

**ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej**

# Aplikacje webowe na platformę .NET

W13 – Angular - podstawy

# Syllabus

- Aplikacja SPA
- Angular 16-19
  - instalacja Node.js, Angular 19, konfiguracja VS Code
  - CLI:
    - generowanie aplikacji i składowych aplikacji
    - kompilacja i uruchamianie aplikacji
  - Funkcje wybranych plików aplikacji
  - Język TypeScript a JavaScript
  - Interpolacja napisów
  - Dekoratory (wybrane)
  - Serwisy i wstrzykiwanie
  - Komunikacje jednokierunkowe:
    - atrybuty do przesyłania danych
    - emitery zdarzeń
  - Komunikacja dwustronna
  - Sygnały
- Porównanie z wcześniejszymi wersjami Angulara.

## Aplikacja SPA 1/2

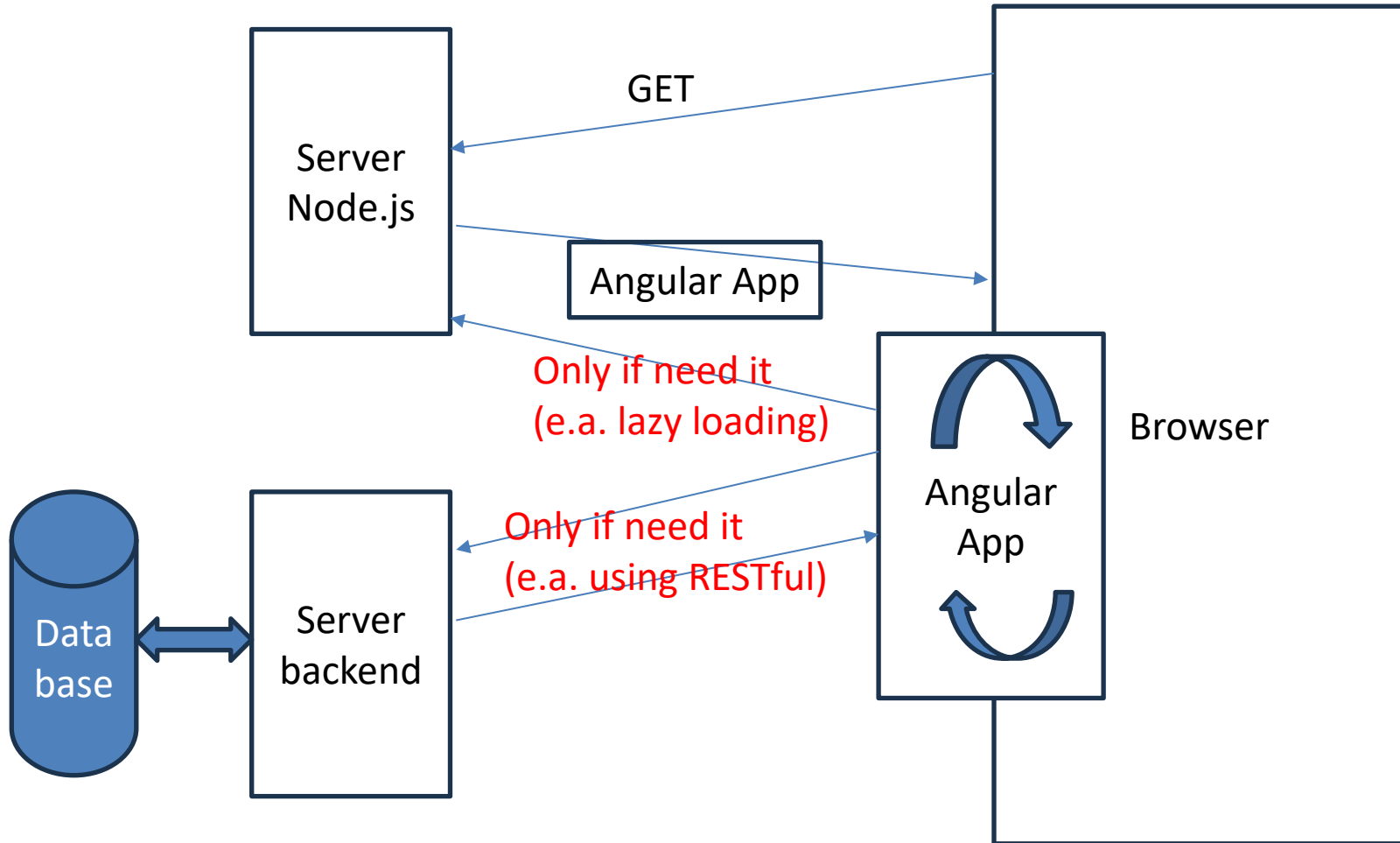
- **Single Page Application (SPA)** to aplikacja webowa, która działa w przeglądarce i dynamicznie aktualizuje treść strony bez konieczności przeładowywania całej strony podczas nawigacji między podstronami. Cała aplikacja jest ładowana raz, a kolejne interakcje użytkownika powodują jedynie wymianę niezbędnych danych z serwerem, co sprawia, że aplikacja działa płynniej i szybciej.
- Kluczowe cechy SPA
  - Brak pełnego przeładowania strony
    - W odróżnieniu od tradycyjnych aplikacji wielostronicowych (MPA – Multi Page Application), SPA nie przeładowuje całej strony przy każdej akcji użytkownika. Zamiast tego, tylko potrzebne części DOM-u są aktualizowane.
  - Dynamiczne załadowanie treści
    - Dane i widoki są ładowane asynchronicznie za pomocą technologii takich jak AJAX lub fetch API, co pozwala na płynne i szybkie aktualizowanie zawartości.
  - Routing po stronie klienta
    - SPA używa routera **po stronie klienta** do zarządzania ścieżkami URL, co pozwala na zmianę adresu w pasku przeglądarki bez przeładowania strony.
  - Lepsze doświadczenie użytkownika (UX)
    - Dzięki szybszej nawigacji i bardziej responsywnemu interfejsowi użytkownika, SPA oferuje wrażenie aplikacji natywnej, gdzie przełączanie między widokami jest płynne.

## Aplikacja SPA 2/2

- Popularne frameworki i biblioteki, które wspierają tworzenie SPA:
  - **Angular**
  - React
  - Vue.js
  - Svelte
- Zalety SPA:
  - Szybsze działanie po początkowym załadowaniu.
  - Płynniejsza nawigacja i lepsze wrażenia użytkownika.
  - Mniejsze obciążenie serwera – tylko dane, a nie całe widoki, są przesyłane przy każdej akcji.
- Wady SPA:
  - Gorsze wsparcie dla SEO (choć można je poprawić poprzez Server-Side Rendering lub Pre-rendering).
    - SEO (Search Engine Optimization) – Optymalizacja dla Wyszukiwarek Internetowych. SEO to zbiór działań mających na celu poprawę widoczności strony internetowej w wynikach wyszukiwania (np. w Google, Bing czy Yahoo). Celem tych działań jest uzyskanie wyższej pozycji w wynikach wyszukiwania dla określonych fraz (słów kluczowych), co prowadzi do zwiększenia liczby odwiedzających stronę.
  - Dłuższy czas początkowego załadowania aplikacji.
  - Potrzeba większej uwagi przy zarządzaniu stanem aplikacji i pamięcią.

# Działanie aplikacji SPA

- Działanie aplikacji typu SPA (np. w Angular-ze) jak poniżej.
  - Wszystkie **podstrony** są **wewnątrz** aplikacji SPA, komunikują się akcjami wewnątrz przeglądarki.
  - Często adresy tych dwóch serwerów różnią się tylko numerem portu.

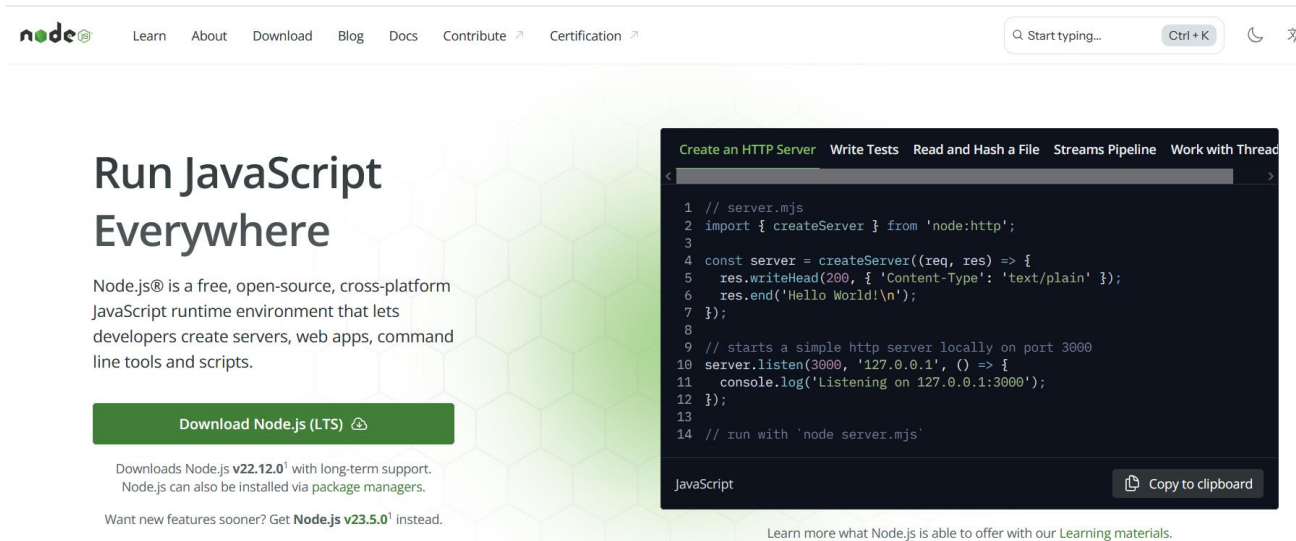


# Angular

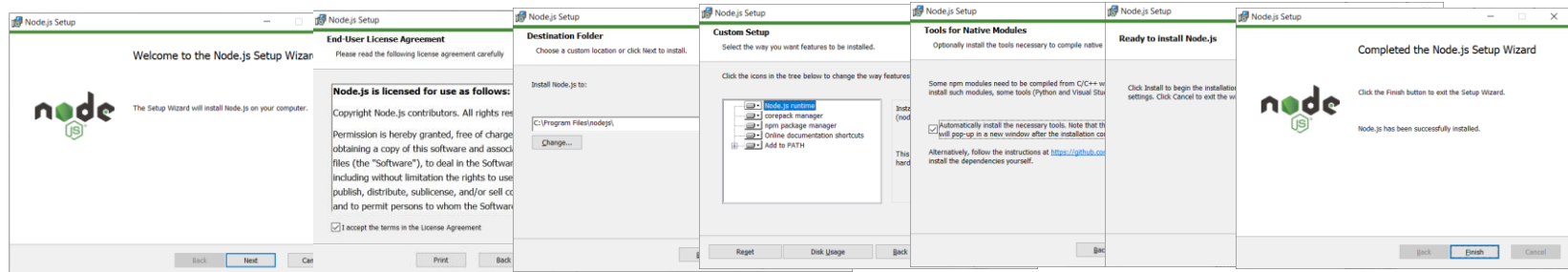
- Framework **front-endowy**
- Oparty na TypeScript-ie (nadjęzyk JavaScript)
  - Dodatkowe notacje związane z określaniem (i pilnowaniem przez kompilator) typu zmiennych itp.
  - Po konwersji powstaje ostatecznie poprawny kod JavaScript, który będzie uruchamiany przez przeglądarkę
    - Użytkownik musi zezwolić na działanie Javascript-u w przeglądarce (co jest obecnie domyślnym ustawieniem)
- Powstaje osobna aplikacja, która pobierana jest z innego serwera niż ewentualny backend (tzn. URL musi się różnić co najmniej numerem portu)
  - Wymaga instalacji serwera Node.js
- Do konwersji kodu na Javascript i uruchamianie serwera z aplikacją frontendową służy CLI, które należy zainstalować.
  - Jednak do poprawnego działania (i instalacji) CLI potrzeba zainstalować Node.js (który w sobie zawiera też narzędzie `npm`)

# Instalacja Node.js

- URL: <https://nodejs.org/en>
- Aktualna wersja: v22.12.0

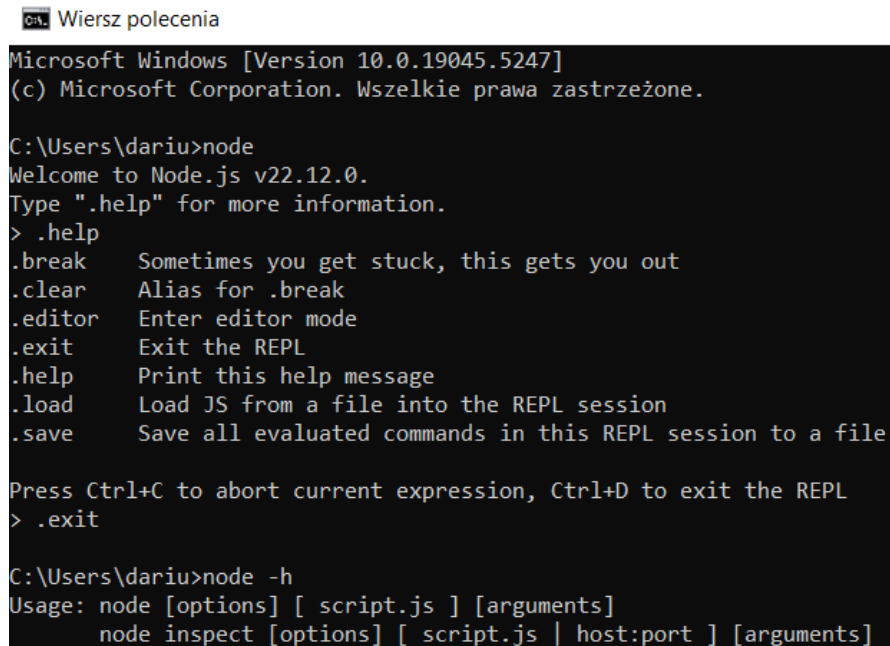


The screenshot shows the Node.js website homepage. At the top is a navigation bar with links: Learn, About, Download, Blog, Docs, Contribute, and Certification. A search bar is on the right. The main heading is "Run JavaScript Everywhere". Below it, a paragraph states: "Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts." A green button says "Download Node.js (LTS)". Below the button, it says "Downloads Node.js v22.12.0<sup>1</sup> with long-term support. Node.js can also be installed via package managers." Another line says "Want new features sooner? Get Node.js v23.5.0<sup>1</sup> instead." On the right, there's a dark-themed code editor showing a simple HTTP server code snippet. Below the code is a "Copy to clipboard" button. At the bottom right, a link says "Learn more what Node.js is able to offer with our Learning materials."



# Sprawdzenie działania Node.js

- Sprawdzenie, czy działa, komenda `node`.
- Zainstalowany został manager pakietów `npm` instalowany z Node.js.
  - Teraz można zainstalować Angular CLI.



```
Ctrl Wiersz polecenia
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\dariu>node
Welcome to Node.js v22.12.0.
Type ".help" for more information.
> .help
.break      Sometimes you get stuck, this gets you out
.clear      Alias for .break
.editor     Enter editor mode
.exit       Exit the REPL
.help       Print this help message
.load       Load JS from a file into the REPL session
.save       Save all evaluated commands in this REPL session to a file

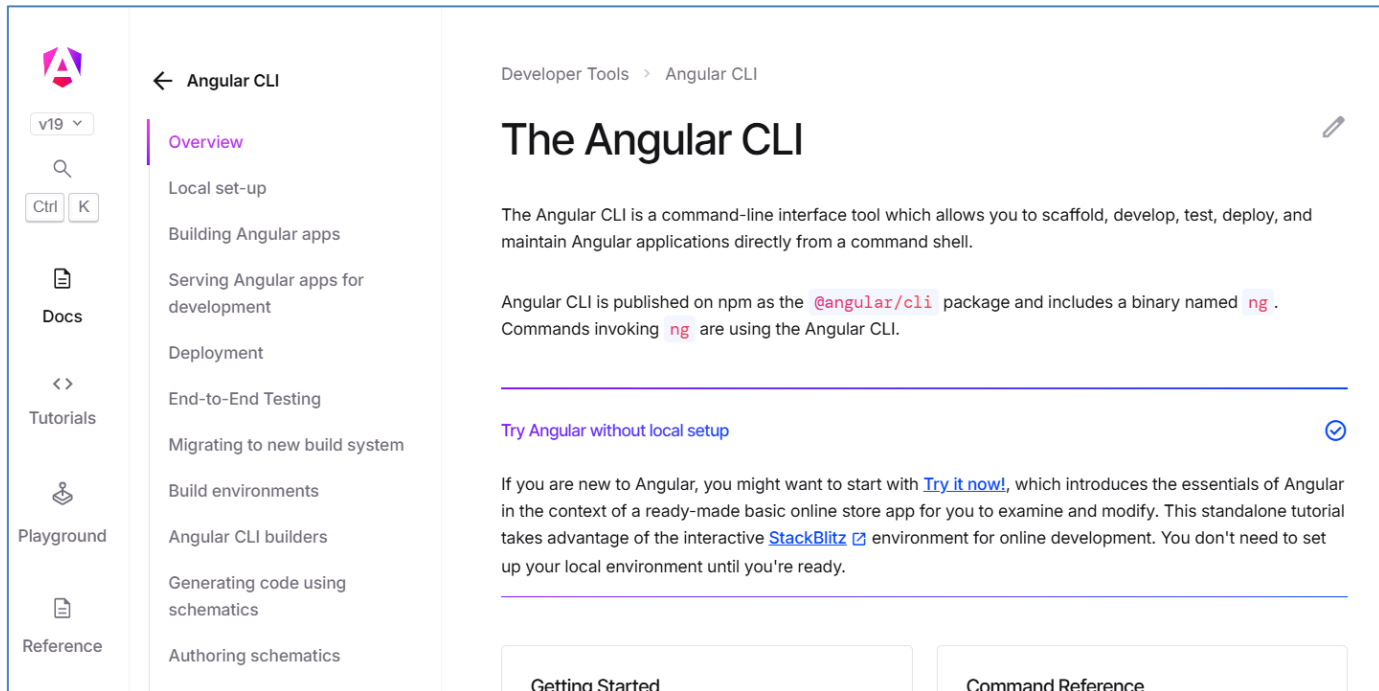
Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
> .exit

C:\Users\dariu>node -h
Usage: node [options] [ script.js ] [arguments]
       node inspect [options] [ script.js | host:port ] [arguments]
```



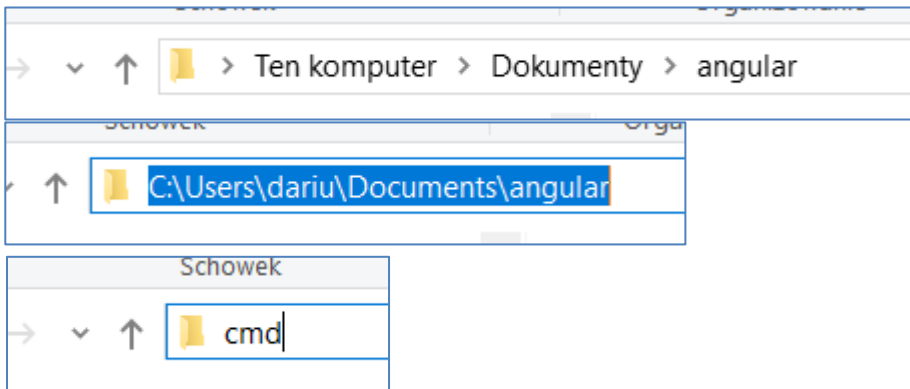
# Angular CLI

- CLI – Command Line Interface
- URL: <https://angular.dev/tools/cli>
- Właściwy URL do strony instalacji: <https://angular.dev/tools/cli/setup-local>
  - W linii komend: `npm install -g @angular/cli`



# Tworzenie nowego projektu

- Ewentualne stworzenie folderu na projekty w Angular-ze.
- Uruchomienie shella (cmd/PowerShell/bash/...) w tym folderze.
- Komenda: `ng new first`
- Wybrać preprocesor dla CSS
- Nie wybierać SSR i SSG
  - SSR to generowanie stron na żądanie **po stronie serwera**, więc w zasadzie to nie SPA.
  - SSG to generowanie stron (statycznych) po stronie serwera podczas kompilacji aplikacji - nie SPA.
- Rozpocznie się proces tworzenia szkieletu aplikacji wraz z instalacją pakietów w folderze `first`.



```
C:\Users\dariu\Documents\angular>ng new first
? Which stylesheet format would you like to use?
  CSS      [ https://developer.mozilla.org/docs/Web/CSS      ]
> Sass (SCSS) [ https://sass-lang.com/documentation/syntax#scss ]
  Sass (Indented) [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less      [ http://lesscss.org ]
```

```
C:\Users\dariu\Documents\angular>ng new first
✓ Which stylesheet format would you like to use? Sass (SCSS) [ https://sass-lang.com/documentation/syntax#scss ]
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N) N
```

```
C:\Users\dariu\Documents\angular>cd first
C:\Users\dariu\Documents\angular\first>dir
Volume in drive C is OS
Volume Serial Number is 24A9-4F73

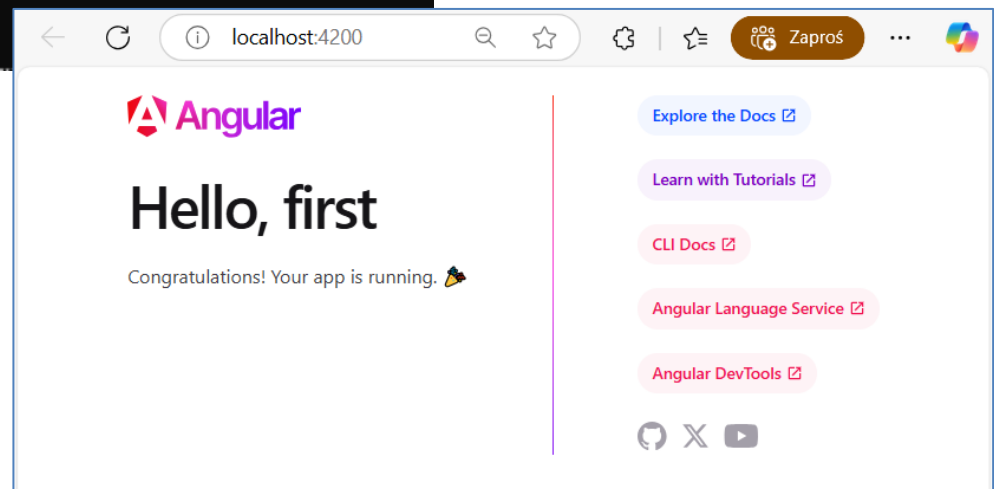
Directory of C:\Users\dariu\Documents\angular\first

03.01.2025  16:07  <DIR>      .
03.01.2025  16:07  <DIR>      ..
03.01.2025  16:06             331 .editorconfig
03.01.2025  16:06             629 .gitignore
03.01.2025  16:06  <DIR>      .vscode
03.01.2025  16:06             2 862 angular.json
03.01.2025  16:07  <DIR>      node_modules
03.01.2025  16:07             531 611 package-lock.json
03.01.2025  16:06             1 074 package.json
03.01.2025  16:06  <DIR>      public
03.01.2025  16:06             1 527 README.md
03.01.2025  16:06  <DIR>      src
03.01.2025  16:06             439 tsconfig.app.json
03.01.2025  16:06             942 tsconfig.json
03.01.2025  16:06             449 tsconfig.spec.json
                                9 File(s)          539 864 bytes
                                6 Dir(s)  424 092 606 464 bytes free
```

# Uruchamianie aplikacji

- Wejście do folderu `first`
- Komenda: `ng serve --open`
- Opcja `--open` (lub `-o`) otwiera aplikację w domyślnej przeglądarce.
- Domyślny (startowy) URL to <http://localhost:4200/>
- W konsoli 'q' lub Ctrl+C przerywa działanie aplikacji.

```
C:\Users\dariu\Documents\angular\first>ng serve --open
Initial chunk files | Names      | Raw size
polyfills.js       | polyfills  | 90.20 kB |
main.js            | main       | 18.18 kB |
styles.css         | styles     | 96 bytes |
                   | Initial total | 108.47 kB
Application bundle generation complete. [1.497 seconds]
Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
Local: http://localhost:4200/
press h + enter to show help
```



## Komendy CLI 1/2

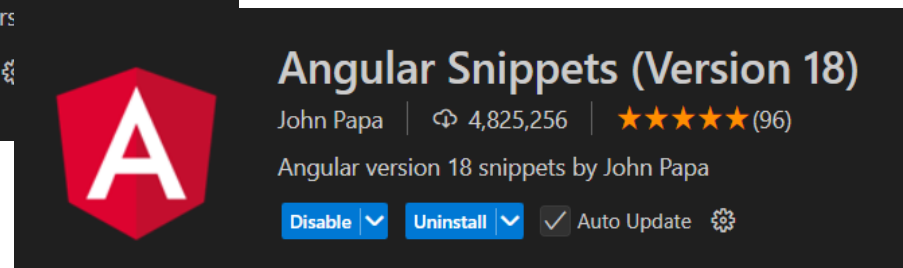
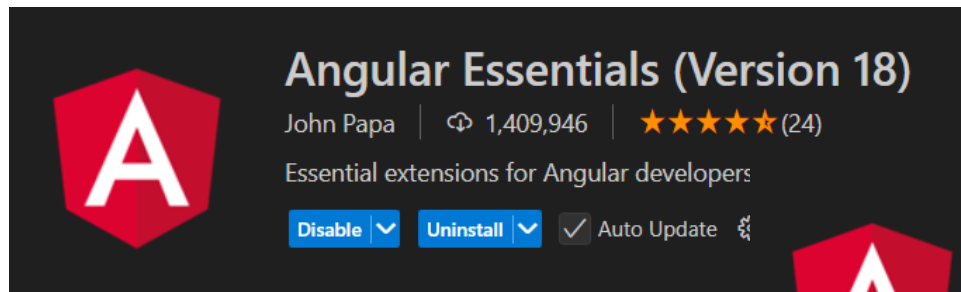
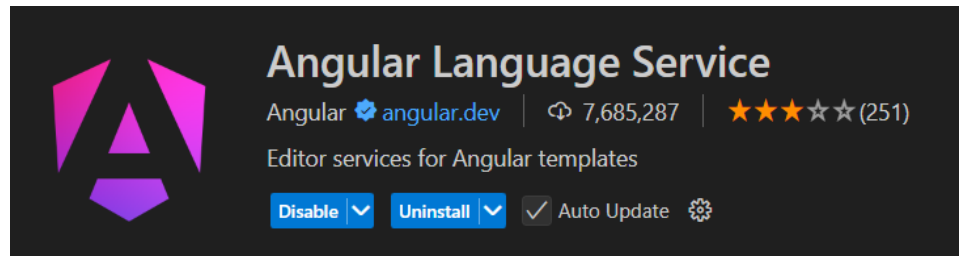
- Wszystkie komendy CLI uruchamiają się poprzez `ng`.
- Ogólny schemat jest:
  - `ng <mainCmd> <subCmd> <subSubCmd> <params>... [ <options> ]`
- Czyli `<mainCmd>` to główna komenda CLI, która wpływa na to jakie podkomendy `<subCmd>`, `<subSubCmd>` itd. mogą być użyte.
- Na końcu (teoretycznie można przed parametrami) następują opcje zaczynające się od podwójnego minusa `--` i nazwa opcji (np. `--open`).
- Część z opcji ma swoją krótką wersję zaczynającą się od pojedynczego minusa `-` (np. `-o`).
- Podobnie część komend i podkomend mają swoje aliasy (krótkie formy).
- **Standardowo** w parametrach podając nazwy (identyfikatory) powinno się używać **kebab-case** (czyli wyrazy łączyć **minusami**).
  - nazwa ta będzie uzupełniona automatycznie o końcówki zależne od tego, jakiego typu jest np. tworzony element, czyli np. dla komponentu nastąpi uzupełnienie o końcówkę `component`.

