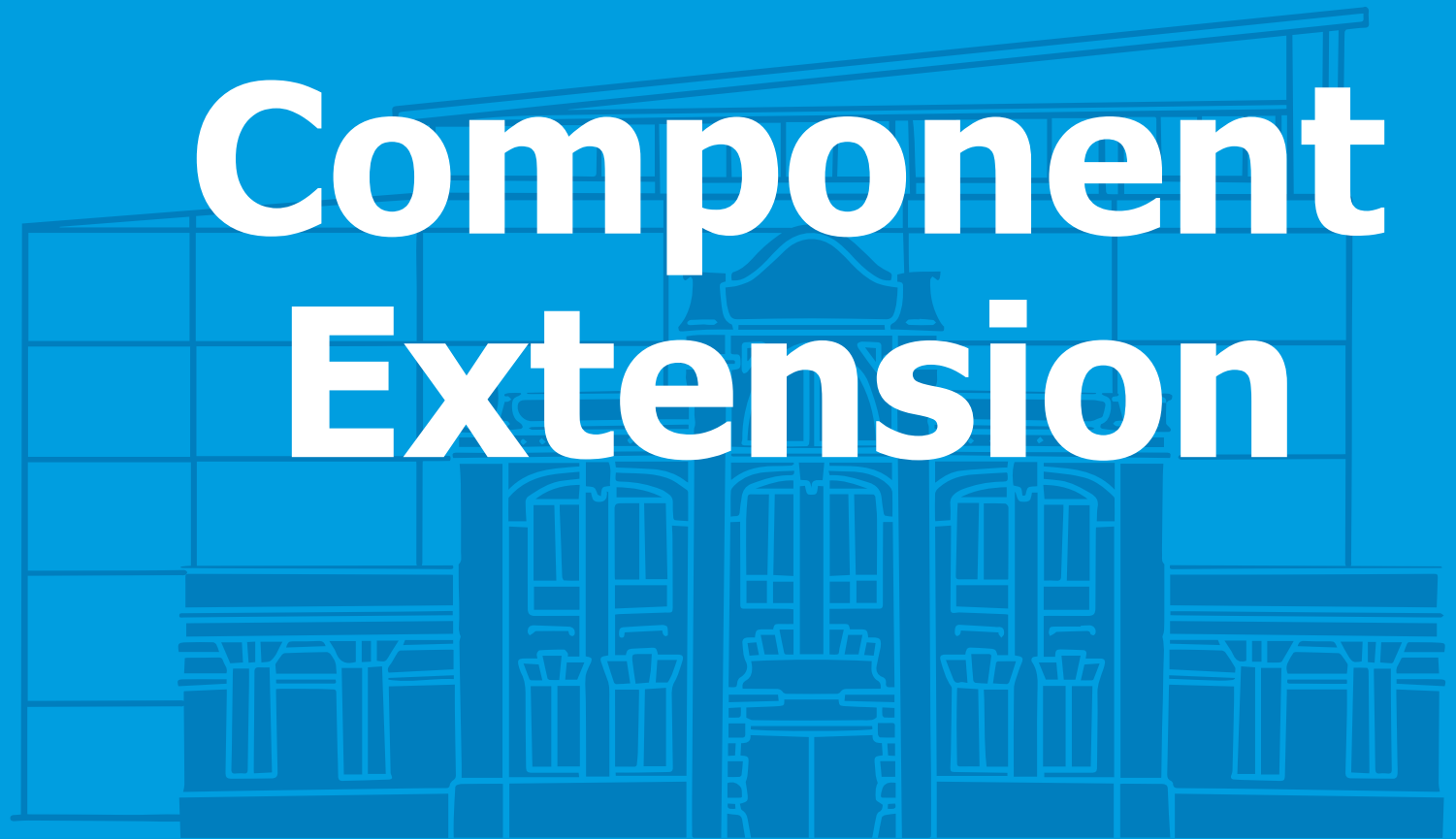
netmind
a bts company

Angular 14 - 09

Component Extension



Component Extension

- We are going to continue exploring the components, and we will see how to extend their basic functionality to perform other tasks.
- To do this, we will take advantage of certain TypeScript features, such as **strong typing**.
- Interfaces allow you to define contracts that those who implement them have to comply with, guaranteeing features that can be used by rendering engines and also by editors.

Benefits of "strong typing"

- Our array of products has no type defined, so the editor cannot check for errors while typing.
- One way to avoid this is to define an **interface** with all the properties that the array should have, and declare the array to be of that interface type.
- Another alternative is defining a **class**, specially if you want to add it business logic.

🔗 **Create a separate file “iproduct.ts" in the “models" directory**

```
ng g i models/IProduct
```

```
export interface IProduct {
    name:string;
    code:string;
    image:string;
    date:Date;
    price:number;
    stars:number;
}
```

🔗 **Create a separate file "product.ts" in the “models" directory**

```
ng g cl models/Product
```

```
export class Product {
  private _name: string;
  private _code: string;
  private _image: string;
  private _date: Date;
  private _price: number;
  private _stars: number;

  constructor(...) {...}
  // getters and setters
  // other methods
}
```

Benefits of "strong typing"

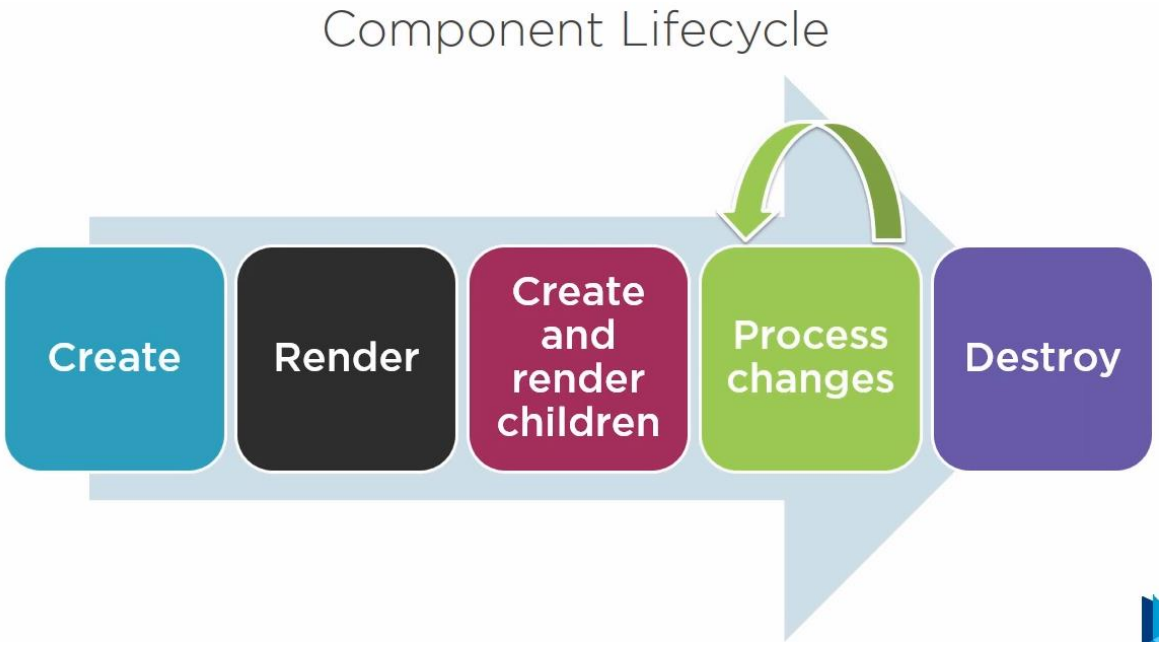
- Next, in the component, we need to perform the import of the interface, and redefine the data to be cast according to the interface:

```
import { IProduct } from 'src/app/models/iproduct';
...
   products: IProduct[] = [
    {
      name: 'Leaf Rake',
      code: 'GUN-0611',
      image: 'LeafRake.png',
      date: new Date(2016,3,19),
      price: 19.95,
      stars: 3,
    },
    {
      name: 'Garden Cart',
      code: 'GUN-0023',
      image: 'GardenCart.png',
      date: new Date(2016,5,21),
      price: 32.99,
      stars: 4,
    },
  ];
```

```
import { IProduct } from 'src/app/models/iproduct';
...
productObjs: Product[] = [
    new Product('Leaf Rake',
    'GUN-0611',
    'LeafRake.png',
    new Date(2016,3,19),
    19.95,
    3),
    new Product('Garden Cart',
    'GUN-0023',
    'GardenCart.png',
    new Date(2016,5,21),
    32.99,
    4)
  ];
```

Components lifecycle

- Components have a **lifecycle** which is the steps Angular takes to create and maintain it, until it is removed from the DOM

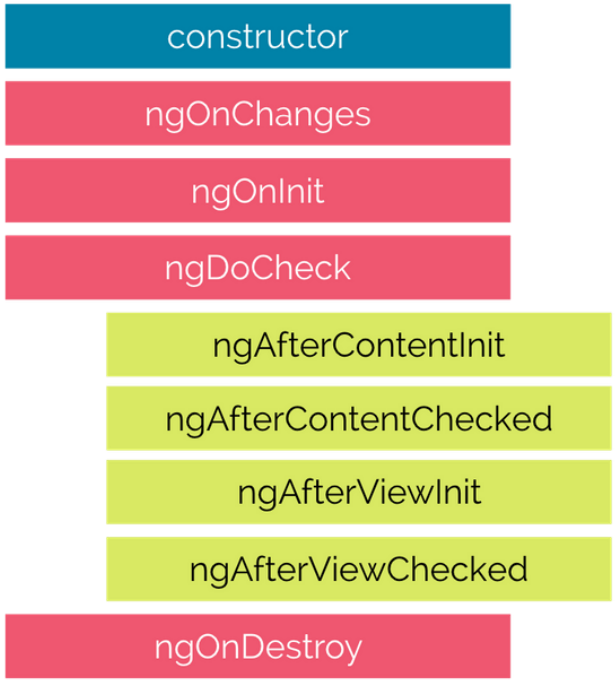


Components lifecycle

- We can intervene between these phases, and include "**lifecycle hooks**" to modify the default behavior.
  - https://angular.io/guide/lifecycle-hooks

- The 3 most used are:
  - **onInit**: for component initialization. It is the perfect step to read data from a "backend" server.
  - **onChanges**: to execute code after a change occurs in an input element.
  - **onDestroy**: Allows to perform memory cleanup tasks before destroying the component.

- We will need to **implement hook interfaces** that are already defined in Angular.
  - This requires the reference corresponding to the hook that we are going to use.
  - And the implementation of the method (similar to that of an event handler).



Components lifecycle

HOOK METHOD	PURPOSE
ngOnChanges()	Respond when Angular sets or resets data-bound input properties. The method receives a SimpleChanges object of current and previous property values. This happens frequently, so any operation you perform here impacts performance significantly.
ngOnInit()	Initialize the directive or component after Angular first displays the data-bound properties and sets the directive or component's input properties.
ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own.
ngAfterContentInit()	Respond after Angular projects external content into the component's view, or into the view that a directive is in.
ngAfterContentChecked()	Respond after Angular checks the content projected into the directive or component.
ngAfterViewInit()	Respond after Angular initializes the component's views and child views, or the view that contains the directive.
ngAfterViewChecked()	Respond after Angular checks the component's views and child views, or the view that contains the directive.
ngOnDestroy()	Cleanup just before Angular destroys the directive or component. Unsubscribe Observables and detach event handlers to avoid memory leaks.

Component Lifecycle Resource Usage

```
import { Component, OnChanges, OnDestroy, OnInit } from '@angular/core’;

@Component()
export class ProductsListComponent implements OnInit, OnChanges, OnDestroy {

...
  ngOnInit() {
    console.log(`Spy #${this.products} onInit`);
  }

  ngOnChanges() {
    console.log(`Spy #${this.products} onInit`);
  }

  ngOnDestroy() {
    console.log(`Spy #${this.products} onDestroy`);
  }
...
}
```

- Implementing interfaces is not required (they don't exist in JavaScript yet), but it is good practice.

Custom pipes

- A Pipe is just another component, therefore it is a decorated class. The decorator in this case is **@Pipe** .

🔗 **To create a pipe we will use the command:**

```
ng g p [pipe_name]
ng g p pipes/FilterProducts
```

- We will use pipes through its **name** property (filterProducts).
- As any other component it is added to the module.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'filterProducts'
})
export class FilterProductsPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }

}
```

Using custom pipes

- We can add business logic to **transform** function:

```
@Pipe({
  name: 'filterProducts'
})
export class FilterProductsPipe implements PipeTransform {

  transform(value: Product[], text: string): Product[] {
    console.log('filtering:', value, text);
    if(text){
      return value.filter(aP=>aP.name.toLocaleLowerCase().indexOf(text.toLocaleLowerCase())>=0);
    }else return value;
  }

}
```

- In the UI, we just use it like any other pipe.

```
<input [(ngModel)]="filter_text" placeholder="input a text to filter"/>
...
<tr *ngFor="let product of productObjs | filterProducts:filter_text">...</tr>
```

Custom directives: attribute

- Directives are classes that add additional behavior to elements .A directive is just another component a decorated class with **@Directive**.
  - https://angular.io/guide/attribute-directives

🔗 **To create a pipe we will use the command:**

```
ng g d [directive_name]
ng g d directives/highlight
```

- We will invoke directives using its **selector** property (highlight).
- As any other component it is added to the module.

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

@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {

  constructor() { }

}
```

Custom directives : attribute

- To modify the element we will need the **ElementRef**. It grants direct access to the host DOM element through its nativeElement property:
- Then we must add ElementRef in the directive's **constructor()** to inject a reference to the host DOM element, the element to which apply Highlight.
- In the constructor we can add the necessary logic that implements the directive functionality.

```
import { Directive, ElementRef } from '@angular/core';

@Directive()
export class HighlightDirective {

  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }

}
```

- In the UI, we just use it in the desired tag.

```
<td highlight>{{product.name}}</td>
```

Custom directives : structural

- Structural directives work in a different way as they are capable of completely changing the element on which they are applied.
  - Examples of structural directives are *ngFor and *ngIf.
  - https://angular.io/guide/structural-directives
- For doing this, it uses **TemplateRef** (reference to content enclosed within the container) and **ViewContainerRef** (Refers to the Container to which directive is applied).
- We can get input parameters using **@Input**.

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[replicate]',
})
export class ReplicateDirective {
  constructor(private templateRef: TemplateRef<any>, private viewContainer: ViewContainerRef) {}

  @Input() set replicate(nTimes: number) {
    for (var i = 0; i < nTimes; i++)
      this.viewContainer.createEmbeddedView(this.templateRef);
  }
}
```

Custom directives : structural

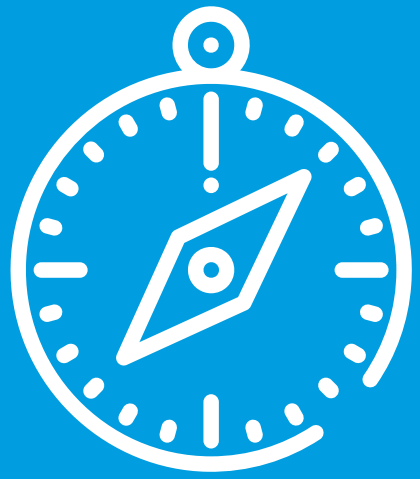
- Structural directives are called using ***[name]**, in this case `*replicate="[number]`“.
- When we apply the directive to a selector, it modifies the container, in this case, to replicate the element.

```
<h2 *replicate="4">Title to be repeated</h2>
```

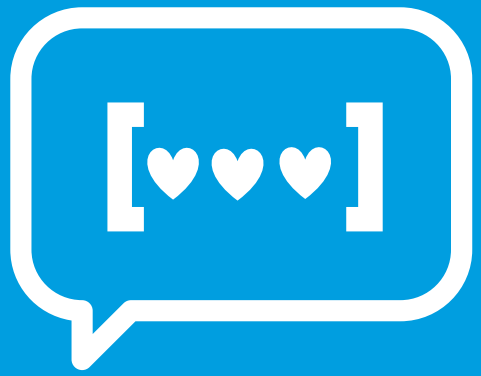
Title to be repeated
Title to be repeated
Title to be repeated
Title to be repeated

Let's put it into practice: Tasks/Projects App

1. Add models for project and task.
2. Create a pipe that allows you to filter the list of tasks based on two parameters: a field and its value
3. Then, make it generic for applying to projects.
4. Create an attribute directive for converting any string to uppercase and enclose in a red border, with 2px span.
5. Create a structural directive that applies a delay of n seconds for showing the element
    - Tip: use timeout function.



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: