Agenda

- Root App module
- Ahead-Of-Time(AOT) Compilation
- Feature modules
- Lazy Loading a Module
- Shared Module

Root App module

- An **NgModule** class is adorned with **@NgModule** decorator function this will tell the angular application how to compile and run the module code.
- Every angular application will have at least one **NgModule** and every angular app has one **root module** class.
- The root module class is standardly termed as **AppModule** we can name it with any name, it exists in **app.module.ts** file.

File: **app.module.ts**

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';  //importing app.component module
@NgModule({
        imports: [BrowserModule],
        declarations: [AppComponent],
        bootstrap: [AppComponent]
})
export class AppModule {  }
```

@NgModule decorator identifies **AppModule** class as **Angular Module class** and it takes the metadata object that tells how to compile and launch the application.

Some of the mostly used modules are **HttpModule** which contains **Http Services, RouterModule** which has router, **BrowserModule** which is needed to execute the application in browser.

The important properties are:

- **imports** - other modules whose exported classes are needed by component templates declared in this module.
- **declarations** - the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.
- **bootstrap** - the main application view, called the root component, that hosts all other app views. Only the **root** module should set this bootstrap property.

- **exports** - the subset of declarations that should be visible and usable in the component templates of other modules. You *can* export any declarable class—components, directives, and pipes—whether it's declared in this module or in an imported module. You *can* re-export entire imported modules, which effectively re-exports all of their exported classes. A module can even export a module that it doesn't import.
- **providers** - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.

Bootstrap in main.ts

Bootstrapping an angular application can be done in many ways, the variations depends upon where we compile the application and where we run it. One of the way is by using **JIT compiler** and its recommended place is in **src/main.ts.**

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

In the previous section we have discussed that **export** keyword for **AppModule** is not required, but it's only required when we need to bootstrap the JIT Compiled based application in a separate file which is **src/main.ts**.

Important key points to be discussed are

- Here we are importing **platformBrowserDynamic** which creates the browser platform for dynamic **JIT** compilations and to bootstrap the **AppModule.**
- The bootstrapping process sets up the execution environment, to place the component selector in the **HTML DOM** first it will dig out the **AppComponent** from the module's bootstrap array and create an instance for your component and then it will insert the element tag.

AOT (Ahead-Of-Compilation) Compilation

- This is one of the compilations like **JIT compiler**, but the difference is **JIT Compiler** is a dynamic online compiler and the platform Browser has a built-in compiler, whereas **AOT** is **static offline compiler** and browser platform doesn't have a compiler.
- The static alternative can produce a much smaller application that launches faster, especially on mobile devices and high latency networks. In the static option, the Angular compiler runs ahead of time as part of the build process, producing a collection of class factories in their own files. Among them is the **AppModuleNgFactory**.
- Both AOT and JIT generate an AppModuleNgFactory class from the same AppModule source code. JIT compiler creates that factory class in the browser but AOT creates it in a physical file and that will be imported form **main.ts.**

In general, the `AppModule` should neither know nor care how it is bootstrapped.

```
// The browser platform without a compiler
```

```
import { platformBrowser } from '@angular/platform-browser';
// The app module factory produced by the static offline compiler
import { AppModuleNgFactory } from './app/app.module.ngfactory';
// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

Feature Modules

- When our root **AppModule** starts growing larger with application classes and its components then it becomes hard to maintain all the components in a single module file **app.module.ts.**
- In these conditions, our **app lacks clear boundaries** and due to this it becomes **harder to assign development responsibilities.**
- To overcome these issues angular has introduced feature modules, a feature module is a class adorned by the **@NgModule** decorator and its metadata, just like a root module**. Feature module** metadata have the **same properties as** the metadata for a **root module**.
- They **share the same dependency injectors**, i.e., services in one module are also available to other modules.
- Main differences between root module and feature module are,
  1. **Root module launches the application** whereas **feature module** will get imported to **extend the application.**
  2. **Feature module** can expose or **hide its implementation** from other modules.
  3. **Feature module helps us to partition the application** into various sets of functionalities that will be focused on an application business domain, user work flow, and facilities like forms, http, routing (or) a collection of related utilities. Whereas **we can do everything within root module.**
  4. **Feature module** doesn't import **BrowerModule** but **CommonModule** because feature is just a part of **RootModule** so no need to import Browser Module in FeatureModules. To make our feature module to be **platform independent** we only need to import CommonModule**.**

Example:

1) Let's create two new folder under **/src/app**

   Folder: **/src/app/Employee**

   Folder: **/src/app/Department**

2) And you can add **directives, pipes, services, components** etc. as per the requirement in respective folders individually as follows.

3) Shared Module

File: Shared.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AgePipe } from './age.pipe';
import { JoinPipe } from './join.pipe';
import { ShowAlertDirective } from './showalert.directive';

@NgModule({
    imports: [CommonModule],
    declarations: [ShowAlertDirective, JoinPipe, AgePipe],
    exports: [ShowAlertDirective, JoinPipe, AgePipe, CommonModule]
})
export class SharedModule { }
```

4) Employee Routing Module.

File: employee-routing.moudule.ts

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

import { EmployeeComponent } from './employee.component';

@NgModule({
    imports: [RouterModule.forChild([
        { path: 'emp', component: EmployeeComponent },
        { path: 'emp/:id', component: EmployeeComponent }
    ])],
```

```
    declarations: [],
    exports: [RouterModule]
})
export class EmployeeRoutingModule { }
```