## Komendy CLI 2/2

- Przykładowe komendy (reszta będzie pojawiać się w ramach potrzeby):
  - `new <workspaceName>` - tworzenie nowego workspace-u Angulara (alias `n`)
  - `serve` – kompilacja i uruchomienie aplikacji (alias `s`).
  - `generate` – tworzenie plików szkieletu nowej składowej aplikacji, bardzo często używane, wiele podkomend (alias `g`):
    - `component` – tworzy 3 pliki dla nowego komponentu (lub 4, jeśli z testami), (alias `c`)
    - `class` – tworzy plik dla klasy
    - `service` – tworzy plik dla serwisu
    - ...
- Wiele opcji zależnych od komendy:
  - dla `generate`:
    - `--skip-tests` – nie tworzyć plików do testowania z rozszerzeniem `.spec.ts`
    - ...
- Przykładowa komenda pełna i z aliasami:
  - `ng generate component student-list --skip-test`
  - `ng g c student-list --skip-test`

# Konfiguracja Visual Studio Code

- Zalecane rozszerzenie do VS Code:
  - Angular Language Service
- Warte rozważenia:
  - Angular Essentials
  - Angular Snippets

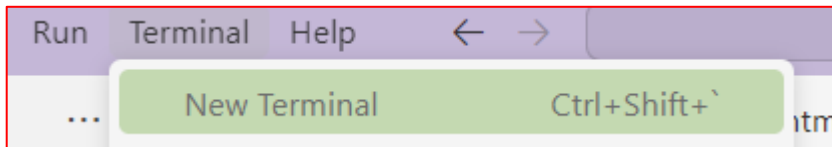


# Uruchamianie aplikacji w terminalu VS Code

- Uruchomienie za pomocą CLI w terminalu:
  - Command prompt (cmd)
  - PowerShell – wymaga wcześniej ustawienia uprawnień:
    - `Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned`
- Klawisz 'q' lub Ctrl+C, lub zamknięcie terminala wyłącza aplikację.
- Uruchomienie aplikacji można wykonać również poprzez `npm start`.

```
PS C:\Users\dariu> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
https://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): Y
```

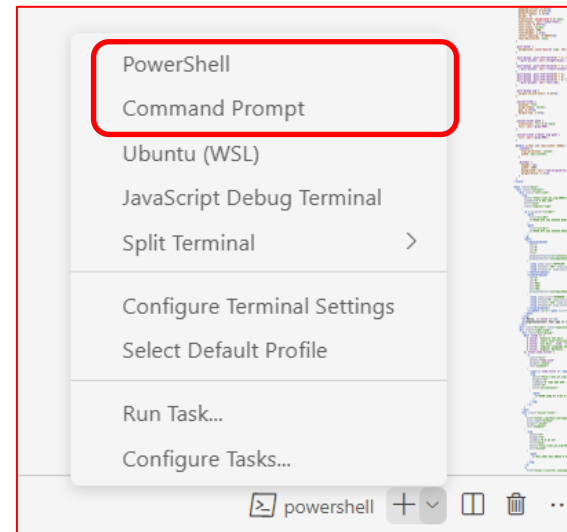


```
PS C:\Users\dariu\Documents\angular\first> ng serve -o
Initial chunk files | Names | Raw size
polyfills.js | polyfills | 90.20 kB |
main.js | main | 18.18 kB |
styles.css | styles | 96 bytes |

| Initial total | 108.47 kB

Application bundle generation complete. [1.352 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
→ Local: http://localhost:4200/
→ press h + enter to show help
```

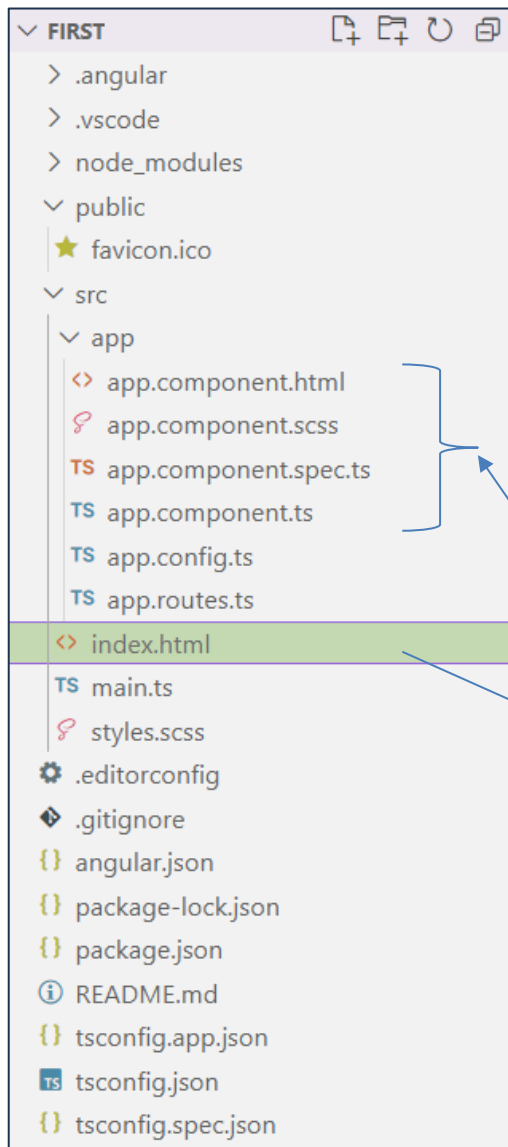


## Nowsze wersje Angulara

- Przedstawione zostaną głównie możliwości nowszej wersji Angulara (v16-v19) – Modern Angular
  - Wiele mechanizmów w nowszych wersjach używa się wygodniej.
    - Wstrzykiwanie można wykonać na wiele sposobów
  - Tworzą bardziej wydajny kod.
  - Kod jest krótszy i łatwiejszy do zrozumienia.
  - Nie używają modułów.
  - Pojawiły się nowe mechanizmy:
    - Sygnały - pozwalają lepiej zarządzać komunikacją między komponentami.
- Mechanizmy, notacje ze starszych wersji (Legacy Angular) nadal działają, na niektórych slajdach przedstawione zostaną przykładowe kody nie korzystające z nowych możliwości.
  - na razie nie powstało zbyt dużo kodów w nowszych wersjach, więc warto wiedzieć jak czytać mniej nowoczesny kod.



# Struktura projektu Angular-a



- Standardowo jako pierwsza strona ładowa się plik `/index.html`.
- Reszta plików jest w większości w folderze `/app`.
- Pierwsza strona zawiera niestandardowy element `<app-root>`
  - Standardowe elementy HTML nie posiadają minusów `-` w identyfikatorze.
  - Jest to **komponent** aplikacji Angulara.
  - Jego definicja znajduje się w plikach zaczynających się od `app.component`:
    - `app/app.component.html`
    - `app/app.component.scss`
    - `app/app.component.ts`
    - `app/app.component.spec.ts`

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>First</title>
6   <base href="/">
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="icon" type="image/x-icon" href="favicon.ico">
9 </head>
10 <body>
11   <app-root></app-root>
12 </body>
13 </html>
```

## Plik `app.component.ts`

- Zawiera kod w TypeScriptie (`.ts`).
- Musi zawierać import-y używanych klas/funkcji itp.
  - Również jeśli są używane tylko w plikach `.html` lub `.scss`
- Jest to w pewnym sensie model komponentu.
  - Wzorcem widoku jest plik z rozszerzeniem `.html`
- Za pomocą dekoratora (adnotacji) `@Component` (importowanego z `@angular/core`) podaje się w obiekcie różne parametry w tym:
  - Selektor.
  - Elementy importowane, które będą użyte **we wzorcu**.
  - Ścieżkę do wzorca (lub sam wzorec za pomocą `template`).
  - Ścieżkę do stylów (lub style wprost za pomocą `styles`).
  - I wiele innych parametrów.

```
1  import { Component } from '@angular/core';
2  import { RouterOutlet } from '@angular/router';
3
4  @Component({
5    selector: 'app-root',
6    imports: [RouterOutlet],
7    templateUrl: './app.component.html',
8    styleUrls: ['./app.component.scss']
9  })
10 export class AppComponent {
11    title = 'first';
12 }
```

Czyli wstaw ten komponent  
w miejsce wystąpienia  
`<app-root>`

Pole `title` z początkową  
wartością `'first'`, z której  
kompilator TypeScript  
wynioskuje typ `string`

Plik `app.component.html` i `index.html` z nową zawartością

- Na początek uprościmy w/w plik do prostszej zawartości jak poniżej.
  - To spowoduje, że w poprzednim pliku powinno się ustawić pustą tablicę dla pola `imports`.

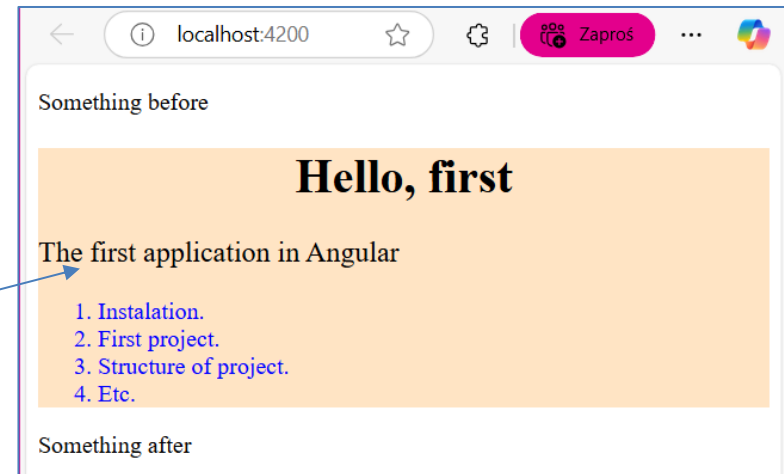
```
1  <main class="main">
2    <div class="content">
3      <h1>Hello, {{ title }}</h1>
4      <p>The first application in Angular</p>
5      <ol>
6        <li> Instalation. </li>
7        <li> First project. </li>
8        <li> Structure of project. </li>
9        <li> Etc. </li>
10     </ol>
11   </div>
12 </main>
```

```
10 <body>
11   <p>Something before</p>
12   <app-root></app-root>
13   <p>Something after</p>
14 </body>
```

## Plik stylu i efekt końcowy

- Plik `app.component.scss` niech zawiera arkusz jak poniżej.
- Wówczas efekt będzie jak po prawej stronie.
  - Style odnoszą się tylko do elementów w ramach wzorca komponentu (np. dla `<p>`)

```
1  .content{
2    background-color: bisque;
3  }
4
5  h1{
6    text-align: center;
7  }
8  li{
9    color: blue;
10 }
11 p{
12   font-size: larger;
13 }
```



# Źródło strony w przeglądarce

- W źródle strony widać, że dodawane są skrypty w JavaScript-ie.
- Jest jednak też element stworzony Angulariem `<app-root>`.

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <script type="module" src="/@vite/client"></script>
5
6   <meta charset="utf-8">
7   <title>First</title>
8   <base href="/">
9   <meta name="viewport" content="width=device-width, initial-scale=1">
10  <link rel="icon" type="image/x-icon" href="favicon.ico">
11  <link rel="stylesheet" href="styles.css"></head>
12 <body>
13   <p>Something before</p>
14   <app-root></app-root>
15   <p>Something after</p>
16  <script src="polyfills.js" type="module"></script><script src="main.js" type="module"></script></body>
17 </html>
18
```

## Uwaga nt. pola standalone w @Component

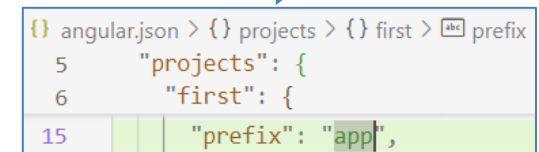
- W wersjach przed Angular 15 komponenty definiowało się w ramach modułów `NgModule`.
- Od wersji 15 do 18 wszystkie komponenty mogą należeć do tzw. komponentów standardowych. Aby to uzyskać należało w obiekcie dla dekorator `@Component` ustawić pole `standalone` na **true** (domyślnie miało wartość **false**).
  - Poniżej po lewej jak wyglądałaby zawartość dla `app.component.ts` takich wersji.
- Od wersji 19 w dekoratorze `@Component` ustawiane jest pole `standalone` **domyślnie** na **true**.
  - Poniżej po prawej.

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    standalone: true,
6    imports: [],
7    templateUrl: './app.component.html',
8    styleUrls: ['./app.component.scss']
9  })
10 export class AppComponent {
11   title = 'first';
12 }
```

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    imports: [],
6    templateUrl: './app.component.html',
7    styleUrls: ['./app.component.scss']
8  })
9  export class AppComponent {
10   title = 'first';
11 }
```

## Pozostałe elementy

- Plik `app.component.spec.ts` służy to testów automatycznych, które nie będą omawiane
  - Plik ten zostanie usunięty
- Plik `angular.json` zawiera konfigurację całej aplikacji. Można w nim znaleźć np. informacje, że:
  - Plik `src/main.ts` jest plikiem startowym.
  - Plik `src/styles.scss` jest plikiem ze stylami globalnymi dla całej aplikacji.
  - itd..
- Pozostałe pliki na razie nie będą zmieniane i zostaną ewentualnie omówione, gdy zajdzie taka potrzeba.
- Chociaż nie jest to wymagane, to warto trzymać się pewnych standardów nazewnictwa plików, klas (np. komponentów) i selektorów:
  - Nazwa klasy powinna zawierać końcówkę oznaczając typ klasy: `Component`, `Service`
  - Selektory w przypadku komponentów powinny mieć stały prefix ze znakiem minus: **app-**, `myapp-`, `univ-`
    - Wtedy na pewno nie nadpiszemy standardowych elementów HTML
  - Nazwa główna pliku (plików) powinna być jak nazwa klasy, ale wyrazy rozdzielone kropką.
  - selektory są pisane z małej litery, wyrazy rozdzielone minusem (kebab-case)
- Przykład:
  - Klasa: `StudentListComponent`
  - Selector: `app-student-list`
  - Plik: `student.list.component.(ts/html/scss)`



The screenshot shows a snippet of the `angular.json` file. A blue arrow points from the text 'app-' in the list above to the 'prefix' value in the JSON. The JSON structure is as follows:

```
{  
  "angular.json": {  
    "projects": {  
      "first": {  
        "prefix": "app",  
      }  
    }  
  }  
}
```

