Angular Modules in Depth

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

NgModule

The purpose of a **NgModule** is to declare objects that belong together within the scope of Angular Module, like components and services.

When your application grows, you won't just have one module, but many of them.

- Main **AppModule** that your app bootstrap
- Featured Modules

`NgModule` and scopes/visibility

Components and Services do not have the same scope/visibility:

Components are in local scope (**private visibility**),

Services are in global scope (**public visibility**).

By default, when a module imports another module, it cannot access its components but it has access to all its services.

Scope Example

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

Root Injector:
AppService,
FeaturedServiceOne,
FeaturedServiceTwo

```
@NgModule({
   declarations: [ FeaturedComponentOne ],
   imports: [ CommonModule ],
   providers: [ FeaturedServiceOne ]
})
export class FeatureModuleOne {}
```

```
@NgModule({
   declarations: [ FeaturedComponentTwo ],
   imports: [ CommonModule ],
   exports: [ FeaturedComponentTwo ]
   providers: [ FeaturedServiceTwo ]
})
export class FeatureModuleTwo {}
```

`NgModule` and scopes/visibility

When you explicitly **`exports`** components from ModuleA, then when you import ModuleA from AppModule, these components are usable in the AppModule. If you need to use them in Module B, you'll have to import ModuleA from ModuleB.

When AppModule imports ModuleA, all ModuleA services end up in the root injector of AppModule, these services are now usable in the whole application. ModuleB does not need to import ModuleA to use it's services.

When to Import NgModule?

We need to know why we import these other modules: Is it to use components or is it to use services?

Given the difference of scope between components and services:

If the module is imported for **components**, 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**, in the first app module.

Angular itself is subdivided in different modules (core, common, router, form, http..etc).

Import Angular Modules

Service modules	Component modules	Hybrid modules
Services	Components	Services & Components
Import once	Import every time module when needed.	Import the main module once for services. Import the components every module when needed.

`HttpClientModule` (Any other module providing services only)	`FormsModule`, `ReactiveFormsModule`, Angular Material Modules (Any other module exporting components, directives or pipes)	`BrowserModule CommonModule`
		
		`RouterModule.forRoot()` `RouterModule.forChild()`

Mixed NgModules

The **RouterModule** gives you a component **<router-outlet>** and a directive **routerLink**, but also services **ActivatedRoute** and **Router**.

The first time in app module, **forRoot()** will give the router components and provide the router services. But the next times in submodules, **forChild()** will only give the router components (and not providing again the services, which would be bad).

```
RouterModule.forRoot(routes) // For the AppModule
RouterModule.forChild(routes) // For submodules
```

BrowserModule has combination of components, directives and services, to use it again in a feature module use: **CommonModule** instead.

The Need for Lazy Loading

Angular is built with the focus on mobile. That's why they put a lot of effort into making small compiled and bundled Angular applications.

One of the techniques they use is **tree shaking**, which helped drop the size of an application to only 5K.

At some point, however, our application will be big enough, that even with this technique, the application file will be too large to be loaded at once.

What is Lazy Loading?

Even if we separate the logic of our code into multiple modules, **modules by default will be eagerly-loaded** and will be part of one bundle. That's where lazy-loading comes into play.

Lazy loading speeds up the application load time by splitting it into multiple bundles, and loading them on demand, as the user navigates throughout the app. As a result, the initial bundle is much smaller, which improves the bootstrap time.

Lazy-Loaded Modules

```
const routes: Routes = [
   { path: 'admin',
     loadChildren: () => import('./lazy/lazy.module').then(m => m.LazyModule) }
];
```

Remember, you need to import the `CommonModule` instead of `BrowserModule` and all other modules of components. But for services, you'll still have access to services already provided in the app (like `HttpClient` and your own services).
However, services provided in your lazy-loaded module will only be available in this lazy-loaded module, not everywhere in your app. Lazy-Modules have their own local-injector.

Example – Eager Loading

```
@NgModule({
  declarations: [AppComponent, HomeComponent],
  imports: [ BrowserModule,
        ModuleTwo,
        RouterModule.forRoot([
          { path: '', component: HomeComponent },
          { path: 'two', component: ComponentTwo } ]),
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

importing **ModuleTwo** will bundle all its components with the same main bundle.

