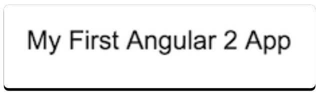
5 MIN QUICKSTART

Our QuickStart goal is to build and run a super-simple Angular 2 application in TypeScript.

See It Run!

Try this live example which loads the sample app in plunker and displays a simple message:


My First Angular 2 App

Of course we don't build apps to run in plunker. The following steps establish a development environment for the documentation samples that also can be the foundation for our real world applications. At a high level, we will

- set up the development environment
- write the app's Angular root component
- write main.ts which tells Angular to display the root component
- write the host web page (`index.html`)

We'll see many code blocks as we pursue this agenda. They're all easy to copy and paste:

```
Click the glyph on the right to copy code snippets to the clipboard ⇨⇨⇨⇨⇨⇨⇨⇨⇨⇨
```

HIDE EXPLANATIONS

Explanations describe the concepts and reasons behind the instructions. Explanations have a thin border on the left like *this* block of text.

Click *Hide Explanations* to show only the instructions. Click it again to see everything again.

Development Environment

We need to set up our development environment:

- install node and npm
- create an application project folder
- add a tsconfig.json to guide the TypeScript compiler
- add a typings.json that identifies missing TypeScript definition files
- add a package.json that defines the packages and scripts we need
- install the npm packages and typings files

Install node and npm if not already on your machine.

Create a **new project folder**

```
mkdir angular2-quickstart
cd    angular2-quickstart
```

Add a **tsconfig.json** file to the project folder and copy/paste the following:

tsconfig.json

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

This `tsconfig.json` file guides the TypeScript compiler. Learn more about it in the TypeScript Configuration chapter.

Add a **typings.json** file to the project folder and copy/paste the following:

typings.json

```
{
  "ambientDependencies": {
    "es6-shim": "github:DefinitelyTyped/DefinitelyTyped/es6-shim/es6-
shim.d.ts#4de74cb527395c13ba20b438c3a7a419ad931f1c",
    "jasmine":
"github:DefinitelyTyped/DefinitelyTyped/jasmine/jasmine.d.ts#bc92442c075929849ec41d28ab618892ba493504"
  }
}
```

> Many JavaScript libraries extend the JavaScript environment with features and syntax that the TypeScript compiler doesn't recognize natively. We teach it about these capabilities with TypeScript type definition files — *d.ts files* — which we identify in a `typings.json` file.
>
> We go a little deeper into *typings* in the TypeScript Configuration chapter.

Add a **package.json** file to the project folder and copy/paste the following:

package.json

```
{
  "name": "angular2-quickstart",
  "version": "1.0.0",
  "scripts": {
    "start": "concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  },
  "license": "ISC",
  "dependencies": {
    "angular2": "2.0.0-beta.8",
    "systemjs": "0.19.22",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.3",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.2",
    "zone.js": "0.5.15"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.1.0",
    "typescript": "^1.8.2",
    "typings":"^0.6.8"
  }
}
```

Adding the libraries we need with *npm*

Angular application developers rely on the *npm* package manager to install the libraries their apps require. The Angular team recommends the starter-set of packages specified in the `dependencies` and `devDependencies` sections. See the npm packages chapter for details.

Helpful scripts

We've included a number of npm scripts in our suggested `package.json` to handle common development tasks:

package.json (scripts)

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

We execute most npm scripts in the following way: `npm run` + *script-name*. Some commands (such as `start` don't require the `run` keyword).

Here's what these scripts do:

- `npm start` - run the compiler and a server at the same time, both in "watch mode"
- `npm run tsc` - run the TypeScript compiler once
- `npm run tsc:w` - run the TypeScript compiler in watch mode; the process keeps running, awaiting changes to TypeScript files and re-compiling when it sees them.
- `npm run lite` - run the lite-server, a light-weight, static file server, written and maintained by John Papa with excellent support for Angular apps that use routing.
- `npm run typings` - runs the *typings* tool
- `npm postinstall` - called by *npm* automatically *after* it successfully completes package installation. This script installs the TypeScript definition files this app requires.

Install these packages by entering the following *npm* command in a terminal window (command window in Windows):

```
npm install
```

Scary **error messages in red** may appear **during** install. The install typically recovers from these errors and finishes successfully.

npm errors and warnings

All is well if there are no console messages starting with `npm ERR!` *at the end* of **npm install**. There might be a few `npm WARN` messages along the way — and that is perfectly fine.

We often see an `npm WARN` message after a series of `gyp ERR!` messages. Ignore them. A package may try to re-compile itself using `node-gyp`. If the re-compile fails, the package recovers (typically with a pre-built version) and everything works.

Just make sure there are no `npm ERR!` messages at the end of `npm install`.

