NgModule FAQs

NgModules help organize an application into cohesive blocks of functionality.

The NgModules guide takes you step-by-step from the most elementary `@NgModule` class to a multi-faceted sample with lazy-loaded modules.

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

These FAQs assume that you have read the [NgModules](guide/ngmodule) guide.

{@a q-what-to-declare}

What classes should I add to *declarations*?

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

Declare these classes in *exactly one* NgModule. Declare them in *this* NgModule if they *belong* to this module.

{@a q-declarable}

What is a *declarable*?

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

{@a q-what-not-to-declare}

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 NgModule.
- An array of directives imported from another NgModule. For example, don't declare FORMS_DIRECTIVES from `@angular/forms`.
- NgModule classes.

- Service classes.
- Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.

{@a q-why-multiple-mentions}

Why list the same component in multiple *@NgModule* properties?

`AppComponent` is often listed in both `declarations` and `bootstrap`. You might see `HeroComponent` 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.
- `HeroComponent` could be imported from another application module (so you can't declare it) and re-exported by this module.
- `HeroComponent` could be exported for inclusion in an external component's template as well as dynamically loaded in a pop-up dialog.

{@a q-why-cant-bind-to}

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.

You also get this error if "x" really isn't a property or if "x" is a private component property (i.e., lacks the `@Input` or `@Output` decorator).

For example, if "x" is `ngModel`, you may not have imported the `FormsModule` from `@angular/forms`.

Perhaps you declared "x" in an application feature module but forgot to export it? The "x" class isn't visible to other components of other NgModules until you add it to the `exports` list.

{@a q-what-to-import}

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 only BrowserModule in the root `AppModule`.

{@a q-browser-vs-common-module}

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 NgModule. *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.

`BrowserModule` throws an error if you try to lazy load a module that imports it.

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

{@a q-reimport}

What if I import the same NgModule twice?

That's not a problem. When three NgModules 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'.

{@a q-what-to-export}

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 class, it stays *private*, visible only to other component 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.

{@a q-what-not-to-export}

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 `HttpModule` because it doesn't export anything. It's only purpose is to add http service providers to the application as a whole.

{@a q-reexport} {@a q-re-export}

Can I re-export classes and NgModules?

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](guide/ngmodule-faq#q-declarable) classes that another NgModule could use. For example, there's no point in re-exporting `HttpModule` because it doesn't export anything. It's only purpose is to add http service providers to the application as a whole.

{@a q-for-root}

What is the *forRoot* method?

The `forRoot` static method is a convention that makes it easy for developers to configure the module's providers.

The `RouterModule.forRoot` method is a good example. 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 NgModule, `AppModule`. Importing it in any other NgModule, particularly in a lazy-loaded NgModule, is contrary to the intent and will likely produce a runtime error.

`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.

Angular doesn't recognize these names but Angular developers do. Follow this convention when you write similar modules with configurable service providers.

{@a q-module-provider-visibility}

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.

This is by design. 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 `HttpModule` 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`.

{@a q-lazy-loaded-module-provider-visibility}

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

Unlike providers of the NgModules 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.

{@a q-module-provider-duplicates}

What if two NgModules provide the same service?

When two imported NgModules, 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.

{@a q-component-scoped-providers}

How do I restrict service scope to an NgModule?

When an NgModule 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 NgModules 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 an NgModule requires a customized `HttpBackend` that adds a special header for all Http requests. If another NgModule elsewhere in the application also customizes `HttpBackend` or merely imports the `HttpModule`, 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 `HttpModule` 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 NgModules 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.

{@a q-root-component-or-module}

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

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 NgModule providers with the application root injector. The services created from root

injector providers are available to the entire application. They are *application-scoped*.

Certain services (such as the `Router`) only work when registered 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 are *component-scoped*.

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 for routed applications, "almost" isn't good enough.

`AppComponent` services don't exist at the root level where routing operates. Lazy-loaded modules can't reach them. In the *NgModules* sample application, if you had registered `UserService` in the `AppComponent`, the `HeroComponent` couldn't inject it. The application would fail the moment a user navigated to "Heroes".

{@a q-component-or-module}

Should I add other providers to an NgModule or a component?

In general, prefer registering feature-specific providers in NgModules (`@NgModule.providers`) to 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, a hero editing component that needs a private copy of a caching hero service should register the `HeroService` with the `HeroEditorComponent`. Then each new instance of the `HeroEditorComponent` gets its own cached service instance. The changes that editor makes to heroes in its service don't touch the hero instances elsewhere in the application.

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

{@a q-why-bad}

Why is it bad if *SharedModule* provides a service to a lazy-loaded NgModule?

This question is addressed in the Why UserService isn't shared section of the NgModules guide, which discusses the importance of keeping providers out of the `SharedModule`.

Suppose the `UserService` was listed in the NgModule's `providers` (which it isn't). Suppose every NgModule imports this `SharedModule` (which they all do).

When the app starts, Angular eagerly loads the `AppModule` and the `ContactModule`.

Both instances of the imported `SharedModule` would provide the `UserService`. Angular registers one of them in the root app injector (see What if I import the same NgModule twice?). Then some component injects `UserService`, Angular finds it in the app root injector, and delivers the app-wide singleton `UserService`. No problem.

Now consider the `HeroModule` *which is lazy-loaded*.

When the router lazy loads the `HeroModule`, 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 `HeroComponent`, it must inject a `UserService`. This time 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.

That's almost certainly a mistake.

To demonstrate, run the live example. Modify the `SharedModule` so that it provides the `UserService` rather than the `CoreModule`. Then toggle between the "Contact" and "Heroes" links a few times. The username flashes irregularly as the Angular creates a new `UserService` instance each time.

{@a q-why-child-injector}

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.

{@a q-is-it-loaded}

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, such as this `CoreModule` constructor.

{@a q-entry-component-defined}

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.

Most application components are loaded declaratively. Angular uses the component's selector to locate the element in the template. It then creates the HTML representation of the component and inserts it into the DOM at the selected element. These aren't entry components.

A few components are only loaded dynamically and are *never* referenced in a component template.

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.

Angular loads `AppComponent` dynamically because it's either listed *by type* in `@NgModule.bootstrap` or bootstrapped imperatively with the NgModule's `ngDoBootstrap` method.

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`.

The compiler can't discover these *entry components* by looking for them in other component templates. You must tell it about them by adding them to the `entryComponents` list.

Angular automatically adds the following types of components to the NgModule's `entryComponents`:

- The component in the `@NgModule.bootstrap` list.
- Components referenced in router configuration.

You don't have to mention these components explicitly, although doing so is harmless.

{@a q-bootstrap*vs*entry_component}

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 (application launch) process. 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 `entryComponent` lists, although doing so is harmless.

{@a q-when-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.

{@a q-why-entry-components}

Why does Angular need *entryComponents*?

Entry components are also declared. Why doesn't the Angular compiler generate code for every component in `@NgModule.declarations` ? Then you wouldn't need entry components.

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.

{@a q-module-recommendations}

What kinds of NgModules 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. The following suggestions and guidelines have wide appeal.

SharedModule

Create a `SharedModule` with the components, directives, and pipes that you use everywhere in your app. This NgModule 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 NgModules have `providers`. If you deviate from this guideline, know what you're doing and why.

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

CoreModule

Create a `CoreModule` with `providers` for the singleton services you load when the application starts.

Import `CoreModule` in the root `AppModule` only. Never import `CoreModule` in any other module.

Consider making `CoreModule` a pure services module with no `declarations`.

This page sample departs from that advice by declaring and exporting two components that are only used within the root `AppComponent` declared by `AppModule`. Someone following this guideline strictly would have declared these components in the `AppModule` instead.

Feature Modules

Create feature modules around specific application business domains, user workflows, and utility collections.

Feature modules tend to fall into one of the following groups:

- Domain feature modules.
- Routed feature modules.
- Routing modules.
- Service feature modules.
- Widget feature modules.

Real-world NgModules are often hybrids that purposefully deviate from the following guidelines. These guidelines are not laws; follow them unless you have a good reason to do otherwise.

Feature Module	Guidelines

{@a domain-feature-module}Domain	Domain feature modules deliver a user experience *dedicated to a particular application domain* like editing a customer or placing an order. They typically have a top component that acts as the feature root. Private, supporting sub-components descend from it. Domain feature modules consist mostly of _declarations_. Only the top component is exported. Domain feature modules rarely have _providers_. When they do, the lifetime of the provided services should be the same as the lifetime of the module. Don't provide application-wide singleton services in a domain feature module. Domain feature modules are typically imported _exactly once_ by a larger feature module. They might be imported by the root `AppModule` of a small application that lacks routing. For an example, see the [Feature Modules](guide/ngmodule#contact-module-v1) section of the [NgModules](guide/ngmodule) guide, before routing is introduced.
{@a routed-feature-module}Routed	_Routed feature modules_ are _domain feature modules_ whose top components are the *targets of router navigation routes*. All lazy-loaded modules are routed feature modules by definition. This page's `ContactModule`, `HeroModule`, and `CrisisModule` are routed feature modules. Routed feature modules _shouldn't export anything_. They don't have to because their components never appear in the template of an external component. A lazy-loaded routed feature module should _not be imported_ by any NgModule. Doing so would trigger an eager load, defeating the purpose of lazy loading. `HeroModule` and `CrisisModule` are lazy-loaded. They aren't mentioned among the `AppModule` imports. But an eagerly loaded, routed feature module must be imported by another NgModule so that the compiler learns about its components. `ContactModule` is eager loaded and therefore listed among the `AppModule` imports. Routed Feature Modules rarely have _providers_ for reasons [explained earlier](guide/ngmodule-faq#q-why-bad). When they do, the lifetime of the provided services should be the same as the lifetime of the NgModule. Don't provide application-wide singleton services in a routed feature module or in an NgModule that the routed module imports.
{@a routing-module}Routing	A [routing module](guide/router#routing-module) *provides routing configuration* for another NgModule. A routing module separates routing concerns from its companion module. A routing module typically does the following: * Defines routes. * Adds router configuration to the module's `imports`. * Re-exports `RouterModule`. * Adds guard and resolver service providers to the module's `providers`. The name of the routing module should parallel the name of its companion module, using the suffix "Routing". For example, `FooModule` in `foo.module.ts` has a routing module named `FooRoutingModule` in `foo-routing.module.ts` If the companion module is the _root_ `AppModule`, the `AppRoutingModule` adds router configuration to its `imports` with `RouterModule.forRoot(routes)`. All other routing modules are children that import `RouterModule.forChild(routes)`. A routing module re-exports the `RouterModule` as a convenience so that components of the companion module have access to router directives such as `RouterLink` and `RouterOutlet`. A routing module *should not have its own

	`declarations`*. Components, directives, and pipes are the *responsibility of the feature module*, not the _routing_ module. A routing module should _only_ be imported by its companion module. The `AppRoutingModule`, `ContactRoutingModule`, and `HeroRoutingModule` are good examples. See also [Do you need a _Routing Module_?](guide/router#why-routing-module) on the [Routing & Navigation](guide/router) page.
{@a service-feature-module}Service	Service modules *provide utility services* such as data access and messaging. Ideally, they consist entirely of _providers_ and have no _declarations_. The `CoreModule` and Angular's `HttpModule` are good examples. Service Modules should _only_ be imported by the root `AppModule`. Do *not* import service modules in other feature modules. If you deviate from this guideline, know what you're doing and why.
{@a widget-feature-module}Widget	A widget module makes *components, directives, and pipes* available to external NgModules. `CommonModule` and `SharedModule` are widget modules. Many third-party UI component libraries are widget modules. A widget module should consist entirely of _declarations_, most of them exported. A widget module should rarely have _providers_. If you deviate from this guideline, know what you're doing and why. Import widget modules in any module whose component templates need the widgets.

The following table summarizes the key characteristics of each *feature module* group.

Real-world NgModules are often hybrids that knowingly deviate from these guidelines.

Feature Module	Declarations	Providers	Exports	Imported By	Examples
Domain	Yes	Rare	Top component	Feature, `AppModule`	`ContactModule` (before routing)
Routed	Yes	Rare	No	Nobody	`ContactModule`, `HeroModule`, `CrisisModule`
Routing	No	Yes (Guards)	`RouterModule`	Feature (for routing)	`AppRoutingModule`, `ContactRoutingModule`, `HeroRoutingModule`
Service	No	Yes	No	`AppModule`	`HttpModule`, `CoreModule`
Widget	Yes	Rare	Yes	Feature	`CommonModule`, `SharedModule`

{@a q-ng-vs-js-modules}

What's the difference between Angular NgModules and JavaScript Modules?

Angular and JavaScript are different yet complementary module systems.

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:

export class AppComponent { ... }

Then you `import` a part in another module:

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

This kind of modularity is a feature of the *JavaScript language*.

An *NgModule* is a feature of *Angular* itself.

Angular's `@NgModule` metadata also have `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.

The NgModule classes differ from JavaScript module class in the following key ways:

- An NgModule bounds declarable classes only. Declarables are the only classes that matter to the Angular compiler.
- Instead of defining all member classes in one giant file (as in a JavaScript module), you list the NgModule's classes in the `@NgModule.declarations` list.
- An NgModule can only export the declarable classes it owns or imports from other NgModules. It doesn't declare or export any other kind of class.

The NgModule is also special in another way. Unlike JavaScript modules, an NgModule can extend the *entire* application with services by adding providers to the `@NgModule.providers` list.

The provided services don't belong to the NgModule nor are they scoped to the declared classes. They are available _everywhere_.

Here's an *@NgModule* class with imports, exports, and declarations.

Of course you use *JavaScript* modules to write NgModules as seen in the complete `contact.module.ts` file:

{@a q-template-reference}

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 NgModule or exported by an NgModule that this one imports.

{@a q-angular-compiler}

What is the Angular compiler?

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

The code you write isn't immediately executable. Consider *components*. 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 NgModules.

{@a q-ngmodule-api}

@NgModule API

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

Property	Description
`declarations`	A list of [declarable](guide/ngmodule-faq#q-declarable) classes, the *component*, *directive*, and *pipe* classes that _belong to this NgModule_. These declared classes are visible within the NgModule but invisible to components in a different NgModule unless they are _exported_ from this NgModule and the other NgModule _imports_ this one. Components, directives, and pipes must belong to _exactly_ one NgModule. The compiler emits an error if you try to declare the same class in more than one NgModule. *Do not re-declare a class imported from another NgModule.*
`providers`	A list of dependency-injection providers. Angular registers these providers with the root injector of the NgModule's execution context. That's the application's root injector for all NgModules loaded when the application starts. Angular can inject one of these provider services into any component in the application. If this NgModule or any NgModule loaded at launch provides the `HeroService`, Angular can inject the same `HeroService` intance into any app component. A lazy-loaded NgModule has its own sub-root injector which typically is a direct child of the application root injector. Lazy-loaded services are scoped to the lazy module's injector. If a lazy-loaded NgModule also provides the `HeroService`, 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 NgModules continue to receive the instance created for the application root.
`imports`	A list of supporting NgModules. Specifically, the list of NgModules whose exported components, directives, or pipes are referenced by the component templates declared in this NgModule. A component template can [reference](guide/ngmodule-faq#q-template-reference) another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module. A component can use the `NgIf` and `NgFor` directives only because its declaring NgModule imported the Angular `CommonModule` (perhaps indirectly by importing `BrowserModule`). You can import many standard directives with the `CommonModule` but some familiar directives belong to other NgModules. A component template can bind with `[(ngModel)]` only after importing the Angular `FormsModule`.

`exports`	A list of declarations—*component*, *directive*, and *pipe* classes—that an importing NgModule can use. Exported declarations are the module's _public API_. A component in another NgModule can [reference](guide/ngmodule-faq#q-template-reference) _this_ NgModule's `HeroComponent` if it imports this module and this module exports `HeroComponent`. Declarations are private by default. If this NgModule does _not_ export `HeroComponent`, no other NgModule can see it. Importing an NgModule does _not_ automatically re-export the imported NgModule's imports. NgModule 'B' can't use `ngIf` just because it imported NgModule `A` which imported `CommonModule`. NgModule 'B' must import `CommonModule` itself. An NgModule can list another NgModule among its `exports`, in which case all of that NgModule's public components, directives, and pipes are exported. [Re-export](guide/ngmodule-faq#q-re-export) makes NgModule transitivity explicit. If NgModule 'A' re-exports `CommonModule` and NgModule 'B' imports NgModule 'A', NgModule 'B' components can use `ngIf` even though 'B' itself didn't import `CommonModule`.
`bootstrap`	A list of components that can be 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 an `entryComponent`.
`entryComponents`	A list of components that are _not_ [referenced](guide/ngmodule-faq#q-template-reference) in a reachable component template. Most developers never set this property. The [Angular compiler](guide/ngmodule-faq#q-angular-compiler) must know about every component actually used in the application. The compiler can discover most components by walking the tree of references from one component template to another. But there's always at least one component that's not referenced in any template: the root component, `AppComponent`, that you bootstrap to launch the app. That's why it's called an _entry component_. Routed components are also _entry components_ because they aren't referenced in a template either. The router creates them and drops them into the DOM near a ``. While the bootstrapped and routed components are _entry components_, you usually don't have to add them to a module's `entryComponents` list. Angular automatically adds components in the module's `bootstrap` list to the `entryComponents` list. The `RouterModule` adds routed components to that list. That leaves only the following sources of undiscoverable components: * Components bootstrapped using one of the imperative techniques. * Components dynamically loaded into the DOM by some means other than the router. Both are advanced techniques that few developers ever employ. If you are one of those few, you must add these components to the `entryComponents` list yourself, either programmatically or by hand.