Launching your app with a root module

Prerequisites

A basic understanding of the following:

- JavaScript Modules vs. NgModules.

An NgModule describes how the application parts fit together. Every application has at least one Angular module, the *root* module, which must be present for bootstrapping the application on launch. By convention and by default, this NgModule is named `AppModule`.

When you use the Angular CLI command `ng new` to generate an app, the default `AppModule` looks like the following:

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

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

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

After the import statements is a class 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.

- ***declarations***—this application's lone component.
- ***imports***—import `BrowserModule` to have browser specific services such as DOM rendering, sanitization, and location.
- ***providers***—the service providers.
- ***bootstrap***—the *root* component that Angular creates and inserts into the `index.html` host web page.

The default application created by the Angular CLI only has one component, `AppComponent`, so it is in both the `declarations` and the `bootstrap` arrays.

{@a declarations}

The `declarations` array

The module's `declarations` array tells Angular which components belong to that module. As you create more components, add them to `declarations`.

You must declare every component in exactly one `NgModule` class. If you use a component without declaring it, Angular returns an error message.

The `declarations` array only takes declarables. Declarables are components, directives and pipes. All of a module's declarables must be in the `declarations` array. Declarables must belong to exactly one module. The compiler emits an error if you try to declare the same class in more than one module.

These declared classes are visible within the module but invisible to components in a different module unless they are exported from this module and the other module imports this one.

An example of what goes into a declarations array follows:

```
declarations: [
  YourComponent,
  YourPipe,
  YourDirective
],
```

A declarable can only belong to one module, so only declare it in one `@NgModule`. When you need it elsewhere, import the module that has the declarable you need in it.

Using directives with `@NgModule`

Use the `declarations` array for directives. To use a directive, component, or pipe in a module, you must do a few things:

1. Export it from the file where you wrote it.
2. Import it into the appropriate module.
3. Declare it in the `@NgModule` `declarations` array.

Those three steps look like the following. In the file where you create your directive, export it. The following example, named `ItemDirective` is the default directive structure that the CLI generates in its own file, `item.directive.ts`:

The key point here is that you have to export it so you can import it elsewhere. Next, import it into the `NgModule`, in this example `app.module.ts`, with a JavaScript import statement:

And in the same file, add it to the `@NgModule` `declarations` array:

Now you could use your `ItemDirective` in a component. This example uses `AppModule`, but you'd do it the same way for a feature module. For more about directives, see Attribute Directives and Structural Directives. You'd also use the same technique for pipes and components.

Remember, components, directives, and pipes belong to one module only. You only need to declare them once in your application because you share them by importing the necessary modules. This saves you time and helps keep your application lean.

{@a imports}

The `imports` array

The module's `imports` array appears exclusively in the `@NgModule` metadata object. It tells Angular about other NgModules that this particular module needs to function properly.

This list of modules are those that export components, directives, or pipes that component templates in this module reference. In this case, the component is `AppComponent`, which references components, directives, or pipes in `BrowserModule`, `FormsModule`, or `HttpClientModule`. A component template can reference another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module.

{@a bootstrap-array}

The `providers` array

The providers array is where you list the services the application needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading. For more information, see Providers.

The `bootstrap` array

The application launches by bootstrapping the root `AppModule`, which is also referred to as an `entryComponent`. 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, most applications have only one component tree and bootstrap a single root component.

This one root component is usually called `AppComponent` and is in the root module's `bootstrap` array.

In a situation where you want to bootstrap a component based on an API response, or you want to mount the `AppComponent` in a different DOM node that doesn't match the component selector, please refer to `ApplicationRef.bootstrap()` documentation.

More about Angular Modules

For more on NgModules you're likely to see frequently in applications, see Frequently Used Modules.