```
@NgModule({
  declarations: [ ComponentTwo ],
  imports: [ CommonModule ],
  providers: [],
  bootstrap: [ ComponentTwo ]
})
export class ModuleTwo {}
```

Example – Lazy Loading

```
@NgModule({
  declarations: [ AppComponent,  HomeComponent ],
  imports: [ BrowserModule,
   RouterModule.forRoot([
    { path: '', component: HomeComponent },
    { path: 'two', loadChildren:
          () => import('./lazy/lazy.module').then(m => m.LazyModule)}
    ]) ],
  providers: [],
  bootstrap: [AppComponent]
})
```

We don't import **ModuleTwo** in **imports[]**

```
@NgModule({
  declarations: [ ComponentTwo ],
  imports: [ CommonModule,
     RouterModule.forChild({ path: '', component: ComponentTwo })
  ],
  providers: [],
  bootstrap: [ ComponentTwo ]
})
export class ModuleTwo { }
```

Preloading Modules

Lazy loading speeds up our application load time by splitting it into multiple bundles, and loading them on demand.

The issue with lazy loading is that when the user navigates to the lazy-loadable section of the application, the router will have to fetch the required modules from the server, which can take time.

To fix this problem we can activate preloading: Now the router can preload lazy-loadable modules in the background while the user is interacting with our application.

How Preloading Works?

1. First, we load the initial bundle, which contains only the components we have to have to bootstrap our application. So it is as fast as it can be.
2. Then, Angular bootstraps the application using this small bundle.
3. At this point the application is running, so the user can start interacting with it. In the background, Angular preload other modules.
4. Finally, when the user clicks on a link going to a lazy-loadable module, the navigation is instant.

Enabling Preloading

The Angular router comes with two strategies: **preload nothing** or **preload all** modules, also you can provide you own strategy.

```
@NgModule({
    bootstrap: [AppCmp],
    imports: [
        RouterModule.forRoot(ROUTES, {preloadingStrategy: PreloadAllModules})
    ]
})

class AppModule {}
```

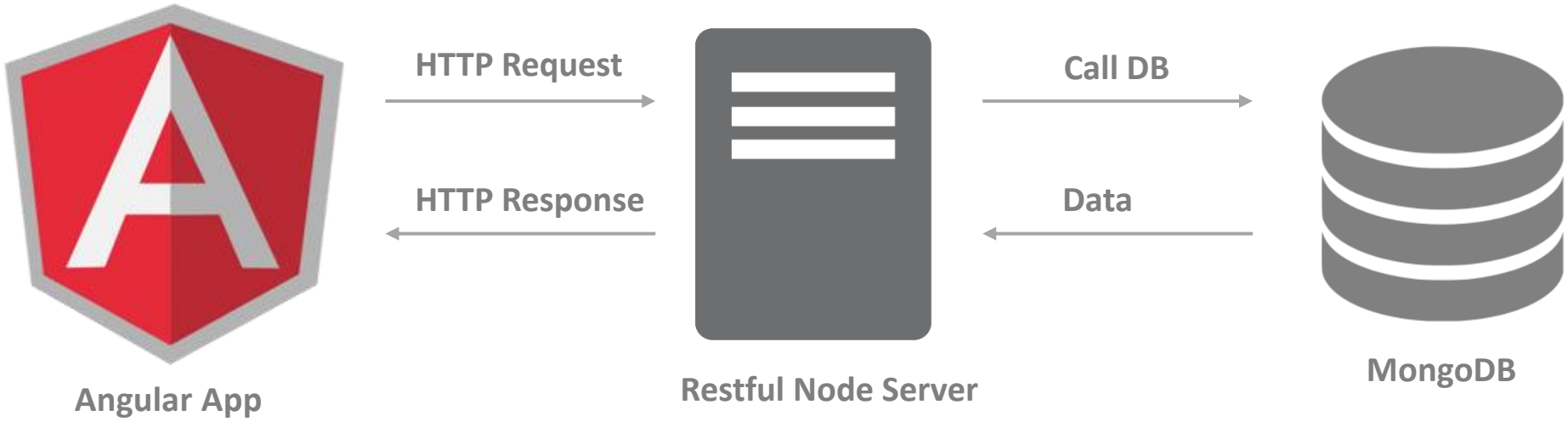
Provider Scope in Lazy Modules

If we have Lazy Module that doesn’t provide a service it will use the instance created from root injector (AppModule)

But if Lazy Module does have provided a service, components of that module will use local instance of that service (not the instance from root injector)

Making HTTP requests from Angular

The Angular **HTTP Client Module** simplifies application programming with the XHR and JSONP APIs. **All async requests return an Observable**.



Using HTTP Service

HTTP has been split into a separate module in Angular. This means that to use it you need to import it from `@angular/common/http` .

```
@NgModule({
        imports: [ BrowserModule, HttpClientModule ],
})
```

Now we can ask for **`HttpClient`** service instance in the constructor:

```
@Component({ })
class MyComponent {
        constructor(public http: HttpClient) {  }
}
```

A Basic Request

```
import {HttpClient, Response} from '@angular/common/http';
@Component({
        template: ` <button type="button" (click)="makeRequest()">Request</button>
                        <div *ngIf="loading">loading...</div>
                        <pre>{{data}}</pre> `
})
export class SimpleHTTPComponent {
        data: Object;
        loading: boolean;
        constructor(public http: HttpClient) { }
        makeRequest(): void {
                this.loading = true;
                this.http.get('http://URL')
                   .subscribe( res => { this.data=res; this.loading=false; });
        }
}
```

This is a bad design. Why?

Separation of Concerns

```
@Injectable()
class MyCustomHttpService {
	constructor(public http: HttpClient) { }
	getMyData() {
		return this.http.get('example.com/resource?myId=123');
	}
}
```

In your component, you may subscribe to that returned Observable and use the response:

```
this.myCustomHttpService.getMyData().subscribe(
	response => console.log(response),
	error => console.error(error),
	completed => console.log('Operation completed!')
);
```

POST Requests

Post requests work like GET requests. The only difference is, that they also require a payload body to be sent with the request.

```
post(url: string, body: any,
       options?: RequestOptionsArgs): Observable<Response>;
```

```
this.http.post('http://link/', { title: 'foo', body: 'bar', userId: 1 })
       .subscribe( res => { console.log(res); },
                   err => { console.log('Error occurred'); } );
```

Optional Http Arguments

You may set up HTTP calls with additional, optional argument. For example, you may want to add certain **`Headers`** to your Http call.

Therefore, the Http service methods take an optional argument in the form of a JS object. This object allows you to set up additional configuration.

```
myHeaders = new Headers();
myHeaders.append('Content-Type', 'application/json');

this.http.get('example.com/resource?myId=123', { headers: myHeaders });
```

HTTP Interceptors

An interceptor sits between the application and a backend API. With interceptors we can manipulate a request coming out from our application before it is actually submitted and sent to the backend. A response arriving from the backend can be altered before it is submitted to our application.

You may use the CLI to generate a new interceptor.

Interceptor Example

```
// src/app/app.interceptor.ts
import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class MyInterceptor implements HttpInterceptor {
        intercept (req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
                const authReq = req.clone(
                        { headers: req.headers.set('Accept-Language', 'Test') });
                return next.handle(authReq).pipe(
                   map(resp => {
                       return resp.clone({ body: [{title: 'CS572'}] });
                   })
                );
        }
}
```

Manipulate Request

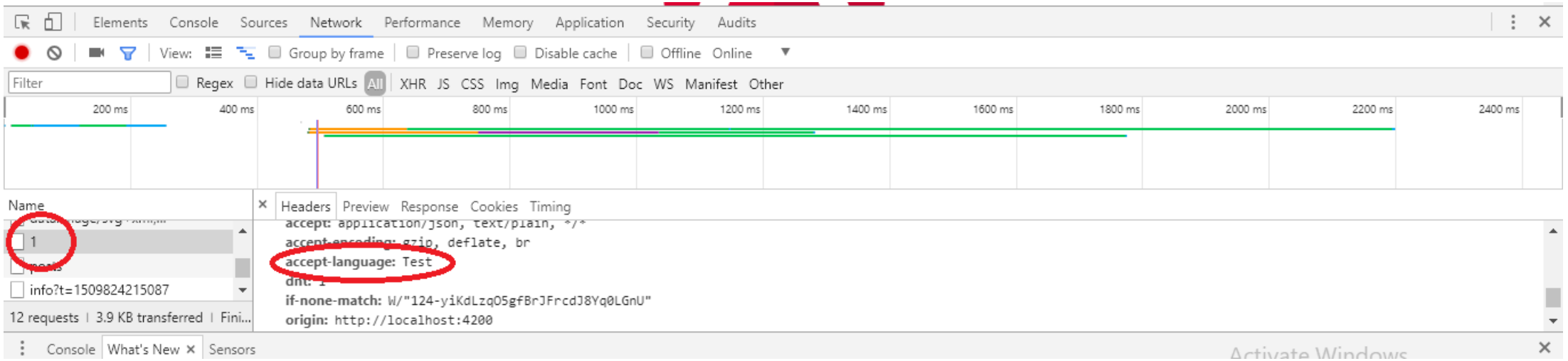
Manipulate Response

Interceptor Provider

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { MyInterceptor } from './app.interceptor';

providers: [{ provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true }]
```

multi: needs to be set to **true** to tell Angular that HTTP_INTERCEPTORS is an array of values, rather than a single value



Manage Subscriptions

```
@Component({ /* ... */
   template: ` <ul *ngIf="books.length > 0">
                  <li *ngFor="let book of books">{{book.name}}</li>
               </ul> ` })
export class BooksComponent implements OnInit, OnDestroy {
   private subscription: Subscription;
   books: Book[];
   constructor(private service: Service) {}
   ngOnInit() {
      this.subscription = this.service.getBooks()
                              .subscribe(books => this.books = books);
   }
   ngOnDestroy(): void {
      this.subscription.unsubscribe();
   }
}
```

Subscriptions with the Async Pipe

```
@Component({ /* ... */
   template: ` <ul *ngIf="(books$ | async).length">
                  <li *ngFor="let book of books$ | async">{{book.name}}</li>
               </ul> ` })
export class TodosComponent implements OnInit {
   books$: Observable<Book[]>;
   constructor(private service: Service) { }
   ngOnInit() {
      this.books$ = this.service.getBooks()
   }
}
```

Advantages of using subscribe()

Unwrapped property can be used in multiple places in the template without the need to rely on various subscriptions.

Unwrapped property is available everywhere in the component. This means it can be used directly in the component's method without the need to pass it from the template. That way, all state can be kept in the component.

Disadvantages of using subscribe()

Using of the **`subscribe()`** introduces complementary need to unsubscribe at the end of the component life-cycle to avoid memory leaks. Developers usually have to unsubscribe manually.
The most RxJS declarative way to do this is to use **`takeUntil(unsubscribe$)`** or **`take(1)`** operator. This solution is very easy and will not lead to any errors just a silent memory leak.
Subscribing to the observable manually in **`ngOnInit()`** doesn't work with **`OnPush`** change detection strategy out of the box.

Advantages of using | async pipe

This solution works with **`OnPush`** change detection out of the box! Just make sure that all your business logic is immutable and always returns new objects.

Angular handles subscriptions of **`| async`** pipes for us automatically so there is no need to unsubscribe manually in the component using **`ngOnDestroy`**. This leads to less verbosity and hence less possibilities for making a mistake.

Disadvantages of using | async pipe

Objects have to be wrapped in the template using "***ngIf="something$ | async**" syntax.

On the other hand, this is not a problem with collections when using ***ngFor="let something of somethings$ | async"**

Properties wrapped in the template using ***ngIf** or ***ngFor** are not accessible in the component's methods. This means we have to pass these properties to the methods from the template as the method parameters which further splits logic between the template and the component itself.

Async Pipe with ngFor

```
@Component({
   selector: 'app-root',
   template: ` <ul>
                  <li *ngFor="let city of cities$ | async">
                      Name: {{ city.name }},
                      Population: {{ city.population }},
                      Elevation: {{ city.elevation }}
                  </li>
               </ul> ` })
export class AppComponent {
   cities$ = of([
         { name: 'Los Angeles', population: '3.9 million', elevation: '233’' },
         { name: 'New York', population: '8,4 million', elevation: '33’' },
         { name: 'Chicago', population: '2.7 million', elevation: '594’' }, ])
  .pipe(delay(1000));
}
```