# Własne typy danych w Typescript

- Główna różnica między Typescriptem i Javascriptem to określanie niezmiennego typu danych dla zmiennych, parametrów itd.
- W notacji oznacza to umieszczenie dwukropka za identyfikatorem oraz określenie typu.
- Określenie typu może być bezpośrednie lub za pomocą nazwy typu.
- Bezpośrednie: np.
  - `let result: undefined | "ok" | "error";`
- Stworzyć identyfikator typu i użyć go, np.
  - `type ResultType = undefined | "ok" | "error";`
  - `let result: ResultType.`
- W powyższym przypadku słowo kluczowe `undefined` oznacza, że zmienna (itp.) może mieć nieokreśloną wartość, albo wartość stringu `"ok"`, albo `"error"`. Jakakolwiek inna wartość przypisana do takiej zmiennej będzie powodowała błąd kompilacji
  - ale już niekoniecznie w wykonaniu, bo tam działa JavaScript.
- Ponieważ często może być taka potrzeba użycia np. początkowej wartości nieokreślonej, to istnieje cukier syntaktyczny dla takich przypadków. Używa się znaku zapytania `?` po nazwie zmiennej, jeśli może mieć też wartość nieokreśloną. Zatem powyższe kody można zapisać inaczej (lepiej):
  - `let result?: "ok" | "error";`
  - `type ResultType = "ok" | "error";`
  - `let result?: ResultType.`
  - `let knownResult: ResultType. // cannot be undefined`



# Definiowanie typów

- Definiować typ można jako:

- interfejs
- unię typów
- klasę

```
interface User {  
    id: number;  
    name: string;  
    email: string;  
    isActive: boolean;  
}
```

```
type Status = 'active' | 'inactive' | 'suspended';  
type Ident = string | number;
```

```
class Product {  
    constructor(  
        public id: number,  
        public name: string,  
        public price: number  
    ) {}  
  
    getFormattedPrice(): string {  
        return `$$${this.price.toFixed(2)}`;  
    }  
}
```

# Definiowanie i używanie typów w różnych plikach

- Najlepiej do definiowania typów tworzyć folder (foldery) o nazwach "models", zawierający tylko definicje modeli.
- Z tego powodu, aby je użyć w innych plikach, trzeba użyć słowa kluczowego `export`. Np.
  - `export type Ident = string | number;`
- W plikach, gdzie chcemy używać wyeksportowanych elementów (np. typów) należy je zaimportować za pomocą konstrukcji:
  - `import { <identyfier1>, <identyfier2>, ...} from <PathToFile>`
  - W ścieżce podaje się nazwę pliku BEZ rozszerzenia `.ts`
- Np.:
  - `import { User } from './models/user';`

```
// example.component.ts
import { Component } from '@angular/core';
import { User } from './models/user';
// we assume that the file is in the models folder

@Component({
  selector: 'app-example',
  imports: [User]
  templateUrl: './example.component.html'
})
export class ExampleComponent {
  user: User = {
    id: 1,
    name: 'John Doe',
    email: 'john@example.com',
    isActive: true
  };
}
```

```
// example.component.html
<div>
  {{ user.name }}
</div>
```

## Definicje typów

- Definicja typów Student, Department, Course.

```
src > models > TS department.model.ts > ...  
1   export type Department = "W1" | "W2" | "W3" | "W4N";
```

```
src > models > TS student.model.ts > ...  
1   import { Department } from "../department.model";  
2  
3   export interface Student {  
4       id: number;  
5       index: number;  
6       firstName: string;  
7       lastName: string;  
8       department: Department;  
9       icon: string;  
10  }
```

```
src > models > TS course.model.ts > ...  
1   export class Course{  
2       constructor(  
3           public id:number,  
4           public name: string,  
5       ) { }  
6   }
```

## Serwisy, wstrzykiwanie

- Program uruchamiający, który znajduje się w pliku `src/main.ts` (kod poniżej) działa podobnie jak ASP .Net, czyli przegląda kod, tworzy kontener wstrzykiwalnych elementów, w tym serwisów.
- Takie wstrzykiwalne elementy oczywiście można tworzyć samemu.
- Stwórzmy zatem serwis, w którym będą przechowywane dane, a który będzie wstrzykiwany do innych elementów aplikacji.

```
src > TS main.ts > ...
```

```
1  import { bootstrapApplication } from '@angular/platform-browser';  
2  import { appConfig } from './app/app.config';  
3  import { AppComponent } from './app/app.component';  
4  
5  bootstrapApplication(AppComponent, appConfig)  
6  | .catch((err) => console.error(err));
```

# Generowanie kodu serwisu

- Service najlepiej wygenerować za pomocą CLI (będąc w podfolderze `services`):
  - `ng generate service University --skip-tests`
  - `ng g s University --skip-tests`
- Powstanie plik `university.service.ts`
  - Zawiera definicję klasy `UniversityService`
    - końcówkę `Service` dodaje generator kodu.
- Pojawia się dekorator `@Injectable` (zaimportowany z `@angular/core`), który oznacza, że klasa może być wstrzykiwana do innych elementów aplikacji. Domyślnie klasa przygotowana jest do eksportu (`export`).

```
src > services > TS university.service.ts > ...
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class UniversityService {
7
8    constructor() { }
9  }
```

# Serwis UniversityService

- Kod ze studentami i kursami.
- Pliki z ikonami znajdują się w folderze `src/assets/students`.

```
src > services > TS university.service.ts >  UniversityService
1  import { Injectable } from '@angular/core';
2  import { Student } from '../models/student.model';
3  import { Course } from '../models/course.model';
4
5  @Injectable({
6    providedIn: 'root'
7  })
8  export class UniversityService {
9    students: Student[] =
10    [
11      {id:1, index:12345, firstName:'Dariusz', lastName:'Konieczny',department:'W1',icon:'icon1.png'},
12      {id:2, index:12346, firstName:'Brayan', lastName:'Newmann',department:'W2',icon:'icon2.png'},
13      {id:3, index:22222, firstName:'Loco', lastName:'Maroco',department:'W4N',icon:'icon3.png'},
14      {id:4, index:33333, firstName:'Adam', lastName:'Babacki',department:'W4N',icon:'icon4.png'}
15    ]
16    courses: Course[] = [
17      {id:1,name:"Algebra"},
18      {id:2,name:"Programming"},
19      {id:3,name:"Algorithms"}
20    ]
21
22    constructor() {
23
24    }
```

## Tworzenie komponentu StudentComponent

- Stwórzmy komponent do pokazywania jednego studenta na (w kolejnych krokach) liście studentów. Niech zawiera imię, nazwisko i ikonę.
- Komenda w folderze app:
  - `ng g c student -skip-tests`
- Tworzą się 3 pliki w nowym folderze app/student:
  - Wzorca komponentu: `student.component.html`
    - z linią: `<p>student works!</p>`
  - Styli komponentu: `student.component.scss`
    - pusty
  - Kodu komponentu: `student.component.ts`
    - Z klasą `StudentComponent` z dekoratorem `@Component` z informacjami o w/w plikach oraz ustawiający selektor komponentu na "app-student"

```
src > app > student > TS student.component.ts > ...
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-student',
5    imports: [],
6    templateUrl: './student.component.html',
7    styleUrls: ['./student.component.scss']
8  })
9  export class StudentComponent {
10
11  }
```

## Przygotowanie komponentu, wstrzyknięcie serwisu

- Celem demonstracji wstrzyknijmy do tego komponentu stworzony serwis i ustawmy dane losowego studenta, które w widoku zostaną wyświetlone.
- Poniżej nowszy, preferowany sposób wstrzykiwania od Angular 15.
  - Użycie metody `inject()` z nazwą oczekiwanego serwisu.
- Wyjaśnienie pozostałego kodu na dalszych slajdach.

```
src > app > student > TS student.component.ts > ...
1  import { Component, inject } from '@angular/core';
2  import { UniversityService } from '../../services/university.service';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11   private univService=inject(UniversityService);
12   private len=this.univService.students.length;
13
14   selectedStudent=this.univService.students[Math.floor(Math.random()*this.len)];
15
16   get imagePath(){
17     return 'assets/students/'+this.selectedStudent.icon;
18   }
19 }
```



## Starszy sposób wstrzykiwania serwisu

- Wcześniej, do Angulara 14, wstrzykiwanie było poprzez konstruktor, co bardziej komplikowało kod.
- Nie trzeba dodawać za `selectedStudent` wykrzyknika, bo pole to jest inicjowane w konstruktorze.
  - Ale ponieważ trzeba wprost podać typ, to trzeba go też zaimportować.

```
src > app > student > TS student.component.ts > ...
1  import { Component } from '@angular/core';
2  import { UniversityService } from '../services/university.service';
3  import { Student } from '../models/student.model';
4
5  @Component({
6    selector: 'app-student',
7    imports: [],
8    templateUrl: './student.component.html',
9    styleUrls: ['./student.component.scss']
10 })
11 export class StudentComponent {
12   private len:number;
13   selectedStudent:Student;
14
15   constructor(private univService:UniversityService){
16     this.len=this.univService.students.length;
17     this.selectedStudent=this.univService.students[Math.floor(Math.random()*this.len)];
18   }
19
20   get imagePath(){
21     return 'assets/students/'+this.selectedStudent.icon;
22   }
23 }
```

# Użycie danych we wzorcu komponentu

- We wzorcach komponentów używa się **interpolacji napisów**, pozwalające wykonać dowolny kod (wyrażenie) generujący napis/liczbę.
- Kompilowane wyrażenie umieszcza się między podwójnymi klamrami:
  - `{{expression}}`
- W wyrażeniach mamy dostęp do wszystkich pól/właściwości/metod obiektu modelu komponentu
  - Bez pisania `this`.
- Poniżej możliwe użycie interpolacji i przykładowy wynik.
  - Warto zauważyć, że w przypadku atrybutu `alt` użyto raz cudzysłowy, a wewnątrz apostrofy.
- Poniżej również style przygotowane dla komponentu.

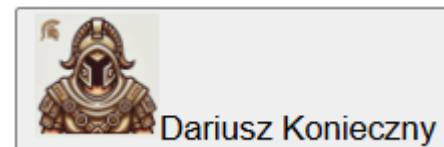
```
src > app > student > <> student.component.html > ...
```

Go to component

```
1 <div>
2   <button>
3     
4     <span>{{selectedStudent.firstName}} {{selectedStudent.lastName}} </span>
5   </button>
6 </div>
```

```
src > app > student > 🔗 student.component.scss >
```

```
1 img{
2   height: 50px;
3   width: auto;
4 }
```



## Skrócony zapis włączenia interpolacji

- Bardzo często w ramach ustawiania **atrybutów** jest to wartość, która w całości jest (lub mogłaby być) wartością interpolowaną. Z tego powodu możliwy jest zapis atrybutu **w nawiasach kwadratowych**, który oznacza, że wartość wyrażona w cudzysłowie jest **w całości** interpolowanym wyrażeniem.
  - Np. `[alt]="selectedStudent.lastName"`
- Nie likwiduje to jednak potrzeby używania cudzysłowy i apostrofów.
  - Z tego powodu (i prostoty kodu wzorca) lepiej przygotowanie takich napisów wykonać w klasie/modelu komponentu.

```
src > app > student > <> student.component.html > ...  
Go to component  
1 <div>  
2   <button>  
3     <img [src]='assets/students/'+this.selectedStudent.icon" [alt]="selectedStudent.lastName">  
4     <span>{{selectedStudent.firstName}} {{selectedStudent.lastName}} </span>  
5   </button>  
6 </div>
```



## Ostateczna wersja wzorca strony

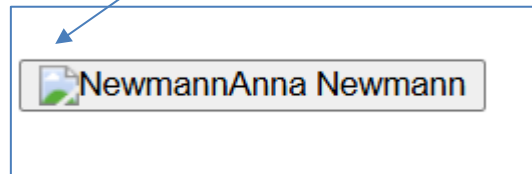
- Ostateczna wersja z wykorzystaniem gettera `imagePath`.

```
src > app > student > TS student.component.ts > ...
1  import { Component } from '@angular/core';
2  import { UniversityService } from '../services/university.service';
3  import { Student } from '../models/student.model';
4
5  @Component({
6    selector: 'app-student',
7    imports: [],
8    templateUrl: './student.component.html',
9    styleUrls: ['./student.component.scss']
10 })
11 export class StudentComponent {
12   private len:number;
13   selectedStudent:Student;
14
15   constructor(private univService:UniversityService){
16     this.len=this.univService.students.length;
17     this.selectedStudent=this.univService.students[Math.floor(Math.random()*this.len)];
18   }
19
20   get imagePath(){
21     return 'assets/students/'+this.selectedStudent.icon;
22   }
23 }
```

```
<div>
  <button>
    <img [src]="imagePath" [alt]="selectedStudent.lastName">
    <span>{{selectedStudent.firstName}} {{selectedStudent.lastName}} </span>
  </button>
</div>
```

# Obrazki - Assets

- Wszelkie pliki (foldery) dodatkowe, oprócz tych tworzących za pomocą kodu aplikację, muszą zostać dodane w pliku `angular.json` w odpowiednim miejscu.
  - inaczej w aplikacji pojawi się placeholder niepoprawnie wczytanego obrazka.

