NgModule API

Contents

At a high level, NgModules are a way to organize Angular apps and they accomplish this through the metadata in the `@NgModule` decorator. The metadata falls into three categories:

- **Static:** Compiler configuration which tells the compiler about directive selectors and where in templates the directives should be applied through selector matching. This is configured via the `declarations` array.
- **Runtime:** Injector configuration via the `providers` array.
- **Composability/Grouping:** Bringing NgModules together and making them available via the `imports` and `exports` arrays.

```
@NgModule({
  // Static, that is compiler configuration
  declarations: [], // Configure the selectors
  entryComponents: [], // Generate the host factory

  // Runtime, or injector configuration
  providers: [], // Runtime injector configuration

  // Composability / Grouping
  imports: [], // composing NgModules together
  exports: [] // making NgModules available to other parts of the app
})
```

`@NgModule` metadata

The following table summarizes the `@NgModule` metadata properties.

Property	Description
`declarations`	A list of declarable classes, (*components*, *directives*, and *pipes*) that *belong to this module*. 1. When compiling a template, you need to determine a set of selectors which should be used for triggering their corresponding directives.

2. The template is compiled within the context of an NgModule—the NgModule within which the template's component is declared—which determines the set of selectors using the following rules:
    - All selectors of directives listed in `declarations`.
    - All selectors of directives exported from imported NgModules.

Components, directives, and pipes must belong to *exactly* one module. The compiler emits an error if you try to declare the same class in more than one module. Be careful not to re-declare a class that is imported directly or indirectly from another module.

`providers`

A list of dependency-injection providers.

Angular registers these providers with the NgModule's injector. If it is the NgModule used for bootstrapping then it is the root injector.

These services become available for injection into any component, directive, pipe or service which is a child of this injector.

A lazy-loaded module has its own injector which is typically a child of the application root injector.

Lazy-loaded services are scoped to the lazy module's injector. If a lazy-loaded module also provides the `UserService`, any component created within that module's context (such as by router navigation) gets the local instance of the service, not the instance in the root application injector.

Components in external modules continue to receive the instance provided by their injectors.

For more information on injector hierarchy and scoping, see Providers and the DI Guide.

`imports`

A list of modules which should be folded into this module. Folded means it is as if all the imported NgModule's exported properties were declared here.

Specifically, it is as if the list of modules whose exported components, directives, or pipes are referenced by the component templates were declared in this module.

A component template can reference another component, directive, or pipe when the reference is declared in this module or if the imported module has exported it. For example, a component can use the `NgIf` and `NgFor` directives only if the module has imported the Angular `CommonModule` (perhaps indirectly by importing `BrowserModule`).

You can import many standard directives from the `CommonModule` but some familiar directives belong to other modules. For example, you can use `[(ngModel)]` only after importing the Angular `FormsModule`.

`exports`	A list of declarations—*component*, *directive*, and *pipe* classes—that an importing module can use. Exported declarations are the module's *public API*. A component in another module can use *this* module's `UserComponent` if it imports this module and this module exports `UserComponent`. Declarations are private by default. If this module does *not* export `UserComponent`, then only the components within *this* module can use `UserComponent`. Importing a module does *not* automatically re-export the imported module's imports. Module 'B' can't use `ngIf` just because it imported module 'A' which imported `CommonModule`. Module 'B' must import `CommonModule` itself. A module can list another module among its `exports`, in which case all of that module's public components, directives, and pipes are exported. Re-export makes module transitivity explicit. If Module 'A' re-exports `CommonModule` and Module 'B' imports Module 'A', Module 'B' components can use `ngIf` even though 'B' itself didn't import `CommonModule`.
`bootstrap`	A list of components that are automatically bootstrapped. Usually there's only one component in this list, the *root component* of the application. Angular can launch with multiple bootstrap components, each with its own location in the host web page. A bootstrap component is automatically added to `entryComponents`.
`entryComponents`	A list of components that can be dynamically loaded into the view. By default, an Angular app always has at least one entry component, the root component, `AppComponent`. Its purpose is to serve as a point of entry into the app, that is, you bootstrap it to launch the app. Routed components are also *entry components* because they need to be loaded dynamically. The router creates them and drops them into the DOM near a `<router-outlet>`. While the bootstrapped and routed components are *entry components*, you don't have to add them to a module's `entryComponents` list, as they are added implicitly. Angular automatically adds components in the module's `bootstrap` and route definitions into the `entryComponents` list. That leaves only components bootstrapped using one of the imperative techniques, such as `ViewComponentRef.createComponent()` as undiscoverable. Dynamic component loading is not common in most apps beyond the router. If you need to dynamically load components, you must add these components to the

`entryComponents` list yourself.

For more information, see Entry Components.

More on NgModules

You may also be interested in the following:

- Feature Modules.
- Entry Components.
- Providers.
- Types of Feature Modules.

Version 10.0.10-local+sha.84d1ba792b.