Section 22: Angular Modules & Optimizing App: Table of Contents

[308. Module Introduction 1](#_Toc76591571)

[310. Understanding the Different Approaches 2](#_Toc76591572)

[312. Preparing Programmatic Creation 2](#_Toc76591573)

[313. Creating a Component Programmatically 3](#_Toc76591574)

[314. Understanding entryComponents 3](#_Toc76591575)

[315. Data Binding & Event Binding 3](#_Toc76591576)

[317. Useful Resources & Links 4](#_Toc76591577)

# 319. What are Modules?

Modules are what Angular uses (in particular,@NgModules) to understand your application and its features.

Modules are used to define all of the building blocks that your app uses -> Components, Directives, and even Services.

Every application needs at least one module - the app module, but they can also split features into different modules as desired.

Core Angular features, such as the FormsModule, are also able to be loaded in so you don't have to declare all of the module's components/directives individually.

Lastly, a building block (component, directive, etc) cannot be used without first including it in a module.

# 320. Analyzing the App Module

In the @NgModule decorator of a module, we are have a complete understanding of the declarations and imports arrays. For the bootstrap array property, however, we use it to denote the component used for starting our app. It defines what component will be immediately available via the index.html file. You can include more than one component in the bootstrap array, but each component would be detached and have their own root component trees within the application, which ends up messy for component communication and is not advised.

In order to split up your modules and keep them leaner, you need the exports property in @NgModule. Every module works on its own in Angular and does not communicate with other modules. You can only use components in the modules that they are declared in and nowhere else.

# 321. Getting Started with Feature Modules

Beyond the App Module, we have Feature Modules. Feature Modules are not created in a special way. They are just modules that group together components, directives, pipes that are used in a certain feature area of our application.

Splitting your app into Feature Modules is a prerequisite for performance improvements, and ensures that you or other developers quickly figure out where to look for code for specific parts of the app.

If you create a Feature Module with all of the declarations moved in from the app module, and then add all of these declarations to the exports so that they can be used by the app module, you'll get some errors. And that's because these modules work as standalone entities, so things like the FormsModule or the AppRoutingModule that are referenced in the new Feature Module aren't automatically loaded in, and thus will fail.

# 322. Splitting Modules Correctly

After adding the RouterModule to our Feature Module imports array, we clear that error but we still end up getting other errors for features that are made available via other common Angular modules such as the BrowserModule, FormsModule, and ReactiveFormsModule. The HttpClientModule is an exception to this since it only provides services, which work differently and can be used application-wide. These other modules need to be added to the imports if they are used at all in the new Feature Module

We need the BrowserModule in our new Feature Module to use \*ngIf and \*ngFor, but the BrowserModule is a special case and can only be imported once - in the app module. Therefore, all Feature Modules will need to import the CommonModule instead.

# 323. Adding Routes to Feature Modules

With Feature Modules in a project, it can be good practice to handle routing individually in each of the Feature Modules. So we can create a RecipesRoutingModule, for instance, and import the RouterModule and move the recipes routes from the app routing module. However, in order to keep the routing synced between the app routing module and feature routing modules, you want to use the forChild() method on the RouterModule import within the feature routing modules, rather than the for forRoot() method that was used in the app routing module.

Ex in RecipesRoutingModule:

1. @NgModule({
2. imports: [RouterModule.forChild(routes)],
3. exports: [RouterModule]
4. })

# 324. Component Declarations

In the Feature Module, there is no reason to export all of the components now that we have routing handled within our Feature's own routing module, meaning we can delete the exports property content. The app will work all the same.

# 326. Understanding Shared Modules

It can be a good idea to create a shared module to store any components, directives, pipes, or Angular Modules that are commonly used by some of your Feature modules. However, when moving these items to a new shared module, you always want to make sure that you don't have any declarations items listed in more than one module, or you will encounter errors.

# 327. Understanding the Core Module

One pattern that you can use would be the inclusion of a core module for storing all of the services being provided. This is largely unnecessary if you are providing all of your services within their service.ts files. However, if not, it can be useful for keeping the app module lean and having all of your services accounted for in once place. In a core module, all you need to do is provide the services (and make sure they aren't provided elsewhere). There is no need to export anything since services, unlike components, directives, and pipes, can be accessed from anywhere. However, the core module does need to be added to the app module imports.