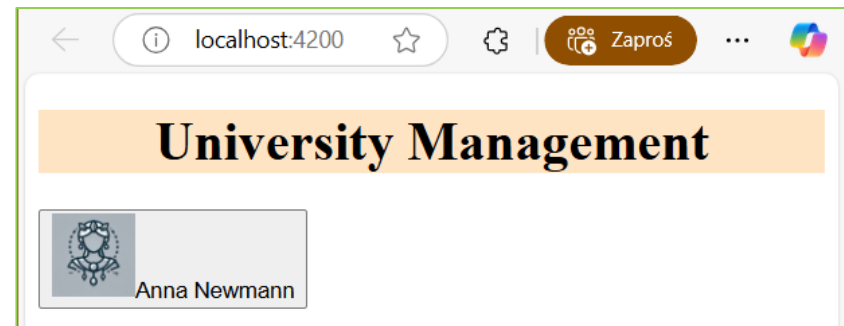
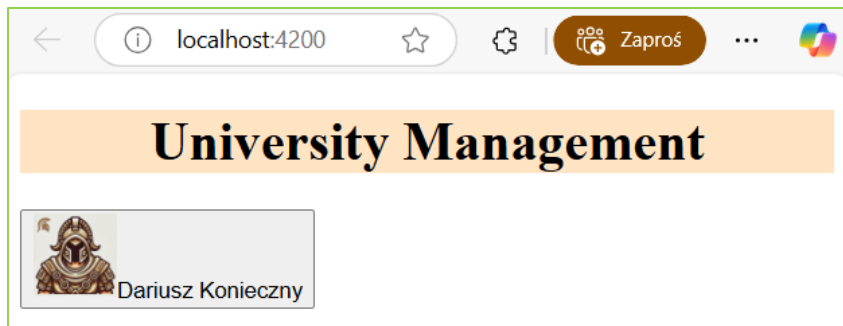
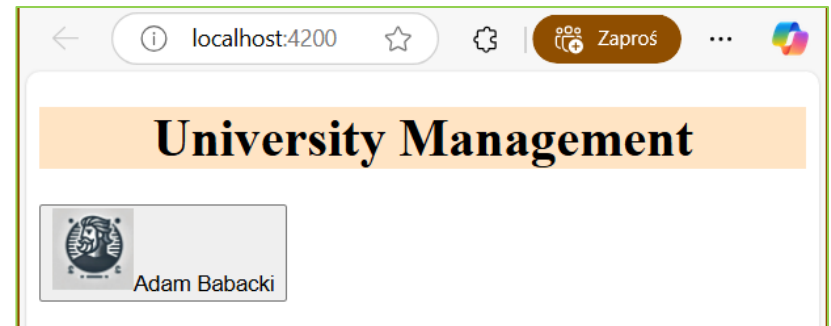
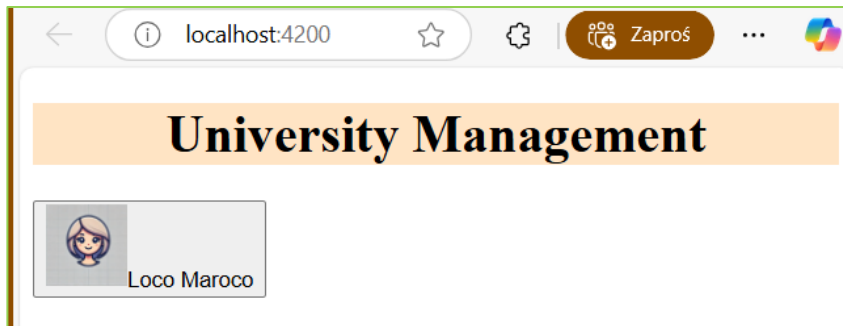


```
angular.json > {} projects > {} first > {} architect > {} build > {} options
5   "projects": {
6     "first": {
16       "architect": {
17         "build": {
19           "options": {
22             "browser": "src/main.ts",
23             "polyfills": [
24               "zone.js"
25             ],
26             "tsConfig": "tsconfig.app.json",
27             "inlineStyleLanguage": "scss",
28             "assets": [
29               {
30                 "glob": "**/*",
31                 "input": "public"
32               }
33             ],
34             "styles": [
```

```
angular.json > {} projects > {} first > {} architect > {} build > {} options >
5   "projects": {
6     "first": {
16       "architect": {
17         "build": {
19           "options": {
22             "browser": "src/main.ts",
23             "polyfills": [
24               "zone.js"
25             ],
26             "tsConfig": "tsconfig.app.json",
27             "inlineStyleLanguage": "scss",
28             "assets": [
29               "src/assets/students",
30               "src/favicon.ico",
31               {
32                 "glob": "**/*",
33                 "input": "public"
34               }
35             ],
36             "styles": [
```

## Efekt działania

- Odświeżenie strony losuje ponownie dane studenta.



# Obsługa zdarzenia

- Niech teraz kliknięcie w ten przycisk pokazuje na konsoli dane studenta, losuje nowego, pokazuje wybrane informacje o tym zdarzeniu.
- Należy dodać obsługę zdarzenia za pomocą wiązania zdarzenia (event binding).
- W Angularze zapisuje się to jako nazwa zdarzenia ujęta w okrągłe nawiasy: (<eventName>), po której następuje znak równości i kod ujęty w cudzysłowy. Jest to najczęściej wywołanie metody (bezparametrowej lub z parametrem o identyfikatorze \$event). Np.
  - (click) = "onClick()"
  - (keypress) = "onModifyString(\$event)"
- Typ parametru \$event zależy od zdarzenia, np. dla (click) będzie to MouseEvent
- Zmodyfikowane wersje plików:

```
17  get imagePath(){
18      |   return 'assets/students/'+this.selectedStudent.icon;
19      |   }
20
21  onSelectStudent(event:MouseEvent) {
22      |   console.log(this.selectedStudent.firstName, this.selectedStudent.lastName);
23      |   console.log(event.screenX,event.screenY);
24      |   this.selectedStudent=this.univService.students[Math.floor(Math.random()*this.len)];
25      |   }
26  }
```

```
src > app > student > <> student.component.html > ...
Go to component
1  <div>
2      <button (click)="onSelectedStudent($event)">
3          <img [src]="imagePath" [alt]="selectedStudent.lastName">
4          <span>{{selectedStudent.firstName}} {{selectedStudent.lastName}} </span>
5      </button>
6  </div>
```

# Efekt działania

- 3 kliknięcia

The screenshots illustrate the state of the 'University Management' application after three clicks. Each screenshot shows a user profile card and a browser console with log messages.

**Screenshot 1 (Top Left):** Shows the 'University Management' title and a user profile card for Dariusz Konieczny. The browser console is open, showing a log message: `Dariusz Konieczny` at `student.component.ts:22` with values `230 217`.

**Screenshot 2 (Top Right):** Shows the 'University Management' title and a user profile card for Adam Babacki. The browser console is open, showing a log message: `Dariusz Konieczny` at `student.component.ts:22` with values `230 217`.

**Screenshot 3 (Middle Left):** Shows the 'University Management' title and a user profile card for Dariusz Konieczny. The browser console is open, showing a log message: `Dariusz Konieczny` at `student.component.ts:22` with values `230 217`.

**Screenshot 4 (Bottom):** Shows the 'University Management' title and a user profile card for Adam Babacki. The browser console is open, showing a log message: `Dariusz Konieczny` at `student.component.ts:22` with values `230 217`.



## Zdarzenia w Angularze

- Mimo, że wydaje się, że zdarzenie kliknięcia przekazywane jest tylko do `StudentComponent`, tak naprawdę (dzięki pakietowi `zone.js`, którego nie używa się wprost, a który wykorzystuje mechanizm detekcji zmian, ang. change detection mechanism) są **informowane wszystkie** obiekty komponentowe stworzone w Angularze, zaczynając od `AppComponent` i przechodząc po **całym** drzewie elementów komponentowych (a nie tylko po ścieżce od korzenia do elementu docelowego, jak to jest w DOM-ie).
  - Wszelkie zdarzenia: ruchu myszką, klawiaturowe, upływu czasu, programisty itd.
  - Bo obsługa zdarzenia mogła zmienić wiele danych w aplikacji i elementy komponentowe muszą zmienić te wartości we wzorcach komponentów (jeśli nastąpiła zmiana wartości).
    - np. na wykresie ma dodać się nowy punkt, jakaś kontrolka w innym miejscu jednocześnie musi zmienić kolor na czerwony itd.
  - Przy bardziej rozbudowanej aplikacji może to powodować niepotrzebne obciążenie aplikacji.
  - Lepiej byłoby, żeby tylko elementy zainteresowany zmianą były informacją
    - czyli wzorzec projektowy ... Obserwator/Słuchacz
- Od wersji Angular 16 powstał **wbudowany** mechanizm sygnałów (ang. **signal**), który umożliwia informowanie tylko tych komponentów (i nie tylko komponentów) zainteresowanych zmianą.
- Metody dla sygnałów są w `@angular/core`.

## Kod z sygnałami

- Wersja po przerobieniu na sygnały.
  - bez informacji wysyłanych do konsoli.
- Ponieważ obecnie `selectedStudent` to sygnał (dokładnie jest klasy `WritableSignal<Student>`), aby odczytać wartość należy go "wywołać" poprzez `()`.
- Natomiast jeśli należy ustawić jego nową wartość, to należy użyć metody `set()` z nową wartością. Wówczas wszystkie elementy "wywołujące" ten sygnał zostaną "poinformowane" o zmianie wartości.

```
10 export class StudentComponent {
11
12     private univService=inject(UniversityService);
13     private len=this.univService.students.length;
14
15     selectedStudent=signal(this.univService.students[Math.floor(Math.random()*this.len)]);
16
17     get imagePath(){
18         return 'assets/students/'+this.selectedStudent().icon;
19     }
20
21     onSelectStudent() {
22         this.selectedStudent.set(this.univService.students[Math.floor(Math.random()*this.len)]);
23     }
24 }
```

```
1 <div>
2   <button (click)="onSelectStudent()">
3     <img [src]="imagePath" [alt]="selectedStudent().lastName">
4     <span>{{selectedStudent().firstName}} {{selectedStudent().lastName}} </span>
5   </button>
6 </div>
```

## Kod z computed()

- Celem lepszego zrozumienia działania zmodyfikowane zostanie działanie getter-a `imagePath` na funkcję `computed()` z `@angular/core`.
- Parametrem tej funkcji musi być bezargumentowa funkcja z kodem zawierającym odczytanie wartości sygnału i zwracającej jakiś wynik.
  - właściwość `imagePath` staje się też sygnałem (typu `Signal<string>`) i jej wartość też należy odczytać poprzez wywołanie `()`.

```
15  selectedStudent=signal(this.univService.students[Math.floor(Math.random()*this.len)]);
16
17  imagePath = computed(()=>'assets/students/'+this.selectedStudent().icon)
18
19  onSelectStudent() {
20    |  this.selectedStudent.set(this.univService.students[Math.floor(Math.random()*this.len)]);
21  }
```

```
1  <div>
2    <button (click)="onSelectStudent()">
3      <img [src]="imagePath()" [alt]="selectedStudent().lastName">
4      <span>{{selectedStudent().firstName}} {{selectedStudent().lastName}} </span>
5    </button>
6  </div>
```

## Jak działają sygnały

- Sygnał reprezentuje komórkę danych, która może się zmieniać w czasie. Sygnały mogą być albo „stanem” (po prostu wartością, która jest ustawiana ręcznie), albo „obliczane” (w przypadku formuły opartej na innych sygnałach w metodzie `compute()`).
- Sygnały implementują wzorzec obserwatora, w którym subskrybują/rejestrują się metody wymagające wywołania przy zmianie tej wartości (poprzez wywołanie nawiasami otwartymi `()`).
- Subskrypcja wykonywana jest często przez kompilator Angulara.
- Poprzez metodę `compute(() => <functionBody>)` wskazujemy bezargumentową metodę, która ma być wykonana, jeśli w `<functionBody>` odczytywany jest zmieniony sygnał.
- kompilator analizuje, które sygnały są odczytywane i subskrybuje do nich tą funkcję.
- wynikiem jest też sygnał, więc też go trzeba wywoływać celem odczytania wartości.
- Sygnał ten też przechowuje swoją wartość i jeśli ta się zmieniła, informuje metody zasubskrybowane do niego.
- Są dwa podstawowe typy sygnałów:
  - `Signal<T>` - tylko do odczytu
  - `WritableSignal<T>` - do odczytu i zapisu

## Metody typu `WriteableSignal<T>`

- `()` - odczytanie wartości przechowywanej w kontenerze (typu `T`)
- `set(value: T)` - wstawienie nowej wartości
- `update(updater: (value: T) => T)` - ustawienie funkcji `updater` do uaktualniania wartości na podstawie poprzedniej wartości. Metoda `updater` ma zwracać tę nową wartość.
- `mutate(mutator: (value: T) => void)` - umożliwia bezpośrednią modyfikację obiektu (używane dla obiektów złożonych). Metoda `mutator` modyfikuje wewnętrznie parametr typu `T`, nie modyfikując samej referencji.
- Gdy wywoływana jest dowolna z metod modyfikujących, po jej skończeniu uruchamiane są wszystkie zasubskrybowane metody, aby odczytały sobie nową wartość sygnału.
  - Co może powodować kaskadowo/drzewiaście wywołanie kolejnych metod jeśli te pierwsze zasubskrybowane są też sygnałami.
  - Nie powinno być cykli w przypadku takich zależności.
- Dzięki temu mechanizmowi uruchamiane są tylko te metody w tych komponentach, które rzeczywiście potrzebują ponownego wykonania

