Angular

João Gonçalves

Interactive Developer / Trainer / Consultant

Applied Mathematics Degree

Master Multimedia Communication

http://joaogoncalves.net

edu@joaogoncalves.net

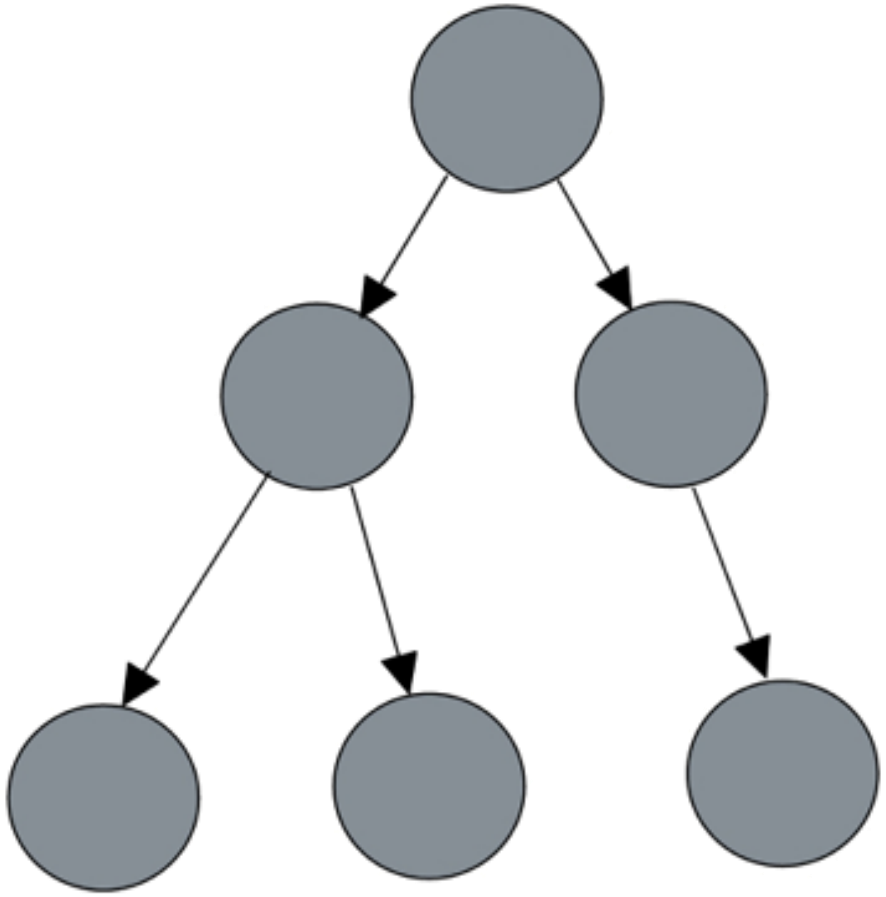
Structuring User Interfaces with Components

Components are the basic **building blocks** of an Angular application

They are responsible for the **presentational logic** of an Angular application

organized in a **hierarchical tree** of components that can interact with each other

can **communicate and interact with one or more components** in the component tree



Structuring User Interfaces with Components

Controlling data representation:

- Displaying data conditionally - @if()
- Iterating through data - @for()
- Switching through templates

Component inter-communication

Angular components expose a public API that allows them to **communicate** with other components

This API encompasses **input properties**

also exposes **output properties** we can bind event listeners

Encapsulating CSS styling

We can set up different levels of **view encapsulation**

Emulated: Entails an emulation of native scoping in shadow DOM by sandboxing the CSS rules under a specific selector that points to a component.

Native: Uses the native shadow DOM encapsulation mechanism of the renderer that works only on browsers that support shadow DOM.

None: Template or style encapsulation is not provided. The styles are injected as they were added into the <head> element of the document

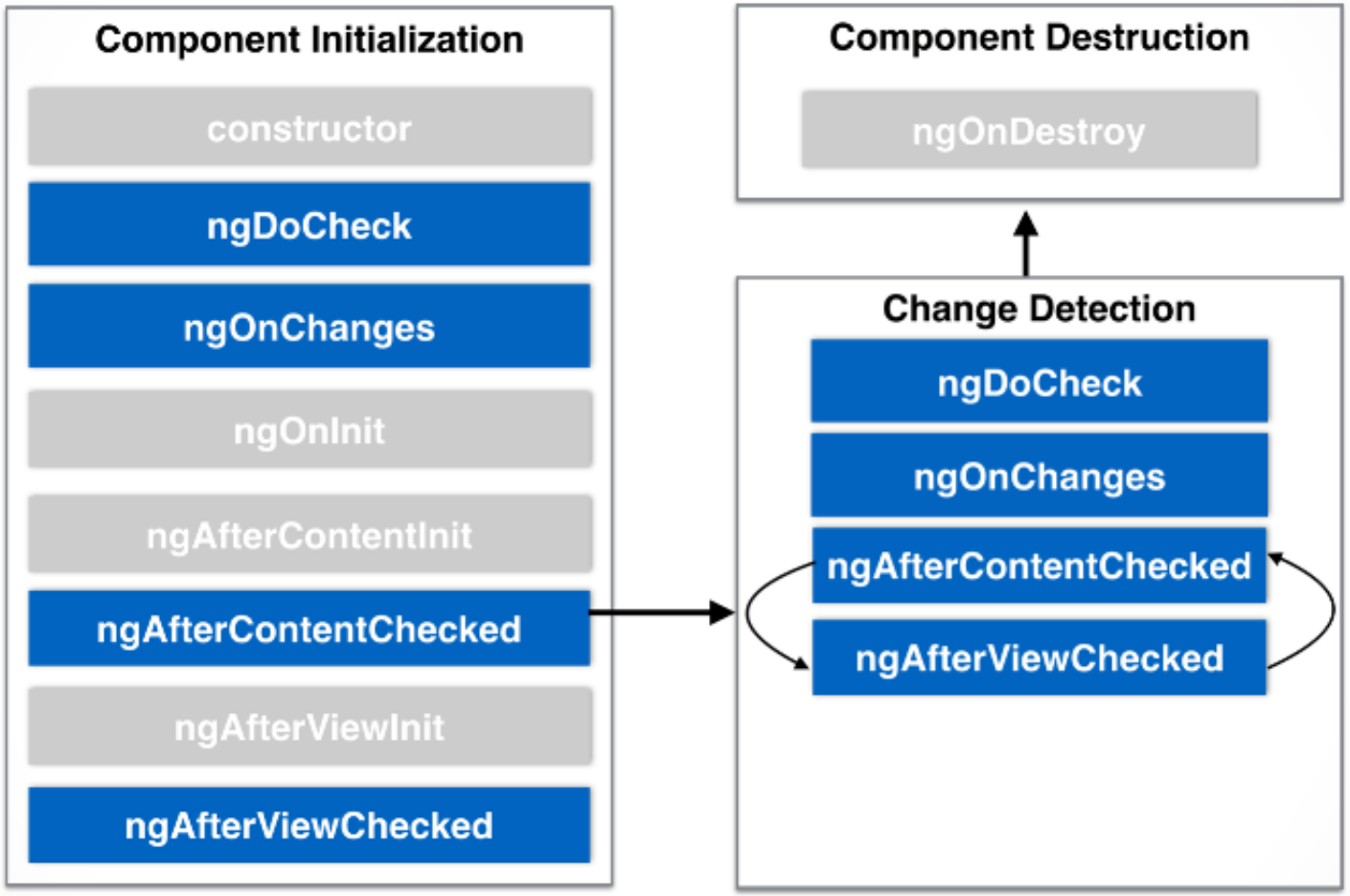
COMPONENT LIFECYCLE

COMPONENT LIFECYCLE

Various **events** happen during the **lifecycle of an Angular component**:

it gets **created, reacts to different events**, and gets **destroyed**

COMPONENT LIFECYCLE



Pipes & Directives

Pipes and Directives

We will take our components to the next level using Angular **pipes** and **directives**

Pipes allow us to digest and transform the information we bind in our templates

Directives enable more ambitious functionalities, such as manipulating the DOM or altering the appearance and behavior of HTML elements

Pipes and Directives

Manipulating data with pipes

Building pipes

Building directives

Pipes and Directives - Manipulating data with pipes

Pipes allow us to transform the outcome of our expressions at the view level

take data as input, transform it into the desired format, and display the output in the template

EX: expression | pipe

EX (params): expression | pipe:param

Pipes and Directives - Manipulating data with pipes

Pipes can be used with interpolation and property binding:

uppercase/lowercase, percent , date ,
currency, json, keyvalue, slice, async

Pipes and Directives - Building directives

Angular directives are HTML attributes that extend the behavior or the appearance of a standard HTML element

When we apply a directive to an HTML element or even an Angular component, we can add custom behavior or alter its appearance.

Pipes and Directives - Building directives

There are **three types of directives**:

Components: Components are directives that contain an associated HTML template.

Structural directives: These add or remove elements from the DOM.

Attribute directives: These modify the appearance of a DOM element or define a custom behavior.

Dependency Injection

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

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

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.**

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

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);
```

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);
```

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)**

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**

DI – benefits, “injected” vs “new”

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

makes your code more **testable and reusable**

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

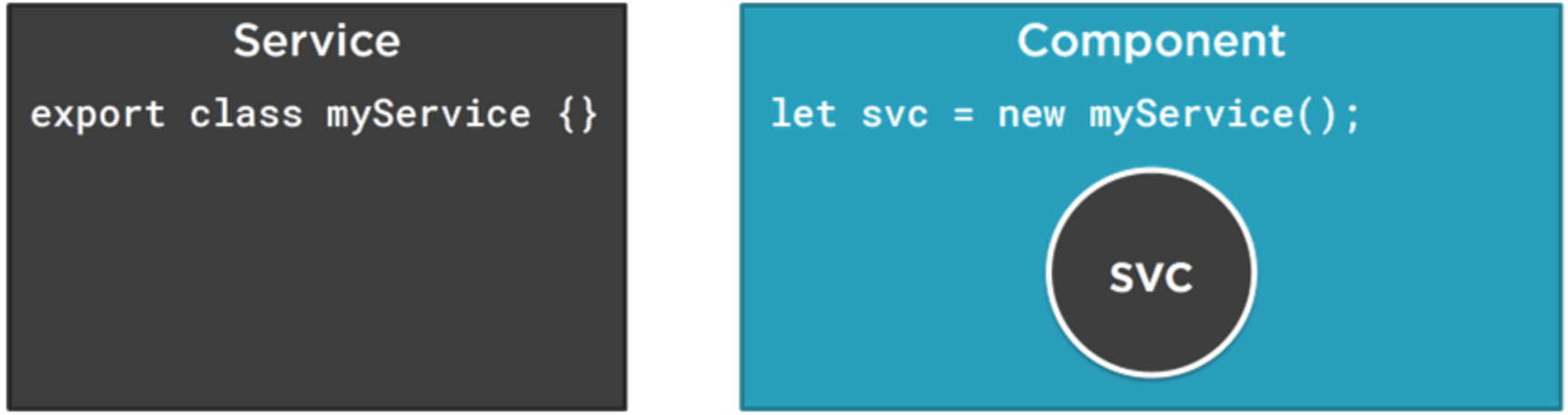
have a **ProductComponent** that **gets product details** using the **ProductService** class

DI – example...

Without DI:

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

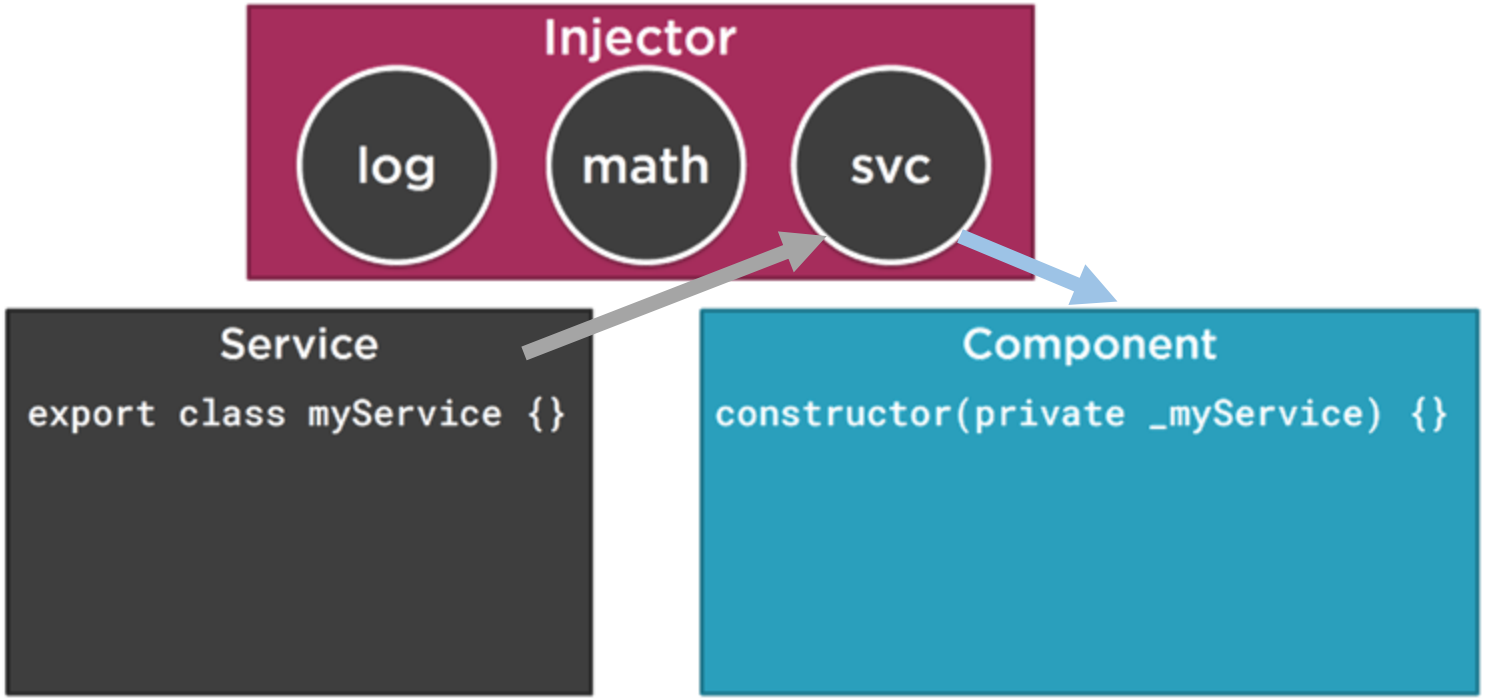
using **new**, calling **getInstance() (singleton),** or invoking some **factory function createProductService()**



DI – example...

DI allows you to **decouple application components and services**

Angular uses **the concept of a token**



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();
  }
}
```

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

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**

DI – Dependency Injection

Reusability:

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

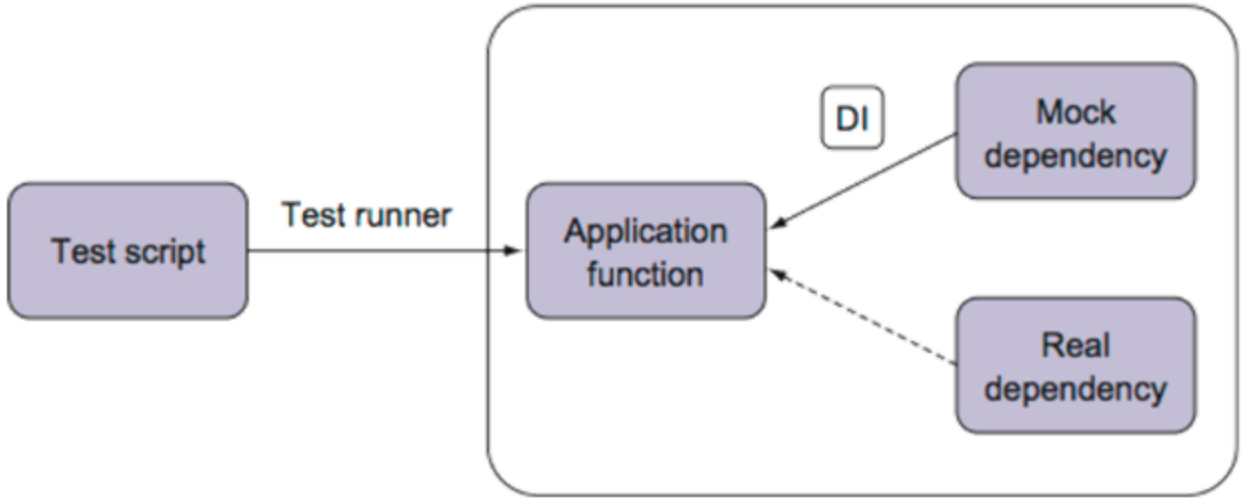
```
providers: [{provide: ProductService, useClass: AnotherProductService}]
```

DI – Dependency Injection

Testability:

DI increases the testability of your components in isolation

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



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

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

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’)

DI – Injectors & Providers

Creating our **first Angular service**

Providing dependencies across the application

Injecting services in the component tree

Overriding providers in the injector hierarchy

Code ...