QUICKSTART

TYPESCRIPT

This QuickStart guide demonstrates how to build and run a simple Angular application.

DON'T WANT TYPESCRIPT?

Although you're getting started in TypeScript, you can also write Angular applications in JavaScript and Dart. Use the language selector in the left nav to switch development languages for this guide.

Overview

The QuickStart application has the structure of a real-world Angular application and displays the simple message:

The live example link opens the finished application in Plunker so that you can interact with the code. You'll find live examples at the start of most sections.

Try it out. Here's a link to a live example.

You can also clone the entire QuickStart application from GitHub.

Build this application!

- Prerequisite: Install Node.js and npm.

- Step 1: Create and configure the project.
- Step 2: Create your application.
- Step 3: Create a component and add it to your application.
- Step 4: Start up your application.
- Step 5: Define the web page that hosts the application.
- Step 6: Build and run the application.
- Step 7: Make some live changes.
- Wrap up and Next Steps

Prerequisite: Install Node.js and npm

If Node.js and npm aren't already on your machine, install them. Our examples require node **v4.x.x** or higher and npm **3.x.x** or higher. To check which version you are using, run `node -v` and `npm -v` in a terminal window.

Step 1: Create and configure the project

In this step you will:

- Create the project folder
- Create configuration files
- Install packages

Create the project folder

Using a terminal window, create a directory for the project, and change into this directory.

```
mkdir angular-quickstart
cd    angular-quickstart
```

Create configuration files

Our typical Angular project needs several configuration files:

- **package.json** identifies npm package dependencies for the project.
- **tsconfig.json** defines how the TypeScript compiler generates JavaScript from the project's files.
- **typings.json** provides additional definition files for libraries that the TypeScript compiler doesn't natively recognize.
- **systemjs.config.js** provides information to a module loader about where to find application modules, and registers all the necessary packages. It also contains other packages that will be needed by later documentation examples.

Create each of these files in your project directory. Populate them by pasting in text from the tabs in the example box below.

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',

      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
```

```
        '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
        '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
        '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
        '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',

        // other libraries
        'rxjs':                      'npm:rxjs',
        'angular-in-memory-web-api': 'npm:angular-in-memory-web-api',
      },
      // packages tells the System loader how to load when no filename and/or no extension
      packages: {
        app: {
          main: './main.js',
          defaultExtension: 'js'
        },
        rxjs: {
          defaultExtension: 'js'
        },
        'angular-in-memory-web-api': {
          main: './index.js',
          defaultExtension: 'js'
        }
      }
    });
})(this);
```

Learn more about these configuration files in the Npm Package Configuration guide and the TypeScript Configuration guide. A detailed discussion of module loading is beyond the scope of this guide.

SYSTEMJS OR WEBPACK?

Although we use SystemJS for illustrative purposes here, it's only one option for loading modules. Use the module loader that you prefer. For Webpack and Angular, see Webpack: an Introduction. Or, learn more about SystemJS configuration in general here.

Install packages

Using npm from the command line, install the packages listed in `package.json` with the command:

```
npm install
```

Error messages—in red—might appear during the install, and you might see `npm WARN` messages. As long as there are no `npm ERR!` messages at the end, you can assume success.

You should now have the following structure:

```
angular-quickstart
 ├─node_modules ...
 ├─typings ...
 ├─package.json
 ├─systemjs.config.js
 ├─tsconfig.json
 └─typings.json
```

If the `typings` folder doesn't show up after running `npm install`, you'll need to install it manually with the command:

```
npm run typings install
```

You're now ready to write some code!

Step 2: Create your application

You compose Angular applications into closely related blocks of functionality with NgModules. Angular itself is split into separate Angular Modules. This makes it possible for you to keep payload size small by only importing the parts of Angular that your application needs.

Every Angular application has at least one module: the *root module*, named `AppModule` here.

Create an app subfolder off the project root directory:

```
mkdir app
```

Create the file `app/app.module.ts` with the following content:

app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ]
})
export class AppModule { }
```

This is the entry point to your application.

Since the QuickStart application is a web application that runs in a browser, your root module needs to import the `BrowserModule` from `@angular/platform-browser` to the `imports` array.

This is the smallest amount of Angular that is needed for a minimal application to run in the browser.

The QuickStart application doesn't do anything else, so you don't need any other modules. In a real application, you'd likely import `FormsModule` as well as

`RouterModule` and `HttpModule`. These are introduced in the Tour of Heroes Tutorial.

Step 3: Create a component and add it to your application

Every Angular application has at least one component: the *root component*, named `AppComponent` here.

