CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

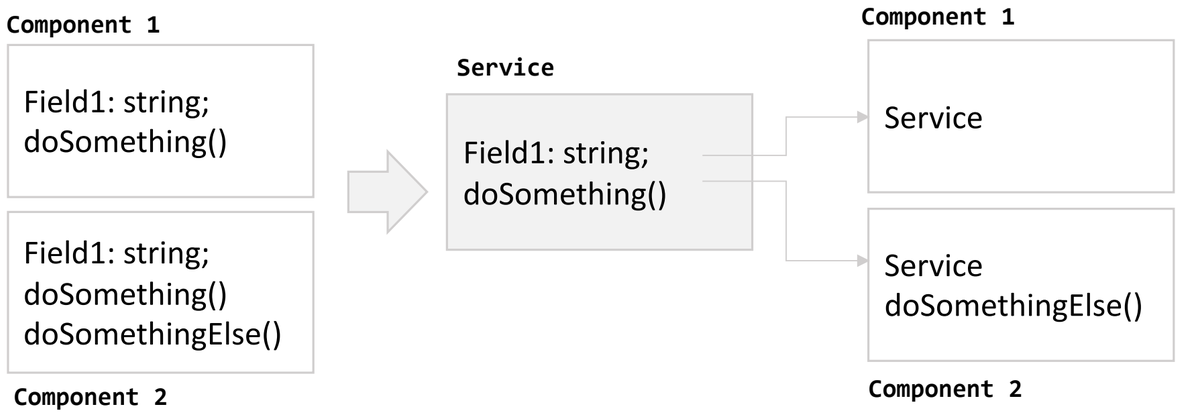
PREV NEXT

OUTLINE

Search...

Services

- Common layer to store and interact with Data (Database on server)
- Communication channel between components
- Services provide access to certain functionality from various places in your application.



Services can be used to centralize certain tasks, remove redundant code duplication or outsource heavy tasks.

Dependency Injection

The idea behind dependency injection is very simple. If you have a component that depends on a service. You do not create that service yourself.

Instead, you request one in the constructor, and the framework will create an instance and provide it to you.

This leads to more **decoupled code** (separation of concerns), which enables testability, easy code refactoring and modularity.

PREV NEXT

OUTLINE

Search...

Example

Let’s imagine we have a `Product` class. Each product has a base price. In order to calculate the full price for this product, we rely on a service that takes the base price of the product when it's been initialized and calculate the price based the state we’re selling it to.

```
class Product {
        constructor(basePrice: number) {
                this.service = new PriceService();
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

A Better Way

```
class Product {
        constructor(service: PriceService, basePrice: number) {
                this.service = service;
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

This technique of injecting the dependencies relies on a principle called the **Inversion of Control (IoC)** principle is also called informally the Hollywood principle (don’t call us, we’ll call you).

When we use DI we are moving towards a more **loosely coupled** architecture where changing bits and pieces of a single component affects the other areas of the application less. And, as long as the interface between those components don’t change, we can even swap them altogether, without any other components even realizing.

6

PREV NEXT

OUTLINE

Search...

Registering Providers

Angular dependency injection framework takes care of creating the required instance, but it needs to know how to create such an instance.

There are two ways to provide services

Global	Locally (component level)

`// in the Service`
`@Injectable({ providedIn: 'root' })`
	`// In the Component:`
	`providers = [ MyService ]`
	`OR`
	`providers = [`
	`  {provide: MyService, useClass: MyService}`
	`]`

All Angular native services are provided at root, you may ask for them in your constructor, Angular knows where to find them and provide an instance for you.

