NgModules

NgModules help organize an application into cohesive blocks of functionality.

An NgModule is a class adorned with the **@NgModule** decorator function. `@NgModule` takes a metadata object that tells Angular how to compile and your code. It identifies the module's own components, directives, and pipes, making some of them public so external components can use them. `@NgModule` may add service providers to the application dependency injectors. And there are many more options covered here.

{@a bootstrap}

For a quick overview of NgModules, consider reading the Bootstrapping guide, which introduces NgModules and the essentials of creating and maintaining a single root `AppModule` for the entire application.

This page covers NgModules in greater depth.

Live examples

This page explains NgModules through a progression of improvements to a sample with a "Heroes" theme. Here's an index to live examples at key moments in the evolution of the sample:

- The initial app
- The first contact module
- The revised contact module
- Just before adding SharedModule
- The final version

Frequently asked questions (FAQs)

This page covers NgModule concepts in a tutorial fashion.

The companion NgModule FAQs guide offers answers to specific design and implementation questions. Read this page before reading those FAQs.

{@a angular-modularity}

Angular modularity

NgModules are a great way to organize an application and extend it with capabilities from external libraries.

Many Angular libraries are NgModules (such as `FormsModule`, `HttpModule`, and `RouterModule`). Many third-party libraries are available as NgModules (such as Material Design, Ionic, AngularFire2).

NgModules consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

NgModules can also add services to the application. Such services might be internally developed, such as the application logger. Services can come from outside sources, such as the Angular router and Http client.

NgModules can be loaded eagerly when the application starts. They can also be *lazy-loaded* asynchronously by the router.

An NgModule is a class decorated with `@NgModule` metadata. By setting metadata properties you tell Angular how your application parts fit together. For example, you can do the following:

- *Declare* which components, directives, and pipes belong to the NgModule.
- *Export* some of those classes so that other component templates can use them.
- *Import* other NgModules with the components, directives, and pipes needed by the components in *this* NgModule.
- *Provide* services at the application level that any application component can use.
- *Bootstrap* the app with one or more top-level, *root* components.

{@a root-module}

The root *AppModule*

Every Angular app has at least one NgModule class, the *root module*. You bootstrap *that* NgModule to launch the application.

By convention, the *root module* class is called `AppModule` and it exists in a file named `app.module.ts`. The **Angular CLI** generates the initial `AppModule` for you when you create a project.

ng new quickstart

The root `AppModule` is all you need in a simple application with a few components.

As the app grows, you may refactor the root `AppModule` into *feature modules* that represent collections of related functionality. For now, stick with the root `AppModule` created by the CLI.

The initial `declarations` array identifies the application's only component, `AppComponent`, the *root*

component at the top of the app's component tree.

Soon you'll declare more components (and directives and pipes too).

The `@NgModule` metadata `imports` a single helper module, `BrowserModule`, which every browser app must import. `BrowserModule` registers critical application service providers. It also includes common directives like `NgIf` and `NgFor`, which become immediately visible and usable in any of this NgModule's component templates.

The `providers` array registers services with the top-level *dependency injector*. There are no services to register ... yet.

Lastly, the `bootstrap` list identifies the `AppComponent` as the *bootstrap component*. When Angular launches the app, it renders the `AppComponent` inside the `<app-root>` element tag of the `index.html`.

Learn about that in the bootstrapping guide.

The CLI-generated `AppComponent` in this guide's sample has been simplified and consolidated into a single `app.component.ts` file like this:

Run the app and follow along with the steps in this guide:

ng serve

{@a declarations} {@a declare-directive}

Declare directives

{@a declarables}

As the app evolves, you'll add directives, components, and pipes (the *declarables*). You must declare each of these classes in an NgModule.

As an exercise, begin by adding a `highlight.directive.ts` to the `src/app/` folder *by hand*.

The `HighlightDirective` is an attribute directive that sets the background color of its host element. Update the `AppComponent` template to attach this directive to the `<h1>` title element:

The screen of the running app has not changed. The `<h1>` is not highlighted. Angular does not yet recognize the `highlight` attribute and is ignoring it. You must declare the `HighlightDirective` in `AppModule`.

Edit the `app.module.ts` file, import the `HighlightDirective`, and add it to the `AppModule` *declarations* like this:

The Angular CLI would have done all of this for you if you'd created the `HighlightDirective` with the CLI command like this:

ng generate directive highlight

But you didn't. You created the file by hand so you must declare the directive by hand.

{@a declare-component}

Declare components

Now add a `TitleComponent` to the app and this time create it with the CLI.

ng generate component title --flat --no-spec --inline-style

The `--flat` flag tells the CLI to generate all files to the `src/app/` folder.
The `--no-spec` flag skips the test (`.spec`) file.
The `--inline-style` flag prevents generation of the `.css` file (which you won't need).
To see which files would be created or changed by any `ng generate` command, append the `--dryRun` flag (`-d` for short).

Open the `AppModule` and look at the `declarations` where you will see that the CLI added the `TitleComponent` for you.

Now rewrite the `title.component.html` like this.

And move the `title` property from `app.component.ts` into the `title.component.ts`, which looks as follows after a little cleanup.

Rewrite `AppComponent` to display the new `TitleComponent` in the `<app-title>` element and get rid of the `title` property.

Error if component not declared

There was no visible clue when you neglected to declare the `HighlightDirective` attribute directive. The Angular compiler doesn't recognize `highlight` as an `<h1>` attribute but it doesn't complain either. You'd discover it was undeclared only if you were looking for its effect.

Now try removing the declaration of the `TitleComponent` from `AppModule`.

The Angular compiler behaves differently when it encounters an unrecognized HTML element. The app ceases to display the page and the browser console logs the following error

Uncaught Error: Template parse errors: 'app-title' is not a known element: 1. If 'app-title' is an Angular component, then verify that it is part of this NgModule. 2. If 'app-title' is a Web Component then add 'CUSTOM*ELEMENTS*SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.

If you don't get that error, you might get this one: Uncaught Error: Component TitleComponent is not part of any NgModule or the module has not been imported into your module.

Always declare your components, directives, and pipes.

{@a providers}

Service providers

The Dependency Injection page describes the Angular hierarchical dependency-injection system and how to configure that system with providers.

NgModule providers

An NgModule can provide services. A single instance of each provided service becomes available for injection into every class created with that NgModule's injector (or one of its descendant injectors).

When Angular boots the application, it creates the root `AppModule` with a root dependency injector. Angular configures the root injector with the providers specified in the module's `@NgModule.providers`.

Later, when Angular creates a new instance of a class— be it a component, directive, service, or module— that new class can be injected with an instance of a service provided to the root injector by the `AppModule`.

Angular also configures the root injector with the providers specified by [imported NgModules](#imports). An NgModule's own providers are registered _after_ imported NgModule providers. When there are multiple providers for the same injection token, the last registration wins.

Compared to Component providers

Providing a service in `@Component.providers` metadata means that a new service instance will be created for each new instance of *that* component and will be available for injection into *all of that component instance's descendant sub-components*.

The service instance won't be injected into any other component instances. Other instances of the same

component class cannot see it. Sibling and ancestor component instances cannot see it.

Component providers always supersede NgModule providers. A component provider for injection token `X` creates a new service instance that "shadows" an NgModule provider for injection token `X`. When the component or any of its sub-components inject `X`, they get the *component* service instance, not the *NgModule* service instance.

Should you provide a service in an *NgModule* or a *component*? The answer depends on how you want to scope the service. If the service should be widely available, provide it in an NgModule. If it should be visible only within a component tree, provide it in the component at the root of that tree.

NgModule provider example

Many applications capture information about the currently logged-in user and make that information accessible through a user service.

Use the CLI to create a `UserService` and provide it in the root `AppModule`.

ng generate service user --module=app

This command creates a skeleton `UserService` in `src/app/user.service.ts` and a companion test file, `src/app/user.service.spec.ts`.

The `--module=app` flag tells the CLI to provide the service class in the NgModule defined in the `src/app/app.module.ts` file.

If you omit the `--module` flag, the CLI still creates the service but *does not provide it* anywhere. You have to do that yourself.

Confirm that the `--module=app` flag did provide the service in the root `AppModule` by inspecting the `@NgModule.providers` array in `src/app/app.module.ts`

Replace the generated contents of `src/app/user.service.ts` with the following dummy implementation.

Update the `TitleComponent` class with a constructor that injects the `UserService` and sets the component's `user` property from the service.

Update the `TitleComponent` template to show the welcome message below the application title.

{@a imports}

NgModule imports

In the revised `TitleComponent`, an `*ngIf` directive guards the message. There is no message if there is no user.

Although `AppModule` doesn't declare the `NgIf` directive, the application still compiles and runs. How can that be? The Angular compiler should either ignore or complain about unrecognized HTML.

Importing *BrowserModule*

Angular does recognize `NgIf` because the `AppModule` imports it indirectly when it imports `BrowserModule`.

Importing `BrowserModule` made all of its public components, directives, and pipes visible to the templates of components declared in `AppModule`, which include `TitleComponent`.

{@a reexport}

Re-exported NgModules

The `NgIf` directive isn't declared in `BrowserModule`. It's declared in `CommonModule` from `@angular/common`.

`CommonModule` contributes many of the common directives that applications need, including `ngIf` and `ngFor`.

`AppModule` doesn't import `CommonModule` directly. But it benefits from the fact that `BrowserModule` imports `CommonModule` **and re-exports it**.

The net effect is that an importer of `BrowserModule` gets `CommonModule` directives automatically as if it had declared them itself.

Many familiar Angular directives don't belong to `CommonModule`. For example, `NgModel` and `RouterLink` belong to Angular's `FormsModule` and `RouterModule` respectively. You must import those NgModules before you can use their directives.

To illustrate this point, you'll extend the sample app with *contact editor* whose `ContactComponent` is a form component. You'll have to import form support from the Angular `FormsModule`.

{@a add-contact-editor}

Add a *contact editor*

Imagine that you added the following *contact editor* files to the project by hand *without the help of the CLI*.

Form components are often complex and this is one is no exception. To make it manageable, all contact-related files are in an `src/app/contact` folder. The `ContactComponent` implementation is spread over three constituent HTML, TypeScript, and css files. There's a [custom pipe](guide/pipes#custom-pipes) (called `Awesome`), a `ContactHighlightDirective`, and a `ContactService` for fetching contacts. The `ContactService` was added to the `AppModule` providers. Now any class can inject the application-wide instances of the `ContactService` and `UserService`.

Import supporting *FormsModule*

The `ContactComponent` is written with Angular forms in the template-driven style.

Notice the `[(ngModel)]` binding in the middle of the component template, `contact.component.html`.

Two-way data binding `[(ngModel)]` is typical of the *template-driven* style. The `ngModel` is the selector for the `NgModel` directive. Although `NgModel` is an Angular directive, the *Angular compiler* won't recognize it for two reasons:

1. `AppModule` doesn't declare `NgModel` (and shouldn't).
2. `NgModel` wasn't imported via `BrowserModule`.

`ContactComponent` wouldn't behave like an Angular form anyway because form features such as validation aren't part of the Angular core.

To correct these problems, the `AppModule` must import *both* the `BrowserModule` *and* the **FormsModule from '@angular/forms'** like this.

You can write Angular form components in template-driven or [reactive](guide/reactive-forms) style. NgModules with components written in the _reactive_ style import the `ReactiveFormsModule`.

Now `[(ngModel)]` binding will work and the user input will be validated by Angular forms, once you declare the new component, pipe, and directive.

Never re-declare

Importing the `FormsModule` makes the `NgModelDirective` (and all of the other `FORMS_DIRECTIVES`) available to components declared in `AppModule`.

Do not also add these directives to the `AppModule` metadata's declarations.

Never re-declare classes that belong to another NgModule. Components, directives, and pipes should be declared in _exactly one NgModule_.

{@a declare-pipe}

Declare pipes

The revised application still won't compile until you declare the contact component, directive, and pipe.

Components and directives are *declarables*. So are **pipes**.

You learned earlier to generate and declare both components and directives with the CLI `ng generate` commands.

There's also a CLI command to generate and declare the `AwesomePipe`:

ng generate pipe awesome

However, if you write these class files by hand or opt-out of declaration with the `--skip-import` flag, you'll have to add the declarations yourself.

You were told to add the *contact editor* files by hand, so you must manually update the `declarations` in the `AppModule`:

Display the *ContactComponent*

Update the `AppComponent` template to display the `ContactComponent` by placing an element with its selector (`<app-contact>`) just below the title.

Run the app

Everything is in place to run the application with its contact editor. Try the example: ## Selector conflicts Look closely at the screen. Notice that the background of the application title text is _blue_. It should be _gold_ (see `src/app/app.component.html`). Only the contact name should be blue (see `src/app/contact/contact.component.html`). What went wrong? This application defines two highlight directives that set the background color of their host elements with a different color (gold and blue). One is defined at the root level (`src/app/highlight.directive.ts`); the other is in the contact editor folder (`src/app/contact/contact-highlight.directive.ts`). Their class names are different (`HighlightDirective` and `ContactHighlightDirective`) but their selectors both match any HTML element with a `highlight` attribute. Both directives are declared in the same `AppModule` so both directives are active for all components declared in `AppModule`. There's nothing

intrinsically wrong with multiple directives selecting the same element. Each could modify the element in a different, non-conflicting way. In _this case_, both directives compete to set the background color of the same element. The directive that's declared later (`ContactHighlightDirective`) always wins because its DOM changes overwrite the changes by the earlier `HighlightDirective`. The `ContactHighlightDirective` will make the application title text blue when it should be gold. Only the contact name should be blue (see `src/app/contact/contact.component.html`). If you cannot rename the selectors, you can resolve the conflicts by creating [feature modules](#feature-modules) that insulate the declarations in one NgModule from the declarations in another.
While it is legal to declare two _directives_ with the same selector in the same NgModule, the compiler will not let you declare two _components_ with the same selector in the same NgModule because it **cannot insert multiple components in the same DOM location**. Nor can you _import_ an NgModule that declares the same selector as another component in this NgModule. The reason is the same: an HTML element may be controlled by at most one Angular component. Either rename the selectors or use [feature modules](#feature-modules) to eliminate the conflict.
Feature modules This tiny app is already experiencing structural issues. * The root `AppModule` grows larger with each new application class. * There are conflicting directives. The `ContactHighlightDirective` in the contact re-colors the work done by the `HighlightDirective` declared in `AppModule` and colors the application title text when it should color only the `ContactComponent`. * The app lacks clear boundaries between contact functionality and other application features. That lack of clarity makes it harder to assign development responsibilities to different teams. _Feature modules_ can help resolve these issues. Architecturally, a feature module is an NgModule class that is dedicated to an application feature or workflow. Technically, it's another class adorned by the `@NgModule` decorator, just like a root `AppModule`. Feature module metadata have the same properties as root module metadata. When loaded together, the root module and the feature module share the same dependency injector, which means the services provided in a feature module are available to all. These two module types have the following significant technical differences: * You _boot_ the root module to _launch_ the app; you _import_ a feature module to _extend_ the app. * A feature module can expose or hide its [declarables](#declarables) from other NgModules. Otherwise, a feature module is distinguished primarily by its intent. A feature module delivers a cohesive set of functionality focused on an application business domain, user workflow, facility (forms, http, routing), or collection of related utilities. Feature modules help you partition the app into areas of specific interest and purpose. A feature module collaborates with the root module and with other NgModules through the services it provides and the components, directives, and pipes that it shares. {@a contact-module-v1}

Make *contact editor* a feature

In this section, you refactor the *contact editor* functionality out of the root `AppModule` and into a dedicated feature module by following these steps.

1. Create the `ContactModule` feature module in its own folder.

2. Copy the *contact editor* declarations and providers from `AppModule` to `ContactModule`.
3. Export the `ContactComponent`.
4. Import the `ContactModule` into the `AppModule`.
5. Cleanup the `AppModule`.

You'll create one new `ContactModule` class and change one existing `AppModule` class. All other files are untouched.

Create the feature module

Generate the *ContactModule* and its folder with an Angular CLI command.

ng generate module contact

Here's the generated `ContactModule`.

After modifying the initial `ContactsModule` as outlined above, it looks like this.

The following sections discuss the important changes.

Import *CommonModule*

Notice that `ContactModule` imports `CommonModule`, not `BrowserModule`. The CLI module generation took care of this for you.

Feature module components need the common Angular directives but not the services and bootstrapping logic in `BrowserModule`. See the NgModule FAQs for more details.

Import *FormsModule*

The `ContactModule` imports the `FormsModule` because its `ContactComponent` uses `NgModel`, one of the `FormsModule` directives.

NgModules don't inherit access to the declarations of the root `AppModule` or any other NgModule. Each NgModule must import what it needs. Because `ContactComponent` needs the form directives, its `ContactModule` must import `FormsModule`.

Copy declarations

The `ContactModule` declares the *contact editor* components, directives and pipes.

The app fails to compile at this point, in part because `ContactComponent` is currently declared in both the

`AppModule` and the `ContactModule`. A component may only be declared in one NgModule. You'll fix this problem shortly.

{@a root-scoped-providers}

Providers are root-scoped

The `ContactModule` provides the `ContactService` and the `AppModule` will stop providing it after refactoring.

Architecturally, the `ContactService` belongs to the *contact editor* domain. Classes in the rest of the app do not need the `ContactService` and shouldn't inject it. So it makes sense for the `ContactModule` to provide the `ContactService` as it does.

You might expect that the `ContactService` would only be injectable in classes declared or provided in the `ContactModule`.

That's not the case. *Any* class *anywhere* can inject the `ContactService` because `ContactModule` providers are *root*-scoped.

To be precise, all _eagerly loaded_ modules— modules loaded when the application starts — are root-scoped. This `ContactModule` is eagerly loaded. You will learn that services provided in [_lazy-loaded_ modules](#lazy-loaded-modules) have their own scope.

Angular does not have *module*-scoping mechanism. Unlike components, NgModule instances do not have their own injectors so they can't have their own provider scopes.

`ContactService` remains an *application*-scoped service because Angular registers all NgModule `providers` with the application's *root injector*. This is true whether the service is provided directly in the root `AppModule` or in an imported feature module like `ContactModule`.

In practice, service scoping is rarely an issue. Components don't accidentally inject a service. To inject the `ContactService`, you'd have to import its *type* and explicitly inject the service into a class constructor. Only *contact editor* components should import the `ContactService` type.

If it's really important to you to restrict the scope of a service, provide it in the feature's top-level component (`ContactComponent` in this case).

For more on this topic, see "How do I restrict service scope to a module?" in the NgModule FAQs.

Export public-facing components

The `ContactModule` makes the `ContactComponent` *public* by *exporting* it.

Declared classes are *private* by default. Private declarables may only appear in the templates of components declared by the *same* NgModule. They are invisible to components in *other* NgModules.

That's a problem for the `AppComponent`. Both components *used to be* declared in `AppModule` so Angular could display the `ContactComponent` within the `AppComponent`. Now that the `ContactComponent` is declared in its own feature module. The `AppComponent` cannot see it unless it is public.

The first step toward a solution is to *export* the `ContactComponent`. The second step is to *import* the `ContactModule` in the `AppModule`, which you'll do when you refactor the *AppModule*.

The `AwesomePipe` and `ContactHighlightDirective` remain private and are hidden from the rest of the application.

The `ContactHighlightDirective`, being private, no longer overrides the `HighlightDirective` in the `AppComponent`. The background of the title text is gold as intended.

{@a refactor-appmodule}

Refactor the *AppModule*

Return to the `AppModule` and remove everything specific to the *contact editor* feature set. Leave only the classes required at the application root level.

- Delete the *contact editor* import statements.
- Delete the *contact editor* declarations and providers.
- Delete the `FormsModule` from the `imports` list (the `AppComponent` doesn't need it).
- Import the `ContactModule` so the app can continue to display the exported `ContactComponent`.

Here's the refactored `AppModule`, presented side-by-side with the previous version.

Improvements

There's a lot to like in the revised `AppModule`.

- It does not change as the *Contact* domain grows.
- It only changes when you add new NgModules.
- It's simpler:
    - Fewer import statements.

- No `FormsModule` import.
- No *contact editor* declarations.
- No `ContactService` provider.
- No *highlight directive* conflicts.

Try this `ContactModule` version of the sample.

Try the live example.

{@a routing-modules} {@a lazy-loaded-modules}

Routing modules

Navigating the app with the Angular Router reveals new dimensions of the NgModule.

In this segment, you'll learn to write *routing modules* that configure the router. You'll discover the implications of *lazy loading* a feature module with the router's `loadChildren` method.

Imagine that the sample app has evolved substantially along the lines of the Tour of Heroes tutorial.

- The app has three feature modules: Contact, Hero (new), and Crisis (new).
- The Angular router helps users navigate among these modules.
- The `ContactComponent` is the default destination when the app starts.
- The `ContactModule` continues to be *eagerly loaded* when the application starts.
- `HeroModule` and the `CrisisModule` are *lazy-loaded*.

There's too much code behind this sample app to review every line. Instead, the guide explores just those parts necessary to understand new aspects of NgModules.

You can examine the complete source for this version of the app in the live example.

{@a app-component-template}

The root *AppComponent*

The revised `AppComponent` template has a title, three links, and a `<router-outlet>` .

The `<app-contact>` element that displayed the `ContactComponent` is gone; you're routing to the *Contact* page now.

The root *AppModule*

The `AppModule` is slimmer now.

The `AppModule` is no longer aware of the application domains such as contacts, heroes, and crises. Those concerns are pushed down to `ContactModule`, `HeroesModule`, and `CrisisModule` respectively and only the routing configuration knows about them.

The significant change from version 2 is the addition of the *AppRoutingModule* to the NgModule `imports`. The `AppRoutingModule` is a routing module that handles the app's routing concerns.

AppRoutingModule

The router is the subject of the Routing & Navigation guide, so this section skips many routing details and concentrates on the *intersection* of NgModules and routing.

You can specify router configuration directly within the root `AppModule` or within a feature module.

The *Router guide* recommends instead that you locate router configuration in separate, dedicated NgModules, called *routing modules*. You then import those routing modules into their corresponding root or feature modules.

The goal is to separate the normal declarative concerns of an NgModule from the often complex router configuration logic.

By convention, a routing module's name ends in `...RoutingModule`. The top-level root module is `AppModule` and it imports its companion *routing module* called `AppRoutingModule`.

Here is this app's `AppRoutingModule`, followed by a discussion.

The `AppRoutingModule` defines three routes:

The first route redirects the empty URL (such as `http://host.com/`) to another route whose path is `contact` (such as `http://host.com/contact`).

The `contact` route isn't defined within the `AppRoutingModule`. It's defined in the *Contact* feature's *own* routing module, `ContactRoutingModule`.

It's standard practice for feature modules with routing components to define their own routes. You'll get to [`ContactRoutingModule`](#contact-routing-module) in a moment.

The remaining two routes use lazy loading syntax to tell the router where to find the modules for the hero and crisis features:

A lazy-loaded NgModule location is a _string_, not a _type_. In this app, the string identifies both the NgModule

file and the NgModule _class_, the latter separated from the former by a `#`.

Routing module imports

A *routing module* typically imports the Angular `RouterModule` so it can register routes.

It may also import a *feature module* which registers routes (either directly or through its companion *routing module*).

This `AppRoutingModule` does both.

It first imports the `ContactModule`, which as you'll see, imports its own `ContactRoutingModule`.

Import order matters! Because "contacts" is the first defined route and the default route for the app, you must import it *before* all other routing-related modules.

The second import registers the routes defined in this module by calling the `RouterModule.forRoot` class method.

The `forRoot` method does two things:

1. Configures the router with the supplied *routes*.
2. Initializes the Angular router itself.

Call `RouterModule.forRoot` exactly once for the entire app. Calling it in the `AppRoutingModule`, the companion to the root `AppModule`, is a good way to ensure that this method is called exactly once. Never call `RouterModule.forRoot` in a feature's _routing module_.

Re-export *RouterModule*

All *routing modules* should re-export the `RouterModule`.

Re-exporting `RouterModule` makes the router directives available to the companion module that imports it. This is a considerable convenience for the importing module.

For example, the `AppComponent` template relies on the `routerLink` directive to turn the user's clicks into navigations. The Angular compiler only recognizes `routerLink` because

- `AppComponent` is declared by `AppModule`,
- `AppModule` imports `AppRoutingModule`,
- `AppRoutingModule` exports `RouterModule`, and
- `RouterModule` exports the `RouterLink` directive.

If `AppRoutingModule` didn't re-export `RouterModule`, the `AppModule` would have to import the `RouterModule` itself.

{@a contact-routing-module}

Routing to a feature module

The three feature modules (`ContactModule`, `HeroModule`, `CrisisModule`) have corresponding routing modules (`ContactRoutingModule`, `HeroRoutingModule`, `CrisisRoutingModule`).

They follow the same pattern as the `AppRoutingModule`. * define routes * register the routes with Angular's `RouterModule` * export the `RouterModule`.

The `ContactRoutingModule` is the simplest of the three. It defines and registers a single route to the `ContactComponent`.

There is **one critical difference** from `AppRoutingModule`: you pass the routes to `RouterModule.forChild`, not `forRoot`.

Always call `RouterModule.forChild` in a feature-routing module. Never call `RouterModule.forRoot`.

ContactModule changes

Because the app navigates to the `ContactComponent` instead of simply displaying it in the `AppComponent` template, the `ContactModule` has changed.

- It imports the `ContactRoutingModule`.
- It no longer exports `ContactComponent`.

The `ContactComponent` is only displayed by the router, No template references its `<app-contact>` selector. There's no reason to make it public via the `exports` array.

Here is the latest version, side-by-side with the previous version.

{@a hero-module}

Lazy-loaded routing

The `HeroModule` and `CrisisModule` have corresponding *routing modules*, `HeroRoutingModule` and `CrisisRoutingModule`.

The app *lazy loads* the `HeroModule` and the `CrisisModule`. That means the `HeroModule` and the

`CrisisModule` are not loaded into the browser until the user navigates to their components.

Do not import the `HeroModule` or `CrisisModule` or any of their classes outside of their respective file folders. If you do, you will unintentionally load those modules and all of their code when the application starts, defeating the purpose of lazy loading. For example, if you import the `HeroService` in `AppModule`, the `HeroService` class and all related hero classes will be loaded when the application starts.

Lazy loading can improve the app's perceived performance because the browser doesn't have to process lazy-loaded code when the app starts. It may *never* process that code.

You cannot tell that these modules are lazy-loaded by looking at their *routing modules*. They happen to be a little more complex than `ContactRoutingModule`. For example, The `HeroRoutingModule` has child routes. But the added complexity springs from intrinsic hero and crisis functionality, not from lazy loading. Fundamentally, these *routing modules* are just like `ContactRoutingModule` and you write them the same way.

{@a lazy-load-DI}

Lazy-loaded NgModule providers

There is a **runtime difference** that can be significant. Services provided by lazy-loaded NgModules are only available to classes instantiated within the lazy-loaded context. The reason has to do with dependency injection.

When an NgModule is *eagerly loaded* as the application starts, its providers are added to the application's *root injector*. Any class in the application can inject a service from the *root injector*.

When the router *lazy loads* an NgModule, Angular instantiates the module with a *child injector* (a descendant of the *root injector*) and adds the module's providers to this *child injector*. Classes created with the *child injector* can inject one of its provided services. Classes created with *root injector* cannot.

Each of the three feature modules has its own data access service. Because the `ContactModule` is *eagerly loaded* when the application starts, its `ContactService` is provided by the application's *root dependency injector*. That means the `ContactService` can be injected into any application class, including hero and crisis components.

Because `CrisisModule` is *lazy-loaded*, its `CrisisService` is provided by the `CrisisModule` *child injector*. It can only be injected into one of the crisis components. No other kind of component can inject the `CrisisService` because no other kind of component can be reached along a route that lazy loads the `CrisisModule`.

Lazy-loaded NgModule lifetime

Both eager and lazy-loaded NgModules are created *once* and never destroyed. This means that their provided service instances are created *once* and never destroyed.

As you navigate among the application components, the router creates and destroys instances of the contact, hero, and crisis components. When these components inject data services provided by their modules, they get the same data service instance each time.

If the `HeroService` kept a cache of unsaved changes and the user navigated to the `ContactComponent` or the `CrisisListComponent`, the pending hero changes would remain in the one `HeroService` instance, waiting to be saved.

But if you provided the `HeroService` in the `HeroComponent` instead of the `HeroModule`, new `HeroService` instances would be created each time the user navigated to a hero component. Previously pending hero changes would be lost.

To illustrate this point, the sample app provides the `HeroService` in the `HeroComponent` rather than the `HeroModule`.

Run the app, open the browser development tools, and look at the console as you navigate among the feature pages.

// App starts ContactService instance created. ... // Navigate to Crisis Center CrisisService instance created. ... // Navigate to Heroes HeroService instance created. ... // Navigate to Contact HeroService instance destroyed. ... // Navigate back to Heroes HeroService instance created.

The console log shows the `HeroService` repeatedly created and destroyed. The `ContactService` and `CrisisService` are created but never destroyed, no matter where you navigate.

Run it

Try this routed version of the sample.

Try the live example.

{@a shared-module}

Shared modules

The app is shaping up. But there are a few annoying problems. There are three unnecessarily different *highlight directives* and the many files cluttering the app folder level could be better organized.

You can eliminate the duplication and tidy-up by writing a `SharedModule` to hold the common components, directives, and pipes. Then share this NgModule with the other NgModules that need these declarables.

Use the CLI to create the `SharedModule` class in its `src/app/shared` folder.

ng generate module shared

Now refactor as follows:

- Move the `AwesomePipe` from `src/app/contact` to `src/app/shared`.
- Move the `HighlightDirective` from `src/app/hero` to `src/app/shared`.
- Delete the *highlight directive* classes from `src/app/` and `src/app/contact`.
- Update the `SharedModule` as follows:

Note the following:

- It declares and exports the shared pipe and directive.
- It imports and re-exports the `CommonModule` and `FormsModule`
- It can re-export `FormsModule` without importing it.

Re-exporting NgModules

Technically, there is no need for `SharedModule` to import `CommonModule` or `FormsModule`. `SharedModule` doesn't declare anything that needs material from `CommonModule` or `FormsModule`.

But NgModules that would like to import `SharedModule` for its pipe and highlight directive happen also to declare components that need `NgIf` and `NgFor` from `CommonModule` and do two-way binding with `[(ngModel)]` from the `FormsModule`.

Normally, they'd have to import `CommonModule` and `FormsModule` as well as `SharedModule`. Now they can just import `SharedModule`. By exporting `CommonModule` and `FormsModule`, `SharedModule` makes them available to its importers *for free*.

A trimmer *ContactModule*

See how `ContactModule` became more concise, compared to its previous version:

Notice the following:

- The `AwesomePipe` and `ContactHighlightDirective` are gone.
- The imports include `SharedModule` instead of `CommonModule` and `FormsModule`.
- The new version is leaner and cleaner.

Why *TitleComponent* isn't shared

`SharedModule` exists to make commonly used components, directives, and pipes available for use in the templates of components in many other NgModules.

The `TitleComponent` is used only once by the `AppComponent`. There's no point in sharing it.

{@a no-shared-module-providers}

Why *UserService* isn't shared

While many components share the same service instances, they rely on Angular dependency injection to do this kind of sharing, not the NgModule system.

Several components of the sample inject the `UserService`. There should be only one instance of the `UserService` in the entire application and only one provider of it.

`UserService` is an application-wide singleton. You don't want each NgModule to have its own separate instance. Yet there is a real danger of that happening if the `SharedModule` provides the `UserService`.

Do *not* specify app-wide singleton `providers` in a shared module. A lazy-loaded NgModule that imports that shared module makes its own copy of the service.

{@a core-module}

The Core module

At the moment, the root folder is cluttered with the `UserService` and `TitleComponent` that only appear in the root `AppComponent`. You didn't include them in the `SharedModule` for reasons just explained.

Instead, gather them in a single `CoreModule` that you import once when the app starts and never import anywhere else.

Perform the following steps:

1. Create a `CoreModule` class in an `src/app/core` folder.
2. Move the `TitleComponent` and `UserService` from `src/app/` to `src/app/core`.
3. Declare and export the `TitleComponent`.
4. Provide the `UserService`.
5. Update the root `AppModule` to import `CoreModule`.

Most of this work is familiar. The interesting part is the `CoreModule`.

You're importing some extra symbols from the Angular core library that you're not using yet. They'll become relevant later in this page.

The `@NgModule` metadata should be familiar. You declare the `TitleComponent` because this NgModule owns it. You export it because `AppComponent` (which is in `AppModule`) displays the title in its template. `TitleComponent` needs the Angular `NgIf` directive that you import from `CommonModule`.

`CoreModule` provides the `UserService`. Angular registers that provider with the app root injector, making a singleton instance of the `UserService` available to any component that needs it, whether that component is eagerly or lazily loaded.

Why bother?

This scenario is clearly contrived. The app is too small to worry about a single service file and a tiny, one-time component. A `TitleComponent` sitting in the root folder isn't bothering anyone. The root `AppModule` can register the `UserService` itself, as it does currently, even if you decide to relocate the `UserService` file to the `src/app/core` folder. Real-world apps have more to worry about. They can have several single-use components (such as spinners, message toasts, and modal dialogs) that appear only in the `AppComponent` template. You don't import them elsewhere so they're not shared in that sense. Yet they're too big and messy to leave loose in the root folder. Apps often have many singleton services like this sample's `UserService`. Each must be registered exactly once, in the app root injector, when the application starts. While many components inject such services in their constructors—and therefore require JavaScript `import` statements to import their symbols—no other component or NgModule should define or re-create the services themselves. Their _providers_ aren't shared. We recommend collecting such single-use classes and hiding their details inside a `CoreModule`. A simplified root `AppModule` imports `CoreModule` in its capacity as orchestrator of the application as a whole.

A trimmer *AppModule*

Here is the updated `AppModule` paired with version 3 for comparison:

`AppModule` now has the following qualities:

- A little smaller because many `src/app/root` classes have moved to other NgModules.
- Stable because you'll add future components and providers to other NgModules, not this one.
- Delegated to imported NgModules rather than doing work.
- Focused on its main task, orchestrating the app as a whole.

{@a core-for-root}

Configure core services with *CoreModule.forRoot*

An NgModule that adds providers to the application can offer a facility for configuring those providers as well.

By convention, the `forRoot` static method both provides and configures services at the same time. It takes a service configuration object and returns a ModuleWithProviders, which is a simple object with the following properties:

- `ngModule` : the `CoreModule` class
- `providers` : the configured providers

The root `AppModule` imports the `CoreModule` and adds the `providers` to the `AppModule` providers.

More precisely, Angular accumulates all imported providers before appending the items listed in `@NgModule.providers`. This sequence ensures that whatever you add explicitly to the `AppModule` providers takes precedence over the providers of imported NgModules.

Add a `CoreModule.forRoot` method that configures the core `UserService` .

You've extended the core `UserService` with an optional, injected `UserServiceConfig` . If a `UserServiceConfig` exists, the `UserService` sets the user name from that config.

Here's `CoreModule.forRoot` that takes a `UserServiceConfig` object:

Lastly, call it within the `imports` list of the `AppModule` .

The app displays "Miss Marple" as the user instead of the default "Sherlock Holmes".

Call `forRoot` only in the root module, `AppModule`. Calling it in any other NgModule, particularly in a lazy-loaded NgModule, is contrary to the intent and can produce a runtime error. Remember to _import_ the result; don't add it to any other `@NgModule` list.

{@a prevent-reimport}

Prevent reimport of the *CoreModule*

Only the root `AppModule` should import the `CoreModule` . Bad things happen if a lazy-loaded NgModule imports it.

You could hope that no developer makes that mistake. Or you can guard against it and fail fast by adding the following `CoreModule` constructor.

The constructor tells Angular to inject the `CoreModule` into itself. That seems dangerously circular.

The injection would be circular if Angular looked for `CoreModule` in the *current* injector. The `@SkipSelf` decorator means "look for `CoreModule` in an ancestor injector, above me in the injector hierarchy."

If the constructor executes as intended in the `AppModule`, there is no ancestor injector that could provide an instance of `CoreModule`. The injector should give up.

By default, the injector throws an error when it can't find a requested provider. The `@Optional` decorator means not finding the service is OK. The injector returns `null`, the `parentModule` parameter is null, and the constructor concludes uneventfully.

It's a different story if you improperly import `CoreModule` into a lazy-loaded NgModule such as `HeroModule` (try it).

Angular creates a lazy-loaded NgModule with its own injector, a *child* of the root injector. `@SkipSelf` causes Angular to look for a `CoreModule` in the parent injector, which this time is the root injector. Of course it finds the instance imported by the root `AppModule`. Now `parentModule` exists and the constructor throws the error.

Conclusion

You made it! You can examine and download the complete source for this final version from the live example.

Frequently asked questions

Now that you understand NgModules, you may be interested in the companion NgModule FAQs page with its ready answers to specific design and implementation questions.