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

Create the component file `app/app.component.ts` with the following content:

app/app.component.ts

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

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

The QuickStart application has the same essential structure as any other Angular component:

- **An import statement**. Importing gives your component access to Angular's core `@Component` decorator function.
- **A @Component decorator** that associates *metadata* with the `AppComponent` component class:
    - a *selector* that specifies a simple CSS selector for an HTML element that represents the component.
    - a *template* that tells Angular how to render the component's view.

- **A component class** that controls the appearance and behavior of a view through its template. Here, you only have the root component, `AppComponent`. Since you don't need any application logic in the simple QuickStart example, it's empty.

You *export* the `AppComponent` class so that you can *import* it into the application that you just created.

Edit the file `app/app.module.ts` to import your new `AppComponent` and add it in the declarations and bootstrap fields in the `NgModule` decorator:

app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

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

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

Step 4: Start up your application

Now you need to tell Angular to start up your application.

Create the file `app/main.ts` with the following content:

app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';
```

```
2.
3.  import { AppModule } from './app.module';
4.
5.  const platform = platformBrowserDynamic();
6.  platform.bootstrapModule(AppModule);
```

This code initializes the platform that your application runs in, then uses the platform to bootstrap your `AppModule`.

Why create separate *`main.ts`*, app module and app component files?

App bootstrapping is a separate concern from creating a module or presenting a view. Testing the component is much easier if it doesn't also try to run the entire application.

BOOTSTRAPPING IS PLATFORM-SPECIFIC

Because the QuickStart application runs directly in the browser, `main.ts` imports the `platformBrowserDynamic` function from `@angular/platform-browser-dynamic`, not `@angular/core`. On a mobile device, you might load a module with Apache Cordova or NativeScript, using a bootstrap function that's specific to that platform.

Step 5: Define the web page that hosts the application

In the *project root* folder, create an `index.html` file and paste the following lines into it:

index.html

```
1.  <html>
2.    <head>
3.      <title>Angular QuickStart</title>
4.      <meta charset="UTF-8">
5.      <meta name="viewport" content="width=device-width, initial-scale=1">
6.      <link rel="stylesheet" href="styles.css">
```

```
7.
8.       <!-- 1. Load libraries -->
9.        <!-- Polyfill(s) for older browsers -->
10.      <script src="node_modules/core-js/client/shim.min.js"></script>
11.
12.      <script src="node_modules/zone.js/dist/zone.js"></script>
13.      <script src="node_modules/reflect-metadata/Reflect.js"></script>
14.      <script src="node_modules/systemjs/dist/system.src.js"></script>
15.
16.      <!-- 2. Configure SystemJS -->
17.      <script src="systemjs.config.js"></script>
18.      <script>
19.        System.import('app').catch(function(err){ console.error(err); });
20.      </script>
21.    </head>
22.
23.    <!-- 3. Display the application -->
24.    <body>
25.      <my-app>Loading...</my-app>
26.    </body>
27.  </html>
```

The noteworthy sections here are:

- JavaScript libraries: `core-js` polyfills for older browsers, the `zone.js` and `reflect-metadata` libraries needed by Angular, and the `SystemJS` library for module loading.
- Configuration file for `SystemJS`, and a script where you import and run the `app` module which refers to the `main` file that you just wrote.
- The `<my-app>` tag in the `<body>` which is *where your app lives!*

Add some style

Styles aren't essential, but they're nice, and `index.html` assumes that you have a stylesheet called `styles.css`.

Create a `styles.css` file in the *project root* folder, and start styling, perhaps with the minimal styles shown below.

styles.css (excerpt)

```
/* Master Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
```

For the full set of master styles used by the documentation samples, see styles.css.

Step 6: Build and run the application

Open a terminal window and enter this command:

```
npm start
```

That command runs the following two parallel node processes:

- The TypeScript compiler in watch mode.

Read more about other useful npm scripts included in this example's `package.json`.

- A static file 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 the following:

Step 7: Make some live changes

Try changing the message in `app/app.component.ts` to "My SECOND Angular App".

The TypeScript compiler and `lite-server` will detect your change, recompile the TypeScript into JavaScript, refresh the browser, and display your revised message.

Close the terminal window when you're done to terminate both the compiler and the server.

Wrap up and next steps

The final project folder structure looks like this:

```
angular-quickstart
  app
    app.component.ts
    app.module.ts
    main.ts
  node_modules ...
  typings ...
  index.html
```

```
package.json
styles.css
systemjs.config.js
tsconfig.json
typings.json
```

To see the file contents, open the live example.

What next?

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

You wrote a little Angular component, created a simple `index.html`, and launched with a static file server.

You also created the basic application setup that you'll re-use for other sections in this guide. From here, the changes you'll make in the `package.json` or `index.html` files are only minor updates to add libraries or some css stylesheets. You also won't need to revisit module loading again.

To take the next step and build a small application that demonstrates real features that you can build with Angular, carry on to the Tour of Heroes tutorial!