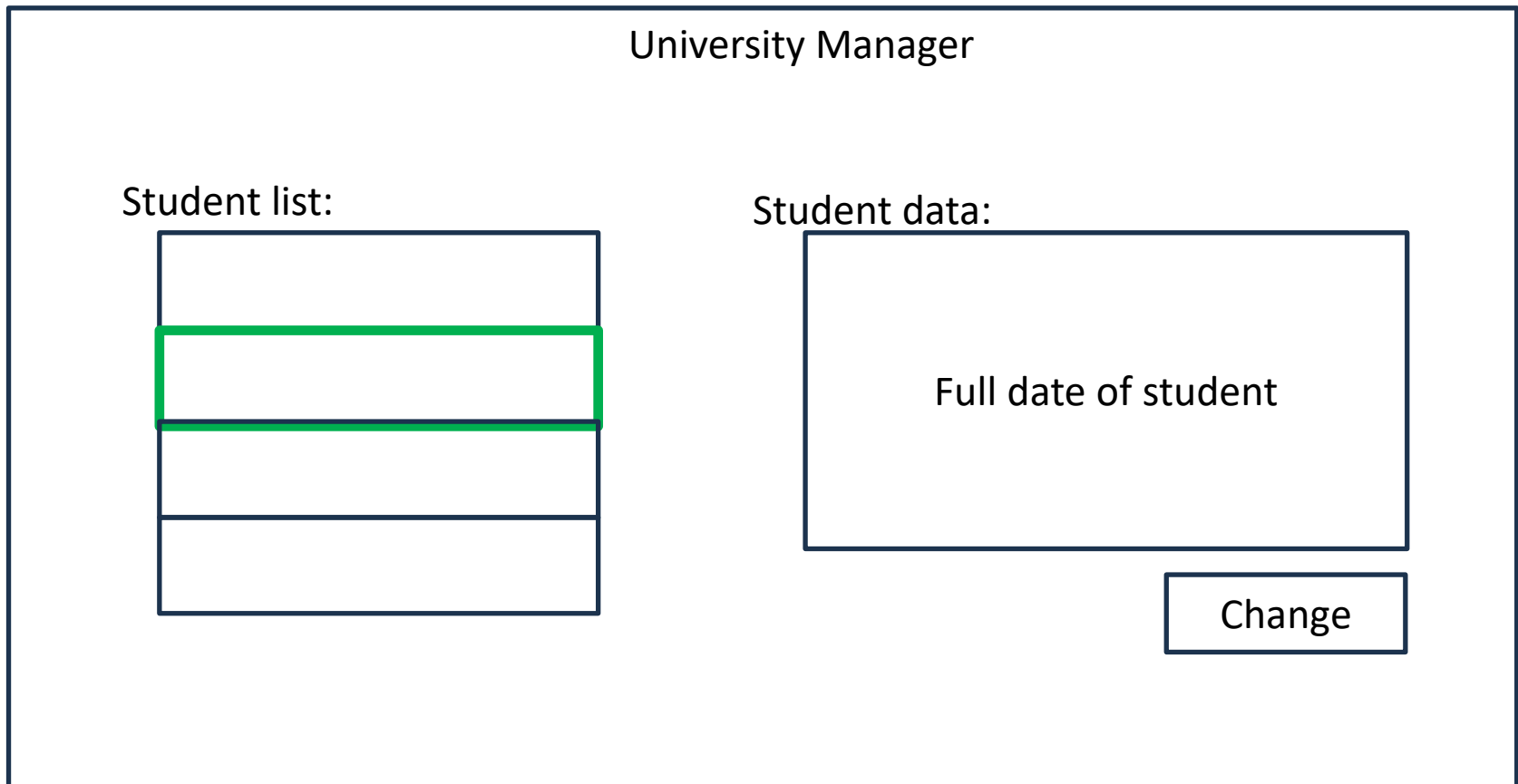
## Odniesienie do przykładu

- We wcześniejszym prostym przykładzie do sygnału `selectedStudent` tworzy się wiele małych funkcji `computed()` związanych z interpolacją w widoku komponentu (zielone ramki).
- Oprócz tego w interpolacji wywoływany jest kod dla wyliczania `src` odczytujący wartość sygnału `imagePath`, który też będzie pewną małą funkcją `compute()` zasubskrybowaną do tego sygnału.
- Po zmianie wartość w kontenerze `selectedStudent` spowodowanej kliknięciem, podczas metody `set()` uruchomione zostaną te w/w funkcje z zielonych ramek oraz funkcja `computed()` dla wyliczania wartości w kontenerze `imagePath` w obiekcie `StudentComponent`.
- Wykonanie tego `computed()` spowoduje zmianę jej wartości, i o tym fakcie zostanie poinformowana funkcja `computed()` wygenerowana na kodu w zielonej ramce.

```
1 <div>
2   <button (click)="onSelectedStudent()">
3     <img [src]="imagePath()" [alt]="selectedStudent().lastName">
4     <span>{{selectedStudent().firstName}} {{selectedStudent().lastName}} </span>
5   </button>
6 </div>
```

# Założenia aplikacji

- W aplikacji ma być lista studentów.
- Wybranie (kliknięciem) studenta wpisuje jego bieżące dane po prawej stronie.
- Dane te można zmienić po wciśnięciu przycisku "Change".
- Otwiera się wtedy okienko modalne do zmiany danych.



## Zmiany oczywiste

- Należy wytworzyć kilka nowych komponentów, na początek jeden do obsługi listy studentów `<app-student-list>`
  - w folderze `src/app` : `ng g c student-list --skip-tests`
  - Powinien zawierać html-ową listę elementów `<app-student>`
  - Na początku w liście będzie tylko jeden element.
- Do modelu tego nowego komponentu zostanie wstrzyknięty serwis do obsługi uniwersytetu.
- Z komponenta studenta usunięte zostanie wstrzykiwanie i losowanie studenta. Zamiast właściwości `selectedStudent` będzie właściwość po prostu `student` typu `Student`.
- Komponent listy studentów dodany zostanie do komponentu głównego aplikacji.



## Stan plików 1/2

```
src > app > TS app.component.ts > ...
1  import { Component } from '@angular/core';
2  import { StudentListComponent } from "../student-list/student-list.component";
3
4  @Component({
5    selector: 'app-root',
6    imports: [StudentListComponent],
7    templateUrl: './app.component.html',
8    styleUrls: ['./app.component.scss']
9  })
10 export class AppComponent {
11   title = "University Management";
12 }
```

```
src > app > <> app.component.html > ...
Go to component
1  <main class="main">
2    <div class="content">
3      <h1>{{ title }}</h1>
4    </div>
5    <app-student-list></app-student-list>
6  </main>
```

- użycie we wzorcu komponentu `<app-student-list>` innego komponentu `<app-student>` wymusza w modelu zaimportowanie `StudentComponent` i dodanie go do tablicy pola `imports` w dekoratorze `@Component`
  - co zaproponuje VS Code z odpowiednią wtyczką.

```
src > app > student-list > TS student-list.component.ts > ...
1  import { Component, inject } from '@angular/core';
2  import { UniversityService } from '../services/university.service';
3  import { StudentComponent } from "../student/student.component";
4
5  @Component({
6    selector: 'app-student-list',
7    imports: [StudentComponent],
8    templateUrl: './student-list.component.html',
9    styleUrls: ['./student-list.component.scss']
10 })
11 export class StudentListComponent {
12   public univService=inject(UniversityService);
13 }
```

```
src > app > student-list > <> student-list.component.html
Go to component
1  <ul>
2    <li>
3      <app-student></app-student>
4    </li>
5  </ul>
```

## Stan plików 2/2

- Musimy używać znaków '?', bo pole `student` może mieć wartość `undefined`.
- Aplikacja działa, ale oczywiście pole `student` ma obecnie wartość `undefined`.
  - i w efekcie mamy dziwną grafikę w interfejsie.

```
src > app > student > TS student.component.ts > ...
1  import { Component } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11   student?: Student;
12
13   imagePath = () => 'assets/students/' + this.student?.icon;
14
15   onSelectStudent() {
16   }
17 }
```

```
src > app > student > <> student.component.html > ...
Go to component
1  <div>
2    <button (click)="onSelectedStudent()">
3      <img [src]="imagePath()" [alt]="student?.lastName">
4      <span>{{student?.firstName}} {{student?.lastName}} </span>
5    </button>
6  </div>
```

### University Management

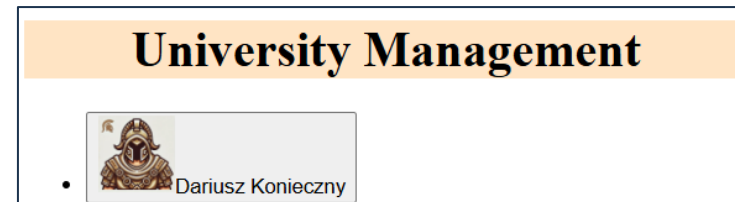
- 

## Przekazywanie danych do komponentu - sposób 1, starszy

- Do komponentu można przekazać dowolnego typu dane poprzez notację, jakby to był **atrybut elementu HTML**.
- Tworzenie nowych atrybutów w `<app-student>` dokonuje się poprzez użycie dekoratora `@Input` (zaimportowane z `@angular/core`) do wybranego pola modelu komponentu
  - Może być wiele nowych, tak stworzonych, atrybutów dla danego komponentu.
  - W tym przypadku będzie to atrybut `student` i będzie on typu `Student`.
- Po dodaniu atrybutu należy we wzorcu listy studentów użyć go.

```
src > app > student > TS student.component.ts > ...
1  import { Component, Input } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11    @Input() student?: Student;
12
13    imagePath = () => 'assets/students/' + this.student?.icon;
14
15    onSelectedStudent() {
16    }
17 }
```

```
src > app > student-list > <> student-list.component.html > ...
Go to component
1  <ul>
2    <li>
3      <app-student [student]="univService.students[0]" >/app-student>
4    </li>
5  </ul>
```



## Sposób 1a i 1b - opis

- Jest to też sposób starszy do wersji Angular 15 (ale nadal działa).
- Zapis pola w postaci:  
`@Input() student: Student;`  
jest niedozwolony, gdyż pole to nie jest nigdzie w kodzie ustawiane na konkretną wartość (mogłoby być tak zrobione w konstruktorze, ale tu nie ma to sensu).
- Trzeba dodać końcówkę `,?` aby definicja była poprawna. Ten znak staje się w pewnym sensie częścią identyfikatora, stąd pojawia się również w kodzie dla metody `imagePath()`.
- Jeśli jesteśmy pewni, że atrybut zawsze będzie podany można zamiast `?` użyć wykrzyknika `!` informując kompilator, że ten atrybut będzie podany.
  - ale możemy jednak **zapomnieć o tym wymogu** w trakcie pisania kodu
- Aby **wymusić obecność atrybutu** w komponencie należy w dekoratorze `@Input` wstawić obiekt z polem `required` równych **true**.
  - wtedy kompilator Angulara sprawdzi, czy atrybut **rzeczywiście** został ustawiony.

## Sposób 1b - pliki

```
src > app > student > TS student.component.ts > ...
1  import { Component, Input } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11   @Input({required:true}) student!:Student;
12
13   imagePath = ()=>'assets/students/'+this.student.icon;
14
15   onSelectStudent() {
16   }
17 }
```

```
src > app > student > <> student.component.html > ...
Go to component
1  <div>
2    <button (click)="onSelectedStudent()">
3      <img [src]="imagePath()" [alt]="student.lastName">
4      <span>{{student.firstName}} {{student.lastName}} </span>
5    </button>
6  </div>
```

- Od wersji Angular 16 preferowane jest użycie sygnałów i funkcji z nimi związanych.

## Wymuszanie atrybutu w komponencie <app-student-list>

- Wykorzystanie interpolacji dla atrybutu student i wstawienie do elementu <app-student> odpowiedniej wartości studenta.
- Nie uzupełnienie tego atrybutu, albo wstawienie wartości niezgodnej typem kończy się błędem kompilacji.

```
src > app > student-list > <> student-list.component.html > ...  
Go to component  
1 <ul>  
2 | <li>  
3 | | <app-student [student]="univService.students[0]"></app-student>  
4 | | </li>  
5 </ul>
```

```
TS student.component.ts(8, 3): Error occurs in the template of component StudentListComponent.  
src > app > student-list > <> student-list.component.ts(8, 3): Error occurs in the template of component StudentListComponent.  
Go to comp (component) StudentComponent  
1 <ul>  
2 | <li> View Problem (Alt+F8) No quick fixes available  
3 | | <app-student></app-student>  
4 | | </li>  
5 </ul>
```

```
TS student.component.ts(8, 3): Error occurs in the template of component StudentListComponent.  
src > app > student-list > <> student-list.component.ts(8, 3): Error occurs in the template of component StudentListComponent.  
Go to component (property) StudentComponent.student: Student  
1 <ul>  
2 | <li> View Problem (Alt+F8) No quick fixes available  
3 | | <app-student student="title"></app-student>  
4 | | </li>  
5 </ul>
```

## Sposób 2 - z sygnałami.

- Od Angulara 16 preferowany inny sposób określania atrybutu.
- Zamiast dekoratora używa się funkcji `input()` (z `@angular/core`), która tworzy pole o nazwie atrybutu, ale o typie `InputSignal<T>` (o możliwościach `ReadOnlySignal<T>`, ale do użycia tylko w komponentach).
  - mimo tego ustawiając wartość tego atrybutu wstawia się wartość typu `T`, a nie `Signal<T>`.
    - Czyli kod we wzorcu `<app-student-list>` nie ulega zmianie.
- Użycie po prostu `input()` tworzy sygnał o typie `unknown`, co nie jest najlepszym wyjściem. Najlepiej albo określić typ poprzez notację generyczną, albo w argumencie podać wartość, z której kompilator wywnioskuje typ.
  - `input<string>()` - wartość początkowa w kontenerze to `undefined`
  - `input("start")` - wartość początkowa ustalona na "start" i typ na `Signal<T>`.
- W przypadku atrybutu jeszcze lepiej używać zmodyfikowanej wersji `input.required<T>()`, co jest w pewnym sensie równoważne `@Input({required: true})`.
- Oczywiście, skoro to sygnały, to podczas odczytu wartości, trzeba sygnał wywołać poprzez `()` (również we wzorcu komponentu).
- I można/trzeba znowu użyć kodu z `computed` dla `imagePath`.

# Pliki z wersją 2, z sygnałami

```
src > app > student > TS student.component.ts > ...
1  import { Component, computed, input } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11    student=input.required<Student>();
12
13    imagePath = computed(()=>'assets/students/'+this.student().icon);
14
15    onSelectStudent() {
16    }
17 }
```

```
src > app > student > <> student.component.html > ...
Go to component
1  <div>
2    <button (click)="onSelectStudent()">
3      <img [src]="imagePath()" [alt]="student().lastName">
4      <span>{{student().firstName}} {{student().lastName}} </span>
5    </button>
6  </div>
```



## Pętle we wzorcach komponentów.

- Jeśli chcemy w `<app-student-list>` stworzyć listę elementów `<li>` na podstawie jakiejś kolekcji/tablicy należy użyć pętli.
- Małe zamieszanie:
  - do Angular 16.0 było to wykonywane poprzez `*ngFor="..."`, które nie wspierało sygnałów.
  - od Angular 16.1 do 18.X eksperymentalnie można było użyć `@for="..."`, które dobrze współdziałało z sygnałami.
  - od Angular 19.0 usunięto `@for`, ale jej właściwości współdziałania z sygnałami przeniesiono do `*ngFor="..."`.
- Gwiazdka oznacza dyrektywę strukturalną. Jest tłumaczona na inny poprawny kod Angulara.
- Składnia: `*ngFor="let <variable> of <collection>"` jako jakby atrybut elementu.
  - wewnątrz tego elementu można używać zmiennej `<variable>`.
- Wymaga: `import {NgFor} from '@angular/common'`
- Wymaga w `@Component: imports:` `[NgFor]`

```
<li *ngFor="let item of items">{{ item }}</li>
```



```
<ng-template [ngForOf]="items" let-item>  
  <li>{{ item }}</li>  
</ng-template>
```





## Pętla w <app-student-list>

- Pętla - Legacy Angular

```
src > app > student-list > <> student-list.component.html > ...  
    Go to component  
1   <ul>  
2   |   <li *ngFor="let student of univService.students">  
3   |       <app-student [student]="student"></app-student>  
4   |   </li>  
5   </ul>
```

```
src > app > student-list > TS student-list.component.ts > ...  
1   import { NgFor } from '@angular/common'  
2   import { Component, inject } from '@angular/core';  
3   import { UniversityService } from '../services/university.service';  
4   import { StudentComponent } from '../student/student.component';  
5  
6   @Component({  
7       selector: 'app-student-list',  
8       imports: [StudentComponent, NgFor],  
9       templateUrl: './student-list.component.html',  
10      styleUrls: ['./student-list.component.scss']  
11  })  
12  export class StudentListComponent {  
13      public univService=inject(UniversityService);  
14  }
```