We're all set. Let's write some code.

Our First Angular Component

Let's create a folder to hold our application and add a super-simple Angular component.

Create an *app* sub-folder off the root directory and make it the current directory

```
mkdir app
cd    app
```

Add a component file named *app.component.ts* and paste the following lines:

app/app.component.ts

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

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

AppComponent is the root of the application

Every Angular app has at least one root component, conventionally named `AppComponent`, that hosts the client user experience.

Components are the basic building blocks of Angular applications. A component controls a portion of the screen — a *view* — through its associated template.

This QuickStart has only one, extremely simple component. But it has the essential structure of every component we'll ever write:

- One or more import statements to reference the things we need.
- A @Component decorator that tells Angular what template to use and how to create the component.
- A component class that controls the appearance and behavior of a view through its template.

Import

Angular apps are modular. They consist of many files each dedicated to a purpose.

Angular itself is modular. It is a collection of library modules each made up of several, related features that we'll use to build our application.

When we need something from a module, we import it. Here we import the Angular `Component` decorator function from the main Angular library module because we need it to define our component.

app/app.component.ts (import)

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

@Component decorator

`Component` is a **decorator** function that takes a *metadata* object. The metadata tell Angular how to create and use this component.

We apply this function to the component class by prefixing the function with the @ symbol and invoking it with the metadata object. just above the class:

app/app.component.ts (metadata)

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

This particular metadata object has two fields, a `selector` and a `template`.

The **selector** specifies a simple CSS selector for an HTML element that represents the component.

> The element for this component is named `my-app`. Angular creates and displays an instance of our `AppComponent` wherever it encounters a `my-app` element in the host HTML.

The **template** specifies the component's companion template, written in an enhanced form of HTML that tells Angular how to render this component's view.

> Our template is a single line of HTML announcing "*My First Angular App*".
>
> A more advanced template could contain data bindings to component properties and might identify other application components which have their own templates. These templates might identify yet other components. In this way an Angular application becomes a tree of components.

Component class

At the bottom of the file is an empty, do-nothing class named `AppComponent`.

app/app.component.ts (class)

```
export class AppComponent { }
```

When we're ready to build a substantive application, we can expand this class with properties and application logic. Our `AppComponent` class is empty because we don't need it to do anything in this QuickStart.

We **export** `AppComponent` so that we can **import** it elsewhere in our application, as we'll see when we create `main.ts`.

Show it with *main.ts*

Now we need something to tell Angular to load the root component

Add a new file , `main.ts`, to the `app/` folder as follows:

app/main.ts

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

bootstrap(AppComponent);
```

We import the two things we need to launch the application:

1. Angular's browser `bootstrap` function
2. The application root component, `AppComponent`.

> Then we call `bootstrap` with `AppComponent`.
>
> **Bootstrapping is platform-specific**
>
> Notice that we import the `bootstrap` function from `angular2/platform/browser`, not `angular2/core`.
>
> Bootstrapping isn't core because there isn't a single way to bootstrap the app. True, most applications that run in a browser call the bootstrap function from this library.
>
> But it is possible to load a component in a different environment. We might load it on a mobile device with Apache Cordova or NativeScript. We might wish to render the first page of our application on the server to improve launch performance or facilitate SEO.
>
> These targets require a different kind of bootstrap function that we'd import from a different library.
>
> **Why create a separate *main.ts* file?**
>
> The *main.ts* file is tiny. This is just a QuickStart. We could have folded its few lines into the `app.component` file and spared ourselves some complexity.
>
> We'd rather demonstrate the proper way to structure an Angular application. App bootstrapping is a separate concern from presenting a view. Mixing concerns creates difficulties down the road. We might launch the `AppComponent` in multiple environments with different bootstrappers. Testing the component is much easier if it doesn't also try to run the entire application. Let's make the small extra effort to do it *the right way*.

Add the `index.html`

The `index.html` is the web page that hosts the application

Navigate to the **project root folder**.

```
cd ..
```

Create an `index.html` file in this root folder and paste the following lines:

index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">

    <!-- 1. Load libraries -->
    <!-- IE required polyfills, in this exact order -->
    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/systemjs/dist/system-polyfills.js"></script>
    <script src="node_modules/angular2/es6/dev/src/testing/shims_for_IE.js"></script>

    <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>

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

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

There are three noteworthy sections of HTML

1. The JavaScript libraries
2. Configuration of SystemJS where we also import and run the `main` file that we just wrote.
3. The <my-app> tag in the `<body>` which is *where our app lives!*

Libraries

We loaded the following scripts

index.html

```
    <!-- IE required polyfills, in this exact order -->
    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/systemjs/dist/system-polyfills.js"></script>
    <script src="node_modules/angular2/es6/dev/src/testing/shims_for_IE.js"></script>