# 329. Understanding Lazy Loading

To start with optimization techniques, we'll first look at using something called Lazy Loading, for which having Feature Modules is a pre-requisite.

Without lazy loading, any time we visit any page, we load everything. Lazy Loading aims to help with just loading what is relevant to a particular page when it is really needed.

With lazy loading, we initially only load our root route (the app module and its components. Only when we visit another module do we load that module's code and its components' code. The advantage is that we download smaller code bundles initially, and download other small pieces when we need them. This gives faster start times overall.

# 330. Implementing Lazy Loading

In order to implement Lazy Loading, you'll need to make the primary path empty within the Feature Model, and add the feature path to the primary app routing module. However, you won't add a component in the app a routing mode - instead you need the loadChildren property. With older versions of Angular, you'll want to put a string with the path name to the feature module.ts file, followed by # and then then the module name. Ex:

1. { path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule' }

With newer versions of Angular, you'd want to set loadChildren as a function with a dynamic import of the feature module, and then a callback to return the module class from the file. Ex:

1. {
2. path: 'recipes',
3. loadChildren: () => import('./recipes/recipes.module').then(m => m.RecipesModule)
4. }

Since we're not pointing to the Feature Module in the app router, the feature model reference in the app module can be removed.

# 332. Preloading Lazy-Loaded-Code

Although lazy loading is an optimization technique, there are some issues. With slower internet connections and bigger modules, you can end up having delays in the app. Therefore, you can tell Angular to preload Lazy Loaded modules to avoid delays. To do this, add an object as a second input to the forRoot() method on the RouterModule within the app routing module file. In there, you can set the preloadingStrategy property to PreloadAllModules, a const imported from '@angular/router'.

While code bundles are still separated with this strategy, Angular attempts to preload bundles as soon as possible. This keeps the initial download bundle small and fast, and then utilized user idle time to get other modules ready and make sure they are fast as well.

# 333. Modules & Services

For services that are provided in the app module or the @Injectable() decorator of it's own file, a single instance of the service is available app-wide and uses the root injector. This method is the default. The same is true if a service is provided in an eager-loaded module, since the code is all initialized in a single bundle. Loading services in eager-loaded modules should be avoided typically, though.

When provided in a component, a component-specific injector is used and the service instance is only available for the specific component tree. Injection into other sibling components will provide new instances for those component trees. This method should be used if a service is only relevant for a specific component tree.

However, with lazy-loaded modules, service instances become available in the loaded modules only and use a child injector. So if a service was provided at both the app level as well as a lazy-loaded modules, there will be a separate instance for the lazy-loaded module. This should be used if a service should be scoped to only the loaded feature module.

# 334. Loading Services Differently

One of the biggest things you need to be careful with when providing services is providing them within a shared module that is imported by several other module. For example, our shared module is imported by both the app module and shopping list module. When imported to the app module, our service is imported in an eager-loaded module, making it available app-wide. However, since it is also imported to the shopping list module, which is lazy-loaded, a new instance of the service is created within the shopping list module, which might not necessarily have been intended.

# 335. Ahead-of-Time Compilation

After the Typescript compiler compiles your code to JavaScript, the Angular compiler does a large amount of work behind the scenes. It compiles all angular template syntax into JavaScript DOM instructions for the user, and is automatically included in built Angular project code.

Angular compilation by default all happens in your browser at runtime since the compiler is part of the code you're shipping. This is called Just-in-Time (JiT) Compilation. The downside to this is that it takes time to compile.

However, we can use Ahead-of-Time (AoT) compilation to do the compiling as part of our development/build process, before app deployment.

To build our app, we will use the command line command: ng build --prod. Instead of creating a development version in the localhost browser, a few files are generated to run our code. This command automatically does Ahead-of-Time Compilation.

One issue that the AoT compiler might find is when you use a .get() method on a FormArray in the template. Typescript doesn't understand that it is trying to access the controls property on a FormArray, so you have to define your getter in the component file and cast it as a FormArray. Ex:

1. get controls() { // a getter!
2. return (this.recipeForm.get('ingredients') as FormArray).controls;
3. }

# 337. Useful Resources & Links

Useful Resources:

* Official Docs: <https://angular.io/guide/ngmodules>
* NgModules FAQ: <https://angular.io/guide/ngmodule-faq>