5 Min Quickstart - js

Let's start from zero and build a super simple Angular 2 application in JavaScript.

Don't want JavaScript?
Although we're getting started in JavaScript, you can also write Angular 2 apps in TypeScript and Dart by selecting either of those languages from the combo-box in the banner.

See It Run!

Running the live example is the quickest way to see an Angular 2 app come to life.

Clicking that link fires up a browser, loads the sample in plunker, and displays a simple message:

Here is the file structure:

angular2-quickstart
app
index.html
license.md
Functionally, it's an `index.html` and two JavaScript files in an `app/` folder. We can handle that!

Of course we won't build many apps that only run in plunker. Let's follow a process that's closer to what we'd do in real life.

1. Set up our development environment
2. Write the Angular root component for our app
3. Bootstrap it to take control of the main web page
4. Write the main page (`index.html`)

We really can build the QuickStart from scratch in five minutes if we follow the instructions and ignore the commentary.

Most of us will be interested in the "why" as well as the "how" and that will take longer.

Development Environment

We'll need a place to stand (the application project folder), some libraries and your editor of choice.

Create a new project folder

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

Add the libraries we need

We recommend the **npm** package manager for acquiring and managing our development libraries.

Don't have npm? Get it now because we're going to use it now and repeatedly throughout this documentation.

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": "npm run lite",
    "lite": "lite-server"
  },
  "license": "ISC",
  "dependencies": {
    "@angular/common":  "2.0.0-rc.1",
    "@angular/compiler":  "2.0.0-rc.1",
    "@angular/core":  "2.0.0-rc.1",
    "@angular/http":  "2.0.0-rc.1",
    "@angular/platform-browser":  "2.0.0-rc.1",
    "@angular/platform-browser-dynamic":  "2.0.0-rc.1",
```

```
    "@angular/router":  "2.0.0-rc.1",
    "@angular/router-deprecated":  "2.0.0-rc.1",
    "@angular/upgrade":  "2.0.0-rc.1",

    "core-js": "^2.4.0",
    "reflect-metadata": "0.1.3",
    "rxjs": "5.0.0-beta.6",
    "zone.js": "0.6.12",

    "angular2-in-memory-web-api": "0.0.9",
    "bootstrap": "^3.3.6"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.0"
  }
}
```

Itching to know the details? We explain in the appendix below

Install these packages by opening a terminal window (command window in Windows) and running this npm command.

```
npm install
```

Scary error messages in red may appear **during** install. Ignore them. The install will succeed. See the appendix below for more information.

Our First Angular Component

The *Component* is the most fundamental of Angular concepts. A component manages a view - a piece of the web page where we display information to the user and respond to user feedback.

Technically, a component is a class that controls a view template. We'll write a lot of them as we build Angular apps. This is our first attempt so we'll keep it ridiculously simple.

Create an application source sub-folder

We like to keep our application code in a sub-folder off the root called `app/`. Execute the following command in the console window.

```
mkdir app
cd    app
```

Add the component file

Now add a file named **app.component.js** and paste the following lines:

app/app.component.js

```
(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 = {}));
```

We're creating a visual component named **`AppComponent`** by chaining the `Component` and `Class` methods that belong to the **global Angular core namespace, `ng.core`**.

app/app.component.js (component schema)

```
  app.AppComponent =
    ng.core.Component({
    })
    .Class({
    });
```

The **`Component`** method takes a configuration object with two properties. The **`Class`** method is where we implement the component itself, giving it properties and methods that bind to the view and whatever behavior is appropriate for this part of the UI.

Let's review this file in detail.

Modules

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

ES5 JavaScript doesn't have a native module system. There are several popular 3rd party module systems we could use. Instead, for simplicity and to avoid picking favorites, we'll create a single global namespace for our application.

We'll call it `app` and we'll add all of our code artifacts to this one global object.

We don't want to pollute the global namespace with anything else. So within each file we surround the code in an IIFE ("Immediately Invoked Function Expression").

app/app.component.js (IIFE)

```
(function(app) {
})(window.app || (window.app = {}));
```

We pass the global `app` namespace object into the IIFE, taking care to initialize it with an empty object if it doesn't yet exist.

Most application files *export* one thing by adding that thing to the `app` namespace. Our `app.component.js` file exports the `AppComponent`.

app/app.component.js (export)