    <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>
```

We began with Internet Explorer polyfills. IE requires polyfills to run an application that relies on ES2015 promises and dynamic module loading. Most applications need those capabilities and most applications should run in Internet Explorer.

Next are the polyfills for Angular2, `angular2-polyfills.js`.

Then the SystemJS library for module loading, followed by the Reactive Extensions RxJS library.

> Our QuickStart doesn't use the Reactive Extensions but any substantial application will want them when working with observables. We added the library here in QuickStart so we don't forget later.

Finally, we loaded the web development version of Angular 2 itself.

We'll make different choices as we gain experience and become more concerned about production qualities such as load times and memory footprint.

SystemJS Configuration

The QuickStart uses SystemJS to load application and library modules. There are alternatives that work just fine including the well-regarded webpack. SystemJS happens to be a good choice but we want to be clear that it was a choice and not a preference.

All module loaders require configuration and all loader configuration becomes complicated rather quickly as soon as the file structure diversifies and we start thinking about building for production and performance.

We suggest becoming well-versed in the loader of your choice. Learn more about SystemJS configuration here.

With those cautions in mind, what are we doing in this QuickStart configuration?

index.html (System configuration)

```
<script>
  System.config({
    packages: {
      app: {
        format: 'register',
        defaultExtension: 'js'
      }
    }
  });
  System.import('app/main')
        .then(null, console.error.bind(console));
</script>
```

The `packages` node tells SystemJS what to do when it sees a request for a module from the `app/` folder.

Our QuickStart makes such requests when one of its application TypeScript files has an import statement like this:

main.ts (excerpt)

```
import {AppComponent} from './app.component'
```

Notice that the module name (after `from`) does not mention a filename extension. The `packages:` configuration tells SystemJS to default the extension to 'js', a JavaScript file.

That makes sense because we transpile TypeScript to JavaScript *before* running the application.

Transpiling in the browser

In the live example on plunker we transpile (AKA compile) to JavaScript in the browser on the fly. That's fine for a demo. That's not our preference for development or production.

We recommend transpiling (AKA compiling) to JavaScript during a build phase before running the application for several reasons including:

- We see compiler warnings and errors that are hidden from us in the browser.
- Pre-compilation simpifies the module loading process and it's much easier to diagnose problem when this is a separate, external step.
- Pre-compilation means a faster user experience because the browser doesn't waste time compiling.
- We iterate development faster because we only re-compile changed files. We notice the difference as soon as the app grows beyond a handful of files.
- Pre-compilation fits into a continuous integration process of build, test, deploy.

The `System.import` call tells SystemJS to import the `main` file (`main.js` ... after transpiling `main.ts`, remember?). `main` is where we tell Angular to launch the application. We also catch and log launch errors to the console.

All other modules are loaded upon request either by an import statement or by Angular itself.

<my-app>

When Angular calls the `bootstrap` function in `main.ts`, it reads the `AppComponent` metadata, finds the `my-app` selector, locates an element tag named `my-app`, and loads our application between those tags.

Compile and run!

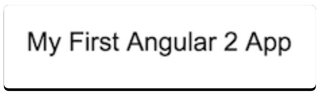
Open a terminal window and enter this command:

```
npm start
```

That command runs two parallel node processes

1. The TypeScript compiler in watch mode
2. A static server called **lite-server** that loads `index.html` in a browser and refreshes the browser when application files change

In a few moments, a browser tab should open and display


My First Angular 2 App

Congratulations! We are in business.

Make some changes

Try changing the message to "My SECOND Angular 2 app".

The TypeScript compiler and `lite-server` are watching. They should detect the change, recompile the TypeScript into JavaScript, refresh the browser, and display the revised message.

It's a nifty way to develop an application!

We close the terminal window when we're done to terminate both the compiler and the server.

Final structure

Our final project folder structure looks like this:

```
angular2-quickstart
├─ app
│  ├─ app.component.ts
│  └─ main.ts
├─ node_modules ...
├─ typings ...
├─ index.html
├─ package.json
├─ tsconfig.json
└─ typings.json
```

And here are the files:

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

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

Wrap Up

Our first application doesn't do much. It's basically "Hello, World" for Angular 2.

We kept it simple in our first pass: we wrote a little Angular component, we added some JavaScript libraries to `index.html`, and launched with a static file server. That's about all we'd expect to do for a "Hello, World" app.

We have greater ambitions.

The good news is that the overhead of setup is (mostly) behind us. We'll probably only touch the `package.json` to update libraries. We'll likely open `index.html` only if we need to add a library or some css stylesheets.

We're about to take the next step and build a small application that demonstrates the great things we can build with Angular 2.

Join us on the Tour of Heroes Tutorial!