Components

The Building Blocks of Our Application

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Questions

Software University

sli.do

#angular

Table of Contents

1. Components Basic Idea
2. Creating Components
3. Bootstrapping
4. Data Bindings & Templates
5. Lifecycle Hooks
6. Signals
7. Component Interaction

Components: Basic Idea

The Main Building Block

The Idea Behind Components

- A component controls **part** of the screen (the view)
- You define **application logic** into the component
- Each component has its **own** HTML/CSS template

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

@Component({
    selector: 'app-root’,
    standalone: true,
    template: `<h1>{{title}}</h1>`,
    styles: [ `h1 {
    background-color: red;}` ]
})

export class AppComponent { title = 'App Title'; }
```

Unique html template and styling

Components as Standalone Units

- In Angular 18, **components** are typically **standalone**, meaning they do not necessarily require an `NgModule` for declaration. This simplifies the architecture and boosts flexibility

```
standalone: true
```

- Standalone components are promoted by Angular 18 as **default approach**
  - Simplifies architecture
  - Enhances modularity

Components as Standalone Units

- Components can exist **independently** without being tied to a module
  - Makes application more performant, due to improved **tree-shaking** (removal of unused code)
  - This is especially useful for small, isolated components or when you want to lazy load components without the need for a full module

Creating Components

And Their Unique Templates

Creating Components Manually

Software University

- To create a component, we need the **Component** decorator

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

- It provides **metadata** and tells **Angular** that we are creating a **Component** and not an **ordinary** class

```
@Component({
    selector: 'app-home',
    template: '<h1>Home View</h1>',
    standalone: true
})
```

We call it whilist adding '@' in front and pass in metadata

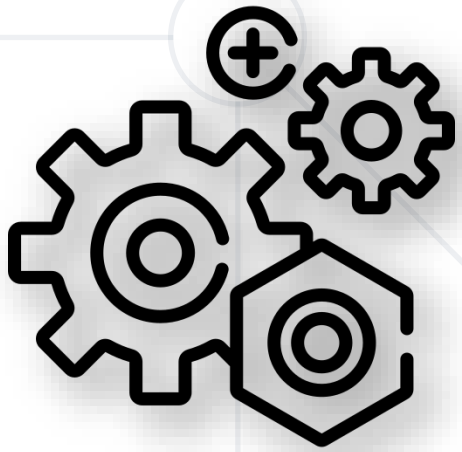
Creating Components Manually

- Component Metadata
  - `selector`
    - The component's **HTML** selector

```
selector: 'app-home'
```

  - `standalone`
    - Declares the component as **standalone**

```
standalone: true
```



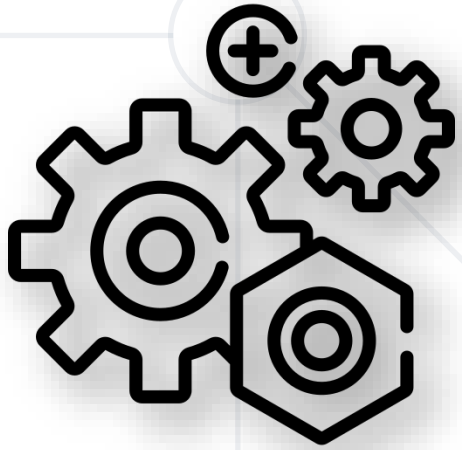
Creating Components Manually

- Component Metadata
  - `template` or `templateUrl`
    - The component's template

```
templateUrl: 'Path to template'
```

  - `styles` or `styleUrls`
    - Unique styles for the **current** component

```
styleUrls: 'Array of paths'
```



Creating Components with the CLI

- We can use the Angular **CLI** to **generate** a **new** component

```
ng generate component home
```

- The **CLI** will **create** the necessary folder structure under `src/app/home/`
- When using the **CLI**, it automatically **imports** the generated component into the relevant module or keeps it as **standalone**

Bootstrapping

Starting the Application

Bootstrapping an Application

- Involves **initializing** the Angular application by specifying the root component that serves as the entry point
- Angular supports **bootstrapping** with standalone components, meaning an **`NgModule`** is no longer **required** for the root component
- Bootstrapping process ensures that the specified root component is **rendered** in the **`index.html`** file, where it replaces the element matching the root component's selector (e.g., **`<app-root>`**)

The Initial Module

- The **bootstrapApplication** method from the **@angular/platform-browser** package is used to bootstrap standalone components directly
- Example **main.ts**

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
```

```
bootstrapApplication(AppComponent)
  .catch((err) => console.error(err));
```

Data Bindings & Templates

Repeater, Enhanced Syntax

Templates & Data Bindings Overview

- A **template** is a form of **HTML** that **tells** Angular how to **render** the component
  - **render** array **properties** using **@for** repeater
  - **render nested** properties of an object
  - **condition** statements using **@if**
  - **attach** events and **handle** them in the component
- They can be both **inline** or in a **separate** file

Render an Array Using @for

Software University

```
export class GamesComponent {
  games : Game[];
  constructor() {
    this.games = [ // Array of games ]
  }
}
```

```
<h1>Games List</h1>
  <p>Pick a game to Buy</p>
<ul>
@for(game of games; track game) {
<li> {{game.title}} </li>
}
</ul>
```

Conditional Statements Using @if and @else

```
<h1>Games List</h1>
<p>Pick a game to Buy</p>
<ul>
@for(game of games; track game) {
<li>
<div>
      {{ game.title }}
</div>
@if (game.price >= 100) {
<span>Price: {{ game.price }} - Expensive</span>
}@else if (game.price >= 50) {
<span>Price: {{ game.price }} - Moderate</span>
}@else {
<span>Price: {{ game.price }} - Cheap</span>}  </li>}
</ul>
```

Attach Events

```
<button (click)="showContent($event)">Show Content</button>
```

```
export class GamesComponent {
    public games: Game[];
    showContent: boolean;

    constructor() {
      this.games = [ // Array of games ]
    }

    showAdditionalContent($event) {
      this.showContent = true;
    }
}
```

Binding Attributes

- Binding attributes

```
export class GamesComponent {
    imgUrl: string;
    constructor() {
     this.imgUrl = "a url to an image"
    }
}
```

```
<img [attr.src]="imgUrl" />
```

The name of the property in the component

Binding CSS Classes or Specific Class Name

Software University

- Binding classes

```
<div [class]="badCurly">Bad curly</div>
```

- You can bind to a specific class name

```
<div [class.special]="isSpecial">
 The class binding is special
</div>

<div class="special"[class.special]="!isSpecial">
 This one is not so special
</div>
```

Toggle class "special" on/off

Binding Styles or Styles with Units

- Binding styles

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
<button [style.background-color]="canSave ? 'cyan': 'grey'" >
 Save
</button>
```

- Or styles with units

```
<button [style.font-size.em]="isSpecial ? 3 : 1">
 Big
</button>
<button [style.font-size.%]="!isSpecial ? 150 : 50">
 Small
</button>
```

Reference and Null-safe Operator

- Reference other elements

```
<input #phone placeholder="phone number">
<button (click)="callPhone(phone.value)">Call</button>
```

Phone refers to the input element

- You can also use the null-safe operator

```
<div>The current hero's name is {{game?.title}}</div>
<div>The null hero's name is {{game && game.name}}</div>
```

Template Expressions

- The text **between** the curly brackets is **evaluated** to a string

```
<p>The sum of two + two + four is {{2 + 2 + 4}}</p>
```

- Template expressions are **not pure** JavaScript
- You **cannot** use these:
  - Assignments (**=**, **+=**, **-=**, **...**)
  - The **new** operator
  - **Multiple** expressions
  - **Increment** or **decrement** operations (**++** or **--**)
  - **Bitwise** operators

Types of Data Binding

- There are **three types** of data binding
  - From data-source to view

```
{{expression}}
[target]="expression"
bind-target="expression"
```

  - From view to data-source

```
(target)="statement"
on-target="statement"
```

  - Two-way

```
[(ngModel)]="expression"
bindon-target="expression"
```

`FormsModule` needed

Lifecycle Hooks

Intersect Through the Loop

Lifecycle Overview

- A component has a lifecycle **managed** by Angular
- Angular offers lifecycle **hooks** that provide **control** over life moments of a component
- Directive and component instances have a **lifecycle** as Angular **creates**, **updates** and **destroys** them

NgOnInit and NgOnDestroy Example

Software University

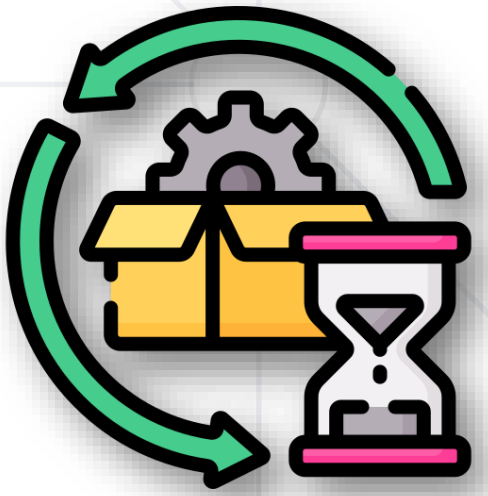
```
import { Component, OnInit, OnDestroy } from '@angular/core';

@Component({..})
export class GamesComponent implements OnInit, OnDestroy {
  games: Game[];

  ngOnInit() {
    console.log('CREATED');
  }

  ngOnDestroy() {
    console.log('DELETED');
  }
}
```

Called shortly after creation

Used for cleanup

Other Lifecycle Hooks

- All lifecycle hooks
  - `ngOnChanges()`
    - When data is changed
  - `ngDoCheck()`
    - Detect your own changes
  - `ngAfterContentInit()`
    - When external content is received



Other Lifecycle Hooks

- All lifecycle hooks
  - `ngAfterContentChecked()`
    - When external content is checked
  - `ngAfterViewInit()`
    - When the views and child views are created
  - `ngAfterViewChecked()`
    - When the above are checked
  - More at: https://angular.dev/guide/components/lifecycle

Signals

Optimize Rendering Updates

What Are Signals

- **Signals** are a new reactivity model introduced in Angular to manage **state** and **reactivity** more predictably and efficiently
- They offer a simpler alternative to **observables** for local state management in components
- A **signal** is a wrapper around a value that notifies interested consumers when that value changes
- Signals can contain **any** value, from primitives to complex data structures

Signal Key Features

- A signal holds a value, and this value is **immutable**, meaning it can only be changed explicitly (using methods like **set**, **update**)
- Signals automatically trigger view updates when their value changes, making them reactive and removing the need for manual subscriptions or **`ChangeDetectorRef`**
- Signals provide a clear, **declarative** syntax to manage state and react to state changes

Signal Core Concepts

- Use the `signal()` function to create a signal and initialize it with a value
- Signals have methods to **modify** their value
  - `set`(value): Replaces the current value
  - `update`(fn): Updates the value based on the previous one
  - `mutate`(fn): Mutates the value if it's a mutable object (e.g., arrays, objects)

Signal Example

```
import { signal } from '@angular/core’;

const count: WritableSignal<number> = signal(0);
console.log('The count is: ' + count());
count.set(3);
count.update(value => value + 1); // Increment the
count by 1
```

Computed Signals

- **Computed** signal are **read-only** signals that derive their value from other signals
- Computed signals are defined using the **computed** function and specifying a derivation
- Computed signals are both **lazily** evaluated and **memorized**
- As a result, you can safely perform **computationally expensive** derivations in computed signals, such as filtering arrays

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

Component Interaction

Passing Data in Between

From Parent to Child

```
import { Component, Input } from '@angular/core';
import { Game } from '../games/game';
```

```
@Component({
  selector: 'game',
  template: `
   <li><div> {{ game.title | uppercase }}
   @if (game.price >= 100) {
   <span>-> Price: {{ game.price }}</span>
   }</div>
   </li>`})
```

```
export class GameComponent {
  @Input('gameProp') game : Game;
}
```

The prop will come from parent

From Parent to Child

```
<h1>Games List</h1>
<p>Pick a game to Buy</p>
<ul>
<game @for=game of games; track game [gameProp]="game">
</game>
</ul>
<button (click)="showAdditionalContent()">Show Image</button>
```

Render the child into the parent template and pass the needed prop

Component Interaction

- In order to pass data from **child** to **parent** component we need the **Output** decorator and an **EventEmitter**

```
import { Output, EventEmitter } from '@angular/core';
export class GameComponent {
 @Input('gameProp') game : Game;
 @Output() onReacted = new EventEmitter<boolean>();

 react(isLiked : boolean) {
  this.onReacted.emit(isLiked);
 }
}
```

The parent will receive the event

Component Interaction

- The **Parent** component handles the event

```
<game @for="game of games" [gameProp]="game"
(onReacted)="onReacted($event)"></game>
```

```
export class GamesComponent {
 games: Game[];
 likes: number;
 dislikes : number;
 onReacted(isLiked: boolean) {
  isLiked ? this.likes++ : this.dislikes++;
 }
}
```

Summary

- Each component has its own template
- There are three types of data binding
- We can intersect the lifecycle of a component

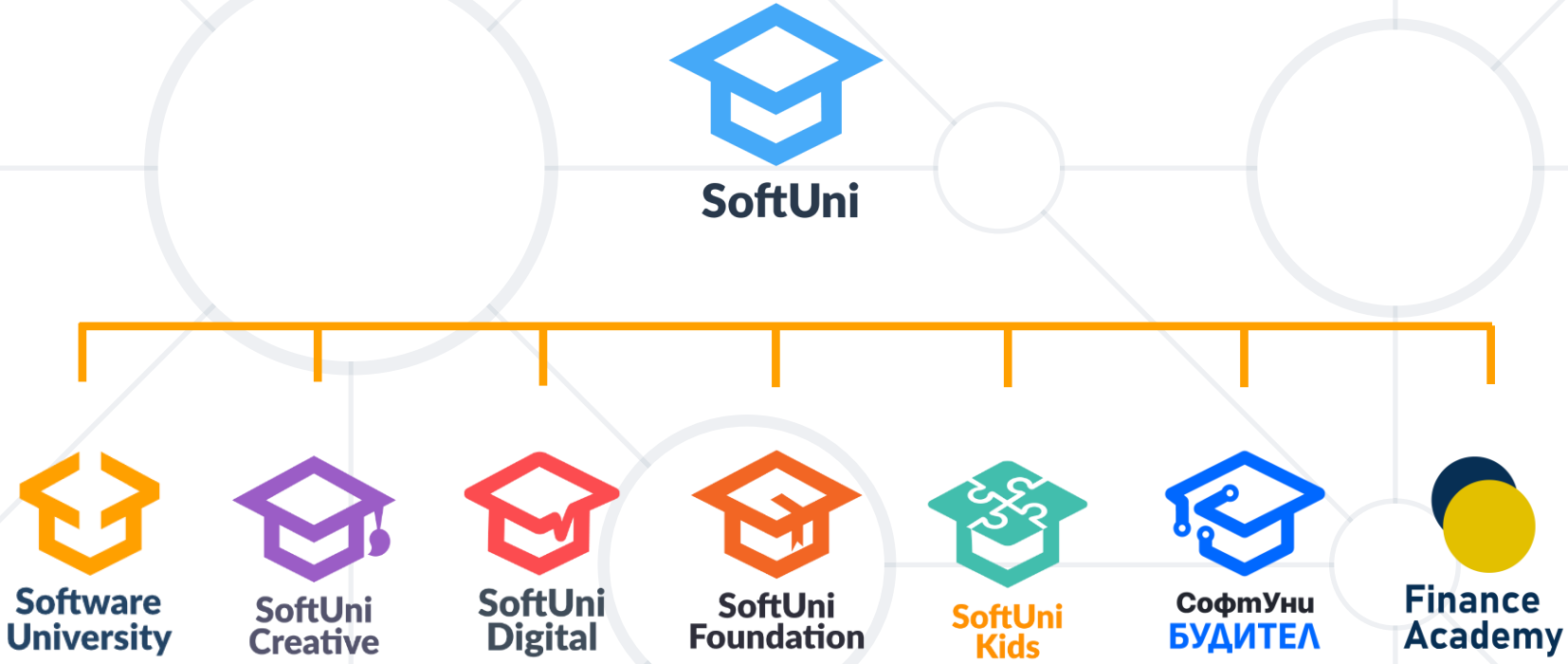
```
ngOnInit() { this.data = // Retrieve data }
```

- Components can interact with each other

```
@Output() fromChild = new EventEmitter<boolen>();
```

- Signals is a system that granularly tracks how and where your state is used throughout an application

Questions?



© SoftUni – https://about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg