Angular Advanced

João Gonçalves

Interactive Developer / Trainer / Consultant

Applied Mathematics Degree

Master Multimedia Communication

http://joaogoncalves.net

edu@joaogoncalves.net

@joaopapin

FLAG

Dependency Injection

FLAG

Modern Angular – Dependency Injection

- Understanding how the dependency injection mechanism works
- Learning about injection contexts
- Using the “inject” function instead of constructor-based dependency injection and the benefits of this approach
- Using the “inject” function to convert class-based guards/resolvers/interceptors to functional ones

FLAG

Modern Angular – Dependency Injection

Dependency injection, or **DI**, is famously the most loved and stable feature that Angular provides as a framework

So what changed, and importantly, why, if it was already so stable?

“inject” (actually not even a new function!), which, almost accidentally, made a minor revolution in Angular projects all over the community

FLAG

DI – Dependency Injection

Introducing **Dependency Injection (DI)** as a design pattern

How Angular **implements DI**

Registering object **providers** and using **injectors**

Angular application is a **collection of components, directives,and services** that may **depend on each other.**

FLAG

DI – Dependency Injection

each component can **explicitly** **instantiate its dependencies**

Angular can do this, using its **dependency injection (DI)** mechanism

A **function that receives an "Object"** as an argument

That **Object is "Injected"** into the function

FLAG

DI – Dependency Injection

the **createShipment() function has a dependency**: **Product**

But the function itself **doesn't know how to create Product**

```
var product = new Product();

createShipment(product);
```

FLAG

DI – Dependency Injection

decoupling the **creation** of the Product object from its **use**

but

both are **located in the same script**

```
var product = new Product();

createShipment(product);
```

FLAG

Dependency Injection pattern is:

If **object A depends** on an **object** identified by **a token (a unique ID) B**

Object A won't explicitly use the new operator to **instantiate the object that B** points at.

it will **have B injected by the environment(IOC)**

FLAG

Dependency Injection pattern is:

Object A just needs to declare,
"**I need an object known as B**"

Object A doesn't request a specific object type (Product) but rather delegates the responsibility of **what to inject to the token B**

FLAG

DI – benefits, “injected” vs “new”

DI helps you write code in a **loosely coupled** way and ...

makes your code more **testable and reusable**

FLAG

DI – Angular

In Angular we **inject services or constants**.

The **services** are **instances of TypeScript classes** that **don't have UI** and just **implement business logic** of your app

have a **ProductComponent** that
gets product details using the **ProductService** class

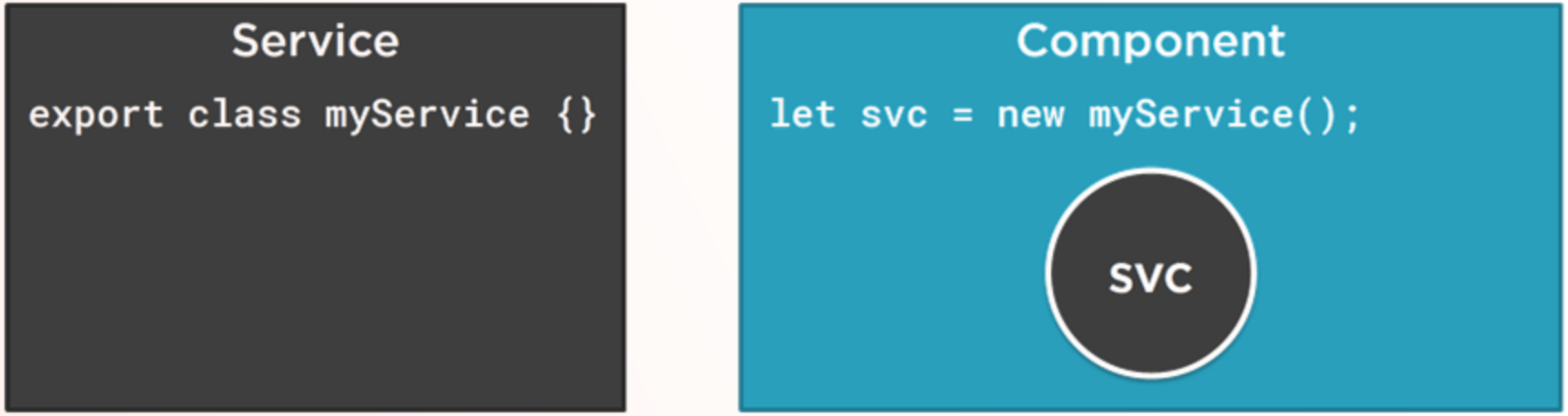
FLAG

DI – example...

Without DI:

ProductComponent needs to know **how to instanteate** the **ProductService** class

using **new**, calling **getInstance() (singleton),** or invoking some **factory function createProductService()**

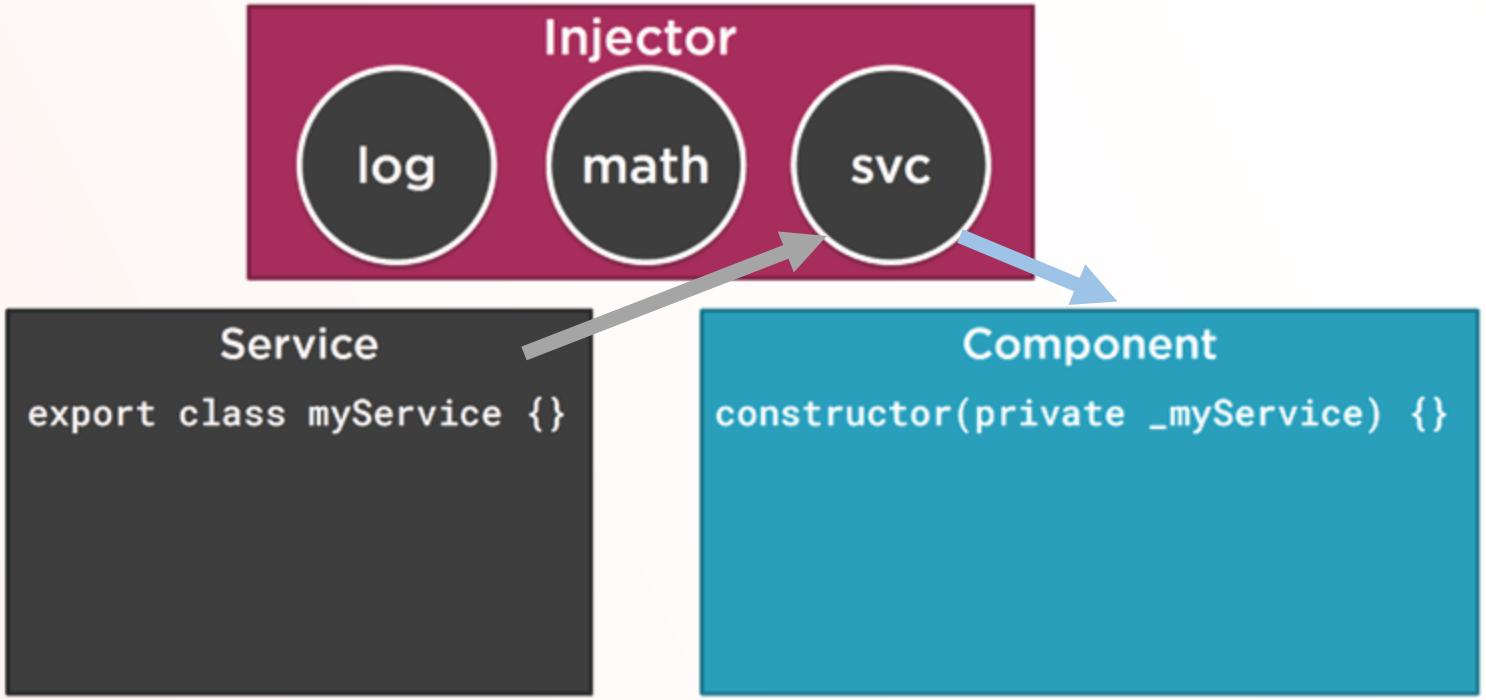


FLAG

DI – example...

DI allows you to decouple application components and services

Angular uses the concept of a token



FLAG

DI – Dependency Injection

A **provider** is an instruction to **Angular** about **how to create an instance of an object for future injection** into a target **component**, **service**, or **directive**

```
@Component({
  providers: [ProductService]  ❶
})
class ProductComponent {
  product: Product;

  constructor(productService: ProductService) {  ❷

    this.product = productService.getProduct();  ❸
  }
}
```

```
@Component({
  providers: [{provide: ProductService, useClass: ProductService}]
})
class ProductComponent {
  product: Product;

  constructor(productService: ProductService) {

    this.product = productService.getProduct();
  }
}
```

FLAG

DI – Dependency Injection

Question:
when the instance of the service is created?

Answer:
depends on the **decorator** in which you specified the **provider** for this service

FLAG

DI – Dependency Injection

inside the **@Component()** decorator.

Angular create an instance of ProductService **when the ProductComponent is created**

Inside de **@NgModule** decorator:

the service instance is **created on the app level** as a **singleton**,
when the first class where the **ProductService is instantiated**,

and all components could **reuse it**

FLAG

DI – Dependency Injection

Reusability:

reuse the same ProductComponent with a **different implementation** of the type ProductService

```
providers: [{provide: ProductService, useClass: AnotherProductService}]
```

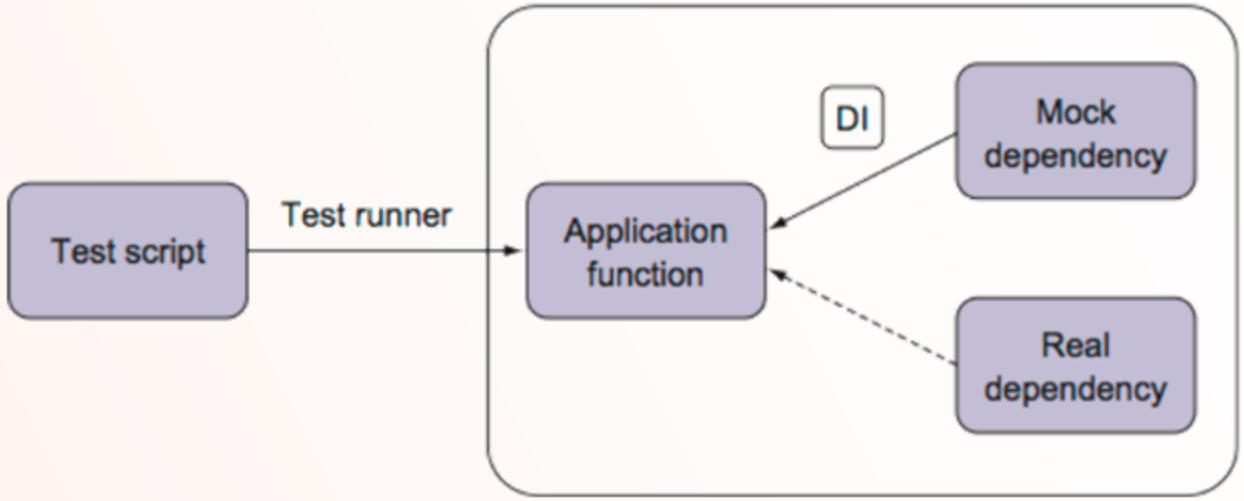
FLAG

DI – Dependency Injection

Testability:

DI increases the testability of your components in isolation

inject **mock objects** if you want to unit-test your code



FLAG

DI – Injectors & Providers

Angular uses a special abstraction, called "**injector**"

This injector is what keeps the registry of dependencies

and allows the retrieval of values via tokens

FLAG

DI – Injectors & Providers

Angular creates special injectors when our application runs, provides dependencies, and then injects them into components (directives, pipes, etc…) implicitly

we just list our dependencies as constructor parameters, and Angular then deduces when to inject what

limits us to only using it on classes, as we need constructor functions to trigger

FLAG

DI – Injectors & Providers

Each **component** have is **Injector instance** capable of injecting objects

Any **Angular application** has a **root injector** available to all of its **modules**

(providedIn: 'root')

(ver imagem, EmployeeList)

This process will repeat every time an EmployeeListComponent is created

FLAG

DI – Injectors & Providers

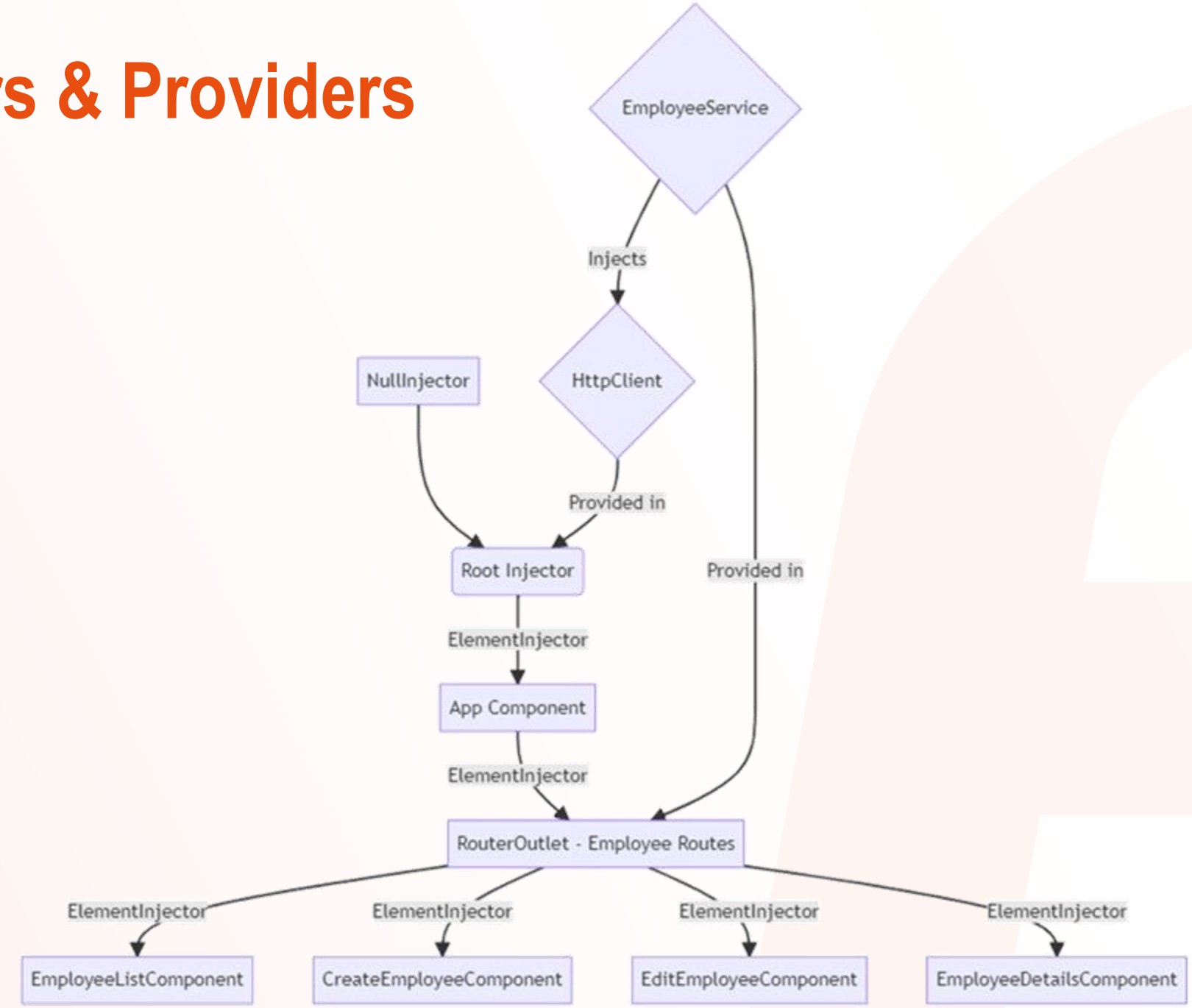
understanding that “**dependency**”, often meaning just “some service”,
isn’t always an instance of a service class

we are able to inject references to DOM elements, or other directives or components
into our components and directives

if we write a directive, we need the reference of the element on which the directive is called, this is done via injecting **the ElementRef**

FLAG

DI – Injectors & Providers



DI – Injection Contexts

Angular we rely on class constructors to inject dependencies

another necessary parameter is that we also need a class decorated by one of Angular's decorators, like Component, Directive, Pipe, or Injectable

limits our ability to use DI with anything other than Angular building blocks

FLAG

DI – Injection Contexts

In every programming language or framework, the code that we run is executed in some sort of context

Angular's dependency injection works in a similar fashion

DI happens when we try to make an instance of a component or directive

that instantiation may happen under different circumstances

each time when DI is invoked, there is a different context in which this will happen

this is how we are able to use different injectors , so we can get different references while requesting the same token

FLAG

DI – Injection Contexts

Angular injection contexts are very specific:

Creation of a class that is instantiated by the DI system, (Injectable, Component, etc)

Initializer fields of such a class, because the TypeScript gets compiled to JavaScript in the end, all property initializations end up in the constructor

Factory function for the useFactory option when providing a dependency

Factory function for an InjectionToken

FLAG

DI – The inject function

the problem with not being able to inject dependencies in places other than classes is primarily the absence of a constructor where we can name the dependencies

we would need a function that would take a token, like the name of the service we want to inject, as a parameter

(Code , change EmployeeList)

FLAG

DI – The inject function

Injecting dependencies outside classes

Code ...

FLAG

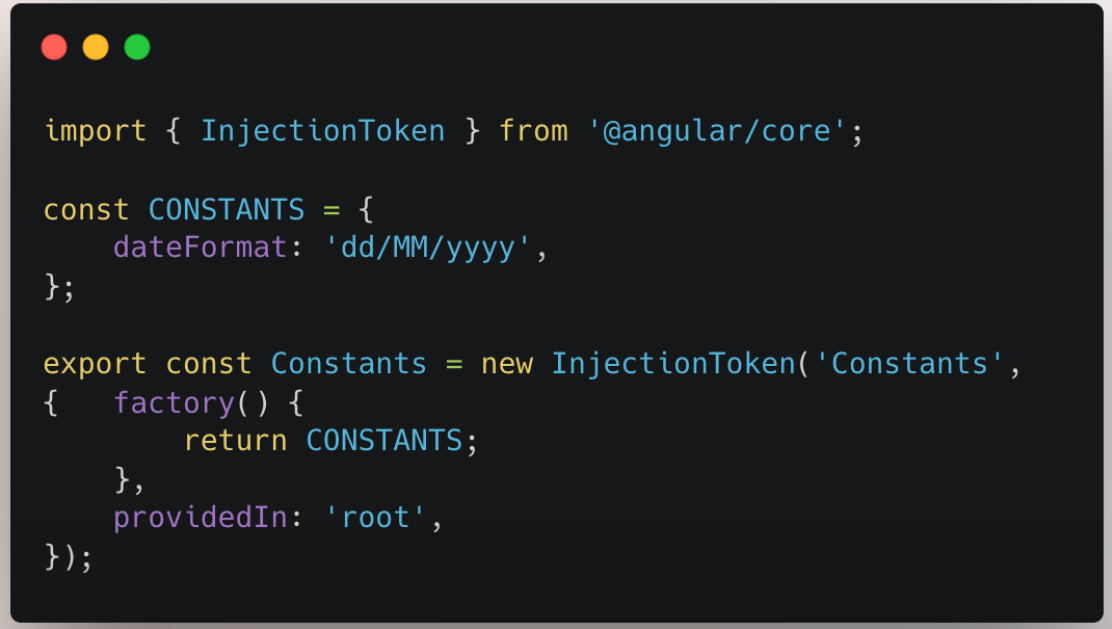
Modern Angular – Dependency Injection

Why we should always use inject?

- Improved reusability: we can now use DI inside functions, meaning we can extend service functionality to places where previously it would not have been possible
- Type inference of InjectionTokens: Not all dependencies are services, sometimes, we need to be able to inject some constant values or even functions directly

FLAG

Modern Angular – Why use inject?



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

const CONSTANTS = {
    dateFormat: 'dd/MM/yyyy',
};

export const Constants = new InjectionToken('Constants',
{   factory() {
        return CONSTANTS;
    },
    providedIn: 'root',
});
```

Type inference of InjectionTokens :

shared constants that are used in various parts of the application, like formatting dates in our application

```
import { Inject } from '@angular/core';
import { Constants, CONSTANTS } from 'app/shared/constants.ts';

export class MyComponent {
    constructor(@Inject(Constants) constants: typeof
CONSTANTS);
}
```

```
export class MyComponent {
    constants = inject(Constants);
}
```

FLAG

Modern Angular – Why use inject?

- Easier component inheritance :

inheriting components is not considered to be the best of practices, but, there are scenarios where this can be useful

```
export class ParentClass {
    constructor(
        private router: Router,
    ) {}
}

@Component({
    ...
})
export class ChildComponent extends ParentClass {
    constructor(
        private router: Router,
        private http: HttpClient,
    ) {
        super(router);
    }
}
```

```
export class ParentClass {
    private router = inject(Router);
}

@Component({
    ...
})
export class ChildComponent extends ParentClass {
    private http = inject(HttpClient);
}
```

FLAG

Modern Angular – Dependency Injection

Why we should always use inject?

- Custom RxJS operators: developers also create their own custom RxJS operators. those operators somehow need a dependency from the DI tree
- Ditching the constructor altogether

FLAG

Modern Angular – Dependency Injection

Building Functional guards, resolvers, and interceptors

Building an AuthGuard

Code …

FLAG

Modern Angular – Dependency Injection

Building Functional guards, resolvers, and interceptors

Building a Employee **Resolver**

common task when developing complex web apps is ensuring some data is loaded via an HTTP request before rendering the component that requires that data

Code …

FLAG

Modern Angular – Dependency Injection

Building Functional guards, resolvers, and interceptors

Adding tokens to HTTP requests (interceptor)

Code …

FLAG

Modern Angular – Migrating to functional guards/resolvers/interceptors

Class-based guards/resolvers/interceptors are deprecated
as per **v15** in favor of **functional** ones

think about ditching the previous versions
of those building blocks and switching to functions

Angular provides some utilities we can use
to provide backward compatibility

FLAG

Modern Angular – Migrating to functional guards/resolvers/interceptors

Migrating guards and resolvers

if we already had our AuthGuard as a class, wanted to use it as a function on a route's canActivate option

we could use the **mapToCanActivate** function to achieve this

The mapToCanActivate function will convert the array of class-based guards to an array of CanActivateFn functions

```
{
    path: 'employees',
    providers: [EmployeeService],
    canActivate: mapToCanActivate([AuthGuard]),
    loadChildren: () => {
        return import('./pages/employees/employees.routes').then(
            (m) => m.routes,
        );
    },
},
```

FLAG

Modern Angular – Migrating to functional guards/resolvers/interceptors

This function, is also available to support guards
for other scenarios and also resolvers:

mapToResolve()

mapToCanDeactivate()

mapToCanMatch()

mapToCanActivateChild()

FLAG

Migrating **interceptors**

In the case of interceptors, registering them previously worked a bit differently, with Angular exposing a special HTTP_INTERCEPTORS InjectionToken

with the **withInterceptors** functions in the standalone setup,
older registration of class-based interceptors
using the HTTP_INTERCEPTORS can still be included,
witdth a special function called withInterceptorsFromDi()

Modern Angular –

Exercicio Biblioteca

FLAG

Dependency Injection

Deep Dive

FLAG

Modern Angular – DI

We already covered the basic and most common use cases for **dependency injection** in Angular

learn about **manipulating DI** in specific ways to further simplify our code

We already see about how dependency injection works

provide a token, then request it somewhere, then Angular searches the DI tree starting from the current component up to the NullInjector, finds the value (or fails to find one), and returns it

FLAG

Modern Angular – DI

it is possible to alter this process

we can specify where the lookup should start

only look for dependencies in this current context

using special decorators like Self, Optional, SkipSelf, and Host

FLAG

Modern Angular – DI

- **@Optional**: this decorator marks a dependency as optional, if Angular does not find it, it will not go up to the NullInjector and throw an error.
  (ex: developer can provide a global value for some dependency, but a component wants to use a default one if the global one is not provided)
- **@Self**: this decorator limits the search for a given dependency to the component's own ElementInjector

FLAG

Modern Angular – DI

- **@SkipSelf**: this one is the opposite of @Self, and starts the search for a given dependency from its parent injector
  (can be used to enforce using a global provider instead of a local one in some scenarios)
- **@Host**: finally, this decorator will limit the search for a dependency to the component and its direct parent
  (useful for components that are usually used in pair (for instance, A ListComponent and ListItemComponent)

FLAG

Modern Angular – DI

Before the inject function became publicly available, with classes

```
export class MyComponent {
  constructor(
    @SkipSelf() @Optional() private readonly someDependency: SomeDependency,
  ) {}
}
```

now

```
export class MyComponent {
    someDependency = inject(SomeDependency, {optional: true, skipSelf: true});
}
```

FLAG

Modern Angular – DI

Let's see an Example, to create Truncate text directive

- Directive will be used in multiple places
- By default, it will limit text to 80 characters
- An optional Input can be provided to change the limit on some given element
- If the developers want to have a different character limit globally, they can provide it and not have to type the input every time they use the directive
- If a global value is provided, but in some component (based on its relation to the viewport) we want a different limit, we should be able to provide that limit to that component only

FLAG

Summary

Since Angular v14, the **inject** function became publicly available

The inject function will take any dependency token and return its provided value

used to inject dependencies and services into functions, as opposed to only classes

It is now commonly preferred to use the inject function instead of the constructor DI

Summary

Guards/resolvers/interceptors have now switched to being fully functional

Legacy class-based guards/resolvers can still be used with helper functions

Legacy interceptors can still be used with the withInterceptorsFromDi function

The inject function also supports dependency lookup modifiers like host, optional, self and skipSelf