

# ANGULAR

## KOMPONENTY:

Czyli elementy służące do budowania aplikacji.

Na początek stwórzmy sobie folder **interfejsy** a w nim dwa pliki:

ang\_komponenty\_uslugi > src > app > interface > TS Car.ts > ...

```
1 import { Type } from './Type';
2
3 export interface Car{
4     brand:string;
5     model:string;
6     image:string;
7     data:string[];
8     type:Type;
9 }
```

ang\_komponenty\_uslugi > src > app > interface > TS Type.ts > Type

```
1 export enum Type{
2     sport = 1,
3     terrain,
4     camper
5 }
```

Teraz tworzymy tablicę obiektów **cars**:

```
cars: Car[] = [
  {
    brand: "McLaren",
    model: "P1",
    image: "assets/images/mclaren_p1.jpg",
    data: ["903KM", "979 Nm", "2,8 s", "350 km/h"],
    type: Type.sport
  },
  {
    brand: "Ferrari",
    model: "F488 spider",
    image: "assets/images/ferrari_f488_spider.jpg",
    data: ["502KM", "530 Nm", "3,9 s", "315 km/h"],
    type: Type.terrain
  },
  {
    brand: "Mercedes",
    model: "G63",
    image: "assets/images/mercedes_g63.jpg",
    data: ["2500KM", "503 Nm", "5,5 s", "250 km/h"],
    type: Type.camper
  }
]
```

Tworzymy zmienne (i funkcję) użyte na stronie HTML:

```
car: Car = this.cars[0];
kolorb:string = "white";
kolort:string = "black";
aktywna:boolean = true;
showImage:boolean = false;

//funkcja do zmiany kolorów
changeColour() {
  this.kolorb= this.kolorb === "white" ? "black":"white";
  this.kolort= this.kolort === "black" ? "white":"black";
}
```

```
type = Type;
```

Wygląd strony robimy na DIV.

```
<div id="content">
  <div id="banner" [style.background-color]="kolorb" [style.color] = "kolort">
    Wybierz samochód marzeń:
    <select [(ngModel)] = car>
      <option [ngValue] = "null">Wybierz samochód</option>
      <option *ngFor="let item of cars" [ngValue]="item">
        {{item.brand}} {{item.model}}</option>
      </select>
      <button type="button" (click)="changeColour()"
        [class.przycisk]="aktywna"
        [style.color]="kolort">
        Zmień Kolor
      </button>
    </div>
  </div>
```

ngModel → patrz wyjaśnienie w pliku: 14\_angular\_wiazanie\_dwukierunkowe.pdf

[ngModel] = car → powiązanie car z wybranym modelem czyli dopisanie nowego modelu

Zauważ że do stylów odwołujemy się bezpośrednio za pomocą angulara:

**[style.background-color]="kolorb" [style.color] = "kolort"**

gdzie niebieskie napisy oznaczają wartości zapisane w zmiennych w pliku **app.component.ts**.

Strona wygląda jak na rysunku poniżej:

Wybierz samochód marzeń: McLaren P1 Zmień Kolor

☐ Pokaż zdjęcie

McLaren P1

SPORTOWY

DANE:

- 903KM
- 979 Nm
- 2.8 s
- 350 km/h

Czyli mamy baner → z wyborem samochodu

Poniżej mamy div w którym są trzy panele → lewy, środkowy i prawy (w nim nazwa auta i dane)

Zaznaczenie opcji pokaż zdjęcie:

Wybierz samochód marzeń: McLaren P1   
☒ Pokaż zdjęcie

McLaren P1



SPORTOWY

DANE:

- 903KM
- 979 Nm
- 2,8 s
- 350 km/h

Kod odpowiedzialny za pokaż zdjęcie:

```
<label class="check">  
  <input type="checkbox" [(ngModel)]="showImage">Pokaż zdjęcie  
</label>
```

Po zaznaczeniu checkbox automatycznie showImage  
Przyjmuje wartość **true**

Dodatkowo należy umieścić kod odpowiedzialny za wyświetlenie obrazka albo napisu, że brak obrazka:

```
<p class="akapit">  
  {{car.brand}} {{car.model}}  
  <ng-template #noImage>  
    <p class="center">Brak zdjęcia</p>  
  </ng-template>  
</p> <br>  
<img src={{car.image}} [class.zdjecie] = "aktywna" *ngIf="showImage else noImage">
```

Jeśli showImage ma wartość false to  
Pojawi się to

Jak to ma wartość TRUE  
To pojawi się obrazek

Panel prawy to kod:

```
<ng-container [ngSwitch]="car.type">  
  <p *ngSwitchCase="type.sport">SPORTOWY</p>  
  <p *ngSwitchCase="type.terrain">TERENOWY</p>  
  <p *ngSwitchCase="type.camper">KAMPER</p>  
</ng-container>  
<p>DANE:</p>  
<ul>  
  <li *ngFor="let item of car.data">{{item}}</li>  
</ul>
```

car.type →  
jest to  
wartość z  
obiektu car

Czyli już to co mieliśmy wcześniej → plik: 15\_dyrektywy.pdf

# Zadania

1. Wykonaj stronę internetową podobną do powyższej.

## Teraz będziemy tworzyć komponenty.

### 2.5.2. Budowa komponentu i zagnieżdżanie

Każdy komponent ma klasę, plik HTML i plik stylów. Aby wygenerować komponent, należy się posłużyć poleceniem **ng generate component <nazwa\_komponentu>**. Dozwolona jest skrócona forma: **ng g c <nazwa\_komponentu>**.

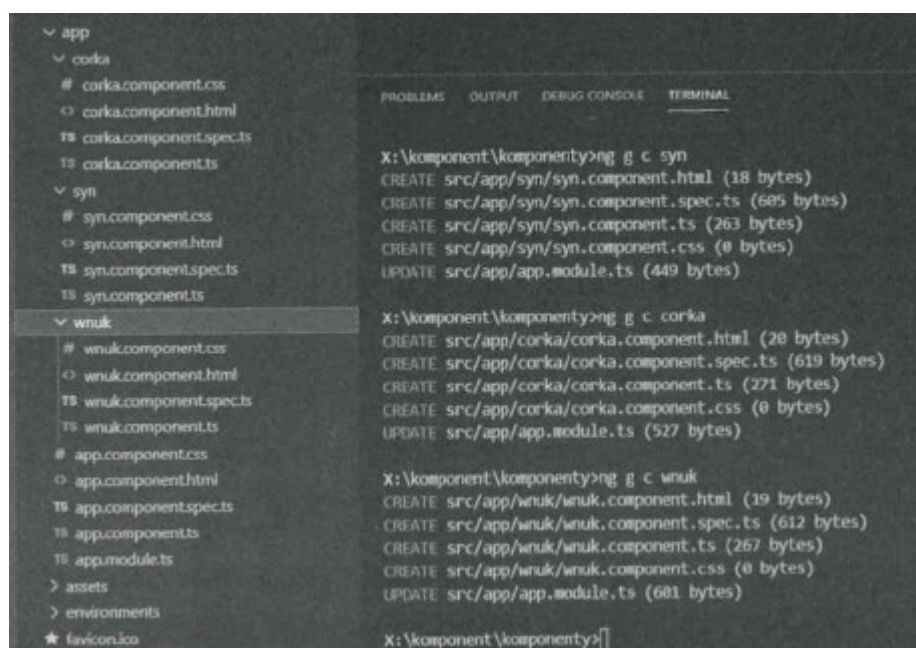
Spróbujmy utworzyć i połączyć ze sobą cztery komponenty o nazwach:

- komponent główny aplikacji (rodzic) — app-root,
  - » komponent syn,
  - » komponent corka,
  - komponent wnuk.

Komponenty o nazwie syn i corka znajdują się na jednym poziomie i są umiejscowione bezpośrednio w komponencie rodzica, natomiast komponent wnuk jest zagnieżdżony w komponencie corka.

Rysunek 2.64 przedstawia proces generowania komponentów. Jak można zaobserwować, dla każdego z nich zostają wygenerowane cztery pliki: *<nazwa\_komponentu>.component.html* to plik szablonu, *<nazwa\_komponentu>.component.css* to arkusz stylów, *<nazwa\_komponentu>.component.ts* to plik TypeScript, w którym znajdują się zmienne oraz metody, a plik *<nazwa\_komponentu>.component.spec.ts* odpowiada za testy.

Generowanie komponentów:



```

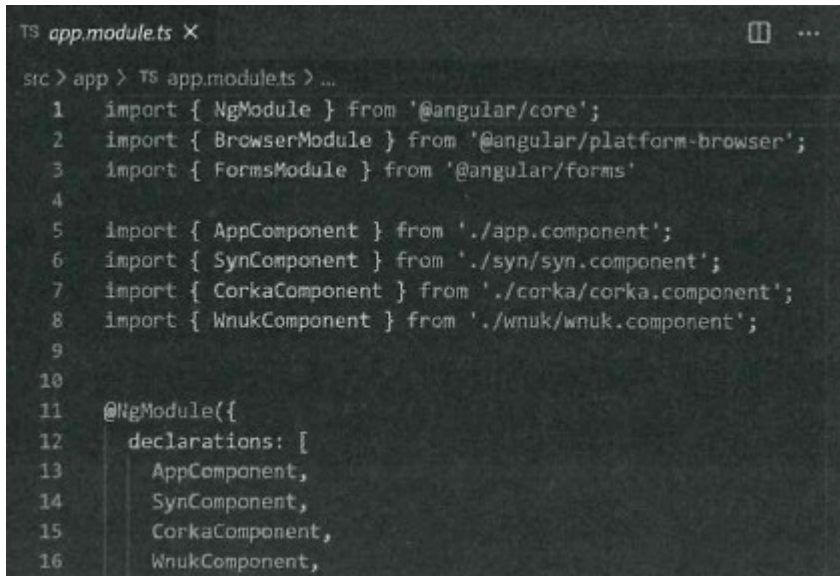
X:\komponent\komponenty>ng g c syn
CREATE src/app/syn/syn.component.html (18 bytes)
CREATE src/app/syn/syn.component.spec.ts (685 bytes)
CREATE src/app/syn/syn.component.ts (263 bytes)
CREATE src/app/syn/syn.component.css (0 bytes)
UPDATE src/app/app.module.ts (449 bytes)

X:\komponent\komponenty>ng g c corka
CREATE src/app/corka/corka.component.html (28 bytes)
CREATE src/app/corka/corka.component.spec.ts (619 bytes)
CREATE src/app/corka/corka.component.ts (271 bytes)
CREATE src/app/corka/corka.component.css (0 bytes)
UPDATE src/app/app.module.ts (527 bytes)

X:\komponent\komponenty>ng g c wnuk
CREATE src/app/wnuk/wnuk.component.html (19 bytes)
CREATE src/app/wnuk/wnuk.component.spec.ts (612 bytes)
CREATE src/app/wnuk/wnuk.component.ts (267 bytes)
CREATE src/app/wnuk/wnuk.component.css (0 bytes)
UPDATE src/app/app.module.ts (681 bytes)

X:\komponent\komponenty>
```

Po wygenerowaniu komponentów można je użyć w naszej aplikacji:



```
TS app.module.ts X
src > app > TS app.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule } from '@angular/forms'
4
5  import { AppComponent } from './app.component';
6  import { SynComponent } from './syn/syn.component';
7  import { CorkaComponent } from './corka/corka.component';
8  import { WnukComponent } from './wnuk/wnuk.component';
9
10
11  @NgModule({
12    declarations: [
13      AppComponent,
14      SynComponent,
15      CorkaComponent,
16      WnukComponent,
```

Stwórzmy własny komponent: plik → **syn.component.ts** znajduje się w folderze **components** trzeba stworzyć ten folder. Dodatkowo trzeba stworzyć w tym samym folderze **syn.component.css**

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

```
@Component({
  selector: 'app-syn',           //użycie w HTML <app-syn></app-syn>
  templateUrl: './syn.html',    //szablon HTML dla synka
  styleUrls: ['./app.component.css']
})
```

```
export class SynComponent implements OnInit{
  @Input() zmiennaInput!: string[]; //Do przesłania danych od RODZICA

  constructor(){}

  ngOnInit():void{
    //Musi to być ponieważ -> implements OnInit
  }
}
```

**syn.html:**

```
<h1>SYNEK TO DZIAŁA</h1>
```

Zaimportujmy go do projektu:



```
//importowanie własnych komponentów uwaga na nazwy!!!  
import {SynComponent} from './components/syn.component';
```

```
@NgModule({  
  declarations: [  
    AppComponent,  
    SynComponent  
  ],  
})
```

Należy jeszcze dopisać zmienną do głównego pliku aplikacji: **app.component.ts**

```
export class AppComponent {  
  
  zmiennaExport:string[] = ["17:00","pływalnia"];
```

Teraz możemy użyć go na naszej stronie:

```
ang_komponenty_uslugi > src > app > <> komponenty.html > <div#content> <div#banner>  
1 <app-syn></app-syn>  
2 <div id="content">  
3   <div id="banner" [style.background
```

Wynik na stronie internetowej:

# SYNEK TO DZIAŁA

Wybierz samochód marzeń:

## *Żądania*

2. Dodaj dodatkowy komponent do strony HTML: córka

Przekazanie danych do syna. Umieszczamy wpis w HTML głównym **nie w syn.html**

```
<app-syn [zmiennaInput]="zmiennaExport"></app-syn>
```

Zmienna z pliku:  
**syn.component.ts**

Zmienna z pliku:  
**app.component.ts**

syn.html:

```
<h1>SYNEK TO DZIAŁA</h1>
```

```
<h4>Teraz można użyć zmiennych tutaj. <br>  
O {{zmiennaInput[0]}} przed {{zmiennaInput[1]}}</h4>
```

Wynik na stronie HTML:

**SYNEK TO DZIAŁA**

Teraz można użyć zmiennych tutaj.  
O 17:00 przed pływalnia

Wybierz samochód marzeń:

# *3 zadania*

3. Stwórz komponent dane osobowe. Przekaż do niego obiekt, który zawiera informację: imię, nazwisko wiek i wyświetl je na stronie HTML.

## Przekazywanie danych z komponentu podrzędnego do nadrzędnego.

Na powiadomienie wysłane od rodzica (komponent nadrzędny) odpowie syn (komponent podrzędny).

Wysyłanie informacji w drugą stronę jest bardziej skomplikowane, a to ze względu na konieczność utworzenia tzw. **event emittera**, który inicjujemy przez zaimportowanie komponentów @Output oraz EventEmitter z @angular/core w pliku **.ts** komponentu podrzędnego. W drugim kroku łączymy z dekoratorem @Output nowy obiekt EventEmitter (obiekt będzie przysyłał dane typu string) o nazwie odpowiedz. Aby dana zmienna/wartość mogła zostać wyeksportowana, musimy użyć metody emit(). Jako jej argument podajemy to, co chcemy wysłać (listing 2.85).

Dopisz tylko to co na niebiesko do pliku **app.component.ts**

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

@Component({
  selector: 'app-syn',
  templateUrl: './syn.component.html',
  styleUrls: ['./syn.component.css']
})
export class SynComponent implements OnInit {
  @Input() zmiennaImport!: string[];

  @Output() odpowiedz = new EventEmitter<string>();

  constructor() { }

  ngOnInit(): void { }

  wyslanie() {
    this.odpowiedz.emit('Będę czekał');
  }
}
```

Umieść w pliku **syn.component.ts** → dane wysłamy po naciśnięciu przycisku

```
<button (click)="wyslanie()" type="button" >Wyślij odpowiedź do rodzica</
```

Do głównego wyglądu programu dopisz (to co na niebiesko):

```
<app-syn [zmiennaImport]="zmiennaEksport" (odpowiedz)="onOdpowiedz($event)"></app-syn>
<app-corka></app-corka>
```

Do pliku **app.component.ts** dopisz (to co na niebiesko)

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'projekt';
  zmiennaEksport: string[] = ['17:00', 'pływalnią'];

  onOdpowiedz(otrzymane: string) {
    alert(otrzymane);
  }
}
```



# *3adania*

4. Na podstawie przykładu samochód marzeń stwórz komponent który będzie wyświetlał szczegóły wybranego samochodu → pamiętaj o przekazaniu odpowiednich danych.

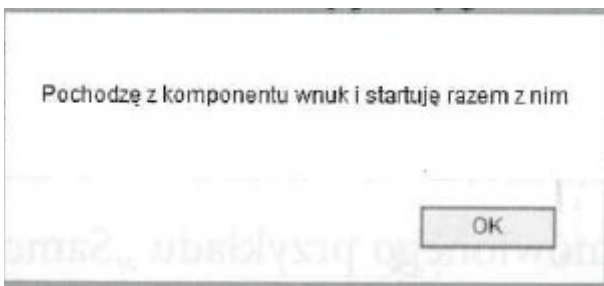
## 2.5.4. Kompozycja i cykl życia komponentu

W nowo utworzonym komponencie znajduje się metoda `ngOnInit()`, która jest wywoływana podczas inicjalizacji komponentu. Aby zweryfikować działanie metody, dodajmy instrukcję, która wywoła metodę `alert`. Instrukcja deklaruje stałą o nazwie `wiadomosc`, której wartość zostaje przekazana do metody `alert` (listing 2.93) i wyświetlona w oknie dialogowym (rysunek 2.72).

↓

```
ngOnInit(): void {  
    const wiadomosc = "Pochodzę z komponentu wnuk i startuję razem z nim";  
    alert(wiadomosc);  
}
```

**alert(wiadomosc):**



# *3adania*

5. Na podstawie przykładu samochód marzeń stwórz komponent który będzie wyświetlał szczegóły wybranego samochodu → pamiętaj o przekazaniu odpowiednich danych.

Tak jak możemy inicjować zasoby, tak **możemy te zasoby zwalniać**.

W pliku *corka.component.ts* dodano komunikat, który zostanie użyty, gdy komponent zostanie odłączony od drzewa dokumentów.

```
ngOnDestroy(): void {  
    console.log('Zostałem zniszczony');  
}
```

Przyłączenie i odłączenie komponentu realizuje dyrektywa `ngIf`. W tym celu w komponencie rodzica w pliku HTML (*app.component.html*) umieszczono kod:

```
<input [(ngModel)]="formularz" />  
<app-corka [name]="formularz" *ngIf="formularz.length>0"></app-corka>
```

Należy również uaktualnić główny plik *app.component.ts* o deklarację zmiennej `formularz`:

```
formularz: string = "";
```

Teraz jak zaczniemy wpisywać dane do formularza pojawi się komponent *corka*, jak skazujemy wszystkie dane komponent *corka* zostanie zniszczony → zobacz komunikat w konsoli

## USŁUGI:

**Usługa** to obiekt zapewniający pewną funkcjonalność. To oznacza, że jeśli mamy stały blok kodu wykonujący określone zadanie, a to jest wywoływane w różnych miejscach, możemy je zdefiniować jako usługę. Dzięki temu kod nie jest powielany.

Aby móc odwołać się do usługi z poziomu innych komponentów, należy jej nazwę umieścić w tabeli *providers* (nazwę usługi poznasz po przejściu do pliku *<nazwa\_usługi>.service.ts*) oraz zaimportować ją za pomocą znanego Ci już polecenia `import`. Wszystkie te czynności przeprowadzamy po otwarciu pliku *app.module.ts* (rysunek 2.76).

Powiązanie usługi z konstruktorem sprawi, że będziemy mogli się do niej odwoływać (listing 2.94).

### *Listing 2.94. Zawartość pliku app.component.ts*

```
import { Component } from '@angular/core';  
import { KalkulatorService } from './kalkulator.service';  
  
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css'],  
    providers: [KalkulatorService]  
})  
export class AppComponent {  
    title = 'projekt';  
    constructor(kalkulator: KalkulatorService) { }  
}
```

Dodanie usługi do *app.modules.ts*

```

ts app.module.ts X
src > app > ts app.module.ts > $ AppModule
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppComponent } from './app.component';
5  import { KalkulatorService } from './kalkulator.service';
6
7  @NgModule({
8    declarations: [
9      AppComponent
10   ],
11   imports: [
12     BrowserModule
13   ],
14   providers: [KalkulatorService],
15   bootstrap: [AppComponent]
16 })
17 export class AppModule {}
18

```

Stwórz plik kalkulator.services.ts i umieść w nim poniższy kod:

**Listing 2.95. Zawartość pliku kalkulator.service.ts**

```

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

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

  constructor() { }

  dodaj(...liczby: number[]): number {
    let wynik = 0;
    for (let wartosci of liczby) {
      wynik += wartosci;
    }
    return wynik;
  }
}

```

**Listing 2.96. Użycie metody — app.component.ts**

```

import { Component } from '@angular/core';
import { KalkulatorService } from './kalkulator.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [KalkulatorService]
})
export class AppComponent {
  title = 'projekt';
  dodawanie: number = 0;
  constructor(kalkulator: KalkulatorService) {
    this.dodawanie = kalkulator.dodaj(3, 6, 7, 9, 2);
  }
}

```

### **Listing 2.97. Wyświetlenie wyniku w szablonie**

```
<h1>
  Suma liczb wynosi {{dodawanie}}
</h1>
```

### **Listing 2.98. Implementacja usługi KalkulatorService w kolejnym komponencie**

```
import { Component, OnInit } from '@angular/core';
import { KalkulatorService } from '../kalkulator.service';
```

```
@Component({
  selector: 'app-syn',
  templateUrl: './syn.component.html',
  styleUrls: ['./syn.component.css']
})
export class SynComponent implements OnInit {

  dodawanie: number = 0;
  constructor(kalkulator: KalkulatorService) {
    this.dodawanie = kalkulator.dodaj(1, 2, 3, 4, 5);
  }

  ngOnInit(): void {
  }
}
```

Oczywiście należy również uaktualnić plik szablonu komponentu (*syn.component.html*).

```
<h1>
  Pochodzę z komponentu syn: Suma liczb wynosi {{dodawanie}}
</h1>
```

## ***3 zadania***

6. Rozszerz funkcjonalność kalkulatora poprzez dodanie do niego kolejnych działań: odejmowanie, mnożenie, dzielenie.