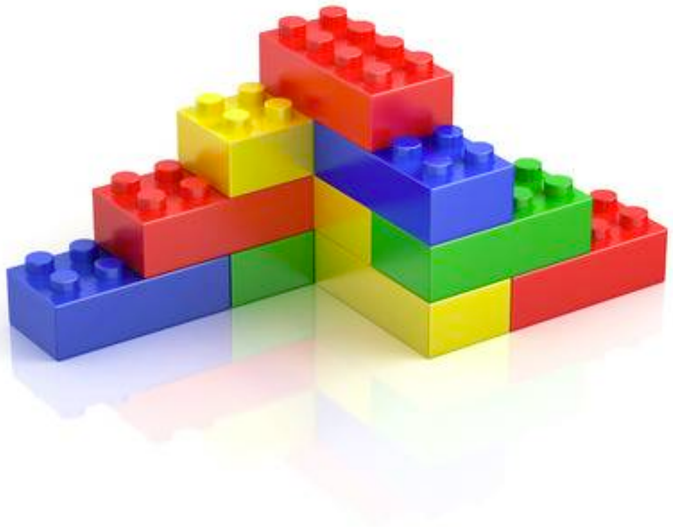
Angular

Modules



Sebastiaan Stuij - 28/07/2020

Introduction

What do we mean by modules in Angular?

- Modules are buckets for storing related entities for easy reuse and distribution
- 2 kind of modules commonly used in Angular projects: JavaScript modules & Angular modules

JS modules vs. Angular Modules

What?

JavaScript modules are language constructs used to separate code into different files (that can be loaded as needed)

Why?

Reusability / composability / isolation / organisation

History

Global scope -> IIFE -> CommonJS/UMD/AMD (+bundlers) -> ES(6/2015) modules

```
// utils.js

// Not exported
function once(fn, context) {
    …
}

// Exported
export function first (arr) {
  return arr[0]
}
```

```
// app.js


import * as utils from './utils'

utils.first([1,2,3]) // 1
```

JS modules vs. **Angular Modules**

What?

Angular modules are logical constructs used for organising similar groups of entities.

Why?

- Reusability / composability / isolation / organisation

- needed by Angular to understand what needs to be loaded and which dependencies exist

- NgModules configure the injector and the compiler and help organise related things together

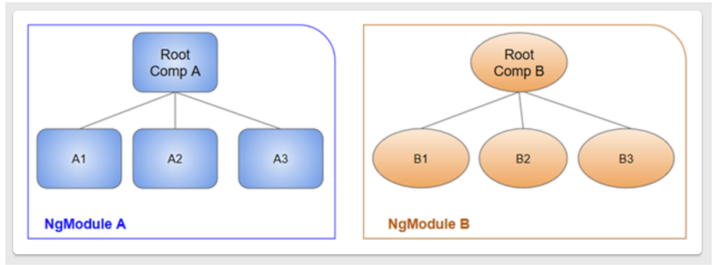
- Angular Modules are the unit of reusability

- Fun fact: Angular itself loads as a collection of **JavaScript modules**

```
// app.module.ts

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Angular modules

- Angular Modules logically group different Angular artifacts such as components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.
- Angular Modules in the form of the @NgModule decorator provide metadata to the Angular compiler which in turn can better “reason about our application” structure and thus introduce optimizations
- NgModules provide a compilation context for their components
- Important features such as lazy loading are done at the Angular Module level
- **Purpose of NgModule is to declare and organise everything you create in Angular (think Java packages or C# namespaces)**



Angular modules

Flavours

- Official Angular modules (@angular/core, @angular/common)
- Third party Angular modules (@angular/material, @ngx-translate/core)
- Your own/company modules (@company/company-design-components)

Angular modules

Module anatomy

ES modules ‘import’

NgModule Decorator

NgModule Metadata object

ES modules ‘export’

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

imports: other modules with the components, directives, and pipes that components in this module need

providers: provides services that the other application components can use.*

declarations: declare which components, directives, and pipes belong to this module

exports: makes some of those components, directives, and pipes public so that other module's component templates can use them

(bootstrap): needed to define the entry point (once)

*not necessary needed anymore since Angular 6 due to ‘providedIn’ property of a Service

Angular modules

Using NgModules

- An NgModule describes how the application parts fit together. Every application has at least one Angular module, the root module, which must be present for bootstrapping the application on launch. By convention and by default, this NgModule is named AppModule.

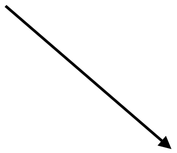
- **Main usage scenario 1)** if the module is imported for components, you'll need to import it in each module needing them

- **Main usage scenario 2)** if the module is imported for application wide/ singleton services, you'll need to import it only once (*or since angular 6 via providers themselves)

Entry point

App module

the *root* component that Angular creates and inserts into the index.html host web page



app.module.ts

```
/* JavaScript imports */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

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

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

main.ts

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

platformBrowserDynamic()
    .bootstrapModule(AppModule)
```

Frequently used NG modules

NgModule	Import it from	Why you use it
`BrowserModule`	`@angular/platform-browser`	When you want to run your app in a browser
`CommonModule`	`@angular/common`	When you want to use `NgIf`, `NgFor`
`FormsModule`	`@angular/forms`	When you want to build template driven forms (includes `NgModel`)
`ReactiveFormsModule`	`@angular/forms`	When you want to build reactive forms
`RouterModule`	`@angular/router`	When you want to use `RouterLink`, `.forRoot()`, and `.forChild()`
`HttpClientModule`	`@angular/common/http`	When you want to talk to a server

app.module.ts

```ts
/* import modules so that AppModule can access them */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [ /* add modules here so Angular knows to use them */
    BrowserModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Feature modules

5 categories

- Domain feature modules (hierarchy of feature components)
- Routed feature modules (lazy loaded modules by default)
- Routing modules (route setups/guards etc.)
- Service feature modules (API calls etc.)
- Widget feature modules (utility/pipes/directives)

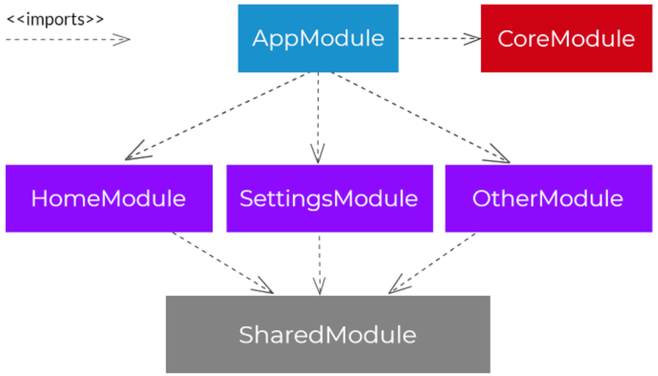
Feature Module	Declarations	Providers	Exports	Imported by
Domain	Yes	Rare	Top component	Feature, AppModule
Routed	Yes	Rare	No	None
Routing	No	Yes (Guards)	RouterModule	Feature (for routing)
Service	No	Yes	No	AppModule
Widget	Yes	Rare	Yes	Feature

Module architecture

- **Feature module** is a best practice (vs everything in the Root module) which represent a collection of related functionality. Think in terms of a specific user workflow, routing or forms

- Create a **SharedModule** for commonly used directives, pipes and components (e.g. button). But don’t declare services (due to lazy loading)

- Create a **CoreModule** for code that will be used across the app but only needs to be imported once (e.g. auth http interceptor, nav component). Providers isn’t needed anymore since Angular 6, but useful to apply a Core folder structure with Singleton services.



