Bootstrapping

An Angular Module (NgModule) class describes how the application parts fit together. Every application has at least one Angular Module, the *root* module that you bootstrap to launch the application. You can call the class anything you want. The conventional name is `AppModule`.

The **Angular CLI** produces a new project with the following minimal `AppModule`. You evolve this module as your application grows.

After the `import` statements, you come to a class adorned with the **`@NgModule`** *decorator*.

The `@NgModule` decorator identifies `AppModule` as an `NgModule` class. `@NgModule` takes a *metadata* object that tells Angular how to compile and launch the application.

The `@NgModule` properties for the minimal `AppModule` generated by the CLI are as follows:

- ***declarations*** — declares the application components. At the moment, there is only the `AppComponent`.
- ***imports*** — the `BrowserModule`, which this and every application must import in order to run the app in a browser.
- ***providers*** — there are none to start but you are likely to add some soon.
- ***bootstrap*** — the *root* `AppComponent` that Angular creates and inserts into the `index.html` host web page.

The Angular Modules (NgModules) guide dives deeply into the details of `@NgModule`. All you need to know at the moment is a few basics about these four properties.

{@a declarations}

The *declarations* array

You tell Angular which components belong to the `AppModule` by listing it in the module's `declarations` array. As you create more components, you'll add them to `declarations`.

You must declare *every* component in an Angular Module class. If you use a component without declaring it, you'll see a clear error message in the browser console.

You'll learn to create two other kinds of classes — directives and pipes — that you must also add to the `declarations` array.

Only _declarables_ — _components_, _directives_ and _pipes_ — belong in the `declarations` array. Do not put any other kind of class in `declarations`. Do _not_ declare `NgModule` classes. Do _not_ declare service classes. Do _not_ declare model classes.

{@a imports}

The *imports* array

Angular Modules are a way to consolidate features that belong together into discrete units. Many features of Angular itself are organized as Angular Modules. HTTP services are in the `HttpClientModule`. The router is in the `RouterModule`. Eventually you may create your own modules.

Add a module to the `imports` array when the application requires its features.

This application, like most applications, executes in a browser. Every application that executes in a browser needs the `BrowserModule` from `@angular/platform-browser`. So every such application includes the `BrowserModule` in its *root* `AppModule`'s `imports` array. Other guide pages will tell you when you need to add additional modules to this array.

Only `@NgModule` classes go in the `imports` array. Do not put any other kind of class in `imports`. The `import` statements at the top of the file and the NgModule's `imports` array are unrelated and have completely different jobs. The _JavaScript_ `import` statements give you access to symbols _exported_ by other files so you can reference them within _this_ file. You add `import` statements to almost every application file. They have nothing to do with Angular and Angular knows nothing about them. The _module's_ `imports` array appears _exclusively_ in the `@NgModule` metadata object. It tells Angular about specific _other_ Angular Modules—all of them classes decorated with `@NgModule`—that the application needs to function properly.

{@a providers}

The *providers* array

Angular apps rely on *dependency injection (DI)* to deliver services to various parts of the application.

Before DI can inject a service, it must create that service with the help of a *provider*. You can tell DI about a service's *provider* in a number of ways. Among the most popular ways is to register the service in the root `ngModule.providers` array, which will make that service available *everywhere*.

For example, a data service provided in the `AppModule`s *providers* can be injected into any component anywhere in the application.

You don't have any services to provide yet. But you will create some before long and you may chose to provide many of them here.

{@a bootstrap-array}

The *bootstrap* array

You launch the application by [*bootstrapping*](bootstrapping) the root `AppModule`. Among other things, the *bootstrapping* process creates the component(s) listed in the `bootstrap` array and inserts each one into the browser DOM.

Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.

While you can put more than one component tree on a host web page, that's not typical. Most applications have only one component tree and they bootstrap a single *root* component.

You can call the one *root* component anything you want but most developers call it `AppComponent`.

Which brings us to the *bootstrapping* process itself.

{@a main}

Bootstrap in *main.ts*

While there are many ways to bootstrap an application, most applications do so in the `src/main.ts` that is generated by the Angular CLI.

This code creates a browser platform for dynamic compilation and bootstraps the `AppModule` described above.

The *bootstrapping* process sets up the execution environment, digs the *root* `AppComponent` out of the module's `bootstrap` array, creates an instance of the component and inserts it within the element tag identified by the component's `selector`.

The `AppComponent` selector — here and in most documentation samples — is `app-root` so Angular looks for a `<app-root>` tag in the `index.html` like this one ...

<body> <app-root></app-root> </body>

... and displays the `AppComponent` there.

The `main.ts` file is very stable. Once you've set it up, you may never change it again.

More about Angular Modules

Your initial app has only a single module, the *root* module. As your app grows, you'll consider subdividing it into multiple "feature" modules, some of which can be loaded later ("lazy loaded") if and when the user chooses to visit those features.

When you're ready to explore these possibilities, visit the Angular Modules guide.