**Using Angular Modudes and Optimizing Apps**

Our application has a couple of components, Directives, Pipes, Services, things like that. The idea behind it is you clearly have to tell angular what your app consists of. So What are the elements of your app. Which components to use, Which directives do you use, Which services do you use. And by using modules and being clear about what you need, you also are very clear about what you don’t need. That’s the idea behind the modules and right now we only use one module in our app. That’s not wrong or bad but we can improve our app by using multiple modules

**Understanding the App Module**

In App module have: declarations array, import array, providers array and bootstrap array

Declaration: we define which components, directives or pipes our app

AppComponent,

HeaderComponent,

RecipesComponent,

RecipeListComponent,

RecipeDetailComponent,

RecipeItemComponent,

ShoppingListComponent,

ShoppingEditComponent,

DropdownDirective,

ItemdecorateDirective,

RecipeStartComponent,

RecipeEditComponent,

SignupComponent,

SigninComponent

],

Import: We define modules which is from angular ship and also contain routing module

imports: [

BrowserModule,

FormsModule,

AppRoutingModule,

ReactiveFormsModule,

HttpModule

],

Providers: what services we may use in the Module, every thing we provide here, it mean we provide for whole application. If you think about how a dependency injection worked then we will use in the same instance

Bootstrap: simply defines what’s our root component, the root component is something different than the root module

**Understanding Feature Modules**

Now I want to dive into the question which additional modules we may build and why a typical module you may add to your application, so a custom module built by you would be a feature module. **Separate component to feature module to make our code to be clearer**

**Create a Recipe Feature Module**

Browser Module basically contains all the features of the common module , therefore you should only use browser module in the app module and common module in all our modules

@NgModule({

declarations : [

RecipesComponent,

RecipeStartComponent,

RecipeListComponent,

RecipeEditComponent,

RecipeDetailComponent,

RecipeItemComponent,

],

imports: [

CommonModule,

ReactiveFormsModule,

],

providers: [],

exports: [],

bootstrap: []

})

export class RecipesModule {

}

Can’t have duplicate declaration between two modules

**But we remain get another error regarding the routing because you have keep in mind that in our recipes component we also have a router outlet:** and while we do define our routes here in the app module, we import the app-routing module here and that’s not enough. That doesn’t travel down to our recipe module.

**Keep in mind a module is only able to use what we define in that module. And Services and kind of an exception. Therefore this is not working with the routing**

**Register Routes in a Feature Module**

const recipesRouting: Routes = [

{

path: 'recipes', component: RecipesComponent,

children: [

{path: '', component: RecipeStartComponent},

{path: 'new', canActivate: [AuthGuard], component: RecipeEditComponent, },

{path: ':id', component: RecipeDetailComponent},

{path: ':id/edit', canActivate: [AuthGuard] , component: RecipeEditComponent, }

]

}

]

@NgModule({

imports: [

RouterModule.forChild(recipesRouting)

],

exports: [RouterModule]

})

export class RecipesRoutingModule {

}

RouterModule must forChild, not for Root, But It remain doesn’t work perfectly because, they are missing dropdowndirective, but if we put dropdown directive to recipe component, it will error, because that directive are used at header. So we should have common or shared module

**Understanding Shared Modules**

@NgModule({

declarations: [

DropdownDirective,

ItemdecorateDirective

],

exports : [

DropdownDirective, ItemdecorateDirective

]

})

export class SharedModule {

}

Do you know why must we export DropdownDirective and ItemdecorateDirective?

The declaring it here is very important because every component, directive or pipe you are going to use has to be declare somewhere in our application in some module, it has to be declared one, only one time, not multiple times. However the ideal behind shared module is that we will now be able to import shared module into the other module and **therefore to be able to use dropdown directive** we also have to export it**, because keep in mind by default everything you set up in a module is only available in that module** and it isn’t accessible from outside to make some features some components accessible from outside

@NgModule({

declarations: [

DropdownDirective,

ItemdecorateDirective

],

exports : [

CommonModule,

DropdownDirective, ItemdecorateDirective

]

})

export class SharedModule {

}

**Loading Components via Selectors vs Routing**

We do reference a shopping list component here (app-routing-module) and therefore we have to add or leave import in this file because as mentioned, the import here is a language feature and typescript need to know where it find this component where it finds this type we are referencing here. **So since this is only a typescript thing and the shopping list component is declared in your shopping list module, you could correctly say well how can we use it(shopping-list-component) in the app-routing-module then because the app-routing-module is not directly related to shopping-list-module, we declare shopping-list-component somewhere and then we use it, That’s the special thing about routing and** That’s the same for all components

In a routing. It’s not important that you declared a component in the same file as to route live. It’s just important that you declared him anywhere in your application before you get a chance of visiting that route. It is very good and dynamic

**But If you decide to use the selector**

If you decide to use the selector of a component if you decide to use the app shopping list selector in app.component.html, you will get error here because shopping list is not known as an element, because it’s not known to the app component which is a part of app.module because ShoppingList Component isn’t declare in app module

**So That is difference between using the selector and routing for the selector**

**A Common GotCha**

FormsModule located in ShoppingListModule, so we can’t access it any where but signIn and SignOut component are also using FormModule. That is reason it make a problems.

@NgModule({

declarations: [

ShoppingListComponent,

ShoppingEditComponent,

],

imports: [

CommonModule,

FormsModule

],

exports: [

FormsModule

]

})

We must export FormModule from ShoppingListModule to outside, because ShoppingListModule is also included in AppModule, so when you exclude it outside, all component in AppModule can use it.

**But it is implicit**

Let fix it with another way by creating own module for SignIn and SignOut component to can import explicit FormModule to it.

**Improve Performance**

**Understand LazyLoading**

We have a root router which might have some child routes but basically we have our root router where we register routes by forRoot:

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports : [RouterModule]

})

export class AppRoutingModule {

}

We also might have a feature module with its own router, the child routes with 4 child now in our project, this modules here might not be loaded at all times. For example: User never use our shopping list section or the recipe section or in a bigger application maybe the admin section or anything like that. A lot of code might never used because if the user never visits the recipe section so all the code related to that including all the templates and everything will never be used. So we downloaded too much code at the beginning

**That is where lazy loading comes into play**

We can lazy load our feature module and its child router. That means this module is only loaded if we actually visit a route leading to his module. Now let’s take a closer look in the next sections by implanting lazy loading into our app

**Adding Lazy Loading to the Recipe Module**

Add lazy loading to our recipe book application. Now the recipe is load eagerly because we imported it here in app module

imports: [

BrowserModule,

AppRoutingModule,

HttpModule,

RecipesModule, //This one

SharedModule,

ShoppingListModule,

AuthModule

],

We don’t want load RecipesModule eagerly here, we remove it out of array list here , doesn’t add this recipe module to our initial bundle anymore. We want to load it lazily. Now we can load it lazily by going to the app routing file and here we reintroduce a route we had there before. It points to recipe but now we no longer load the recipes component here

At AppRoutingModule

const appRoutes: Routes = [

// {

// path: '', redirectTo: '/recipes', pathMatch: 'full'

// },

{

path: '', component: HomeComponent, pathMatch : 'full'

},

{

path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule'

},

{

path: 'shopping-list', component: ShoppingListComponent

}

]

We use

{

path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule'

},

With recipes.module is name of file module and slash and the name of class RecipesModule

**Then we change little bit in recipes-routing module:**

const recipesRouting: Routes = [

{

path: '', component: RecipesComponent,

children: [

{path: '', component: RecipeStartComponent},

{path: 'new', canActivate: [AuthGuard], component: RecipeEditComponent, },

{path: ':id', component: RecipeDetailComponent},

{path: ':id/edit', canActivate: [AuthGuard] , component: RecipeEditComponent, }

]

}

]

**Protecting Lazy Load with canLoad**

Protecting Lazy Loaded Routes with canLoad

Section 21, Lecture 274

What if you want to use route protection (canActivate  to be precise) on lazily loaded routes?

You can add canActivate to the lazy loaded routes but that of course means, that you might load code which in the end can't get accessed anyways. It would be better to check that BEFORE loading the code.

You can enforce this behavior by adding the canLoad  guard to the route which points to the lazily loaded module:

{ path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule', canLoad: [AuthGuard] }

In this example, the AuthGuard  should implement the [CanLoad interface](https://angular.io/docs/ts/latest/api/router/index/CanLoad-interface.html" \t "_blank).

