Guidelines for creating NgModules

This topic provides a conceptual overview of the different categories of NgModules you can create in order to organize your code in a modular structure. These categories are not cast in stone—they are suggestions. You may want to create NgModules for other purposes, or combine the characteristics of some of these categories.

NgModules are a great way to organize an application and keep code related to a specific functionality or feature separate from other code. Use NgModules to consolidate components, directives, and pipes into cohesive blocks of functionality. Focus each block on a feature or business domain, a workflow or navigation flow, a common collection of utilities, or one or more providers for services.

For more about NgModules, see Organizing your app with NgModules.

For the example application used in NgModules-related topics, see the .

Summary of NgModule categories

All applications start by bootstrapping a root NgModule. You can organize your other NgModules any way you want.

This topic provides some guidelines for the following general categories of NgModules:

- Domain: A domain NgModule is organized around a feature, business domain, or user experience.
- Routed: The top component of the NgModule acts as the destination of a router navigation route.
- Routing: A routing NgModule provides the routing configuration for another NgModule.
- Service: A service NgModule provides utility services such as data access and messaging.
- Widget: A widget NgModule makes a component, directive, or pipe available to other NgModules.
- Shared: A shared NgModule makes a set of components, directives, and pipes available to other NgModules.

The following table summarizes the key characteristics of each category.

NgModule	Declarations	Providers	Exports	Imported by
Domain	Yes	Rare	Top component	Another domain, AppModule
Routed	Yes	Rare	No	None
Routing	No	Yes (Guards)	RouterModule	Another domain (for routing)
Service	No	Yes	No	AppModule
Widget	Yes	Rare	Yes	Another domain
Shared	Yes	No	Yes	Another domain

{@a domain}

Domain NgModules

Use a domain NgModule to deliver a user experience dedicated to a particular feature or application domain, such as editing a customer or placing an order. One example is `ContactModule` in the .

A domain NgModule organizes the code related to a certain function, containing all of the components, routing, and templates that make up the function. Your top component in the domain NgModule acts as the feature or domain's root, and is the only component you export. Private supporting subcomponents descend from it.

Import a domain NgModule exactly once into another NgModule, such as a domain NgModule, or into the root NgModule (`AppModule`) of an application that contains only a few NgModules.

Domain NgModules consist mostly of declarations. You rarely include providers. If you do, the lifetime of the provided services should be the same as the lifetime of the NgModule.

For more information about lifecycles, see Hooking into the component lifecycle.

{@a routed}

Routed NgModules

Use a routed NgModule for all lazy-loaded NgModules. Use the top component of the NgModule as the destination of a router navigation route. Routed NgModules don't export anything because their components never appear in the template of an external component.

Don't import a lazy-loaded routed NgModule into another NgModule, as this would trigger an eager load, defeating the purpose of lazy loading.

Routed NgModules rarely have providers because you load a routed NgModule only when needed (such as for routing). Services listed in the NgModules' `provider` array would not be available because the root injector wouldn't know about the lazy-loaded NgModule. If you include providers, the lifetime of the provided services should be the same as the lifetime of the NgModule. Don't provide app-wide singleton services in a routed NgModule or in an NgModule that the routed NgModule imports.

For more information about providers and lazy-loaded routed NgModules, see Limiting provider scope.

{@a routing}

Routing NgModules

Use a routing NgModule to provide the routing configuration for a domain NgModule, thereby separating routing concerns from its companion domain NgModule. One example is `ContactRoutingModule` in the , which provides the routing for its companion domain NgModule `ContactModule` .

For an overview and details about routing, see In-app navigation: routing to views.

Use a routing NgModule to do the following tasks:

- Define routes.
- Add router configuration to the NgModule's import.
- Add guard and resolver service providers to the NgModule's providers.

The name of the routing NgModule should parallel the name of its companion NgModule, using the suffix `Routing` . For example, `ContactModule` in `contact.module.ts` has a routing NgModule named `ContactRoutingModule` in `contact-routing.module.ts` .

Import a routing NgModule only into its companion NgModule. If the companion NgModule is the root `AppModule` , the `AppRoutingModule` adds router configuration to its imports with `RouterModule.forRoot(routes)` . All other routing NgModules are children that import `RouterModule.forChild(routes)` .

In your routing NgModule, re-export the `RouterModule` as a convenience so that components of the companion NgModule have access to router directives such as `RouterLink` and `RouterOutlet` .

Don't use declarations in a routing NgModule. Components, directives, and pipes are the responsibility of the companion domain NgModule, not the routing NgModule.

{@a service}

Service NgModules

Use a service NgModule to provide a utility service such as data access or messaging. Ideal service NgModules consist entirely of providers and have no declarations. Angular's `HttpClientModule` is a good example of a service NgModule.

Use only the root `AppModule` to import service NgModules.

{@a widget}

Widget NgModules

Use a widget NgModule to make a component, directive, or pipe available to external NgModules. Import widget NgModules into any NgModules that need the widgets in their templates. Many third-party UI component libraries are provided as widget NgModules.

A widget NgModule should consist entirely of declarations, most of them exported. It would rarely have providers.

{@a shared}

Shared NgModules

Put commonly used directives, pipes, and components into one NgModule, typically named `SharedModule`, and then import just that NgModule wherever you need it in other parts of your application. You can import the shared NgModule in your domain NgModules, including lazy-loaded NgModules. One example is `SharedModule` in the , which provides the `AwesomePipe` custom pipe and `HighlightDirective` directive.

Shared NgModules should not include providers, nor should any of its imported or re-exported NgModules include providers.

To learn how to use shared modules to organize and streamline your code, see Sharing NgModules in an app.

Next steps

You may also be interested in the following:

- For more about NgModules, see Organizing your app with NgModules.
- To learn more about the root NgModule, see Launching an app with a root NgModule.
- To learn about frequently used Angular NgModules and how to import them into your app, see Frequently-used modules.
- For a complete description of the NgModule metadata properties, see Using the NgModule metadata.

If you want to manage NgModule loading and the use of dependencies and services, see the following:

- To learn about loading NgModules eagerly when the application starts, or lazy-loading NgModules asynchronously by the router, see Lazy-loading feature modules.
- To understand how to provide a service or other dependency for your app, see Providing Dependencies for an NgModule.
- To learn how to create a singleton service to use in NgModules, see Making a service a singleton.