how it would be done in Angular 1.

Add the component selector element to the index.html. This is where our main component will be rendered.

index.html

```
<h1>Angular Getting Started</h1>

<my-app>Loading...</my-app>
```

To bootstrap angular you should call the bootstrap method and pass it the main component. Update the main.ts file to do that:

main.ts

```
import 'zone.js/dist/zone.min.js';
import 'reflect-metadata';

import {bootstrap} from 'angular2/platform/browser';
import {AppComponent} from "./app-component";

bootstrap(AppComponent);
```

If you still have live-server running then the browser should refresh automatically:

Binding to DOM properties

One of the more significant changes that Angular 2 bring with it is the DOM property binding. You can now find to any property of the DOM itself (n anymore), thus getting more power to control any DOM behaviour via binding and requiring less custom directives. This is done using square brac ***sion***". The template expression value can be a component property, function or a statement. Update the AppComponent to set a random colour fo

app/app-component.ts

```
@Component({
    selector: 'my-app',
    template: `<h3 [style.color]="color">{{title}}</h3>`
})
export class AppComponent {
    private title:string = "Angular 2.0 is in the house.";
    private color:string = "#"+((1<<24)*Math.random()|0).toString(16);
}
```

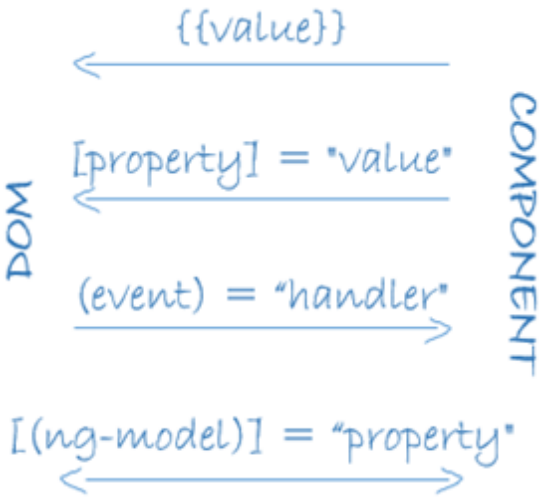
Event Bindings

Similar to DOM properties, Angular 2 provides an easy way to find to any DOM events using braces. **(event)="statement"** will bind an event to a p
Update the AppComponent to provide a way to update the colour on click:

app/app-component.ts

```
@Component({
    selector: "my-app",
    template: `<h3 [style.color]="color" (click)="updateColor()">{{title}}</h3>`
})
export class AppComponent {
    private title:string = "Angular 2.0 is in the house.";
    private color:string;
 constructor() {
  this.updateColor();
 }
    getColor() {
      return "#"+((1<<24)*Math.random()|0).toString(16);
    }
    updateColor() {
      this.color = this.getColor();
    }
}
```

Read more about data binding in the Angular Architecture Overview



The demise of $apply and $digest

In Angular 2 there is no need to worry about applying asynchronous actions and asking for it to run digest manually. This is all taken care of autom
s, which created an execution context that spans even across asynchronous actions and event handlers.

For example, try updating the above code to set the colour asynchronously after 1 second:

app/app-component.ts

```
updateColor() {
  this.color = "lightgrey";
  window.setTimeout(() => this.color = this.getColor(),1000);
}
```

Amazingly, Angular knows when to apply these changes even though it's asynchronous and we haven't used any Angular wrapper. Magic!

Notice the use of the arrow function scoping. You got to love this! (Pun intended).

Watch this short video explaining how zone.js works:

Try looking at the window.setTimeout function in your browser with and without zone.js to see what zone.js does to "know" about asynchronous fu
to the same execution context.