{

path: 'recipes', loadChildren: './recipes/recipes.module#RecipesModule', canLoad: [AuthGuard]

},

@Injectable()

export class AuthGuard implements CanActivate, CanLoad {

canLoad(route: Route): boolean | Observable<boolean> | Promise<boolean> {

return this.authService.isAuthenticated();

}

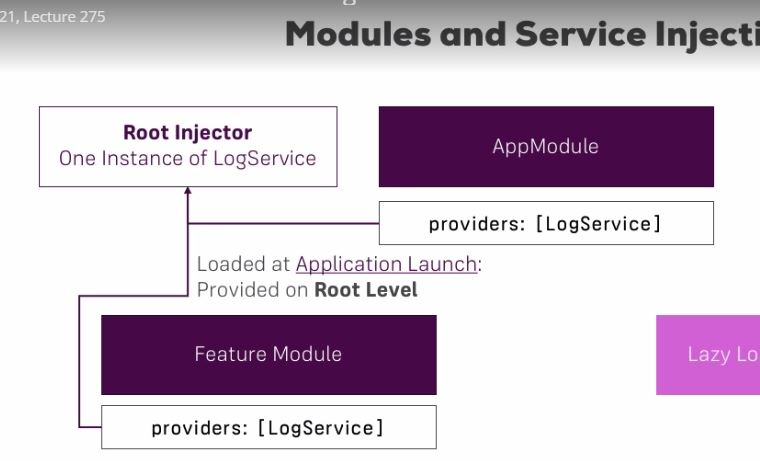
**Module and Service work together**

How module and Injection Services are connected how it works behind the scenes

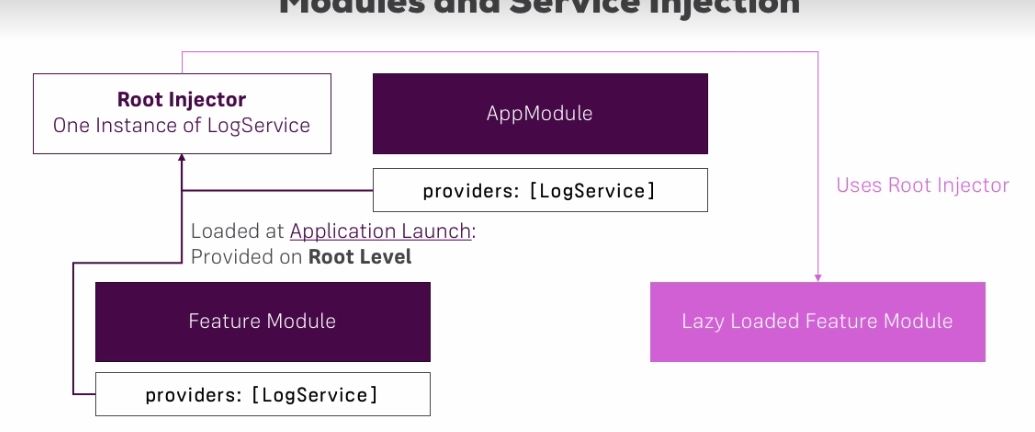
Consider our application at this present: We get the app module and feature module like shopping-list-module which is not loaded lazily in our application right now and the recipe modules and another feature module which is loaded lazily.

Let’s say we provide a log service in this application and we add it to provider’s array to both app module and the feature module. **Something I told you, you probably shoudn’t do but here it won’t hurt you if you do that hehe**. What happens in that in the end we have a root injector for the whole application which created by angular at the point of time the application start and since feature module(shopping-list-module) is loaded not lazily but added to the import or array of the app module basically there is only one root injector and all the services we provide in either of the modules is added to the rule unchecked, that means in the application and the whole application.

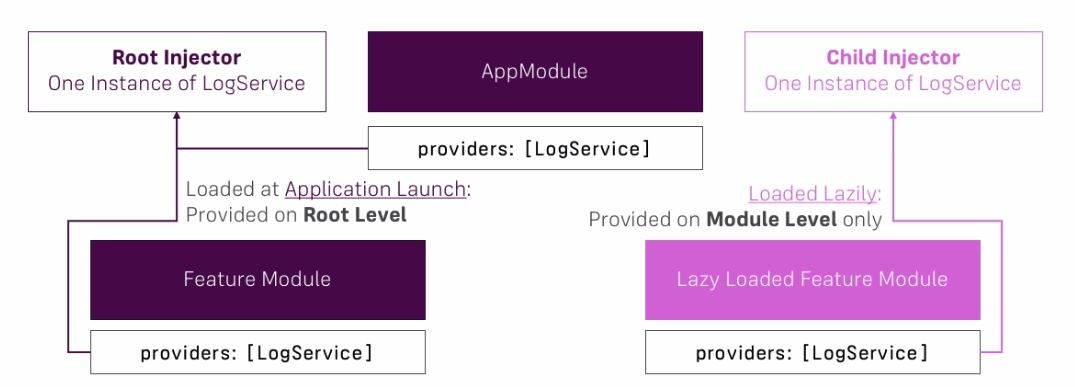
If you now inject the log service you’re going to use one in the same instance and there won’t be a special instance for the feature module here



Now if we were to inject this service in the lazy loaded feature module(recipe-module)

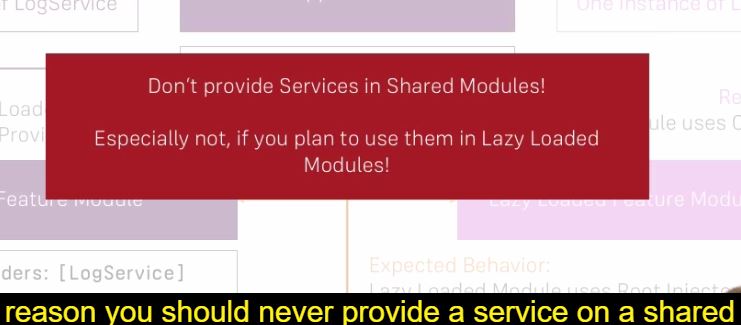


But if every module we declare Log Service:



Angular provide a service in lazy load module so we don’t have same instance. So the fact when you load a module lazy or eager can change how many instances of a service you are going to use And of course therefore it’s absolutely fine to add the provider’s service to lazy loaded module but you have to keep in mind you are not going to use same instance

**Beside that : Just important to keep in mind as a side note if you want to enforce module scope in eager loaded modules to then you’ll simply have to provide the service on that component of this module so other module can use this service**, you can easily to add the provider array in the component decorator of component instead of the module



**Core Module**



**Creating basic core module**

We can also use the core module to basically bundle all our imports and providers in there because unlike the shared module the core module will only be imported by the root module, the app module here

@NgModule({

declarations: [

HeaderComponent,

HomeComponent,

],

imports: [

CommonModule,

SharedModule

],

exports: [

HeaderComponent

]

})

export class CoreModule {

}

Not enough, because header need app-routing-module, but header locate in CoreModule, so we must import AppRoutingModule to this module

imports: [

CommonModule,

SharedModule,

AppRoutingModule

],

**Restruturing services to use Child Injector**

@NgModule({

imports: [

RouterModule.forChild(recipesRouting)

],

exports: [RouterModule],

providers: [AuthGuard]

})

import AuthGuard to Recipe Routing Module because it loaded lazily

**Ahead-of-Time Compilation**

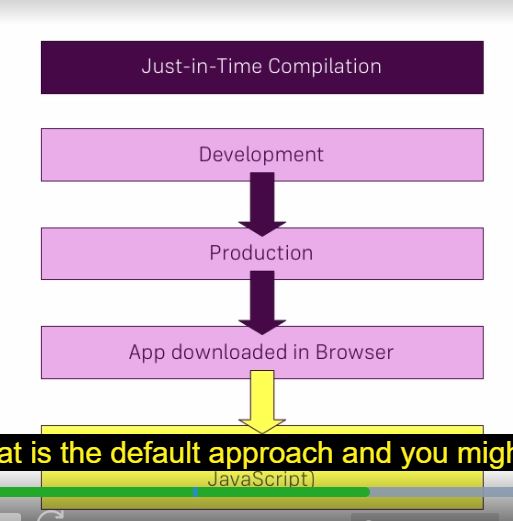
Enough about the modules, you learn how to use module, how to restructure your app using modules. Why that is a good practice and how that makes your module cleaner and more focus and even how you can gain some advantage. Here is another big advantage you can gain ahead of time compilation.

**Basically angular offer two types of compiling your code**

It is not mean compiling typescript to javascript that is done by CI not related to Angular. Angular also needs to compile your templates. You write your template in HTML code, you can say in these template file, Angular parse html files and compiles your HTML code into javascript. You can represent your HTML code in javascript



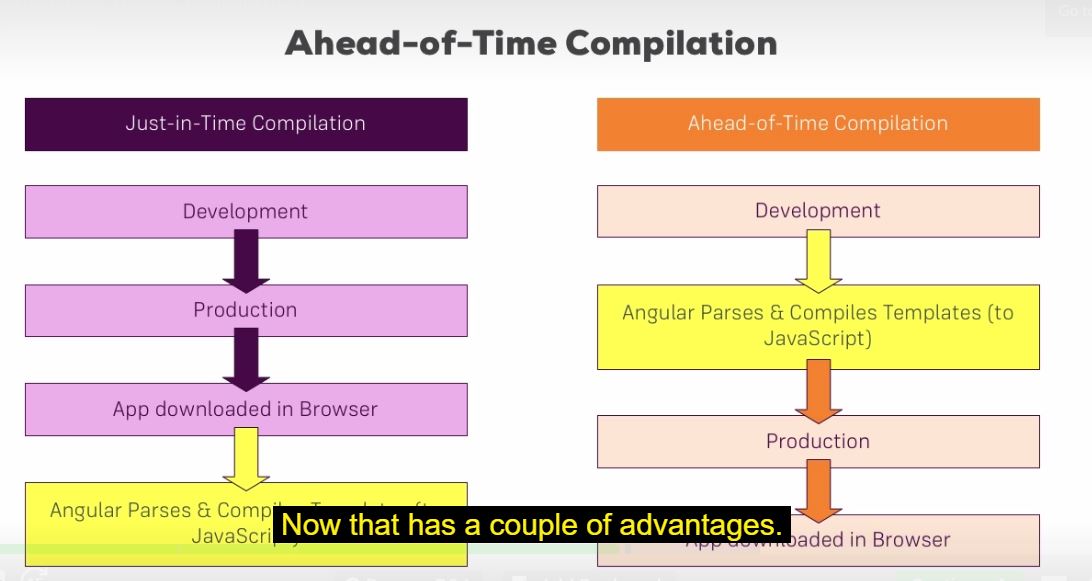
**Default approach**



This is default approach and you migth already see a disadvantage in this approach. I also said id didn’t have a chance of doing this earlier but it only didn’t have a chance because we didn’t let it. We can change the proceduce.

**Ahead-of-Time Compilation**

We still have a development step when using ahead of time compilation **But then we basically allow angular compile the templates because after we’re done developing** . We allow angular understand our templates at an earlier point of time you could say. Sow now our code is already compile and now we ship it to production. So we compile at an earlier point of time ahead of time where we run the app in browser. Now that has a couple of advantages



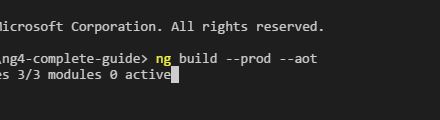
**Advantage of AoT compilation:**

The first advantage is that our application is able to start faster. We don’t have to compile that in the browser anymore because we did it before during development or after developement therefore the application can start faster. It also means that our templates get checked during or right after development when we build our code when we allow angular process it and with checking here. **I means all these errors which you see the javascript console in your running apps otherwise will be thrown in**

**How to use AoT compilation with CLI**

C:\Users\Do My Duyen\Desktop\Capture.JPG

**4.5 MB**



**Preloading Lazy Loaded Routes(use lazy loading but preLoad that code to make it faster)**

We learned How to use module and how to use lazy loading module to also remove the initial chunk, file chunk. Now these are the basic optimizations you should be thinking about in most apps

**If you lazy load in your app you still have that effect that as soon if you visit the app at /recipes you load this whole chunk at this point of time** . So that might give you a little window of time where the application kind of hangs(bi treo) because that code need to be downloaded and depending on the connection that may take a couple of milliseconds or seconds. **So it would be nice if we used lazy loading but still preload to code.** Which mean at the point of time you visit the web page you don’t load the lazy loaded code, but when once the user is using your web page and it’s **using different areas which are not lazy loaded**, you preload lazy load features so that if user decides to lay the Wizard one of these feature areas we have code already available and angular makes implementing better pre-loading strategy

imports: [

RouterModule.forRoot(appRoutes, {preloadingStrategy: })

],

We are using the default pre-loading strategy which is don’t preload:

imports: [

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

],

This is a strategy which as the name implies pre-load is all lazy-loading modules after the app has been loaded. **So not at the point of time you load the app initially but once it runs**