```
app.AppComponent =
```

A more sophisticated application would have child components that descended from `AppComponent` in a visual tree. A more sophisticated app would have more files and modules, at least as many as it had components.

Quickstart isn't sophisticated; one component is all we need. Yet modules play a fundamental organizational role in even this small app.

Modules rely on other modules. In JavaScript Angular apps, when we need something provided by another module, we get it from the `app` object. When another module needs to refer to `AppComponent`, it gets it from the `app.AppComponent` like this:

app/main.js (import)

```
ng.platformBrowserDynamic.bootstrap(app.AppComponent);
```

Angular is also modular. It is a collection of library modules. Each library is itself a module made up of several, related feature modules.

When we need something from Angular, we use the `ng` object.

The Class definition object

At the bottom of the file is an empty, do-nothing class definition object for our `AppComponent` class. When we're ready to build a substantive application, we can expand this object with properties and application logic. Our `AppComponent` class has nothing but an empty constructor because we don't need it to do anything in this QuickStart.

app.component.js (class)

```
    .Class({
      constructor: function() {}
    });
```

The Component definition object

`ng.core.Component()` tells Angular that this class definition object *is an Angular component*. The configuration object passed to the `ng.core.Component()` method has two fields, a `selector` and a `template`.

app.component.js (component)

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

The `selector` specifies a simple CSS selector for a host HTML element named `my-app`. Angular creates and displays an instance of our `AppComponent` wherever it encounters a `my-app` element in the host HTML.

Remember the `my-app` selector! We'll need that information when we write our `index.html`

The `template` property holds the component's companion template. A template is a form of HTML that tells Angular how to render a view. Our template is a single line of HTML announcing "My First Angular 2 App".

Now we need something to tell Angular to load this component.

Bootstrap it!

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

app/main.js

```
(function(app) {
  document.addEventListener('DOMContentLoaded', function() {
    ng.platformBrowserDynamic.bootstrap(app.AppComponent);
  });
})(window.app || (window.app = {}));
```

We need two things to launch the application:

1. Angular's browser `bootstrap` function
2. The application root component that we just wrote.

We have them both in our 'namespaces'. Then we call `bootstrap`, passing in the **root component type**, `AppComponent`.

Learn why we need `bootstrap` from `ng.platformBrowserDynamic` and why we create a separate *main.js* file in the appendix below.

We've asked Angular to launch the app in a browser with our component at the root. Where will Angular put it?

Add the `index.html`

Angular displays our application in a specific location on our `index.html`. It's time to create that file.

We won't put our `index.html` in the `app/` folder. We'll locate it **up one level, in the project root folder**.

```
cd ..
```

Now create the`index.html` file and paste the following lines:

index.html

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

    <!-- 1. Load libraries -->
    <!-- IE required polyfill -->
    <script src="node_modules/core-js/client/shim.min.js">
</script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js">
</script>

    <script src="node_modules/rxjs/bundles/Rx.umd.js"></script>
    <script src="node_modules/@angular/core/core.umd.js">
</script>
    <script src="node_modules/@angular/common/common.umd.js">
</script>
    <script
src="node_modules/@angular/compiler/compiler.umd.js"></script>
    <script src="node_modules/@angular/platform-
browser/platform-browser.umd.js"></script>
    <script src="node_modules/@angular/platform-browser-
dynamic/platform-browser-dynamic.umd.js"></script>

    <!-- 2. Load our 'modules' -->
```

```
    <script src='app/app.component.js'></script>
    <script src='app/main.js'></script>

  </head>

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

</html>
```

There are three noteworthy sections of HTML:

1. We load the JavaScript libraries we need; learn about them below.
2. We load our JavaScript files, paying attention to their order (`main.js` needs `app.component.js` to be there first).
3. We add the `<my-app>` tag in the `<body>`. **This is where our app lives!**

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

Run!

Open a terminal window and enter this command:

```
npm start
```

That command runs 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

Congratulations! We are in business.

Make some changes

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

`lite-server` is watching, so it should detect the change, 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 the server.

Final structure

Our final project folder structure looks like this:

angular2-quickstart
And here are the files:

```
1. (function(app) {
2.   app.AppComponent =
3.     ng.core.Component({
4.       selector: 'my-app',
5.       template: '<h1>My First Angular 2 App</h1>'
6.     })
7.     .Class({
8.       constructor: function() {}
9.     });
10. })(window.app || (window.app = {}));
```

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. Besides adding in the script files for our app 'modules', 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!

Appendices

The balance of this chapter is a set of appendices that elaborate some of the points we covered quickly above.

There is no essential material here. Continued reading is for the curious.

Appendix: Libraries

We loaded the following scripts

index.html

```
    <!-- IE required polyfill -->
    <script src="node_modules/core-js/client/shim.min.js">
</script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js">
</script>

    <script src="node_modules/rxjs/bundles/Rx.umd.js"></script>
    <script src="node_modules/@angular/core/core.umd.js">
</script>
    <script src="node_modules/@angular/common/common.umd.js">
</script>
```

```
    <script
src="node_modules/@angular/compiler/compiler.umd.js"></script>
    <script src="node_modules/@angular/platform-
browser/platform-browser.umd.js"></script>
    <script src="node_modules/@angular/platform-browser-
dynamic/platform-browser-dynamic.umd.js"></script>
```

We began with an Internet Explorer polyfill. IE requires a polyfill 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, `zone.js` and `Reflect.js`, 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.

Appendix: package.json

npm is a popular package manager and Angular application developers rely on it to acquire and manage the libraries their apps require.

We specify the packages we need in an npm package.json file.

The Angular team suggests the packages listed in the `dependencies` and `devDependencies` sections listed in this file:

package.json (dependencies)

```
{
  "dependencies": {
    "@angular/common": "2.0.0-rc.1",
    "@angular/compiler": "2.0.0-rc.1",
    "@angular/core": "2.0.0-rc.1",
```

```
    "@angular/http": "2.0.0-rc.1",
    "@angular/platform-browser": "2.0.0-rc.1",
    "@angular/platform-browser-dynamic": "2.0.0-rc.1",
    "@angular/router": "2.0.0-rc.1",
    "@angular/router-deprecated": "2.0.0-rc.1",
    "@angular/upgrade": "2.0.0-rc.1",
    "core-js": "^2.4.0",
    "reflect-metadata": "0.1.3",
    "rxjs": "5.0.0-beta.6",
    "zone.js": "0.6.12",
    "angular2-in-memory-web-api": "0.0.9",
    "bootstrap": "^3.3.6"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.0"
  }
}
```

There are other possible package choices. We're recommending this particular set that we know work well together. Play along with us for now. Feel free to make substitutions later to suit your tastes and experience.

A `package.json` has an optional **scripts** section where we can define helpful commands to perform development and build tasks. We've included a number of such scripts in our suggested `package.json`:

package.json (scripts)

```
{
  "scripts": {
    "start": "npm run lite",
    "lite": "lite-server"
  }
}
```

We've seen how we can run the server with this command:

```
npm start
```

We are using the special `npm start` command, but all it does is run `npm run lite`.

We execute npm scripts in that manner: `npm run` + *script-name*. Here's what these scripts do:

- `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.

Appendix: 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 very end of `npm install`.

Appendix: *main.js*

Bootstrapping is platform-specific

We use the `bootstrap` function from `ng.platformBrowserDynamic`, not `ng.core`. There's a good reason.

We only call "core" those capabilities that are the same across all platform targets. True, most Angular applications run only in a browser and we'll call the bootstrap function from this library most of the time. It's pretty "core" if we're always writing for a browser.

But it is possible to load a component in a different enviroment. 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 do we create a separate *main.js* file?

The *main.js* file is tiny. This is just a QuickStart. We could have folded its few lines into the `app.component.js` file and spared ourselves some complexity.

We didn't for what we believe to be good reasons:

1. Doing it right is easy
2. Testability
3. Reusability
4. Separation of concerns
5. We learned about import and export

It's easy

Sure it's an extra step and an extra file. How hard is that in the scheme of things?

We'll see that a separate `main.js` is beneficial for *most* apps even if it isn't critical for the QuickStart. Let's develop good habits now while the cost is low.

Testability

We should be thinking about testability from the beginning even if we know we'll never test the QuickStart.

It is difficult to unit test a component when there is a call to `bootstrap` in the same file. As soon as we load the component file to test the component, the `bootstrap` function tries to load the application in the browser. It throws an error because we're not expecting to run the entire application, just test the component.

Relocating the `bootstrap` function to `main.js` eliminates this spurious error and leaves us with a clean component module file.

Reusability

We refactor, rename, and relocate files as our application evolves. We can't do any of those things while the file calls `bootstrap`. we can't move it. We can't reuse the component in another application. We can't pre-render the component on the server for better performance.

Separation of concerns

A component's responsibility is to present and manage a view.

Launching the application has nothing to do with view management. That's a separate concern. The friction we're encountering in testing and reuse stems from this unnecessary mix of responsibilities.

Import/Export

While writing a separate `main.js` file we learned an essential Angular skill: how to 'export' from one 'module' and 'import' into another via our simple namespace abstraction. We'll do a lot of that as we learn more Angular.