PREV NEXT

Create Services from CLI

- Injecting a singleton instance of a class is probably the most common type of injection.
- When you provide the service at the root level, Angular creates a single, shared instance of service and injects into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.
- To create a new service from Angular CLI:

```
ng g s serviceName
```


PREV NEXT

Global Service

`@Injectable()` tells Angular that if this Service asked for anything in the `constructor()` then Angular will be able to inject it.

```
@Injectable({ providedIn: 'root' })
export class LogService {
        constructor() {}
        logText(input: string) {
                console.log(input);
        }
}
```

This tells Angular to provide the Service at the Root level and the whole application can use the same instance (singleton)

```
@Component({ selector: 'comp', template: ` `, })
export class MyComponent {
        constructor (private logService: LogService) {}
        onLog(value: string) {
                this.logService.logText(value);
        }
}
```

10

PREV NEXT

Local Service

This service is not provided at root. It should be provided at the Component level

```
@Injectable()
export class LogService {
        constructor() {}
        logText(input: string) {
                console.log(input);
        }
}
```

```
@Component({ selector: 'comp', template: ` `, providers: [LogService]})
export class MyComponent {
        constructor (private logService: LogService) {}
        onLog(value: string) {
                this.logService.logText(value);
        }
}
```

11

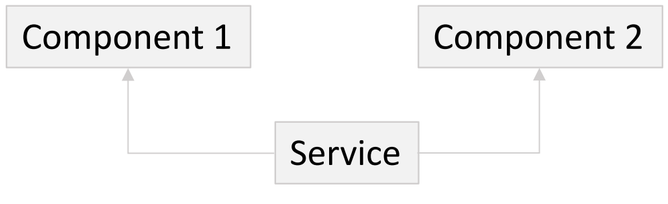
Hierarchical Injector

Angular Dependency Injector is **hierarchical**. This basically means, that providers may be set up on different levels of the application, leading to different outcomes. **Normally you get an instance per provider.**

- The most common pattern is that you provide all your services at root so your application can share the **same instance**.
- If providers are specified on each individual component, **different instances** will be created.

Cross-Component Communication

We can implement **Push-Notifications** like feature using Services:




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

@Injectable({ providedIn: 'root' })
export class MyService {
        // A channel so component1 and component2 can exchange data
        emitter = new EventEmitter<string>();
        emitValue(value: string) {
                this.emitter.emit(value);
        }
}
```

18

Cross-Component Communication

emit()

```
export class Comp1Component {
        constructor (private myService: MyService) {}
        onSend(value: string) {
                this.myService.emitValue(value);
        }
}
```

subscribe()

```
export class Comp2Component implements OnInit {
        private value = '';
        constructor (private myService: MyService) {}
        ngOnInit() {
             this.myService.emitter.subscribe(data => this.value = data);
        }
}
```

Main Points

- Dependency injection is a key component of Angular.
- You can configure dependency injection at the component or root level.
- Dependency injection allows us to depend on interfaces rather than concrete types. This results in more decoupled code and improves testability.

PREV NEXT

OUTLINE

Search...

Routing

Routing means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

Defining routes in our application is useful because we can:

- Separate different areas of the app
- Maintain the state in the app
- Protect areas of the app based on certain rules

Client-Side vs Server-Side Routing

The server must have one route "/" to serve your SPA application. All other routes should be for API to persist/retrieve data. Additional "*" unknown route is required to redirect user to Angular app.

Client-side routing will have difference routes to display various component trees to the user. Angular changes the URL with pushState API to support (refresh, bookmark and shareability to your routes).

With client-side routing we're not necessarily making a request to the server on every URL change.

Components of Angular routing

There are three main components that we use to configure routing in Angular:

- **Routes** describes the routes our application supports. Will be passed to `RouterModule` and imported in our `NgModule`.
- **RouterOutlet** is a placeholder component that gets expanded to each route's content.
- **RouterLink** directive is used to link to routes so our browser won't refresh when we change routes.

RouterModule

```
import { RouterModule, Routes } from "@angular/router";

const MY_ROUTES: Routes = [
        { path: '', redirectTo: 'home', pathMatch: 'full' },
        { path: 'home', component: HomeComponent },
        { path: 'about', component: AboutusComponent },
        { path: 'aboutus', redirectTo : 'about' },
        { path: '**', redirectTo : 'home' }
];
@NgModule({ . . .
        imports: [ BrowserModule, RouterModule.forRoot(MY_ROUTES) ]
                . . . })
export class AppModule { }
```

RouterLink [routerLink]

If we tried creating links that refer to the routes directly using pure HTML, it will result to links when clicked they trigger a **page reload**

```
<a href="/home">Home</a>
```

To solve this problem, we will use the `RouterLink` directive:

```
<a [routerLink]="['home']">Home</a>
```

RouterOutlet `<router-outlet>`

When we change routes, we want to keep our outer layout template and only substitute the inner section of the page with the route's component.

In order to describe to Angular **where** in our page we want to render the contents for each route, we use the **`router-outlet`** directive.

We are going to use our `AppComponent` as a layout which contains all our `RouterLink` and `RouterOutlet` directives.

Example of Layout Component

```
@Component({
        selector: 'AppComponent',
        template: ` <div>
                <nav>
                   <ul>
                       <li><a [routerLink]="['home']">Home</a></li>
                       <li><a [routerLink]="['about']">About</a></li>
                       <li><a [routerLink]="['contact']">Contact us</a></li>
                   </ul>
                </nav>
                <router-outlet></router-outlet>
                </div> `
})
class RoutesDemoApp {}
```

Using [routerLink] will instruct Angular to take ownership of the click event and then initiate a route switch to the right place, based on the route definition.

PREV NEXT

Base Tag <base href="/">

This tag is traditionally used to tell the browser where to look for images and other resources declared using **relative paths**.

Angular Router also relies on this tag to determine how to construct its routing information.

If we have a route with a path of `/hello` and our base element declares `<base href="/app">`, the application will use `/app/hello` as the concrete path.

Routes and Component Lifecycle

Angular Router is pretty efficient when it comes to components lifecycle. It only loads the component once it's been asked for, once created, it does not destroy it and recreate a new one the next time you ask for the same component, instead, **it will reuse the same instance** that you already have in memory (**snapshot**)

Route Parameters – Mandatory Params

We can specify that a route takes a parameter by putting a colon in front of the path segment like this `/route/:param`

```
const routes: Routes = [{ path: 'articles/:id', component: ArticlesComponent } ];
```

In order to read route parameters, we need to first import `ActivatedRoute` Service and we inject it into the constructor of our component:

```
constructor(private route: ActivatedRoute) {
        route.params.subscribe( params => { this.id = params['id']; });
}
```

`route.params` is an observable

PREV NEXT

Query Parameters – Optional Query Params

```
constructor(private route: ActivatedRoute) {
        route.queryParams.subscribe( params => { this.id = params['id']; });
}
```

route.queryParams is an observable

Observable Subscription

You **must subscribe** to an observable in order to listen to its events.

When subscribing to the observable we generate a Subscription, after we finish from the component, this subscription will stay alive and will cause **memory leak**. You should always destroy your observable subscription.

You have to `unsubscribe()` from the Observable in the `ngOnDestroy` method/lifecycle hook

Example

```
import { Component, OnDestroy } from '@angular/core';
import { ActivatedRoute } from "@angular/router";
import { Subscription } from "rxjs";

@Component({ selector: 'component', template: `...` })
export class Component implements OnDestroy{
        private subscription: Subscription;
        id: string;
        constructor(private route: ActivatedRoute) {
                this.subscription = route.params.subscribe(
                                (param: any) => this.id = param['id'];
                );
        }
        ngOnDestroy() {
                this.subscription.unsubscribe();
        }
}
```

Imperative Routing

You can also navigate to a route imperatively (in your code), you need to inject the **`Router`** service then you may call **`navigate()`** like this:

```
import { Router } from '@angular/router';

constructor(private router: Router) {}

this.router.navigate(['home'])
```

Remember: Angular Services don't need to be provided. Angular knows how to create an instance of its services.

Router Link Best Practices

Template

```
<a [routerLink]="['users', 'update', id.value]">Update</a>
// users/update/1
<a [routerLink]="['users', 'update']" [queryParams]="{id: id.value}">Update</a>
// users/update?id=1
```

Imperative Routes

```
this.router.navigate(['users', 'update'], { queryParams: { id: id.value} })
// users/update?id=1
this.router.navigate(['home'], fragment: 'section1'})
// home#section1
```

Nested Routes

Nested routes is the concept of containing routes within other routes. With nested routes we're able to encapsulate the functionality of parent routes and have that functionality apply to the child routes.

We can have multiple, nested `router-outlet`. So each area of our application can have their own child components, that also have their own `router-outlet`.

```
const MY_ROUTES: Routes = [
       { path: 'parent', component: ParentComponent,
         children: [
              { path: 'child', component: ChildComponent }
          ]
       }];
```

Must have an outlet, so its children are rendered in it

Router Hooks

There are times that we may want to do some actions when changing routes (for example authentication).

Let’s say we have a login route and a protected route. We want to only allow the app to go to the protected route if the correct username and password were provided on the login page. In order to do that, we need to **hook into the lifecycle of the router** and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

Guards

Guards are useful Services which allow you to control access to and from a Route/Component.

`canActivate` called when you are serving into the route

`canDeactivate` called when leaving the route.

Guard classes have to be registered in **`providers[]`**, why?

```
const ROUTES: Routes = [
        { path: 'my-path', component: MyComponent,
          canActivate: [MyGuard],
          canDeactivate: [MyOtherGuard, AnotherGuard] }
];
```

Example CanActivate

```
import { CanActivate,
         RouterStateSnapshot,
         ActivatedRouteSnapshot } from "@angular/router";
import { Observable } from "rxjs";

export class MyGuard implements CanActivate {
        canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Observable<boolean> | boolean {

                // Your logic goes here
                // return true to continue
                // otherwise, you will have to redirect to another route
                return confirm('Are you sure?');
        }
}
```

NgModule

The purpose of a **NgModule** is to declare each object you have in Angular Module, like components and services which belong together.

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

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**).

When AppModule imports ModuleA, 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.

PREV NEXT

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.

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', children: { path:'', 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 { }
```


PREV NEXT

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)

PREV NEXT

Example

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

Root Injector: AppService

```
@NgModule({
   declarations: [ FeaturedComponent ],
   imports: [ CommonModule ]
})
export class LZ {}
```

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

Example

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

Root Injector: AppService

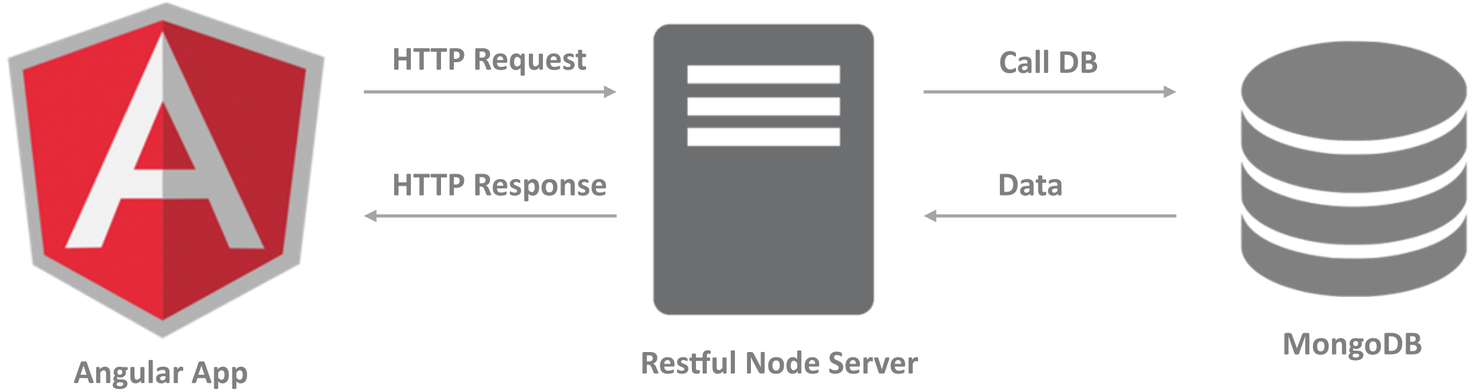
```
@NgModule({
    declarations: [ FeaturedComponent ],
    providers: [ AppService ],
    imports: [ CommonModule ]
})
export class LZ {}
```

Local Injector: AppService

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 into our components

```
@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!')
);
```

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 });
```

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'); } );
```

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.

PREV NEXT

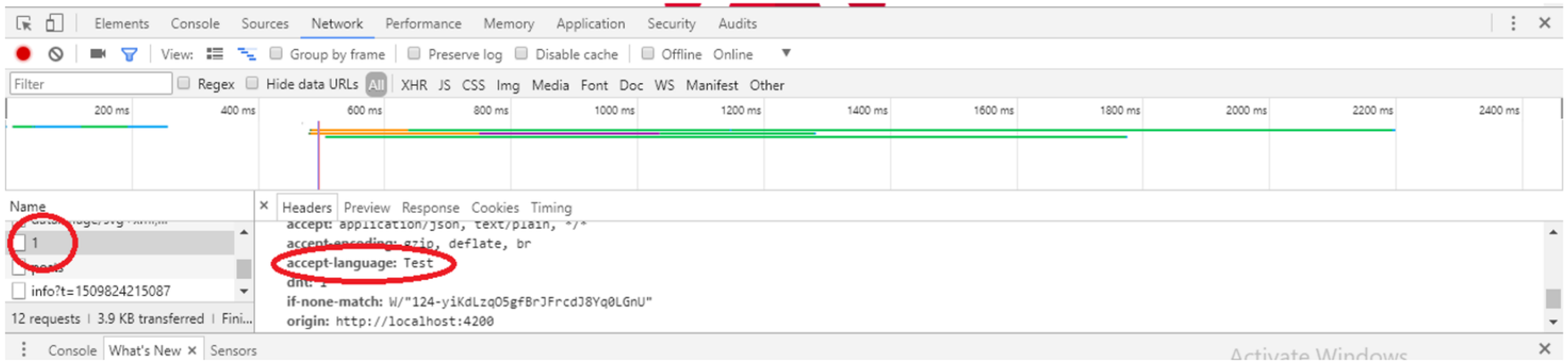
Interceptor Example

```
// 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'}] });
                   })
                );
        }
    }
```

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()
   }
}
```

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$ = Observable.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));
}
```

Async Pipe with ngIf

```
@Component({
   selector: 'app-root',
   template: ` <span *ngIf="(course$ | async)?.length > 0; else othercourse">
                    5xx level: {{ course$.value }}
                </span>
                <ng-template #othercourse>
                    4xx level: {{ course$.value }}
               </ng-template> ` })
export class AppComponent {
   course$ = Observable.of('CS572 Modern Web Applications');
}
```

Async Pipe with ngIf and ngFor

```
@Component({
    selector: 'app-root',
    template: ` <div *ngIf="(users$ | async)?.results.length; else loading">
                    <div *ngFor="let user of (users$ | async)?.results">
                        {{user.name.first}} {{user.name.last}}
                    </div>
                    <ng-template #loading>
                        <div>Loading...</div>
                    </ng-template> ` })
export class AppComponent {
    private users$;
    constructor(private http: HttpClient) {
        this.users$ = this.http.get('https://randomuser.me/api/?results=5')
    }
}
```