Lazy-loading feature modules

By default, NgModules are eagerly loaded, which means that as soon as the application loads, so do all the NgModules, whether or not they are immediately necessary. For large applications with lots of routes, consider lazy loading—a design pattern that loads NgModules as needed. Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.

For the final sample application with two lazy-loaded modules that this page describes, see the .

{@a lazy-loading}

Lazy loading basics

This section introduces the basic procedure for configuring a lazy-loaded route. For a step-by-step example, see the step-by-step setup section on this page.

To lazy load Angular modules, use `loadChildren` (instead of `component`) in your `AppRoutingModule` `routes` configuration as follows.

const routes: Routes = [{ path: 'items', loadChildren: () => import('./items/items.module').then(m => m.ItemsModule) }];

In the lazy-loaded module's routing module, add a route for the component.

const routes: Routes = [{ path: ", component: ItemsComponent }];

Also be sure to remove the `ItemsModule` from the `AppModule`. For step-by-step instructions on lazy loading modules, continue with the following sections of this page.

{@a step-by-step}

Step-by-step setup

There are two main steps to setting up a lazy-loaded feature module:

1. Create the feature module with the CLI, using the `--route` flag.
2. Configure the routes.

Set up an app

If you don't already have an app, follow the following steps to create one with the CLI. If you already have an app, skip to Configure the routes. Enter the following command where `customer-app` is the name of your app:

ng new customer-app –routing

This creates an application called `customer-app` and the `--routing` flag generates a file called `app-routing.module.ts`, which is one of the files you need for setting up lazy loading for your feature module. Navigate into the project by issuing the command `cd customer-app`.

The `--routing` option requires Angular/CLI version 8.1 or higher. See Keeping Up to Date.

Create a feature module with routing

Next, you'll need a feature module with a component to route to. To make one, enter the following command in the terminal, where `customers` is the name of the feature module. The path for loading the `customers` feature modules is also `customers` because it is specified with the `--route` option:

ng generate module customers –route customers –module app.module

This creates a `customers` folder having the new lazy-loadable feature module `CustomersModule` defined in the `customers.module.ts` file and the routing module `CustomersRoutingModule` defined in the `customers-routing.module.ts` file. The command automatically declares the `CustomersComponent` and imports `CustomersRoutingModule` inside the new feature module.

Because the new module is meant to be lazy-loaded, the command does NOT add a reference to the new feature module in the application's root module file, `app.module.ts`. Instead, it adds the declared route, `customers` to the `routes` array declared in the module provided as the `--module` option.

Notice that the lazy-loading syntax uses `loadChildren` followed by a function that uses the browser's built-in `import('...')` syntax for dynamic imports. The import path is the relative path to the module.

String-based lazy loading

In Angular version 8, the string syntax for the `loadChildren` route specification was deprecated in favor of the `import()` syntax. However, you can opt into using string-based lazy loading (`loadChildren: './path/to/module#Module'`) by including the lazy-loaded routes in your `tsconfig` file, which includes the lazy-loaded files in the compilation.

By default the CLI generates projects with stricter file inclusions intended to be used with the `import()` syntax.

Add another feature module

Use the same command to create a second lazy-loaded feature module with routing, along with its stub component.

ng generate module orders –route orders –module app.module

This creates a new folder called `orders` containing the `OrdersModule` and `OrdersRoutingModule`, along with the new `OrdersComponent` source files. The `orders` route, specified with the `--route` option, is added to the `routes` array inside the `app-routing.module.ts` file, using the lazy-loading syntax.

Set up the UI

Though you can type the URL into the address bar, a navigation UI is straightforward for the user and more common. Replace the default placeholder markup in `app.component.html` with a custom nav so you can navigate to your modules in the browser:

To see your application in the browser so far, enter the following command in the terminal window:

ng serve

Then go to `localhost:4200` where you should see "customer-app" and three buttons.

These buttons work, because the CLI automatically added the routes to the feature modules to the `routes` array in `app-routing.module.ts`.

{@a config-routes}

Imports and route configuration

The CLI automatically added each feature module to the routes map at the application level. Finish this off by adding the default route. In the `app-routing.module.ts` file, update the `routes` array with the following:

The first two paths are the routes to the `CustomersModule` and the `OrdersModule`. The final entry defines a default route. The empty path matches everything that doesn't match an earlier path.

Inside the feature module

Next, take a look at the `customers.module.ts` file. If you're using the CLI and following the steps outlined in this page, you don't have to do anything here.

The `customers.module.ts` file imports the `customers-routing.module.ts` and `customers.component.ts` files. `CustomersRoutingModule` is listed in the `@NgModule` `imports` array giving `CustomersModule` access to its own routing module. `CustomersComponent` is in the `declarations` array, which means `CustomersComponent` belongs to the `CustomersModule`.

The `app-routing.module.ts` then imports the feature module, `customers.module.ts` using JavaScript's dynamic import.

The feature-specific route definition file `customers-routing.module.ts` imports its own feature component defined in the `customers.component.ts` file, along with the other JavaScript import statements. It then maps the empty path to the `CustomersComponent`.

The `path` here is set to an empty string because the path in `AppRoutingModule` is already set to `customers`, so this route in the `CustomersRoutingModule`, is already within the `customers` context. Every route in this routing module is a child route.

The other feature module's routing module is configured similarly.

Verify lazy loading

You can check to see that a module is indeed being lazy loaded with the Chrome developer tools. In Chrome, open the developer tools by pressing `Cmd+Option+i` on a Mac or `Ctrl+Shift+j` on a PC and go to the Network Tab.

Click on the Orders or Customers button. If you see a chunk appear, everything is wired up properly and the feature module is being lazy loaded. A chunk should appear for Orders and for Customers but only appears once for each.

To see it again, or to test after working in the project, clear everything out by clicking the circle with a line through it in the upper left of the Network Tab:

Then reload with `Cmd+r` or `Ctrl+r`, depending on your platform.

`forRoot()` and `forChild()`

You might have noticed that the CLI adds `RouterModule.forRoot(routes)` to the `AppRoutingModule imports` array. This lets Angular know that the `AppRoutingModule` is a routing module and `forRoot()` specifies that this is the root routing module. It configures all the routes you pass to it, gives you access to the router directives, and registers the `Router` service. Use `forRoot()` only once in the application, inside the `AppRoutingModule`.

The CLI also adds `RouterModule.forChild(routes)` to feature routing modules. This way, Angular knows that the route list is only responsible for providing additional routes and is intended for feature modules. You can use `forChild()` in multiple modules.

The `forRoot()` method takes care of the *global* injector configuration for the Router. The `forChild()` method has no injector configuration. It uses directives such as `RouterOutlet` and `RouterLink`. For more information, see the `forRoot()` pattern section of the Singleton Services guide.

{@a preloading}

Preloading

Preloading improves UX by loading parts of your application in the background. You can preload modules or component data.

Preloading modules

Preloading modules improves UX by loading parts of your application in the background so users don't have to wait for the elements to download when they activate a route.

To enable preloading of all lazy loaded modules, import the `PreloadAllModules` token from the Angular `router`.

import { PreloadAllModules } from '@angular/router';

Still in the `AppRoutingModule`, specify your preloading strategy in `forRoot()`.

RouterModule.forRoot(appRoutes, { preloadingStrategy: PreloadAllModules })

Preloading component data

To preload component data, use a `resolver`. Resolvers improve UX by blocking the page load until all necessary data is available to fully display the page.

Resolvers Create a resolver service. With the CLI, the command to generate a service is as follows:

ng generate service

In the newly-created service, implement the `Resolve` interface provided by the `@angular/router` package:

import { Resolve } from '@angular/router';

. . .

/* An interface that represents your data model */ export interface Crisis { id: number; name: string; }

export class CrisisDetailResolverService implements Resolve { resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable { // your logic goes here } }

Import this resolver into your module's routing module.

import { CrisisDetailResolverService } from './crisis-detail-resolver.service';

Add a `resolve` object to the component's `route` configuration.

{ path: '/your-path', component: YourComponent, resolve: { crisis: CrisisDetailResolverService } }

In the component's constructor, inject an instance of the `ActivatedRoute` class that represents the current route.

import { ActivatedRoute } from '@angular/router';

@Component({ . . . }) class YourComponent { constructor(private route: ActivatedRoute) {} }

Use the injected instance of the `ActivatedRoute` class to access `data` associated with a given route.

import { ActivatedRoute } from '@angular/router';

@Component({ . . . }) class YourComponent { constructor(private route: ActivatedRoute) {}

ngOnInit() { this.route.data .subscribe(data => { const crisis: Crisis = data.crisis; // . . . }); } }

For more information with a working example, see the routing tutorial section on preloading.

Troubleshooting lazy-loading modules

A common error when lazy-loading modules is importing common modules in multiple places within an application. Test for this condition by first generating the module using the Angular CLI and including the `--route route-name` parameter, where `route-name` is the name of your module. Next, generate the module without the `--route` parameter. If the Angular CLI generates an error when you use the `--route` parameter, but runs correctly without it, you might have imported the same module in multiple places.

Remember, many common Angular modules should be imported at the base of your application.

For more information on Angular Modules, see NgModules.

More on NgModules and routing

You might also be interested in the following: * Routing and Navigation. * Providers. * Types of Feature Modules. * Route-level code-splitting in Angular * Route preloading strategies in Angular