Read more about zones here: Understanding Zones and Zones in Angular 2

Attribute Directives

In Angular 2, attribute directives changes the appearance or behaviour of a DOM element. for example, it lets you control the style of an element. I
colour changes to a random-color-directive.ts directive component:

app/random-color-directive.ts

```
import {Directive, ElementRef, Renderer} from 'angular2/core'
@Directive({
    selector: '[randomColor]',
    host: {
      '(click)': 'onClick()'
    }
})
export class RandomColorDirective {
    constructor(private element: ElementRef, private renderer: Renderer) {
      this.updateColor();
    }
    getColor() {
      return "#"+((1<<24)*Math.random()|0).toString(16);
    }
    onClick() {
      this.updateColor("lightgrey");
      window.setTimeout(() => this.updateColor(),1000);
    }
    updateColor(color = this.getColor()) {
      this.renderer.setElementStyle(this.element.nativeElement, 'color', color);
    }
}
```

Things to note in the directive code:

- A selector is defined, used to reference the directive from elements
- The "host" property is used to attach events to the element this directive is used on
- ElementRef and Renderer are injected into the constructor. We will learn more about Dependency Injection in 02 Creating an Angular 2 A now to note that they are provided by Angular and used as private properties with a special TypeScript notation to avoid creating these pr them in the constructor - this is done for us by the TypeScript transpiler
- The renderer and element are used to set a style on the native element. We are not using the native element directly ourselves for suppor targets in the future. Other renderers can be used for setting the "color" properly on other platforms (like mobile, for example)
- The updateColor function sets a default value for the "color" argument, if not passed in, to be a random color

Now, we can simplify the AppComponent to use the directive:

app/app-component.ts

```
import {Component} from 'angular2/core';
import {RandomColorDirective} from "./random-color-directive";
@Component({
    selector: "my-app",
    template: `<h3 randomColor>{{title}}</h3>`,
    directives: [RandomColorDirective]
})
export class AppComponent {
    private title:string = "Angular 2.0 is in the house.";
}
```

The directive is used as an attribute on the element and defined in the "directives" property of the component definition.

Building an Existing Application

There is no need to install project dependencies manually every time. The only pre-requisite for running projects is having JSPM installed. Then, a you can install all dependencies using "jspm install" in the folder of an existing project.

For example, clone https://github.com/InfomediaLtd/angular2-tutorial. Go into the angular2-getting-started and you'll see that it has no jspm packag dependencies using "jspm install". Once the package installation process finishes all dependencies will show up in the jspm_packages folder and y "live-server" to test this project.

The final code for this tutorial step can be found in the Angular Tutorial repository on GitHub.

Environment Settings

tsconfig.json

JSPM and SystemJS take care of the runtime facilities, but for IDE support (Intellisense or getting as-you-type warnings), or if you ever want to cor files yourself you'll need a tsconfig.json configuration file that specifies the root files and the compiler options required to compile the project.

Create the following file in the root folder of your project:

tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "outDir": "build"
  },
  "exclude": [
    "jspm_packages",
    "node_modules",
    "dist"
  ]
}
```

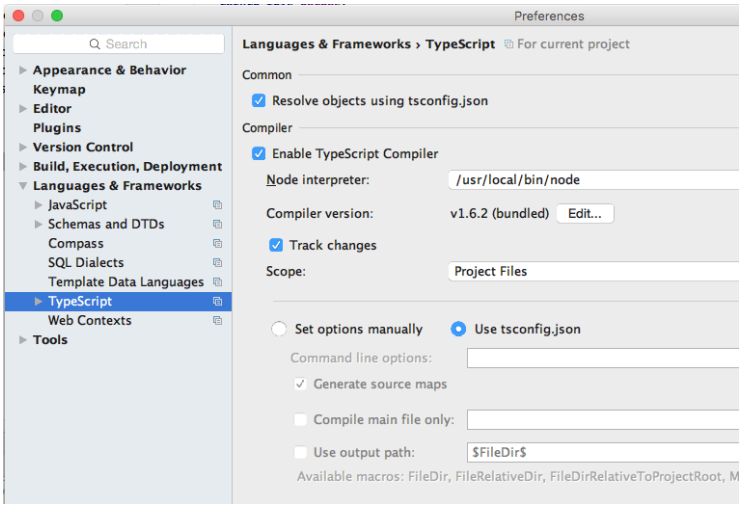
The exclusions above make sure we don't compile any of our dependencies that may reside in any of the specified folders.

We may add more options to this file, so let's also tell SystemJS to use the additional configuration in there by adding "tsconfig":true to the typescri config.js:

```
typescriptOptions: {
  "module": "commonjs",
  "emitDecoratorMetadata": true,
  "tsconfig": true
},
```

IDE Configuration

Your IDE can also compile TypeScript for you and provide errors and warnings. IntelliJ can do that using its built-in TypeScript support. You'll need settings. Atom does that by default if you have the appropriate plugin.

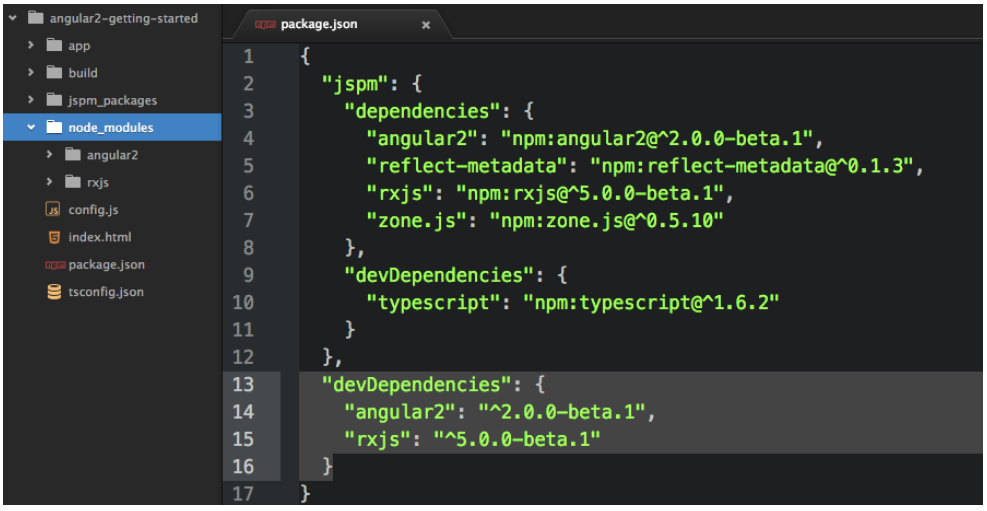


Library Resolution and Typings

Current version of TypeScript (1.7) does not support referencing files in custom locations (like jspm_packages in our case). Thus, it would still not i that resides in jspm_packages/npm/angular2@2.0.0-beta.1 (when using the beta version). Until this is resolved in version 1.8 of TypeScript (https:/ TypeScript/issues/5039), to work around this problem, you will be using the npm package typings and place the libraries for auto completion under easiest way to do that is to add an npm dependency to your package.json:

```
npm install rxjs angular2 --save-dev
```

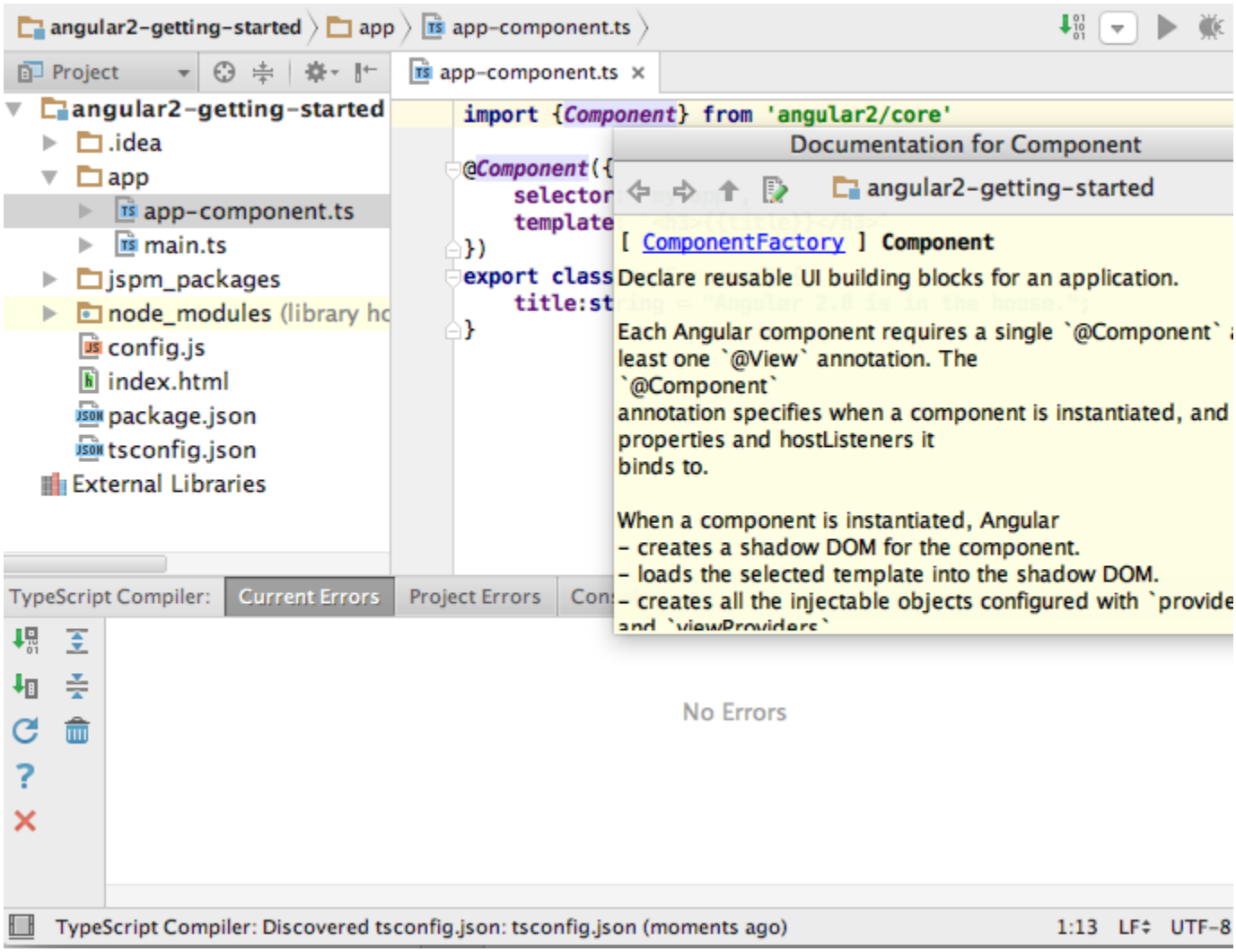
This will update your package.json npm file with the appropriate rxjs and angular2 versions and install the angular library under node_modules/anc can just copy these files to the required location manually (or create a symbolic link to the jspm_packages folder), but you should not do that! It's a required per library, but it should be resolved when the next TypeScript version is out.



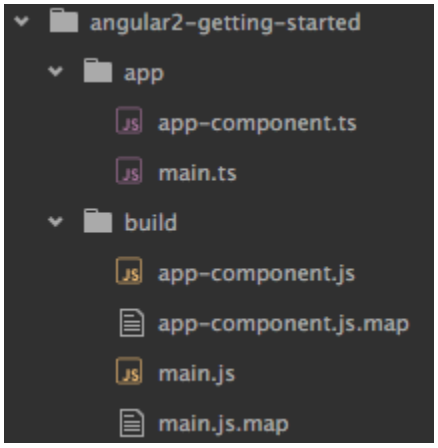
> Another optional alternatives for TypeScript type resolution is to use the Typings project and provide d.ts definition files for all required libra more about this option in 08 Dependency Typings tutorial step

IDE Compilation and IntelliSense

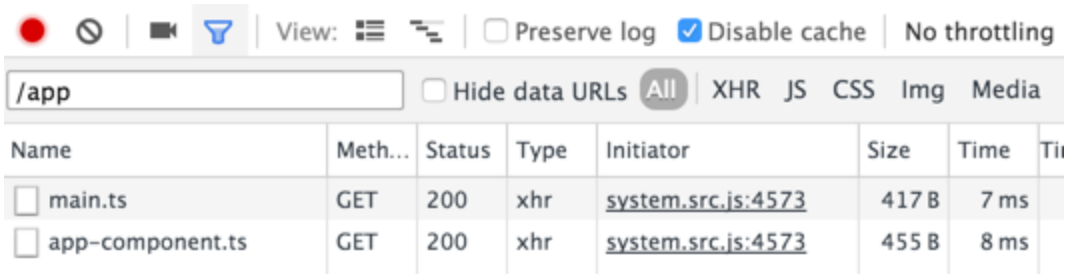
If everything worked well you should be able to view documentation and drill down to the TypeScript implementation:



Notice that now your IDE creates the JavaScript and source maps files during background TypeScript compilation. This is why we specified **"outD** tsconfig.json, to avoid messing up our source folders:



You should never check these files in to source control. They are not supposed to be used for other purposes than reference for the developers in files are never loaded into the browser. Only the TypeScript files are loaded into the browser and getting transpiled on the fly:



To avoid checking in these files make sure you ignore them along with jspm_packages and node_modules folders in your .gitignore:

```
build
dist
jspm_packages
node_modules
dev
out

*.iml
.idea/
.idea_modules/

.project
.classpath
.settings/
```

Debugging with Source Maps

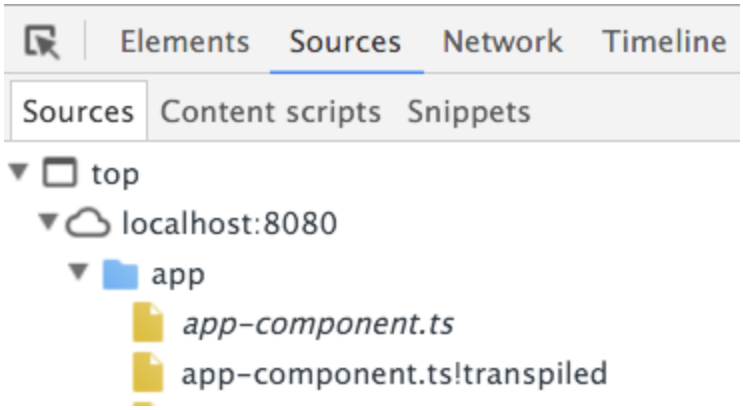
Once our Typescript code is transpiled it's quite difficult to read and debug it in the browser. Source maps help us map the transpiled code to the o

Add the following example code into the AppComponent:

```
constructor() { this.test1(); }
test1() {
    [1,2,3]
        .map(item =>
            `(${item})`)
        .filter(item =>
            item.length>1)
        .forEach(value =>
            console.log(value));
    const testObj = {a:1, b:2, c:3};
    const {a,b} = testObj;
    this.test2(a,b,testObj);
}
test2(a = 1, b = 2, {c = 3}) {
    console.log(a, b, c);
}
```

By default, the JSPM Typescript plugin will generate source maps so there are no additional steps required for debugging this in the browser.

You can see that the original TypeScript file is loaded into the browser sources alongside the transpiled one:



The transpiled code is still readable, but quite a bit different than the original code:

Transpiled Code

```
AppComponent.prototype.test1 = function () {
    [1, 2, 3]
        .map(function (item) {
         return ("(" + item + ")"); })
        .filter(function (item) {
         return item.length > 1; })
        .forEach(function (value) {
         return console.log(value); });
    var testObj = { a: 1, b: 2, c: 3 };
    var a = testObj.a, b = testObj.b;
    this.test2(a, b, testObj);
};
AppComponent.prototype.test2 = function (a, b, _a) {
    if (a === void 0) { a = 1; }
    if (b === void 0) { b = 2; }
    var _b = _a.c, c = _b === void 0 ? 3 : _b;
    console.log(a, b, c);
};
```

Put a breakpoint anywhere in the original TypeScript source and try debugging the code: