Introduction to modules

Angular apps are modular and Angular has its own modularity system called *NgModules*. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, the *root module*, which is conventionally named `AppModule` and resides in a file named `app.module.ts`. You launch your app by *bootstrapping* the root NgModule.

While a small application might have only one NgModule, most apps have many more *feature*

modules. The *root* NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

NgModule metadata

An NgModule is defined by a class decorated with `@NgModule()`. The `@NgModule()` decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

- `declarations`: The components, *directives*, and *pipes* that belong to this NgModule.
- `exports`: The subset of declarations that should be visible and usable in the *component templates* of other NgModules.
- `imports`: Other modules whose exported classes are needed by component templates declared in *this* NgModule.
- `providers`: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
- `bootstrap`: The main application view, called the *root component*, which hosts all other app views. Only the *root NgModule* should set the `bootstrap` property.

Here's a simple root NgModule definition.

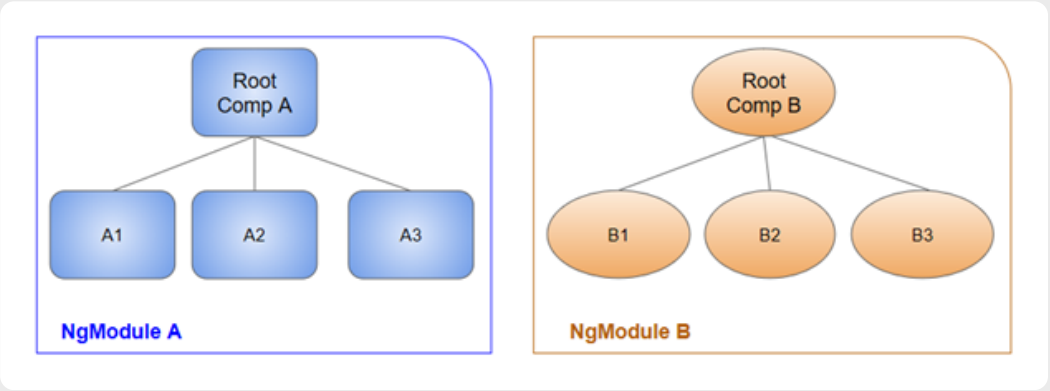
src/app/app.module.ts

```
import { NgModule }      from
'@angular/core';
import { BrowserModule } from
'@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

`AppComponent` is included in the `exports` list here for illustration; it isn't actually necessary in this example. A root NgModule has no reason to *export* anything because other modules don't need to *import* the root NgModule.

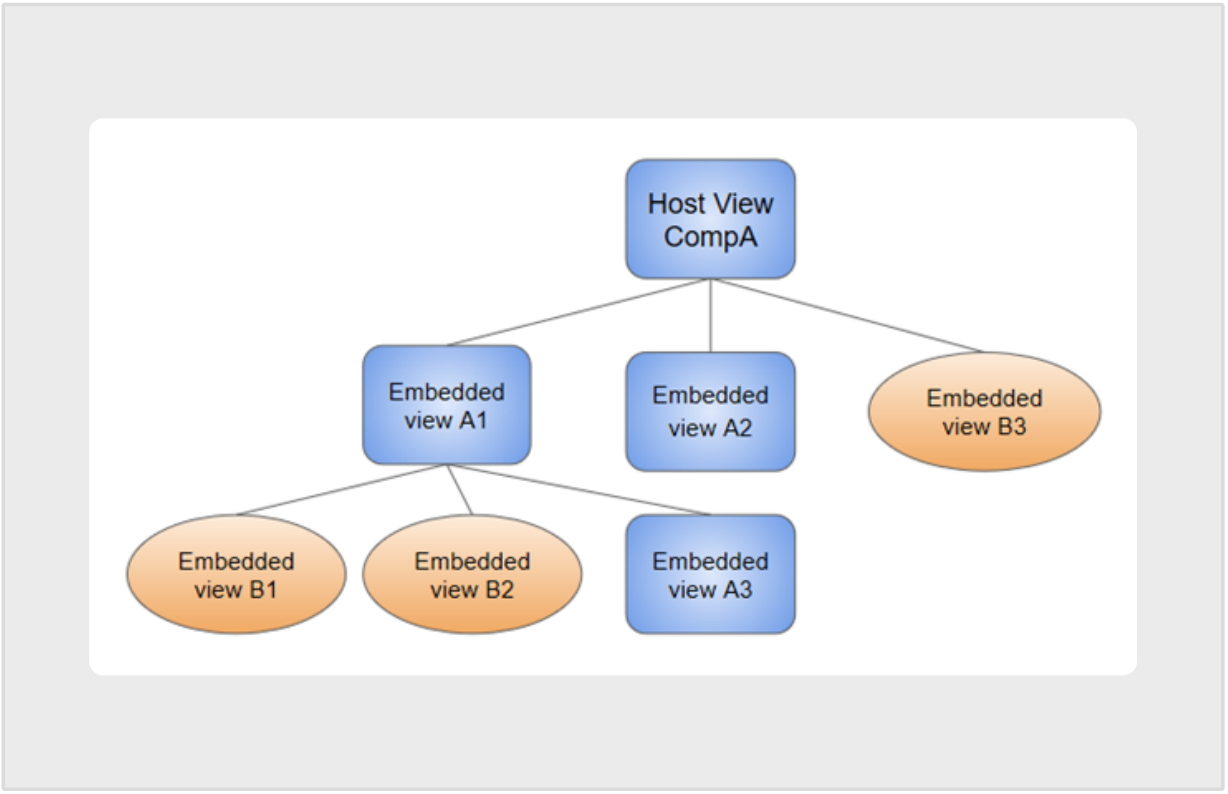
NgModules and components

NgModules provide a *compilation context* for their components. A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components, which can be loaded through the router or created through the template. The components that belong to an NgModule share a compilation context.



A component and its template together define a *view*. A component can contain a *view hierarchy*, which allows you to define arbitrarily complex areas of the

screen that can be created, modified, and destroyed as a unit. A view hierarchy can mix views defined in components that belong to different NgModules. This is often the case, especially for UI libraries.



When you create a component, it's associated directly with a single view, called the *host view*. The host view can be the root of a view hierarchy, which can contain *embedded views*, which are in turn the host views of other components. Those components can be in the same NgModule, or can be imported from other NgModules. Views in the tree can be nested to any depth.

> **Note:** The hierarchical structure of views is a key factor in the way Angular detects and responds to changes in the DOM and app data.

NgModules and JavaScript modules

The NgModule system is different from and unrelated to the JavaScript (ES2015) module system for managing collections of JavaScript objects. These are *complementary* module systems that you can use together to write your apps.

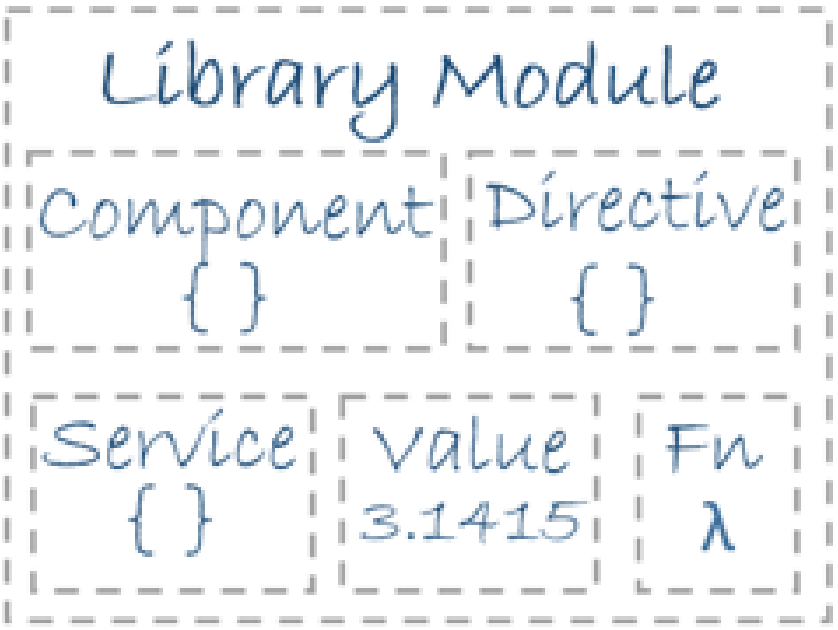
In JavaScript each *file* is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the `export` key word. Other JavaScript modules use *import statements* to access public objects from other modules.

```
import { NgModule }      from
'@angular/core';
import { AppComponent } from
'./app.component';
```

```
export class AppModule { }
```

> Learn more about the JavaScript module system on the web.

Angular libraries



Angular loads as a collection of JavaScript modules. You can think of them as library modules. Each Angular library name begins with the `@angular` prefix. Install them with the node package manager `npm` and import parts of them with JavaScript `import` statements.

For example, import Angular's `Component` decorator from the `@angular/core` library like this.

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

You also import NgModules from Angular *libraries* using JavaScript import statements. For example, the following code imports the `BrowserModule` NgModule from the `platform-browser` library.

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

In the example of the simple root module above, the application module needs material from within `BrowserModule`. To access that material, add it to the `@NgModule` metadata `imports` like this.

```
imports:      [ BrowserModule ],
```

In this way you're using the Angular and JavaScript module systems *together*. Although it's easy to confuse the two systems, which share the common vocabulary of "imports" and "exports", you will become familiar with the different contexts in which they are used.

Learn more from the NgModules guide.