Spis treści

Łukasz Kumiec

- Programuje od 2016 najpierw w AngularJS, następnie w Angular od wersji 2.0
- Posiadam artykuł na stronie angular.love o Angular Elements
- W Q-perior od czerwca 2021
- Moje zainteresowania to: podróże, motoryzacja, sport walki




Podstawowe narzędzia

Co to jest Angular?

- Framework oparty na komponentach do budowanie skalowalnych aplikacji
- Zestaw dobrze zintegrowanych bibliotek
  - Routing
  - Forms
  - Http
  - Material / CDK
- Zestaw narzędzi które ułatwiają rozwijanie, testowanie i aktualizowanie

Czemu stosujemy Angular?

- Wspierany przez Google i posiada dobre community
- Napisane jest w Typescript
- Posiada wszystko w jednym miejscu

Components

- Bloczki z których budujemy aplikacje
- Tworzymy za pomocą typescript decorator - @Component()
- Posiada selektor CSS który używamy w template
- Zawiera template HTML lub templateUrl który wskazuje plik HTML
- Możemy dodać opcjonalne style CSS lub tablice styleUrls które wskazują na pliki CSS

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

@Component({
  selector: 'app-overview',
  template: `
    <p>This is my first component!</p>
  `,
  styles: [`
    p {
        text-algin: center;
    }
  `]
})
export class OverviewComponent {}
```

```
<app-overview></app-overview>
```

Templates - interpolation

- Każdy Component ma template HTML
- Angular rozszerza template o dodatkowy syntax do wyświetlania dynamicznych wartości. Zmiana wartości w Component automatycznie odświeża wartość w template
- Dynamiczne wartości wyświetlamy w podwójnych nawiasach {{ }} – nazywa się to interpolacja

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

@Component({
  selector: 'app-overview',
  template: `
    <p>{{ message }}</p>
  `,
})
export class OverviewComponent {
    message = 'This is my first component!';
}
```

Templates – property binding

- Do każdego elementu możemy dopisać property binding
- Używamy do tego nawiasów kwadratowych

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

@Component({
  selector: 'app-overview',
  template: `
    <p [style.color]="fontColor">{{ message }}</p>
  `,
})
export class OverviewComponent {
    message = 'This is my first component!';
    fontColor = 'red';
}
```

Templates - event listeners

- Posiadamy wsparcie dla event listeners
- Binding z template odbywa sie za pomocą nawiasów ( )

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

@Component({
  selector: 'app-overview',
  template: `
    <button type="button" (click)="sayMessage()">
        Send messge to console
    </button>
  `,
})
export class OverviewComponent {
    sayMessage() {
        console.log('hello')
    }
}
```

Templates - directives

- Posiadamy wbudowane dyrektywy
- Możemy tworzyć własne dyrektywy
- Najbardziej popularne to: *ngIf i *ngFor

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

@Component({
  selector: 'app-overview',
  template: `
    <div *ngIf="showMessage; else noMessage">
        <p>This should not render.</p>
        <p *ngFor="let message of messages">
            {{ message }}
        </p>
    </div>
    <ng-template #noMessage>
        <p>This should render</p>
    </ng-template>
  `,
})
export class OverviewComponent {
    showMessage = false;
    messages = ['Hello world', 'Hello app']
}
```

Dependency injection

- DI pozwala nam stworzyć klasę Typescript bez dbania o jej instancje.
- Angular zajmie się stworzeniem instancji za nas
- Zyskujemy bardziej testowalny i elastyczny kod

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

@Injectable({providedIn: 'root'})
export class LoggerService {
  log(msg: string) {
    console.log(msg);
  }
}
```

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

@Component({
  selector: 'app-overview',
  template: `
    <button (click)="logToConsole('say hello')">
        Log to console
    </button>
  `,
})
export class OverviewComponent {
    constructor(private logger: LoggerService) { }

    logToConsole(msg: string) {
      this.logger.log(msg);
    }
}
```

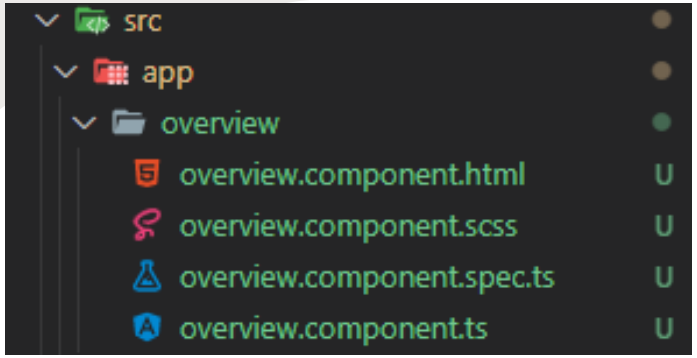
Angular CLI

Angular CLI to najszybszy, prosty i zalecany sposób tworzenia aplikacji Angular.

- ng new – tworzy nowy workspace
- ng build – kompiluje aplikacje do wersji produkcyjnej
- ng serve – uruchamia serwer do developmentu, przy kazdej zmiania jest automatycznie odświeżany
- ng generate – generuje pliki oparte na schematics

Component – jak stworzyć?

- ng generate component overview

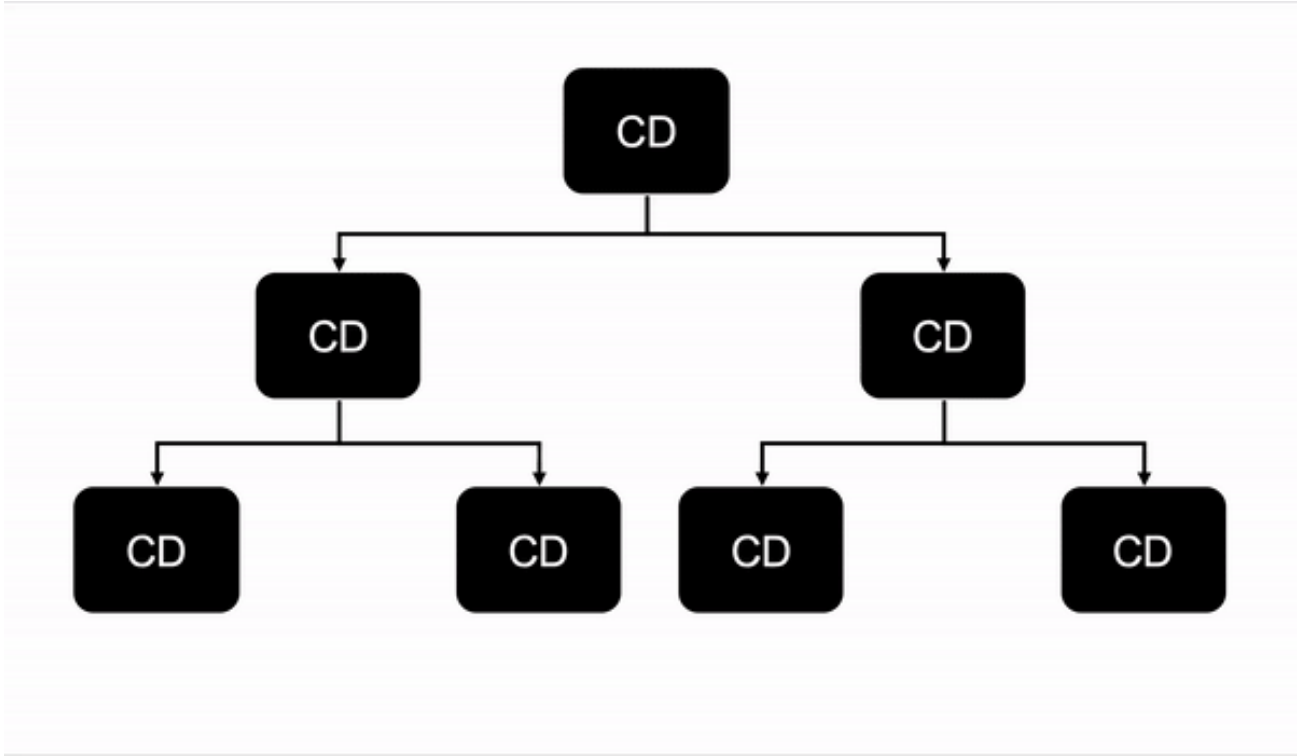


- Najlepszym sposobem jest skorzystanie z Angular CLI
  ng generate component <component-name>
- Defaultowo zostaną wygenerowane 4 pliki
  - <component-name>.component.ts
  - <component-name>.component.html
  - <component-name>.component.css
  - <component-name>.component.spec.ts

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

@Component({
  selector: 'app-overview',
  templateUrl: './overview.component.html',
  styleUrls: ['./overview.component.scss']
})
export class OverviewComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

Component - ChangeDetection

- Wykrywanie zmian to proces, dzięki któremu Angular sprawdza, czy stan aplikacji uległ zmianie i czy jakikolwiek DOM wymaga aktualizacji.
- Najlepiej sobie zobrazowac infografiką



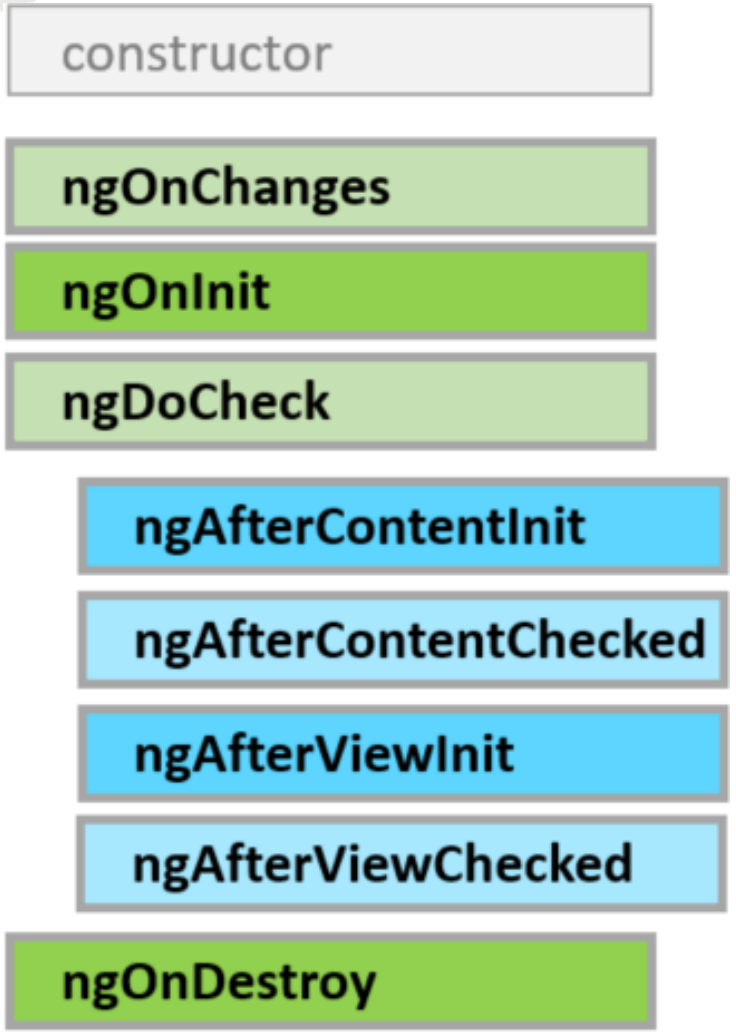
Component – Lifecycle hooks

- Zaczynają się gdy Angular tworzy instancje i renderuje ją. Kończy gdy Angular usuwa instancje i templatke z DOM.
- Dyrektywy też posiadają podobne lifecycle hooks
- Do component możemy dodać jedną lub parę hooks

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

@Component({
  selector: 'app-overview',
  templateUrl: './overview.component.html',
  styleUrls: ['./overview.component.scss']
})
export class OverviewComponent implements OnInit, OnDestroy,
OnChanges {
  @Input() message: string = 'Hello world';
  ngOnChanges(changes: SimpleChanges): void {
    if (changes['message'].isFirstChange())  {
      console.log('first change');
    }
  }
  ngOnInit(): void {
    console.log('init component')
  }
  ngOnDestroy(): void {
    console.log('destroy component')
  }
}
```

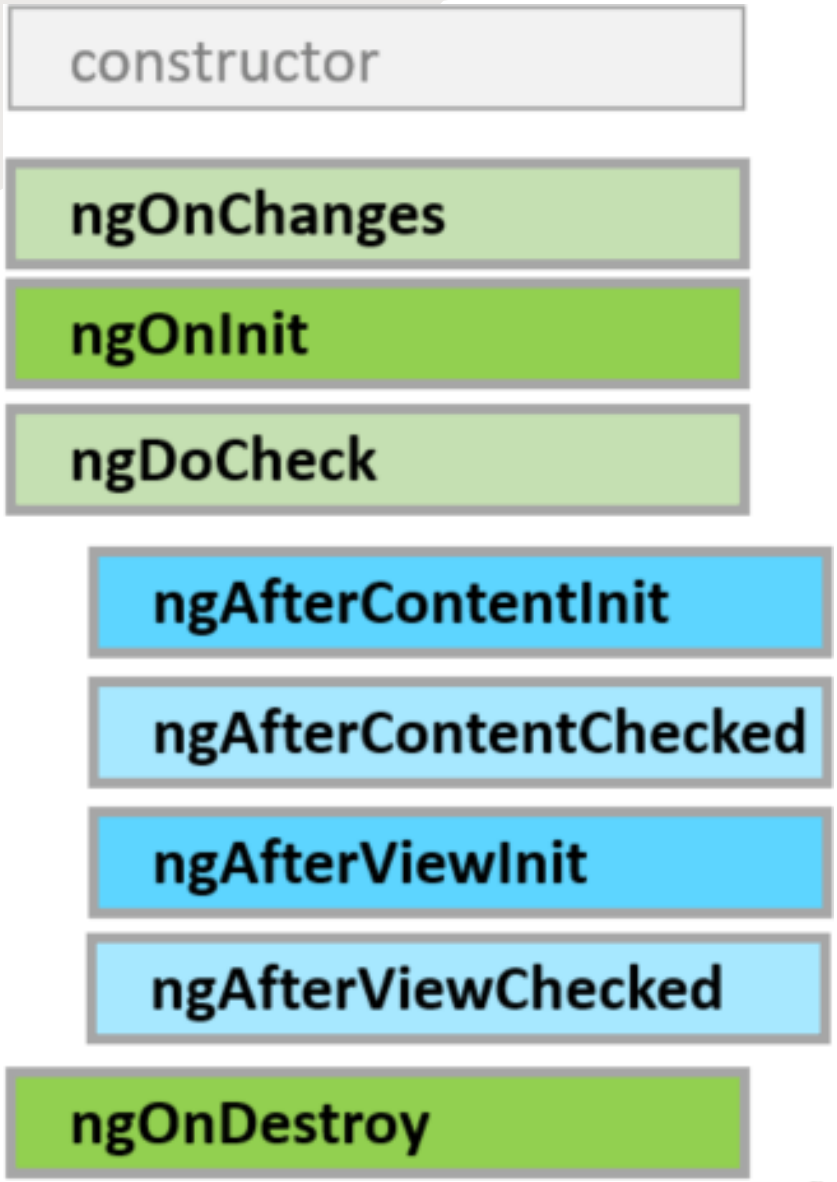
Component – Lifecycle hooks

- OnChanges – wielokrotnie, kiedy Input lub Output się zmienia
- OnInit – po inicjalizacji komponentu
- DoCheck – niestandardowe wykrywania zmian
- AfterContentInit – po inicjalizacji content
- AfterContentChecked – po każdej zmianie content
- AfterViewInit – po zaincjalizowaniu widoku (HTML)
- AfterViewChecked – po kazdej zmianie widoku (HTML)
- OnDestroy – zaraz przed usunięciem



Component – Lifecycle hooks

- ngOnInit()
  - Często wykorzystujemy lifecycle zamiast constructor
  - Gdy potrzebujemy wykonac operacje na @Input()
  - Dobre miejsce do pobrania danych inicjalizujacych
- ngOnDestroy()
  - Służy do sprzątania aby zapobiec memory lead
  - Anulowanie Observable i DOM events
  - Zatrzymanie interval timers
  - Wyrejestrowaniu wszystkich callbacks



Component – View encapsulation

- Style przypisywane są do komponentu i nie wpływają na resztę aplikacji
- Możemy kontrolować style za pomocą encapsulation
  - ShadowDom – używamy wbudowanego Shadow DOM API. Tworzy z komponentu element nadrzędny.
  - Emulated – Przypisujemy style tylko do komponentu
  - None – Dodajemy style globalnie

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

@Component({
  selector: 'app-overview',
  templateUrl: './overview.component.html',
  styleUrls: ['./overview.component.scss'],
  encapsulation: ViewEncapsulation.None
})
export class OverviewComponent {}
```

```
encapsulation: ViewEncapsulation.ShadowDom
```

```
encapsulation: ViewEncapsulation.Emulated
```

Component – Interaction – Input()

- Przekazywanie danych od rodzica do dziecka odbywa sie za pomoca Input binding

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>Hello {{ username }}</h3>
    <p>{{ message }}</p>
  `
})
export class OverviewComponent {
  @Input() message!: string;
  @Input('name') username = '';
}
```

```
<app-overview [message]="'Hello world'" [name]="'Lukas'"></app-overview>
<app-overview message="Hello world" name="Lukas"></app-overview>
```

Component – Interaction – Input()

- Mozemy używać na Inputach getter i setter

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

@Component({
  selector: 'app-overview',
  template: `
    <p>{{ message }}</p>
  `
})
export class OverviewComponent {
  @Input()
  get message(): string { return this._message; }
  set message(message: string) {
    this._message = message.toUpperCase() || 'HELLO WORLD';
  }
  private _message = '';
}
```

```html
<app-overview [message]="'This is my message'"></app-overview>
<app-overview message="This is my message"></app-overview>
```

Component – Interaction – Output()

- Słuchanie zmian dziecka do rodzica obywa się za pomocą @Output() i EventEmitter

```ts
import { Component, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-overview',
  template: `
    <h3>Chose name</h3>
    <button type="button" (click)="emitName('Luk')">Luk</button>
    <button type="button" (click)="emitName('John')">John</button>
  `
})
export class OverviewComponent {
  @Output() setName = new EventEmitter<string>();

  emitName(name: string) {
    this.setName.emit(name);
  }
}
```

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

@Component({
  selector: 'app-overview-parent',
  template: `
    <app-overview (setName)="onSetName($event)"></app-overview>

    <h3>My name is {{ name }}</h3>
  `
})
export class OverviewParentComponent {
  name = '';

  onSetName(name: string) {
    this.name = name;
  }
}
```

Component – Interaction – Local variable

- Słuchanie zmian dziecka do rodzica obywa się za pomocą @Output() i EventEmitter

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>You have: {{ messagesCounter }}</h3>
  `
})
export class OverviewComponent {
  messagesCounter = 0;

  addMessage() {
    this.messagesCounter += 1;
  }

  removeMessage() {
    this.messagesCounter += 1;
  }
}
```

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

@Component({
  selector: 'app-overview-parent',
  template: `
    <app-overview #child></app-overview>

    <button type="button" (click)="child.addMessage()">Add</button>
    <button type="button" (click)="child.removeMessage()">Remove</button>
  `
})
export class OverviewParentComponent {}
```

Component – Interaction – @ViewChild

- Słuchanie zmian dziecka do rodzica obywa się za pomocą @Output() i EventEmitter

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>You have: {{ messagesCounter }}</h3>
  `
})
export class OverviewComponent {
  messagesCounter = 0;

  addMessage() {
    this.messagesCounter += 1;
  }

  removeMessage() {
    this.messagesCounter += 1;
  }
}
```

```ts
import { Component, ViewChild } from '@angular/core';
import { OverviewComponent } from '../overview/overview.component';

@Component({
  selector: 'app-overview-parent',
  template: `
    <app-overview></app-overview>

    <button type="button" (click)="add()">Add</button>
    <button type="button" (click)="remove()">Remove</button>
  `
})
export class OverviewParentComponent {
  @ViewChild(OverviewComponent)
  private overviewComponent!: OverviewComponent;

  add() {
    this.overviewComponent.addMessage();
  }

  remove() {
    this.overviewComponent.removeMessage();
  }
}
```

Component – Interaction – Service

```ts
import { Injectable } from '@angular/core';

@Injectable({providedIn: 'root'})
export class OverviewService {
  messagesCounter = 0;

  addMessage() {
    this.messagesCounter += 1;
  }

  removeMessage() {
    this.messagesCounter += 1;
  }
}
```

```ts
import { Component, OnInit } from '@angular/core';
import { OverviewService } from '../overview.service';

@Component({
  selector: 'app-overview',
  template: `
    <h3>You have: {{ messagesCounter }}</h3>
  `
})
export class OverviewComponent implements OnInit {
  messagesCounter!: number;

  constructor(private overviewService: OverviewService) {}

  ngOnInit(): void {
    this.messagesCounter = this.overviewService.messagesCounter;
  }
}
```

```ts
import { Component } from '@angular/core';
import { OverviewService } from '../overview.service';

@Component({
  selector: 'app-overview-parent',
  template: `
    <app-overview></app-overview>

    <button type="button" (click)="add()">Add</button>
    <button type="button" (click)="remove()">Remove</button>
  `
})
export class OverviewParentComponent {
  constructor(private overviewService: OverviewService) {}

  add() {
    this.overviewService.addMessage();
  }

  remove() {
    this.overviewService.removeMessage();
  }
}
```

Component – Content projection

Single-slot content projection

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>Single-slot content projection</h3>
    <ng-content></ng-content>
  `
})
export class OverviewComponent {}
```

```
<app-overview>
  <p>This should be render in ng-content</p>
</app-overview>
```

Component – Content projection

Multi-slot content projection

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>Multi-slot content projection</h3>
    Name:
    <ng-content select="[name]"></ng-content>
    Message:
    <ng-content></ng-content>
  `
})
export class OverviewComponent {}
```

```
<app-overview>
  <p name>Luk</p>
  <p>This should be render in ng-content</p>
</app-overview>
```

Component – Content projection

Conditional content projection

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

@Component({
  selector: 'app-overview',
  template: `
    <h3>Multi-slot content projection</h3>
    Name:
    <ng-content select="[name]"></ng-content>
    Message:
    <ng-content></ng-content>
  `
})
export class OverviewComponent {}
```

```
<app-overview>
  <p name>Luk</p>
  <p>This should be render in ng-content</p>
</app-overview>
```

Component – Dynamic component

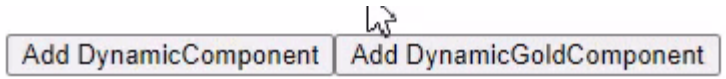
Możemy też dodawać componenty dynamicznie poprzez ViewContainerRef

```
@Component({
  selector: 'app-dynamic-gold',
  template: `
    <span style="color: gold;">This is DynamicGoldComponent, Hello {{ user }}</span>
  `
})
export class DynamicGoldComponent {
  @Input() user!: string;
}
```

```
@ViewChild('container', { read: ViewContainerRef, static: true })
private container!: ViewContainerRef;

renderComponent(comp: string): void {
  this.container.clear();
  const commponentRef = this.container.createComponent(
    comp === 'dynamic' ? DynamicComponent : DynamicGoldComponent
  );
  commponentRef.instance.user = 'Luk';
}
```

```
@Component({
  selector: 'app-dynamic',
  template: `
    <span>This is DynamicComponent, Hello {{ user }}</span>
  `
})
export class DynamicComponent {
  @Input() user!: string;
}
```

```
<button type="button" (click)="renderComponent('dynamic')">Add DynamicComponent</button>
<button type="button" (click)="renderComponent('dynamicGold')">Add DynamicGoldComponent</button>

<p><ng-container #container></ng-container></p>
```



Templates – Binding

Text interpolation

```
message = 'This is my message';
```

```
<p>{{ message }}</p>
```

Property binging

```
fieldId = 'field-id'
```

```
<p [id]="fieldId"></p>
```

Event binding

```
<button type="button" (click)="sendMessage()">Send message</button>
```

```
sendMessage() {
  console.log('sending message...')
}
```

Two-way binding

```
<input type="text" [(ngModel)]="inputValue">
```

```
inputValue = 'some input value';
```

Templates – Attribute binding

Atribute, ARIA

```
<button type="button" [attr.aria-label]="actionName">{{ actionName }} with Aria</button>
```

Colspan

```
<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

Templates – Class binding

Class CSS

```
isWarning = true;
isError = false;
```

```
<p [class.warning]="isWarning">This is my message</p>
<p [class]="'warning'">This is my message</p>
<p [class]="{'warning': isWarning, 'error': isError }">This is my message</p>
<p [class]="['warning', 'error']">This is my message</p>
```

Templates – Style binding

Style CSS

```
colorOrange = 'orange'
```

```
<p [style.color]="'orange'">This is my message</p>
<p [style.color]="colorOrange">This is my message</p>
```

```
<p [style.fontSize]="'2rem'">This is my message</p>
<p [style.fontSize.rem]="2">This is my message</p>
<p [style]="{'fontSize': '2rem', 'color': 'orange'}">This is my message</p>
<p [style]="'fontSize: 2rem; color: orange;'">This is my message</p>
<p [style]="getStyle()">This is my message</p>
```

```
getStyle() {
  return {
    fontSize: '2rem',
    color: 'orange'
  }
}
```

Templates – Event binding

```
<button type="button" (click)="onClick($event)">Click event</button>
```

```
onClick(e: MouseEvent) {
  console.log(e);
}
```

```
▼PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, …}
   isTrusted: true
   altKey: false
   altitudeAngle: 1.5707963267948966
   azimuthAngle: 0
   bubbles: true
   button: 0
   buttons: 0
   cancelBubble: false
   cancelable: true
   clientX: 143
   clientY: 792
   composed: true
```

Templates – Property binding

```
itemImageUrl = 'https://bit.ly/3OmvwVe';
```

```
<img alt="item" [src]="itemImageUrl">
```

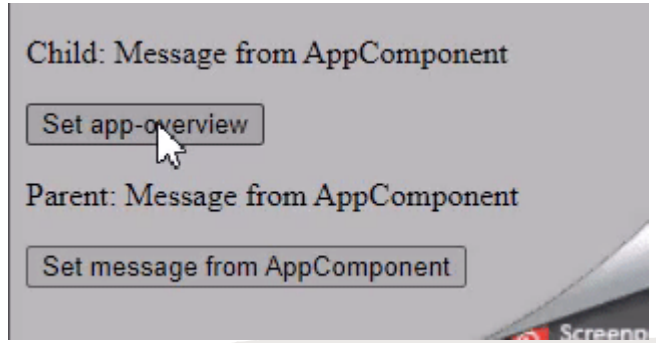
```
isDisabled = true;
```

```
<button type="button" [disabled]="isDisabled">Disabled Button</button>
```

Templates – Two-way binding

```ts
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-overview',
  template: `
    <p>Child: {{ message }}</p>
    <button type="button" (click)="setMessage()">Set app-overview</button>
  `
})
export class OverviewComponent {
  @Input() message!: string;
  @Output() messageChange = new EventEmitter<string>();

  setMessage() {
    this.message = 'Message from app-overview';
    this.messageChange.emit(this.message);
  }
}
```

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

@Component({
  selector: 'app-root',
  template: `
    <app-overview [(message)]="messageApp"></app-overview>
    <p>Parent: {{ messageApp }}</p>
    <button (click)="messageApp = 'Message from AppComponent'">
      Set message from AppComponent
    </button>
  `
})
export class AppComponent {
  messageApp = 'lbd';
}
```



Templates – Pipes

Pipe transformuja wartość do wyświetlenia

Jest to prosta funkcja używana w template która przyjmuje i zwraca przetranformowaną wartość.

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

@Component({
  selector: 'app-overview',
  template: `
    <p>{{ 100 | currency }}</p>
    <p>{{ newDate | date }}</p>
    <p>{{ 'Hi i have very important message' | titlecase }}</p>

    <div *ngFor="let item of { John: 'Hi my name is John', Luk: 'Other Message' } | keyvalue">
      <p>Hello {{ item.key }}, your message: <b>{{ item.value }}</b></p>
    </div>
  `
})
export class OverviewComponent {
  newDate = new Date();
}
```

```
<p>$100.00</p>
<p>Jul 20, 2022</p>
<p>Hi I Have Very Important Message</p>

<div>
  <p>Hello John, your message: <b>Hi my name is John</b></p>
</div>
<div>
  <p>Hello Luk, your message: <b>Other Message</b></p>
</div>
```

Templates – Pipes

Do Pipe możemy przekazać argumenty

```
<p>{{ 100 | currency:'EUR' }}</p>
<p>{{ newDate | date:'MM/dd/yy' }}</p>
```

```
<p>€100.00</p>
<p>07/20/22</p>
```

Możemy też wykorzystywać 2 pipe na jednym inpucie

```
<p>{{ newDate | date | uppercase }}</p>
```

```
<p>JUL 20, 2022</p>
```

Templates – Pipes

Do bardziej skomplikowanych rzeczy możemy stworzyć własne @Pipe.

```
import { Pipe, PipeTransform } from '@angular/core';
import { TranslateService } from './translate.service';

@Pipe({
  name: 'translate'
})
export class TranslatePipe implements PipeTransform {

  constructor(private translateService: TranslateService) {}

  transform(value: string): string {
    return this.translateService.translate(value);
  }
}
```

```
<p>{{ 'Title.App' | translate }}</p>
```

```
<p>Cyberpunk 2077</p>
```

Templates – Pipes

Pipe domyślnie wykonuje zmiane jak tylko zmieni się input value. Niesety wykrywa zmiany dla prymitive value (string, numer) a nie dla object referance (Date, Array, Object). Chyba że tworzymy nowa instacje obiektu (immuntable).

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

@Pipe({
  name: 'join',
  pure: false
})
export class JoinPipe implements PipeTransform {
  transform(arr: string[], separator: string): string {
    return arr.join(separator);
  }
}
```

Angular daje nam możliwość ustawienia pipe na impure, wtedy zmiana odbywa sie na kazde nacisniecie klawisza lub ruch myszy. Dlatego trzeba obchodzic sie z tym ostrożnie

```
strings = ['a', 'b', 'c'];
separator = '-';
```

```
<button type="button" (click)="strings.push('d')">Add element</button>
<button type="button" (click)="separator = '*'">Change separator</button>

<span>{{ strings | join:separator }}</span>
```

Directives – Wbudowane

Klasy które dodają dodatkowe zachowanie do elementów.

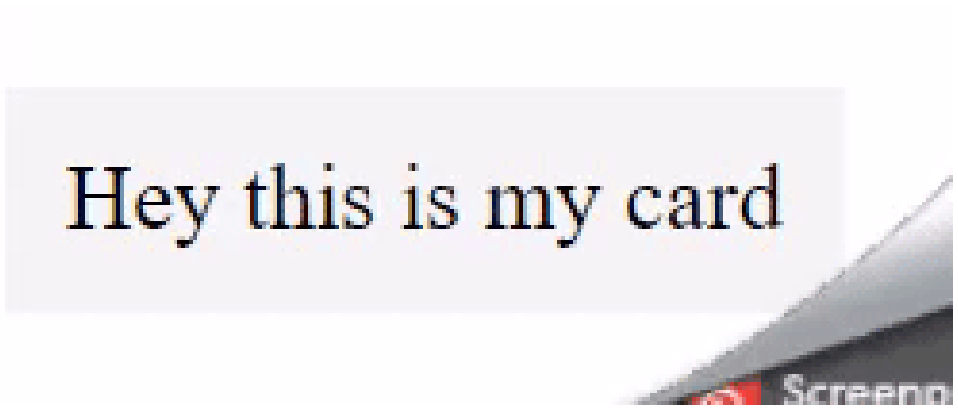
- NgClass – dodaje i usuwa klasy CSS
- NgStyle – dodaje i usuwa style CSS
- NgModel – dodaje two-way data binding
- NgIf – dodaje i usuwa elmenty w komponencie
- NgFor – prezentuje liste w komponencie

```
<p [ngClass]="isWarning ? 'warning' : ''">This is my message</p>
<p [ngStyle]="{'color': isWarning ? 'orange' : 'black'}">This is my message</p>
<input type="text" [(ngModel)]="inputControl">
<p *ngIf="isWarning">Show warning</p>
<p *ngFor="let msg of ['First message', 'Second message']">{{ msg }}</p>
```

Directives – Attribute

Zmienia wygląd lub zachowanie elementów DOM i komponentów Angular

```
<div appCard>
  Hey this is my card
</div>
```




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

@Directive({
  selector: '[appCard]'
})
export class CardDirective {
  @HostBinding('style')
  get styles(): Record<string, string> {
    return {
      padding: '1rem',
      fontSize: '1.5rem'
    }
  }

  @HostBinding('style.backgroundColor') backgroundColor = '#f4f4f4';

  @HostListener('mouseenter') onMouseEnter() {
    this.backgroundColor = '#dbdbdb';
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.backgroundColor = '#f4f4f4';
  }
}
```

Directives – Attribute

Dodajmy drugi parametr

```
<div appCard>
  Hey this is my card
</div>

<div appCard size="large">
  Hey this is my card
</div>
```

Hey this is my card

Hey this is my card

```
import { Directive, HostBinding, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appCard]'
})
export class CardDirective {
  @Input() size: 'normal' | 'large' = 'normal';

  @HostBinding('style')
  get styles(): Record<string, string> {
    return {
      padding: this.size === 'normal' ? '1rem' : '2rem',
      fontSize: this.size === 'normal' ? '1.5rem' : '3rem',
    }
  }

  @HostBinding('style.backgroundColor') backgroundColor = '#f4f4f4';

  @HostListener('mouseenter') onMouseEnter() {
    this.backgroundColor = '#dbdbdb';
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.backgroundColor = '#f4f4f4';
  }
}
```

Directives – Structural

- Dyrektywy strukturalne zmieniają strukturę DOM. Dodają i usuwają elementy.
- Znamy juz wbudowane dyrektywy strukturalne - *ngIf, *ngFor
- Na jednym elemencie możemy dodać jedna dyrektywe strukturalną

Directives – Structural

```ts
import { Injectable } from '@angular/core';


@Injectable({providedIn: 'root'})
export class RightsService {
  private rights: Record<string, boolean> = {
    admin: true,
    editContent: false
  };

  constructor() { }

  hasRight(right: string): boolean {
    return !!this.rights[right];
  }
}
```

```ts
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
import { RightsService } from './rights.service';

@Directive({
  selector: '[appHasRight]'
})
export class HasRightDirective {
  private hasView = false;

  @Input() set appHasRight(right: string) {
    const hasRight = this.rightsService.hasRight(right);
    if (hasRight && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (!hasRight && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }

  constructor(
    private rightsService: RightsService,
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }
}
```

```html
<div *appHasRight="'admin'">
  Show for admin
</div>

<div *appHasRight="'editContent'">
  Editable content
</div>
```

Dependency injection

Jest to wzorzec projektowy, w którym klasa żąda zależności z zewnętrznych źródeł, zamiast je tworzyć.

W Angular takie klasy nazywają się service, aby utworzyć taka klase uruchamiamy z terminala:

ng g service nazwa_service

Po uruchomieniu powinnismy miec wygenerowany pliki:

- nazwa_service.service.ts
- nazwa_service.service.spec.ts

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

@Injectable({
  providedIn: 'root'
})
export class OverviewService {

  constructor() { }
}
```

Dependency injection

Korzystanie z service odbywa sie poprzez wstrzykiwanie do constructora

```
constructor(private overviewService: OverviewService) {}
```

Mozemy korzystac z service w innych service

```
import { Injectable } from '@angular/core';
import { TranslateService } from './translate.service';

@Injectable({
  providedIn: 'root'
})
export class OverviewService {

  constructor(private translateService: TranslateService) { }

  sendMessage(msg: string) {
    this.translateService.translate(msg);
  }
}
```

Dependency injection - providers

Możemy dla każdego service skonfigurować DI token który jest używany w runtime.

Defaultowy token DI jest to nazwa klasy service którą stworzyliśmy

```
providers: [OverviewService]
```

Mozemy to zapisac w bardziej przejrzysty sposób

```
providers: [{ provide: OverviewService, useClass: OverviewService }]
```

- provide – token którym się posługujemy
- Drugi parametr to definicja jak mamy stworzyć obiekt. Zamiast useClass możemy użyć useExisting, useValue i useFactory

Najlepszym sposobem do zobrazowania jak DI w Angular działą jest ta inforgrafika

Dependency injection - InjectionToken

Możemy dla każdego service skonfigurować DI token który jest używany w runtime.

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

export interface AppConfig {
  title: 'Cyberpunk 2077',
  logger: 'warn'
}

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

```
export const OVERVIEW_DI_CONFIG: AppConfig = {
  title: 'Cyberpunk 2077',
  logger: false
}
```

```
providers: [{ provide: APP_CONFIG, useValue: OVERVIEW_DI_CONFIG }]
```

```
title: string;

constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title
}
```