Angular modules
Advanced

- Scope & DI
- Lazy/eager/pre-loaded modules
- Entry components
- forRoot & forChild
- Tree shaking
- Best practices

Scope

Components / Directives

Private visibility →

(confusing: ~public) →

feature.module.ts

```
@NgModule({
  declarations: [
    ListComponent, DetailComponent, SpecialPipe, SpecialDirective
  ],
  imports: [
    CommonModule
  ],
  providers: []
})
export class FeatureModule { }
```

list.component.html

```
<ul>
    <li *ngFor="let item of list">
      <detail-component [details]=“item”> </detail-component>
    </li>
</ul>
```

other.component.html

```
<detail-component [details]=“item”> </detail-component>
```

Public visibility* →

*(when imported)

feature.module.ts

```
@NgModule({
  declarations: [
    ListComponent, DetailComponent, SpecialPipe, SpecialDirective
  ],
  imports: [
    CommonModule
  ],
  providers: [],
  exports: [
   ListComponent, DetailComponent, SpecialPipe, SpecialDirective
  ]
})
export class FeatureModule { }
```

list.component.html

```
<ul>
    <li *ngFor="let item of list">
      <detail-component [details]=“item”> </detail-component>
    </li>
</ul>
```

other.component.html

```
<detail-component [details]=“item”> </detail-component>
```

Scope

Services (DI)

Private visibility →

Public visibility when imported in every consumer module →

Confusing: ~public visibility (if eagerly loaded) →

feature.module.ts

```
@NgModule({
  declarations: [
    ListComponent, DetailComponent, SpecialPipe, SpecialDirective
  ],
  imports: [
    CommonModule
  ],
exports: [
   ListComponent, DetailComponent, SpecialPipe, SpecialDirective
  ]
  providers: [MyHttpProvider]
})
export class FeatureModule { }
```

Tree-shakeable providers in Angular 6

public visibility →

myhttp.provider.ts

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

@Injectable({
  providedIn: 'root'
})
export class MyHttpProvider {
  constructor() { }
}
```

Scope

Advice

- If the module is imported for **components/directives**, you'll need to import it in each module needing them
- If the module is imported for **services**, you'll need to import it only once, all providers are added to the root injector (besides lazy loaded modules)
- Advice mainly use Singleton services >= Angular 6: use '**providedIn: root**'

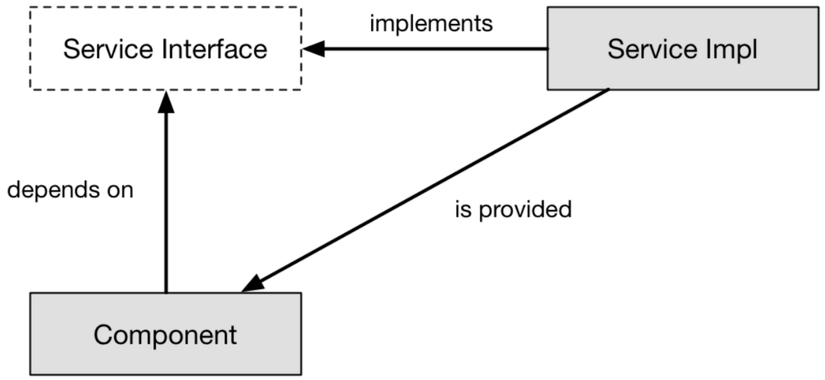
*Sidestep: Dependency Injection

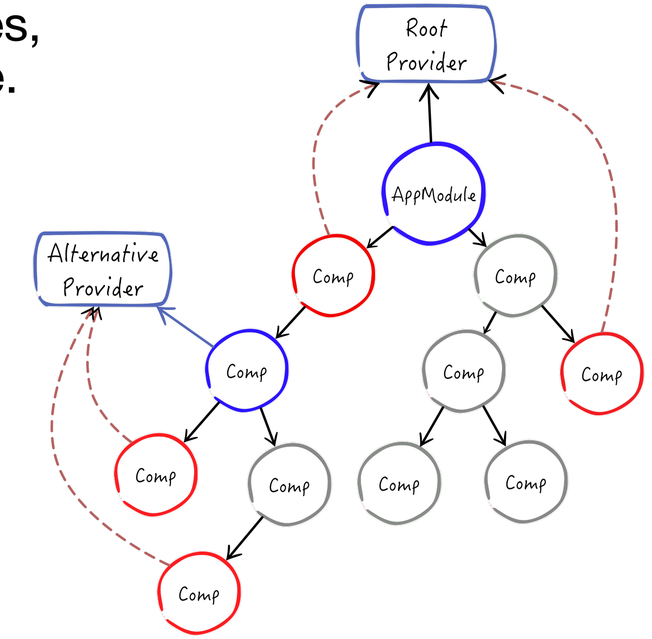
- Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.

- Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

- By doing so components depend on interfaces instead of concrete types, which leads to more decoupled code and greater testability for example.

- Angular consists of a **hierarchy of injectors**





1. What is an injector?

An injector is basically a key/value map. Here is a code example showing how an injector is created and used under the hood:

```
// Register
const injector = Injector.create([
  { provide: "color", useValue: "blue" }
]);

// Retrieve
injector.get("color"); // returns "blue"
```

2. Recommended way

This is what you need to know to get started with Angular DI. How to register a service and then how to inject it into a component. This is the most basic and most common use case.

Register

Creates a 🌴 tree-shakable, singleton service in the root injector.

```
@Injectable({
  providedIn: "root"
})
export class MyService {}
```

Inject into a Component

```
@Component({ ... })
export class MyComponent {
    constructor(private myService: MyService)
}
```

Resolution

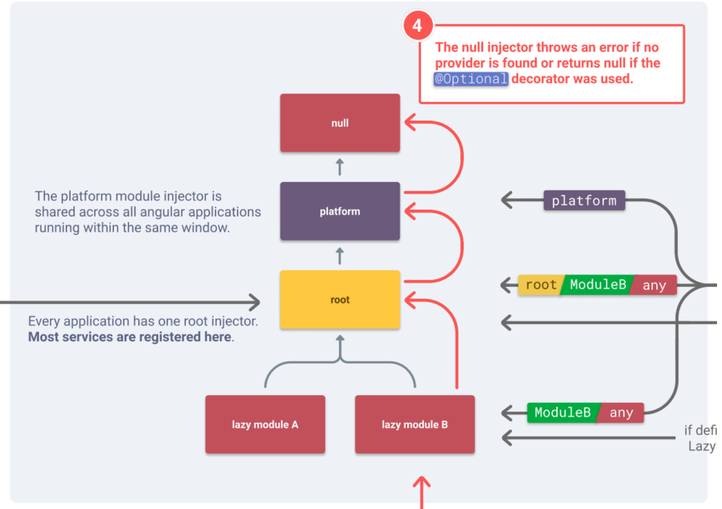
MyService class is used as a token. Like calling Injector.get(MyService) directly.

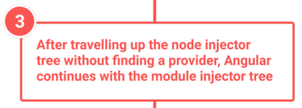
3. Hierarchical injectors (under the hood)

This is how the injector tree looks under the hood. It's a hierachical tree of injectors.

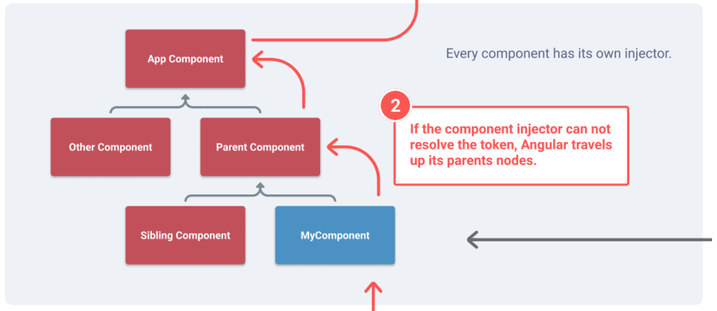
✅ Every injector has a parent (except for the null injector).
✅ Actually is not one but two trees. The node injector and the module injector tree.
✅ The resolution always starts with the current components node injector.
✅ Every component has its own node injector.
✅ In a common application, all services are registered in the root injector.

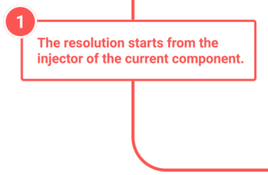
Module Injector Tree





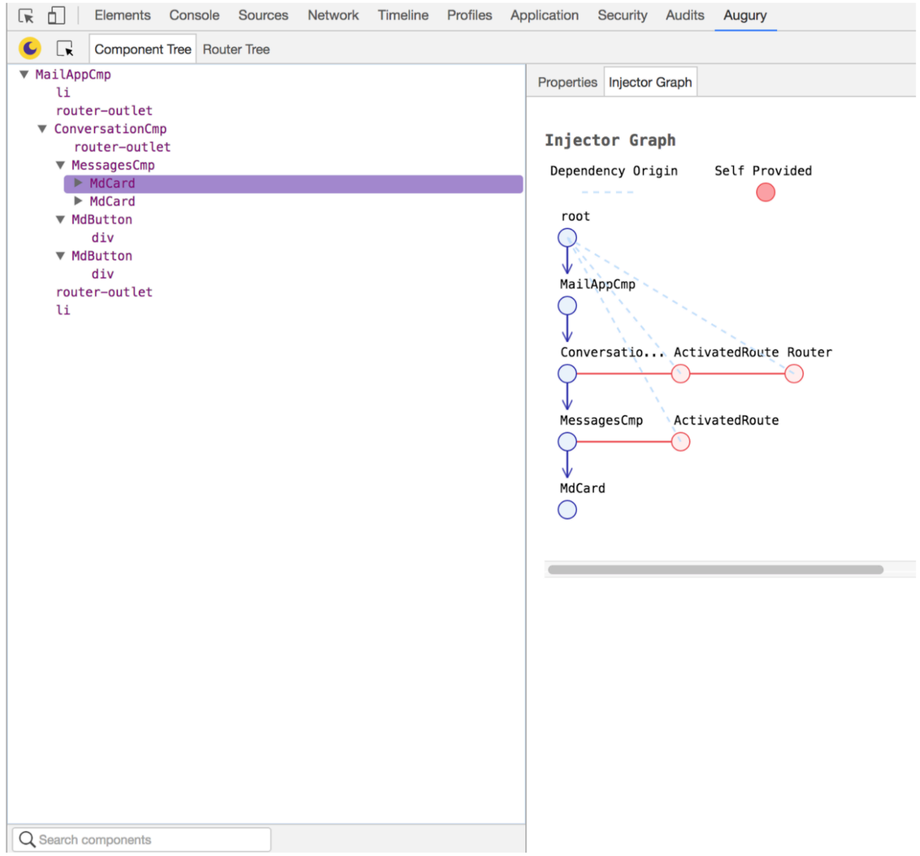
Node Injector Tree (Element Injector before Ivy)





https://dev.to/christiankohler/angular-dependency-injection-infographic-1bjm

*Angular DI tip: Augury



https://augury.rangle.io/

Eager/lazy/pre-loaded modules

- **Eager:** feature modules are loaded **before** the application starts. This is the default module-loading strategy => small applications or core modules (everything required to start the app)

- **Lazy:** feature modules are loaded **on demand after** the application starts. It helps to start application faster => larger apps and everything that is not initially required

- **Pre:** feature Modules are loaded **automatically after** the application starts => medium sized apps and for modules that are likely to be used after initialisation

Lazy-loaded modules

- Lazy loading is the process of loading features of your Angular app only when you navigate to their routes (for the first time)
- This can increase app performance and decreasing the size of the initial bundle
- Angular 8+: changed syntax (dynamic import syntax: https://javascript.info/modules-dynamic-imports)
- Angular provides the loadChildren property of a route's path to specify the module that needs to be lazy loaded when it's first navigated to
- After configuring the lazy-loaded module add separate routing for components of the lazy-loaded module (*forChild vs. forRoot)
- *Extra: lazy loaded modules vs lazy loaded services

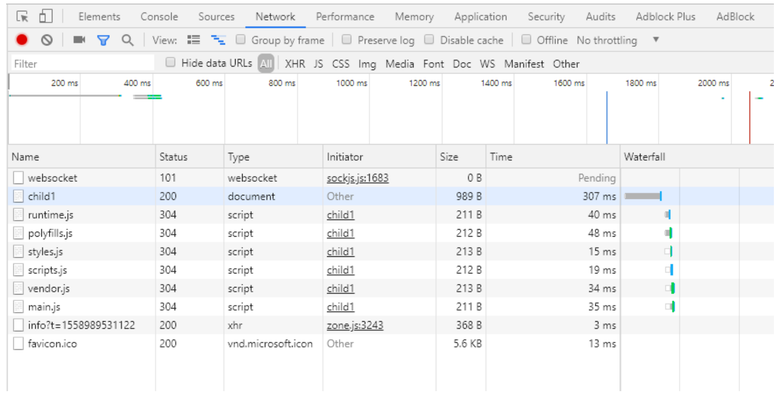
Eager/lazy/pre-loaded modules

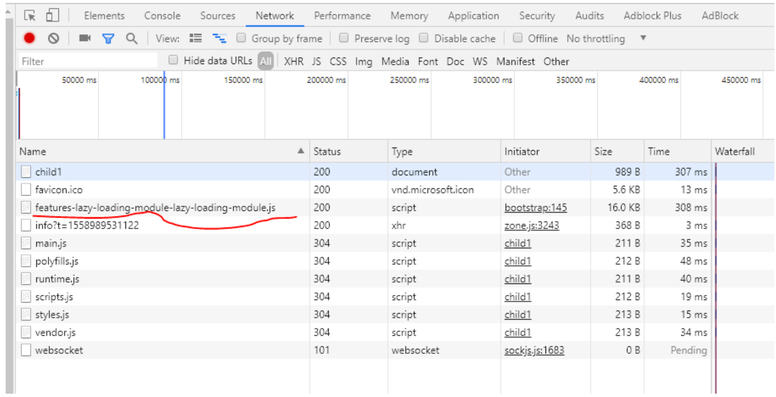
app-routing.module.ts

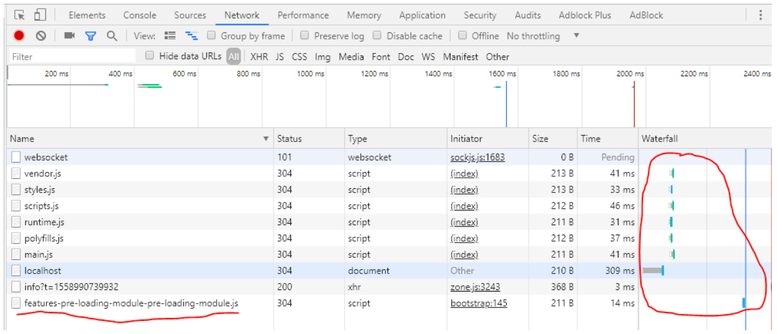
```
const routes: Routes = [
  { path: '', redirectTo: 'eager-loading', pathMatch: 'full' },
  {
    path: 'eager-loading', component: EagerHomeComponent, children: [
      { path: '', redirectTo: 'child1', pathMatch: 'full' },
      { path: 'child1', component: EagerChild1Component },
      { path: 'child2', component: EagerChild2Component },
      { path: '**', redirectTo: 'child1' }
    ]
  },
  {
    path: 'lazy-loading',
    loadChildren: './features/lazy-loading-module/lazy-loading.module#LazyLoadingModule'
    loadChildren:  () => import(`./features/lazy-loading-module/lazy-loading.module`).then(m => m.LazyLoadingModule)
  },
  {
    path: 'pre-loading',
    loadChildren: './features/pre-loading-module/pre-loading.module#PreLoadingModule',
    data: { applyPreload: true }
  },
  { path: '**', redirectTo: 'eager-loading' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes,
    { preloadingStrategy: CustomPreloadingStrategy }
  )],
  exports: [RouterModule]
})
```

Angular 7

Angular 8+







Entry components

- An entry component is any component that Angular loads imperatively by type (which means you're not referencing it in the template). You specify an entry component by bootstrapping it in an NgModule, via its entryComponents or by including it in a routing definition

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
  ],
  entryComponents: [
    ExampleDialogComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
const routes: Routes = [
  {
    path: '',
    component: CustomerListComponent
  }
];
```

forRoot & forChild

- It’s mainly a convention
- Mainly used for configuring provider scope (less relevant since Angular 6’s provideIn)
- Static methods to indicate where it should be imported:
  - forRoot (“injector”) in ‘AppModule’
  - forChild (“injector”) for lazy-loaded modules

```
static forRoot(config: UserServiceConfig):
ModuleWithProviders<GreetingModule> {
  return {
    ngModule: GreetingModule,
    providers: [
      {provide: UserServiceConfig, useValue: config }
    ]
  };
}
```

Treeshaking

- Tree shaking is a step in the build process to remove unused code
- This improves an app’s performance (and less boilerplate code)
- Does this by mainly looking at import statement (problem with Angular <6 way of providing services)
- Mainly beneficial for third-party libraries with a lot of exposed services

Common confusions

- Module encapsulation: declarations vs exports (components/directives: encapsulated) vs providers (services: not encapsulated)
- Module hierarchy: there is **none**, all modules are merged during compilation
- Lazy loading: this is an exception to the hierarchy (but it creates a hierarchy of injectors, **not** modules)
- forRoot/forChild: Use forRoot/forChild convention only for shared modules with providers that are going to be imported into both eager and lazy module modules.
- Multiple module imports: what if I import the same module twice (in lazy and eager module)? => no problem, all existing module loaders cache the module they load

Workshop

- Exercise 1: fix broken app
- Exercise 2: improve app structure
- Exercise 3: add a new module/component
- Exercise 4: optimize the app
- Exercise 5 (bonus): make it cooler

Links

Angular Modules:
https://tylermcginnis.com/javascript-modules-iifes-commonjs-esmodules/
https://dzone.com/articles/angular-modules-vs-es6-modules
https://angular.io/guide/ngmodules
https://medium.com/@cyrilletuzi/understanding-angular-modules-ngmodule-and-their-scopes-81e4ed6f7407

Dependency Injection:
https://dev.to/christiankohler/angular-dependency-injection-infographic-1bjm
https://medium.com/angular-in-depth/angular-dependency-injection-and-tree-shakeable-tokens-4588a8f70d5d
https://www.freecodecamp.org/news/angular-services-and-dependency-injection-explained/
https://medium.com/@tomastrajan/total-guide-to-angular-6-dependency-injection-providedin-vs-providers-85b7a347b59f
https://www.youtube.com/watch?v=jIMXSQYzhgs
https://itnext.io/understanding-provider-scope-in-angular-4c2589de5bc

Lazy loading:
https://medium.com/@lifei.8886196/eager-loading-lazy-loading-and-pre-loading-in-angular-2-what-when-and-how-798bd107090c
https://www.techiediaries.com/angular-lazy-load-module-example/
https://javascript.info/modules-dynamic-imports
https://pusher.com/tutorials/lazy-loading-angular-7
https://github.com/dscheerens/ngx-eager-provider-loader/blob/master/eager-loading-in-angular-2.md
https://github.com/dscheerens/ngx-inject

Module best practices:
https://angular.io/guide/architecture-modules
https://indepth.dev/avoiding-common-confusions-with-modules-in-angular/#482d
https://itnext.io/how-to-optimize-angular-applications-99bfab0f0b7c
https://medium.com/@cyrilletuzi/architecture-in-angular-projects-242606567e40
https://nitayneeman.com/posts/restructuring-an-angular-application-with-feature-modules/
https://levelup.gitconnected.com/where-shall-i-put-that-core-vs-shared-module-in-angular-5fdad16fcecc

Misc.
https://stackoverflow.com/questions/48942691/how-angular-builds-and-runs

Questions or feedback?