NgModule FAQ

Contents ›

NgModules help organize an application into cohesive blocks of functionality.

This page answers the questions many developers ask about NgModule design and implementation.

What classes should I add to the `declarations` array?

Add declarable classes—components, directives, and pipes—to a `declarations` list.

Declare these classes in *exactly one* module of the application. Declare them in a module if they belong to that particular module.

What is a *declarable*?

Declarables are the class types—components, directives, and pipes—that you can add to a module's `declarations` list. They're the only classes that you can add to `declarations`.

What classes should I *not* add to `declarations`?

Add only declarable classes to an NgModule's `declarations` list.

Do *not* declare the following:

- A class that's already declared in another module, whether an app module, @NgModule, or third-party module.
- An array of directives imported from another module. For example, don't declare `FORMS_DIRECTIVES` from `@angular/forms` because the `FormsModule` already declares it.
- Module classes.
- Service classes.
- Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.

Why list the same component in multiple `NgModule` properties?

`AppComponent` is often listed in both `declarations` and `bootstrap`. You might see the same component listed in `declarations`, `exports`, and `entryComponents`.

While that seems redundant, these properties have different functions. Membership in one list doesn't imply membership in another list.

- `AppComponent` could be declared in this module but not bootstrapped.
- `AppComponent` could be bootstrapped in this module but declared in a different feature module.
- A component could be imported from another app module (so you can't declare it) and re-exported by this module.
- A component could be exported for inclusion in an external component's template as well as dynamically loaded in a pop-up dialog.

What does "Can't bind to 'x' since it isn't a known property of 'y'" mean?

This error often means that you haven't declared the directive "x" or haven't imported the NgModule to which "x" belongs.

> Perhaps you declared "x" in an application sub-module but forgot to export it. The "x" class isn't visible to other modules until you add it to the `exports` list.

What should I import?

Import NgModules whose public (exported) declarable classes you need to reference in this module's component templates.

This always means importing `CommonModule` from `@angular/common` for access to the Angular directives such as `NgIf` and `NgFor`. You can import it directly or from another NgModule that re-exports it.

Import `FormsModule` from `@angular/forms` if your components have `[(ngModel)]` two-way binding expressions.

Import *shared* and *feature* modules when this module's components incorporate their components, directives, and pipes.

Import BrowserModule only in the root `AppModule`.

Should I import `BrowserModule` or `CommonModule`?

The root application module, `AppModule`, of almost every browser application should import `BrowserModule` from `@angular/platform-browser`.

`BrowserModule` provides services that are essential to launch and run a browser app.

`BrowserModule` also re-exports `CommonModule` from `@angular/common`, which means that components in the `AppModule` module also have access to the Angular directives every app needs, such as `NgIf` and `NgFor`.

Do not import `BrowserModule` in any other module. *Feature modules* and *lazy-loaded modules* should import `CommonModule` instead. They need the common directives. They don't need to re-install the app-wide providers.

Importing `CommonModule` also frees feature modules for use on *any* target platform, not just browsers.

What if I import the same module twice?

That's not a problem. When three modules all import Module 'A', Angular evaluates Module 'A' once, the first time it encounters it, and doesn't do so again.

That's true at whatever level `A` appears in a hierarchy of imported NgModules. When Module 'B' imports Module 'A', Module 'C' imports 'B', and Module 'D' imports `[C, B, A]`, then 'D' triggers the evaluation of 'C', which triggers the evaluation of 'B', which evaluates 'A'. When Angular gets to the 'B' and 'A' in 'D', they're already cached and ready to go.

Angular doesn't like NgModules with circular references, so don't let Module 'A' import Module 'B', which imports Module 'A'.

What should I export?

Export declarable classes that components in *other* NgModules are able to reference in their templates. These are your *public* classes. If you don't export a declarable class, it stays *private*, visible only to other components declared in this NgModule.

You *can* export any declarable class—components, directives, and pipes—whether it's declared in this NgModule or in an imported NgModule.

You *can* re-export entire imported NgModules, which effectively re-exports all of their exported classes. An NgModule can even export a module that it doesn't import.

What should I *not* export?