5) Employee Module

File: emp.module.ts importing my employee services

```
import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { EmployeeComponent } from "./employee.component"
import { EmployeeRoutingModule } from './employee-routing.module';

@NgModule({
    imports: [SharedModule, EmployeeRoutingModule],
    declarations: [EmployeeComponent],
})
export class EmployeeModule { }
```

6) App Routing Module

File: app-routing.module.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';

const routes: Routes = [
    { path: '', redirectTo: 'home', pathMatch: 'full' },
    { path: 'home', component: HomeComponent },
];
// loadChildren: './app/Employee/employee.module#EmployeeModule'
export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

7) App Module

File: app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { EmployeeModule } from "./employee/employee.module"
import { DepartmentModule } from "./department/department.module"

import { HomeComponent } from "./home.component"
import { HeadComponent } from "./head.component"

import { routing } from './app.routing';

@NgModule({
    imports: [BrowserModule, routing, DepartmentModule, EmployeeModule],
    declarations: [AppComponent, HeadComponent, HomeComponent],
    bootstrap: [AppComponent],
})
```

```
export class AppModule { }
```

Lazy Loading a Module

Lazy loading is the concept of loading the required piece of code only when it's on demand, i.e. when **NgModule** launches the angular application it loads all the components and modules which are declared within, due to this application **start up time will be increased** and this show impact on performance of our application.

When a module is lazy loaded, Angular is going to create a child injector (which is a child of the root injector from the root module) and will create an instance of our services there.

- Application does **not need to load everything at start-up.**
- Lazy loading modules helps us **decrease the start-up time**.
- Modules that are **lazily loaded** will only be loaded **when the user navigates to their routes**.

So let's create lazy loading modules with a router. Here I've create few modules as follows.

We are having two modules one is **EagerComponent** and other is **LazyComponent.**

Our agenda

1. Eager Component should load in start up
2. Lazy Component should load only when we navigate to its module.

File: **eager.component.ts**

```
import { Component } from '@angular/core';
@Component({
   template: '<p>Eager Component</p>'
})
export class EagerComponent { }
```

File: **lazy.component.ts**

```
import { Component } from '@angular/core';
@Component({
  template: '<p>Lazy Component</p>'
})
export class LazyComponent {}
```

File: **lazy.routing.ts**

When registering routes in sub modules and lazy loaded submodules then we need to use **forChild(routes)** as follows.

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LazyComponent } from './lazy.component';
```

```
const routes: Routes = [
  { path: 'Lazy', component: LazyComponent }
];
export const routing: ModuleWithProviders = RouterModule.forChild(routes);
```

File: **lazy.module.ts**

```
import { NgModule } from '@angular/core';
import { LazyComponent } from './lazy.component';
import { routing } from './lazy.routing';
@NgModule({
  imports: [routing],
  declarations: [LazyComponent]
})
export class LazyModule { }
```

Now the root module with component

File: **app.component.ts**

Edit head.html

```
Add the following to the HTML Template file:
  <a class="nav-link" routerLink="lazy">Lazy Loading</a>
```

Edit File: **app.routing.ts**

When registering routes in the root module we need to use **forRoot(routes)** as follows.

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { EagerComponent } from './eager.component';
import { LazyComponent } from './lazy.component';

const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: './app/LazyLoadModules/lazy.module#LazyModule' }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);
```

File: **app.module.ts**

```
import { NgModule } from '@angular/core';
```

```
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { EagerComponent } from './eager.component';
import { routing } from './app.routing';

@NgModule({
   imports: [BrowserModule,routing],
   declarations: [AppComponent, EagerComponent],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

Output: click on lazy link it will route you to LazyComponent and render its template.

Eager vs Lazy Loading modules

Eager Lazy

Eager Component

Shared Module

Shared modules are the modules which are exposed to the other modules of your application which can make use of its services. Form the previous example let us add one more module **SharedModule** and **SharedService.**

File: **app.SharedService.ts**

```
import { Injectable } from '@angular/core'
@Injectable()
export class SharedService {
   counter = 0;
}
```

File: **app.SharedModule.ts**

```
import { NgModule } from '@angular/core'
import { SharedService } from './app.SharedService'
@NgModule({
   providers: [SharedService]
})
export class SharedModule {

}
```

File: **lazy.module.ts**

Now import SharedModule and add that import in it's NgModule import array

```
import { SharedModule } from './app.SharedModule';
imports: [routing, SharedModule]
```

File: **lazy.component.ts**

```
import { Component,OnInit } from '@angular/core';
import { SharedService } from './app.SharedService';
@Component({
   template: `<p>Lazy Component</p>
 <button (click)="counter()">Click me</button>
   <p>{{sharedService.counter}}</p>
`
})
export class LazyComponent{
   constructor(private sharedService: SharedService) {
   }
   counter() {
      this.sharedService.counter += 1;
   }
}
```

File: **eager.component.ts**

```
import { Component } from '@angular/core';
import { SharedService } from './app.SharedService';
@Component({
   template: `<p>Eager Component</p>
   <button (click)="counter()">Click me</button>
   <p>{{sharedService.counter}}</p>
   `
})
export class EagerComponent {
   constructor(private sharedService: SharedService) {
   }
   counter() {
      this.sharedService.counter += 1;
   }
}
```

File: **app.module.ts**

Import SharedModule and add that import into its NgModule import array

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { SharedModule } from './app.SharedModule';
import { AppComponent } from './app.component';
import { EagerComponent } from './eager.component';
import { routing } from './app.routing';

imports: [
    BrowserModule,
    SharedModule,
    routing
  ]
```

Output: Counter should maintain its count individually for **EagerComponent** and **LazyComponent**