Don't export the following:

- Private components, directives, and pipes that you need only within components declared in this NgModule. If you don't want another NgModule to see it, don't export it.
- Non-declarable objects such as services, functions, configurations, and entity models.
- Components that are only loaded dynamically by the router or by bootstrapping. Such entry components can never be selected in another component's template. While there's no harm in exporting them, there's also no benefit.
- Pure service modules that don't have public (exported) declarations. For example, there's no point in re-exporting `HttpClientModule` because it doesn't export anything. Its only purpose is to add http service providers to the application as a whole.

Can I re-export classes and modules?

Absolutely.

NgModules are a great way to selectively aggregate classes from other NgModules and re-export them in a consolidated, convenience module.

An NgModule can re-export entire NgModules, which effectively re-exports all of their exported classes. Angular's own `BrowserModule` exports a couple of NgModules like this:

```
exports: [CommonModule, ApplicationModule]
```

An NgModule can export a combination of its own declarations, selected imported classes, and imported NgModules.

Don't bother re-exporting pure service modules. Pure service modules don't export declarable classes that another NgModule could use. For example, there's no point in re-exporting `HttpClientModule` because it doesn't export anything. Its only purpose is to add http service providers to the application as a whole.

What is the `forRoot()` method?

The `forRoot()` static method is a convention that makes it easy for developers to configure services and providers that are intended to be singletons. A good example of `forRoot()` is the `RouterModule.forRoot()` method.

Apps pass a `Routes` object to `RouterModule.forRoot()` in order to configure the app-wide `Router` service with routes. `RouterModule.forRoot()` returns a ModuleWithProviders. You add that result to the `imports` list of the root `AppModule`.

Only call and import a `forRoot()` result in the root application module, `AppModule`. Avoid importing it in any other module, particularly in a lazy-loaded module. For more information on `forRoot()` see the `forRoot()` pattern section of the Singleton Services guide.

For a service, instead of using `forRoot()`, specify `providedIn: 'root'` on the service's `@Injectable()` decorator, which makes the service automatically available to the whole application and thus singleton by default.

`RouterModule` also offers a `forChild()` static method for configuring the routes of lazy-loaded modules.

`forRoot()` and `forChild()` are conventional names for methods that configure services in root and feature modules respectively.

Follow this convention when you write similar modules with configurable service providers.

Why is a service provided in a feature module visible everywhere?

Providers listed in the `@NgModule.providers` of a bootstrapped module have application scope. Adding a service provider to `@NgModule.providers` effectively publishes the service to the entire application.

When you import an NgModule, Angular adds the module's service providers (the contents of its `providers` list) to the application root injector.

This makes the provider visible to every class in the application that knows the provider's lookup token, or name.

Extensibility through NgModule imports is a primary goal of the NgModule system. Merging NgModule providers into the application injector makes it easy for a module library to enrich the entire application with new services. By adding the `HttpClientModule` once, every application component can make HTTP requests.

However, this might feel like an unwelcome surprise if you expect the module's services to be visible only to the components declared by that feature module. If the `HeroModule` provides the `HeroService` and the root `AppModule` imports `HeroModule`, any class that knows the `HeroService` *type* can inject that service, not just the classes declared in the `HeroModule`.

To limit access to a service, consider lazy loading the NgModule that provides that service. See How do I restrict service scope to a module? for more information.

Why is a service provided in a lazy-loaded module visible only to that module?

Unlike providers of the modules loaded at launch, providers of lazy-loaded modules are *module-scoped*.

When the Angular router lazy-loads a module, it creates a new execution context. That context has its own injector, which is a direct child of the application injector.

The router adds the lazy module's providers and the providers of its imported NgModules to this child injector.

These providers are insulated from changes to application providers with the same lookup token. When the router creates a component within the lazy-loaded context, Angular prefers service instances created from these providers to the service instances of the application root injector.

What if two modules provide the same service?

When two imported modules, loaded at the same time, list a provider with the same token, the second module's provider "wins". That's because both providers are added to the same injector.

When Angular looks to inject a service for that token, it creates and delivers the instance created by the second provider.

Every class that injects this service gets the instance created by the second provider. Even classes declared within the first module get the instance created by the second provider.

If NgModule A provides a service for token 'X' and imports an NgModule B that also provides a service for token 'X', then NgModule A's service definition "wins".

The service provided by the root `AppModule` takes precedence over services provided by imported NgModules. The `AppModule` always wins.

How do I restrict service scope to a module?

When a module is loaded at application launch, its `@NgModule.providers` have *application-wide scope*; that is, they are available for injection throughout the application.

Imported providers are easily replaced by providers from another imported NgModule. Such replacement might be by design. It could be unintentional and have adverse consequences.

As a general rule, import modules with providers *exactly once*, preferably in the application's *root module*. That's also usually the best place to configure, wrap, and override them.

Suppose a module requires a customized `HttpBackend` that adds a special header for all Http requests. If another module elsewhere in the application also customizes `HttpBackend` or merely imports the `HttpClientModule`, it could override this module's `HttpBackend` provider, losing the special header. The server will reject http requests from this module.

To avoid this problem, import the `HttpClientModule` only in the `AppModule`, the application *root module*.

If you must guard against this kind of "provider corruption", *don't rely on a launch-time module's `providers`.*

Load the module lazily if you can. Angular gives a lazy-loaded module its own child injector. The module's providers are visible only within the component tree created with this injector.

If you must load the module eagerly, when the application starts, *provide the service in a component instead.*

Continuing with the same example, suppose the components of a module truly require a private, custom `HttpBackend`.

Create a "top component" that acts as the root for all of the module's components. Add the custom `HttpBackend` provider to the top component's `providers` list rather than the module's `providers`. Recall that Angular creates a child injector for each component instance and populates the injector with the component's own providers.

When a child of this component asks for the `HttpBackend` service, Angular provides the local `HttpBackend` service, not the version provided in the application root injector. Child components make proper HTTP requests no matter what other modules do to `HttpBackend`.

Be sure to create module components as children of this module's top component.

You can embed the child components in the top component's template. Alternatively, make the top component a routing host by giving it a `<router-outlet>`. Define child routes and let the router load module components into that outlet.

Though you can limit access to a service by providing it in a lazy loaded module or providing it in a component, providing services in a component can lead to multiple instances of those services. Thus, the lazy loading is preferable.

Should I add application-wide providers to the root `AppModule` or the root `AppComponent`?

Define application-wide providers by specifying `providedIn: 'root'` on its `@Injectable()` decorator (in the case of services) or at `InjectionToken` construction (in the case where tokens are provided). Providers that are created this way automatically are made available to the entire application and don't need to be listed in any module.

If a provider cannot be configured in this way (perhaps because it has no sensible default value), then register application-wide providers in the root `AppModule`, not in the `AppComponent`.

Lazy-loaded modules and their components can inject `AppModule` services; they can't inject `AppComponent` services.

Register a service in `AppComponent` providers *only* if the service must be hidden from components outside the `AppComponent` tree. This is a rare use case.

More generally, prefer registering providers in NgModules to registering in components.

Discussion

Angular registers all startup module providers with the application root injector. The services that root injector providers create have application scope, which means they are available to the entire application.

Certain services, such as the `Router`, only work when you register them in the application root injector.

By contrast, Angular registers `AppComponent` providers with the `AppComponent`'s own injector. `AppComponent` services are available only to that component and its component tree. They have component scope.

The `AppComponent`'s injector is a child of the root injector, one down in the injector hierarchy. For applications that don't use the router, that's almost the entire application. But in routed applications, routing operates at the root level where `AppComponent` services don't exist. This means that lazy-loaded modules can't reach them.

Should I add other providers to a module or a component?

Providers should be configured using `@Injectable` syntax. If possible, they should be provided in the application root (`providedIn: 'root'`). Services that are configured this way are lazily loaded if they are only used from a lazily loaded context.

If it's the consumer's decision whether a provider is available application-wide or not, then register providers in modules (`@NgModule.providers`) instead of registering in components (`@Component.providers`).

Register a provider with a component when you *must* limit the scope of a service instance to that component and its component tree. Apply the same reasoning to registering a provider with a directive.

For example, an editing component that needs a private copy of a caching service should register the service with the component. Then each new instance of the component gets its own cached service instance. The changes that editor makes in its service don't touch the instances elsewhere in the application.

Always register *application-wide* services with the root `AppModule`, not the root `AppComponent`.

Why is it bad if a shared module provides a service to a lazy-loaded module?

The eagerly loaded scenario

When an eagerly loaded module provides a service, for example a `UserService`, that service is available application-wide. If the root module provides `UserService` and imports another module that provides the same `UserService`, Angular registers one of them in the root app injector (see What if I import the same module twice?).

Then, when some component injects `UserService`, Angular finds it in the app root injector, and delivers the app-wide singleton service. No problem.

The lazy loaded scenario

Now consider a lazy loaded module that also provides a service called `UserService`.

When the router lazy loads a module, it creates a child injector and registers the `UserService` provider with that child injector. The child injector is *not* the root injector.

When Angular creates a lazy component for that module and injects `UserService`, it finds a `UserService` provider in the lazy module's *child injector* and creates a *new* instance of the `UserService`. This is an entirely different `UserService` instance than the app-wide singleton version that Angular injected in one of the eagerly loaded components.

This scenario causes your app to create a new instance every time, instead of using the singleton.

Why does lazy loading create a child injector?

Angular adds `@NgModule.providers` to the application root injector, unless the NgModule is lazy-loaded. For a lazy-loaded NgModule, Angular creates a *child injector* and adds the module's providers to the child injector.

This means that an NgModule behaves differently depending on whether it's loaded during application start or lazy-loaded later. Neglecting that difference can lead to adverse consequences.

Why doesn't Angular add lazy-loaded providers to the app root injector as it does for eagerly loaded NgModules?

The answer is grounded in a fundamental characteristic of the Angular dependency-injection system. An injector can add providers *until it's first used*. Once an injector starts creating and delivering services, its provider list is frozen; no new providers are allowed.

When an applications starts, Angular first configures the root injector with the providers of all eagerly loaded NgModules *before* creating its first component and injecting any of the provided services. Once the application begins, the app root injector is closed to new providers.

Time passes and application logic triggers lazy loading of an NgModule. Angular must add the lazy-loaded module's providers to an injector somewhere. It can't add them to the app root injector because that injector is closed to new providers. So Angular creates a new child injector for the lazy-loaded module context.

How can I tell if an NgModule or service was previously loaded?

Some NgModules and their services should be loaded only once by the root `AppModule`. Importing the module a second time by lazy loading a module could produce errant behavior that may be difficult to detect and diagnose.

To prevent this issue, write a constructor that attempts to inject the module or service from the root app injector. If the injection succeeds, the class has been loaded a second time. You can throw an error or take other remedial action.

Certain NgModules, such as `BrowserModule`, implement such a guard. Here is a custom constructor for an NgModule called `GreetingModule`.

src/app/greeting/greeting.module.ts (Constructor)

```
constructor(@Optional() @SkipSelf() parentModule?: GreetingModule) {
  if (parentModule) {
    throw new Error(
      'GreetingModule is already loaded. Import it in the AppModule only');
  }
}
```

What is an `entry component`?

An entry component is any component that Angular loads *imperatively* by type.

A component loaded *declaratively* via its selector is *not* an entry component.

Angular loads a component declaratively when using the component's selector to locate the element in the template. Angular then creates the HTML representation of the component and inserts it into the DOM at the selected element. These aren't entry components.

The bootstrapped root `AppComponent` is an *entry component*. True, its selector matches an element tag in `index.html`. But `index.html` isn't a component template and the `AppComponent` selector doesn't match an element in any component template.

Components in route definitions are also *entry components*. A route definition refers to a component by its *type*. The router ignores a routed component's selector, if it even has one, and loads the component dynamically into a `RouterOutlet`.

For more information, see Entry Components.

What's the difference between a *bootstrap* component and an *entry component*?

A bootstrapped component *is* an entry component that Angular loads into the DOM during the bootstrap process (application launch). Other entry components are loaded dynamically by other means, such as with the router.

The `@NgModule.bootstrap` property tells the compiler that this is an entry component *and* it should generate code to bootstrap the application with this component.

There's no need to list a component in both the `bootstrap` and `entryComponents` lists, although doing so is harmless.

For more information, see Entry Components.

When do I add components to *entryComponents*?

Most application developers won't need to add components to the `entryComponents`.

Angular adds certain components to *entry components* automatically. Components listed in `@NgModule.bootstrap` are added automatically. Components referenced in router configuration are added automatically. These two mechanisms account for almost all entry components.

If your app happens to bootstrap or dynamically load a component *by type* in some other manner, you must add it to `entryComponents` explicitly.

Although it's harmless to add components to this list, it's best to add only the components that are truly *entry components*. Don't include components that are referenced in the templates of other components.

For more information, see Entry Components.

Why does Angular need *entryComponents*?

The reason is *tree shaking*. For production apps you want to load the smallest, fastest code possible. The code should contain only the classes that you actually need. It should exclude a component that's never used, whether or not that component is declared.

In fact, many libraries declare and export components you'll never use. If you don't reference them, the tree shaker drops these components from the final code package.

If the Angular compiler generated code for every declared component, it would defeat the purpose of the tree shaker.

Instead, the compiler adopts a recursive strategy that generates code only for the components you use.

The compiler starts with the entry components, then it generates code for the declared components it finds in an entry component's template, then for the declared components it discovers in the templates of previously compiled components, and so on. At the end of the process, the compiler has generated code for every entry component and every component reachable from an entry component.

If a component isn't an *entry component* or wasn't found in a template, the compiler omits it.

What kinds of modules should I have and how should I use them?

Every app is different. Developers have various levels of experience and comfort with the available choices. Some suggestions and guidelines appear to have wide appeal.

SharedModule

`SharedModule` is a conventional name for an `NgModule` with the components, directives, and pipes that you use everywhere in your app. This module should consist entirely of `declarations`, most of them exported.

The `SharedModule` may re-export other widget modules, such as `CommonModule`, `FormsModule`, and NgModules with the UI controls that you use most widely.

The `SharedModule` should not have `providers` for reasons explained previously. Nor should any of its imported or re-exported modules have `providers`.

Import the `SharedModule` in your *feature* modules, both those loaded when the app starts and those you lazy load later.

Feature Modules

Feature modules are modules you create around specific application business domains, user workflows, and utility collections. They support your app by containing a particular feature, such as routes, services, widgets, etc. To conceptualize what a feature module might be in your app, consider that if you would put the files related to a certain functionality, like a search, in one folder, that the contents of that folder would be a feature module that you might call your `SearchModule`. It would contain all of the components, routing, and templates that would make up the search functionality.

For more information, see Feature Modules and Module Types

What's the difference between NgModules and JavaScript Modules?

In an Angular app, NgModules and JavaScript modules work together.

In modern JavaScript, every file is a module (see the Modules page of the Exploring ES6 website). Within each file you write an `export` statement to make parts of the module public.

An Angular NgModule is a class with the `@NgModule` decorator—JavaScript modules don't have to have the `@NgModule` decorator. Angular's `NgModule` has `imports` and `exports` and they serve a similar purpose.

You *import* other NgModules so you can use their exported classes in component templates. You *export* this NgModule's classes so they can be imported and used by components of *other* NgModules.

For more information, see JavaScript Modules vs. NgModules.

How does Angular find components, directives, and pipes in a template? What is a *template reference*?

The Angular compiler looks inside component templates for other components, directives, and pipes. When it finds one, that's a template reference.

The Angular compiler finds a component or directive in a template when it can match the *selector* of that component or directive to some HTML in that template.

The compiler finds a pipe if the pipe's *name* appears within the pipe syntax of the template HTML.

Angular only matches selectors and pipe names for classes that are declared by this module or exported by a module that this module imports.

What is the Angular compiler?

The Angular compiler converts the application code you write into highly performant JavaScript code. The `@NgModule` metadata plays an important role in guiding the compilation process.

The code you write isn't immediately executable. For example, components have templates that contain custom elements, attribute directives, Angular binding declarations, and some peculiar syntax that clearly isn't native HTML.

The Angular compiler reads the template markup, combines it with the corresponding component class code, and emits *component factories*.

A component factory creates a pure, 100% JavaScript representation of the component that incorporates everything described in its `@Component` metadata: the HTML, the binding instructions, the attached styles.

Because directives and pipes appear in component templates, the Angular compiler incorporates them into compiled component code too.

`@NgModule` metadata tells the Angular compiler what components to compile for this module and how to link this module with other modules.

Version 10.0.10-local+sha.84d1ba792b.