### University Management

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

## Nowe rodzaje pętli

- Od Angulara 17 można pisać kod podobne do Razora. Tworzenie kodu pętli dokonuje się poza kodem a'la HTML: pętla `@for` na zewnątrz elementu, który ma być powielony
  - składnia: `@for( <variable> of <collection>; track <expression> { ...}`, gdzie `<expression>` to kod **unikatowego** identyfikatora.
  - W klamrach kod 'html'.
- Poniżej kod z użyciem takiej pętli:

```
src > app > student-list > <> student-list.component.html > ...
  Go to component
1  <ul>
2    @for(student of univService.students; track student.id){
3    <li>
4      <app-student [student]="student"></app-student>
5    </li>
6    }
7  </ul>
```

## Emitowanie zdarzeń

- Po kliknięciu elementu `<app-student>` ten element powinien być traktowany jako bieżący student. Informacje o tym wyborze powinna trafić do `<app-student-list>`, aby ustalić, który element z listy wszystkich został wybrany (i np. pokazać wszystkie dane o tym elemencie).
- Na początku informacja o tym będzie wypisywana na konsolę.
  - w przyszłości otwierać ona będzie tabelę z danymi bieżącego studenta.
- Do przekazywania danych od dziecka do rodzica służą **emitery zdarzeń**.
- W wersji starszej Angulara (do 15) wykorzystuje się dekorator `@Output` (z `@angular/core`) oraz tworzy się obiekt typu `EventEmitter` przyporządkowany do wybranej właściwości np. `emitter`.
  - `EventEmitter` jest klasą generyczną, co pozwala na emitowanie dowolnych wartości, ale dla większej kontroli lepiej ustawić konkretny typ `T` w zapisie `EventEmitter<T>`.
- Wyemitowanie wartości to wywołanie dla emitera (np. `emitter`) metody `emitter.emit(value:T)`, co powoduje wysłanie jej **do rodzica** jako **zdarzenie**.
- Rodzic odbiera zdarzenie poprzez zapis we wzorcu w nawiasach okrągłych (**jak każde zdarzenie**, np. myszki), któremu przyporządkowuje wybraną funkcję, tym razem najlepiej z parametrem `$event`.
  - `(emitter)="onSelectStudent($event)"`

## Kody z emiterem

- Emiter w wersji Legacy Angular

```
src > app > student > TS student.component.ts > ...
1  import { Component, computed, EventEmitter, input, Output } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11   student=input.required<Student>();
12   @Output() selectedId=new EventEmitter<number>();
13
14   imagePath = computed(()=>'assets/students/'+this.student().icon);
15
16   onSelectStudent() {
17     this.selectedId.emit(this.student().id);
18   }
19 }
```

## Kod pętli z emiterem - Legacy Angular

- W tym przypadku należy w wartości pętli `*ngFor` użyć dodatkowego kodu za średnikiem wskazującego unikatowy identyfikator elementu kolekcji. Składnia:
  - `trackBy: <trackByFunction>`
  - funkcja `<trackByFunction>` otrzymuje dwa parametry: indeks numeru w kolekcji i sam element.
  - Funkcja zwraca niepowtarzalny identyfikator
  - Kod samej funkcji **musi być** w modelu komponentu (nie może to być w tym miejscu wyrażenie lambda)

```
src > app > student-list > <> student-list.component.html > ...
```

Go to component

```
1 <ul>
2   <li *ngFor="let student of univService.students; trackBy: trackById">
3     <app-student [student]="student" (selectedId)="onSelect($event)"></app-student>
4   </li>
5 </ul>
```

```
src > app > student-list > TS student-list.component.ts > ...
```

```
1 import { NgFor } from '@angular/common';
```

```
e';
es/university.service';
udent.component";
odel';
```

```
9 imports: [StudentComponent, NgFor],
10 templateUrl: './student-list.component.html',
11 styleUrls: ['./student-list.component.scss']
12 })
13 export class StudentListComponent {
14   public univService=inject(UniversityService);
15
16   trackById(index: number, item: Student) {
17     return item.id;
18   }
19
20   onSelect(id : number) {
21     console.log(id);
22   }
23 }
```

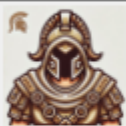
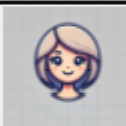

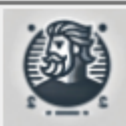
## Kod pętli z emiterem - Modern Angular

- Z nową pętlą od Angular 18 wygląda to jak poniżej:
  - dodatkowy kod po średniku, można wpisać wprost wyrażenie generujące unikatowy identyfikator elementu.

```
src > app > student-list > <> student-list.component.html > ...  
Go to component  
1 <ul>  
2   @for(student of univService.students; track student.id){  
3   <li>  
4     <app-student [student]="student" (selectedId)="onSelect($event)"></app-student>  
5   </li>  
6   }  
7 </ul>
```

# Działanie aplikacji

## University Management

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

Angular is running in development [core.mjs:20876](#) mode.

1	<a href="#">student-list.component.ts:21</a>
2	<a href="#">student-list.component.ts:21</a>
3	<a href="#">student-list.component.ts:21</a>
4	<a href="#">student-list.component.ts:21</a>
2	<a href="#">student-list.component.ts:21</a>

- Efekt po kliknięciu po kolei każdego elementu.



## Wersja nowsza tworzenia emitera

- Zamiast dekoratora można użyć nowszej wersji tworzenia emitera od Angular 16 - funkcji `output()` z `@angular/core`.
- Ta funkcja też jest generyczna, więc najlepiej użyć ją z parametrem `T`, czyli jako `output<T>()`.
- Reszta kodów pozostaje bez zmian:

```
src > app > student > TS student.component.ts > ...
1  import { Component, computed, input, output } from '@angular/core';
2  import { Student } from '../models/student.model';
3
4  @Component({
5    selector: 'app-student',
6    imports: [],
7    templateUrl: './student.component.html',
8    styleUrls: ['./student.component.scss']
9  })
10 export class StudentComponent {
11   student=input.required<Student>();
12   selectedId=output<number>();
13
14   imagePath = computed(()=>'assets/students/'+this.student().icon);
15
16   onSelectStudent() {
17     this.selectedId.emit(this.student().id);
18   }
19 }
```

## Warunkowe dodawanie klasy

- Po kliknięciu elementu `<app-student>` listy przycisków, aktualnie wskazany element należałoby wyróżnić.
- Informacje o tym, który element został wybrany ma klasa `StudentListComponent`.
- Zatem rodzic powinien do nowego atrybutu typu logicznego przekazać informacje o aktualnie wskazanym elemencie, a sam element powinien nadać sobie nowe stylowanie (np. przez dodanie klasy).
- Warunkowe dodanie klasy wykonuje się poprzez składnię:  
`[class.<className>]="<Condition>"`, np.:
  - `[class.active]="isSelected() "`
  - czyli dodaj do elementu klasę "active" jeśli sygnał `isSelected` jest prawdą.

```
src > app > student > student.component.scss > ...
1  img{
2    height: 50px;
3    width: auto;
4  }
5
6  .active{
7    background-color: greenyellow;
8  }
```

# Warunkowe dodawanie klasy - zmiany w kodzie

```
10 export class StudentComponent {
11     student=input.required<Student>();
12     selectedId=output<number>();
13     isSelected=input.required<boolean>();
14
15     imagePath = computed(()=>'assets/students/'+this.student().icon);
16
17     onSelectStudent() {
18         this.selectedId.emit(this.student().id);
19     }
20 }
```

```
src > app > student > <> student.component.html > ...
Go to component
1 <div>
2     <button [class.active]="isSelected()" (click)="onSelectStudent()">
3         <img [src]="imagePath()" [alt]="student().lastName">
4         <span>{{student().firstName}} {{student().lastName}} </span>
5     </button>
6 </div>
```





```
13 export class StudentListComponent {
14     public univService=inject<UniversityService>();
15     public currentId?:number;
16
17     trackById(index: number, item: Student) {
18         return item.id;
19     }
20
21     onSelect(id : number) {
22         this.currentId=id;
23     }
24 }
```

```
src > app > student-list > <> student-list.component.html > ...
Go to component
1 <ul>
2     @for(student of univService.students; track student.id){
3     <li>
4         <app-student
5             [isSelected]="student.id===currentId"
6             [student]="student"
7             (selectedId)="onSelect($event)"></app-student>
8     </li>
9     }
10 </ul>
```





# Działanie

- Po uruchomienie
- Po kliknięciu w trzeci element
- Po kliknięciu w pierwszy element





**University Management**

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

**University Management**

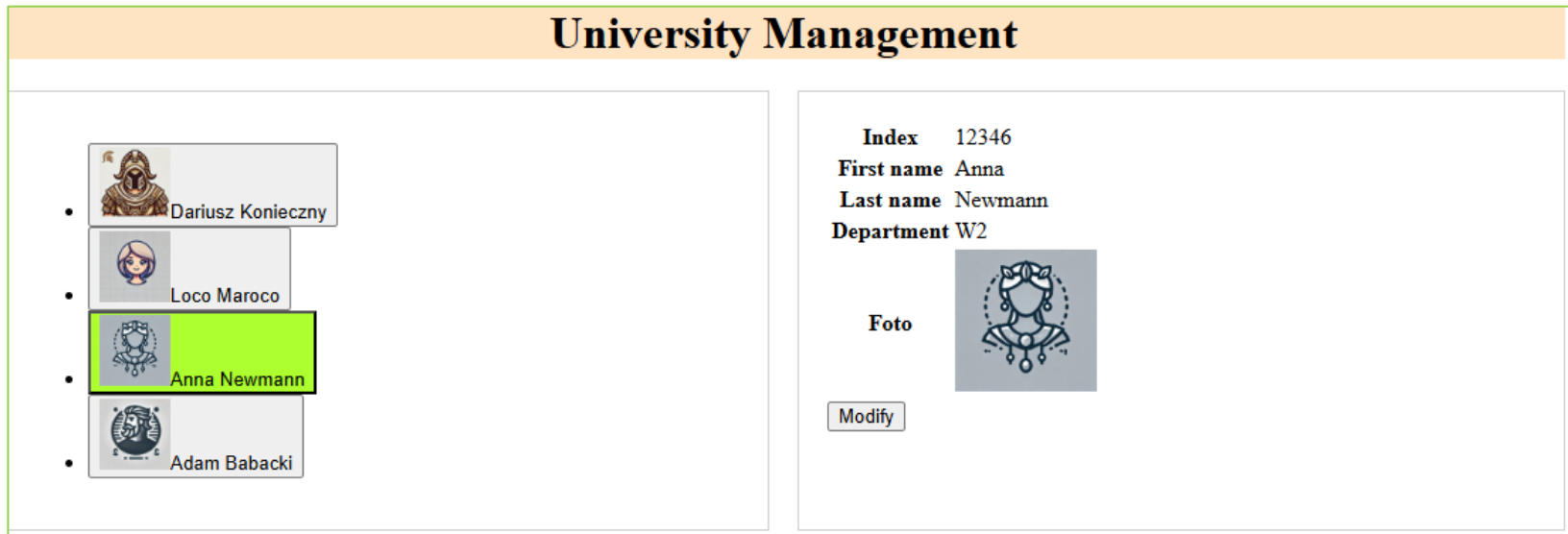
-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

**University Management**

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

# Komponent pokazywania danych - oczekiwany efekt końcowy

- Sterowany przez `<app-student-list>`
- Do pokazywania wszystkich danych bieżącego elementu.
  - pokazuje się po wybraniu go z listy, w przeciwnym przypadku jakiś placeholder.
  - przekazywanie przez atrybut `[student]`.
- Posiada przycisk do zmiany danych studenta, który jest właśnie wyświetlany.
  - generuje zdarzenie `(modifyPress)`.
- Za pomocą CSS podzielić ekran na pół.



# Komponent pokazywania danych i przycisk Change

- Tworzenie: `ng g c student-full --skip-tests`
- Zmodyfikowany plik `student-list.component.html`
  - przygotowanie do użycia CSS
- Zmodyfikowany plik `student-list.component.scss`
- Realizacja warunkowego pokazywania danych studenta lub placeholder:.
  - Wykorzystany zostanie konstrukcja we wzorcu komponentu `@if(<condition>){<body1>} @else {<body2>}`
  - W tej instrukcji może nie być sekcji `@else`
  - Mogą być pomiędzy sekcjami `@if` i `@else` wiele sekcji `@else if`
  - Instrukcja ta jest dostępna od Angular 17.

```
src > app > student-list > <> student-list.component.html > ...
Go to component
1  <div class="container">
2    <div class="half">
3      <ul>
4        @for(student of univService.students; track student.id){
5          <li>
6            <app-student
7              [isSelected]="student.id===currentId"
8              [student]="student"
9              (selectedId)="onSelect($event)"></app-student>
10           </li>
11         }
12       </ul>
13     </div>
14
15     <div class="half">
16       @if (currentId){
17         <app-student-full
18           [student]="currentStudent!"
19           (modifyPress)="onStudentModifyStart()">
20         </app-student-full>
21       } @else if(true){
22         <p>
23           No choosen student.
24         </p>
25       }
26     </div>
27 </div>
```

```
src > app > student-list > <> student-list.component.scss > ...
1  .container {
2    display: flex;
3    gap: 20px;      /* Space between elements */
4  }
5
6  .half {
7    flex: 1;        /* Each element takes up 50% of the width */
8    border: 1px solid #ccc;
9    padding: 20px;
10   box-sizing: border-box;
11 }
```

## Wersja starsza warunkowych komponentów

- Przed Angular 17.
- Możliwość warunkowego dodawania komponentów (fragmentów "html") była obecna od Angular 2, jednak składnia polegała na wpisaniu elementu `*ngIf` jako atrybut jakiegoś elementu.
- Stworzenie sekcji "else" w tym przypadku jest bardziej skomplikowane. Należy przygotować element `<ng-template #<ident>>`, który będzie zawierał kod dla "else" i użyć tego w stringu dla `*ngIf` jak poniżej.
- Wymaga to też importu `NgIf` z `@angular/common` i dopisania w `imports` w dekoratorze `@Component`.

```
15     <div class="half">
16         <app-student-form *ngIf="currentId; else fallback">
17
18     </app-student-form>
19
20     <ng-template #fallback>
21         <p>
22             No chosen student.
23         </p>
24     </ng-template>
25 </div>
```

W kodzie wzorca `#<identifier>` oznacza ogólnie identyfikator `<identifier>` elementu, w którym został umieszczony, ale do użycia **tylko** w ramach **tego wzorca** komponentu.

# Pliki nowego komponentu

src > app > student-full > TS student-full.component.ts > ...

```
4  @Component({
5    selector: 'app-student-full',
6    imports: [],
7    templateUrl: './student-full.component.html',
8    styleUrls: ['./student-full.component.scss']
9  })
10 export class StudentFullComponent {
11   student=input.required<Student>();
12   modifyPress=output();
13
14   imagePath = computed(()=>'assets/students/'+this.student().icon);
15
16   onModify(){
17     this.modifyPress.emit();
18   }
19 }
```

src > app > student-full > S student-full.component.scss >

```
1  img {
2    height: 100px;
3    width: auto;
4  }
```

src > app > student-full > HTML student-full.component.html > ...

```
1  <div>
2    <table>
3      <tbody>
4        <tr>
5          <th>Index</th>
6          <td>{{ student().index }}</td>
7        </tr>
8        <tr>
9          <th>First name</th>
10         <td>{{ student().firstName }}</td>
11       </tr>
12       <tr>
13         <th>Last name</th>
14         <td>{{ student().lastName }}</td>
15       </tr>
16       <tr>
17         <th>Department</th>
18         <td>{{ student().department }}</td>
19       </tr>
20       <tr>
21         <th>Foto</th>
22         <td><img [src]="imagePath()" /></td>
23       </tr>
24     </tbody>
25   </table>
26   <button (click)="onModify()">Modify</button>
27 </div>
```



# Testowanie działania przycisku "Modify"

- Zmodyfikowany plik `student-list.component.ts`
  - dodanie pola aktualnego studenta
  - wypisywanie napisu na konsolę w reakcji na przyciśnięcie przycisku "Modify".

```
src > app > student-list > TS student-list.component.ts > ...
 9  @Component({
10    selector: 'app-student-list',
11    imports: [StudentComponent, StudentFullComponent]
12    templateUrl: './student-list.component.html',
13    styleUrls: ['./student-list.component.scss']
14  })
15  export class StudentListComponent {
16    public univService=inject(UniversityService);
17    public currentId?:number;
18    public currentStudent?:Student;
19
20
21    onSelect(id : number) {
22      this.currentId=id;
23      this.currentStudent=this.univService.students.find((student)=>student.id==id);
24    }
25
26    onStudentModifyStart(){
27      console.log("modify "+this.currentId);
28    }
  }
```





The screenshot shows the 'University Management' application interface. On the left, there is a list of students with their photos and names: Dariusz Konieczny, Loco Maroco, Anna Newmann, and Adam Babacki. The 'Adam Babacki' entry is highlighted in green. On the right, there is a detailed view of the selected student, Adam Babacki, showing his index (33333), first name (Adam), last name (Babacki), and department (W4N). Below this information is a 'Foto' label and a button labeled 'Modify'. The browser console on the right shows the following log messages:

```
Angular is running in development mode. core.mjs:20876
modify 4 student-list.component.ts:28
```

# Formularz - oczekiwany efekt docelowy

## University Management

Two-way binding outside: Loco Maroco

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki


### Modify student data

Index

First name

Last name

Department



Two-way binding in dialog: Loco Maroco

## Logika działania

- Właściwości połączone są za pomocą two-way binding z polami `<input>`. Celem demonstracji 2 w postaci sygnałów, 2 w postaci zwykłych pól.
- Informacja zwracana do komponentu rodzica za pomocą emiterów zdarzeń:
  - po wciśnięciu klawisza typu `"submit"` i napisem `"Save"`.
  - po wciśnięciu klawisza z napisem `"Cancel"`.
- Kliknięcie poza dialogiem traktowane jest jak rezygnacja z modyfikacji.
- W przypadku kliknięcia przycisku typu `"submit"` wysłanie danych o studencie (ale tylko tych, które można było zmienić w formularzu)
  - użycie typu bezpośredniego
- Wewnętrzne przetwarzanie po wciśnięciu klawisza z napisem `"initials"` - zostawienie w nazwisku i imieniu tylko pierwszych liter.
- Do działania potrzeba otrzymać od komponentu rodzica dane o studencie. Dane te mają być przepisane do pól `<input>`.
  - Nie można tego zrobić w konstruktorze, bo wtedy jeszcze komponent nie jest zrenderowany, więc nie działa jeszcze two-way-binding.
  - należy to zrobić w metodzie `ngOnInit()`. Dla pewności warto dopisać implementowanie interfejsu `OnInit` (z `@angular/core`), który zawiera tą metodę.

# Przygotowanie formularza 1/3

- Komenda: `ng g c student-form --skip-tests`
- Formularz będzie w środku elementu `<dialog>`.
- Nowy komponent będzie użyty w `<app-student-list>`.
- Reszta aplikacji w tym czasie wypełniania formularza nie powinna być aktywna:
  - standardowo osiąga się to poprzez javascriptowe `showModal()` dla elementu `<dialog>`, co byłoby kłopotliwe i łamałoby zasady Angulara.
  - zamiast tego tworzenie tła zakrywającego cały viewport za pomocą elementu `div.backdrop` i reguł CSS (kod poniżej).
  - pozostałe elementy CSS dla dialogu i obrazka (kod poniżej).

```
src > app > student-form > student-form.component.scss > ...
1  .backdrop {
2      position:fixed;
3      top:0;
4      left:0;
5      width: 100%;
6      height: 100vh;
7      background-color: rgba(0, 0, 0, 0.5); /* Półprzezroczyste tło */
8  }
9
10 img{
11     height: 200px;
12     width: auto;
13 }
14
15 dialog{
16     width:50%;
17     max-width: 30rem;
18     overflow: hidden;
19     background-color: aqua;
20 }
```

```
src > app > student-form > student-form.component.html > ...
Go to component
1  <div class="backdrop" ></div>
2  <dialog open>
3  <h2>Modify student data</h2>
4  <form (ngSubmit)="onSubmit()">
```

## Przygotowanie formularza 2/3

- W elementach `<input>` atrybut `name` jest wymagany.
- Wytlumaczenie na kolejnych slajdach

```
4 <form (ngSubmit)="onSubmit()"
5   <p>
6     <label for="index">Index</label>
7     <input type="number" id="index" name="index" [(ngModel)]="enteredIndex" />
8   </p>
9   <p>
10    <label for="first-name">First name</label>
11    <input type="text" id="first-name" name="first-name" [(ngModel)]="enteredFirstName"/>
12  </p>
13  <p>
14    <label for="last-name">Last name</label>
15    <input type="text" id="last-name" name="last-name" [(ngModel)]="enteredLastName"/>
16  </p>
17  <p>
18    <label for="department">Department</label>
19    <input type="text" id="department" name="department" [(ngModel)]="enteredDepartment"/>
20  </p>
21  <p>
22    <img [src]="imagePath()" />
23  </p>
24  <p>
25    <button type="button" (click)="onCancel()">Cancel</button>
26    <button type="submit">Save</button>
27    <button type="button" (click)="onClickInitials()">Initials</button>
28  </p>
29 </form>
```



Go to component

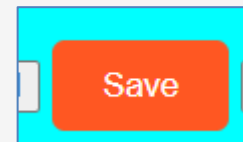
1 <div class="backdrop" (click)="onCancel()"></div>

## Dlaczego (ngSubmit) ?

- Element `<button>` może mieć trzy typy "submit" (domyślny), "reset" i ogólny "button".
- Zamiast użycia `(ngSubmit)=...` dla przycisku o typie "submit" można użyć ogólnego typu i zdarzenia `(click)`.
- Jednak można ustalić globalne CSS dla wszystkich przycisków np. typu "submit" w aplikacji (przykład **poniżej**).
- W przypadku typu ogólnego trzeba dodać klasy do przycisków, żeby je rozróżnić.
- Użycie przycisku typu "submit" bez użycia `(ngSubmit)` prowadzi do standardowego działania, czyli wysłania danych formularza do celu wskazanego w atrybucie `action`, a jeśli go nie ma, do serwera skąd jest działająca aplikacja.

src >  styles.scss > ...

```
1  /* You can add global styles to this file, and also import other style files */
2  form button[type="submit"] {
3      background-color:  #ff5722;
4      color:  white;
5      padding: 10px 20px;
6      border: none;
7      border-radius: 5px;
8      cursor: pointer;
9      margin: 0 5px;
10 }
```



## Przygotowanie formularza 3/3

- Two-way binding za pomocą [ (ngModel) ]
  - Każda zmiana w polu `<input>` zostanie przeniesiona do wskazanej za znakiem równa się '=' właściwości (lub sygnału)
  - każda zmiana we właściwości (w sygnale) zostanie przeniesiona do pola `<input>`
- **Poniżej** tylko celem testowania
  - dodanie paragrafu z interpolacją napisu w ramach `<dialog>`.
  - i drugiego z interpolacją napisu za elementem `<dialog>`.

```
29     </form>
30     <p>
31         Two-way binding in dialog: {{enteredFirstName()}} {{enteredLastName}}
32     </p>
33 </dialog>
34 <p>
35     Two-way binding outside: {{enteredFirstName()}} {{enteredLastName}}
36 </p>
```

## Moduł `FormsModule`

- Moduł `FormsModule` to kolekcja z dyrektywami i funkcjami itd. dla ułatwienia operowania na formularzach.
  - tutaj znajduje się też dyrektywa `ngModule` i zdarzenie `ngSubmit`.



# Model StudentFormComponent 1/2

- Tworzenie sygnału w postaci `output<void>()` znaczy, że będzie wysyłany sygnał bez wartości.
  - To samo znaczy po prostu `output()`.

```
6  @Component({
7    selector: 'app-student-form',
8    imports: [FormsModule],
9    templateUrl: './student-form.component.html',
10   styleUrls: ['./student-form.component.scss']
11 })
12 export class StudentFormComponent implements OnInit {
13   student=input.required<Student>();
14   cancel=output<void>();
15   save=output<{index:number, firstName:string, lastName:string, department:Department}>();
16   enteredIndex=signal(0); // signals - preferred
17   enteredFirstName=signal("");
18   enteredLastName=''; // pure values - old versions
19   enteredDepartment:Department='W1';
20
21   ngOnInit(){
22     this.enteredIndex.set(this.student().index);
23     this.enteredFirstName.set(this.student().firstName);
24     this.enteredLastName=this.student().lastName;
25     this.enteredDepartment=this.student().department;
26   }
27 }
```

## Model StudentFormComponent 2/2

```
29   onClickInitials(){
30     |   this.enteredFirstName.set(this.student().firstName[0]);
31     |   this.enteredLastName=this.student().lastName[0];
32   }
33
34   onCancel(){
35     |   this.cancel.emit();
36   }
37
38   onSubmit(){
39     |   this.save.emit({
40     |     index:this.enteredIndex(),
41     |     firstName:this.enteredFirstName(),
42     |     lastName:this.enteredLastName,
43     |     department:this.enteredDepartment});
44   }
45
46   imagePath = computed(()=>'assets/students/'+this.student().icon);
47
48 }
```

## Użycie nowego komponentu w <app-student-list>

- Do pokazywania/ukrywania <app-student-form> użyta zostanie proste pole logiczne (ale może być też sygnał, wewnętrznie używany) `isStudentModifyOpen`.
  - Komponent <app-student-form> dodany zostanie na początku komponentu listy
    - ponieważ ma on elementy HTML z `position: fixed`, może być dodany w wielu miejscach.
  - otwarcie nastąpi w reakcji na zdarzenie (`modifyPress`) z komponentu <app-student-full>.
- Do formularza wysłane będą dane studenta zapamiętanego w polu `currentStudent` (użyty był też dla komponentu <app-student-full>).
- Z formularza należy odebrać dwa typy zdarzeń:
  - (`cancel`)
    - zamknąć okienko dialogu
  - (`save`)
    - przepisać dane do ze zdarzenia **do serwisu** jak i **do pola** `currentStudent`.

```
src > app > student-list > <> student-list.component.html > ...
Go to component
1  @if(isStudentModifyOpen){
2      <app-student-form \
3          [student]="currentStudent!"
4          (cancel)="onStudentModifyCancel()"
5          (save)="onStudentModifySave($event)">
6      </app-student-form>
7  }
8  <div class="container">
```

```
src > app > student-list > <> student-list.component.html > ...
8  <div class="container">
...
22      <div class="half">
23          @if (currentId){
24              <app-student-full
25                  [student]="currentStudent!"
26                  (modifyPress)="onStudentModifyStart()">
27              </app-student-full>
28          } @else if(true){
29              <p>
30                  No choosen student.
31              </p>
32          }
33      </div>
34  </div>
```

# Modyfikacje StudentListComponent

```
9  @Component({
10    selector: 'app-student-list',
11    imports: [StudentComponent, StudentFullComponent, StudentFormComponent],
12    templateUrl: './student-list.component.html',
13    styleUrls: ['./student-list.component.scss']
14  })
15  export class StudentListComponent {
16    public univService=inject(UniversityService);
17    public currentId?:number;
18    public currentStudent?:Student;
19    isStudentModifyOpen=false;
```

```
28  onStudentModifyStart(){
29    | this.isStudentModifyOpen=true;
30    | }
31
32  onStudentModifyCancel(){
33    | this.isStudentModifyOpen=false;
34    | }
35
36  onStudentModifySave(modifyDate: {index:number, firstName:string, lastName:string,department:Department}){
37    this.isStudentModifyOpen=false;
38    this.univService.students=this.univService.students.map((student)=>student.id==this.currentId?
39      {...student,
40        index:modifyDate.index,
41        firstName:modifyDate.firstName,
42        lastName:modifyDate.lastName,
43        department:modifyDate.department
44      }:student);
45    //this.currentId=undefined;
46    this.currentStudent=this.univService.students.find((student)=>student.id==this.currentId);
47  }
48 }
```

## Wielokropek

- W Javascript **wielokropek przed obiektem** oznacza stworzenie jego **kopii**.
  - użyte musi być między klamrami
- Następnie **po przecinkach** można podawać pola obiektu, które w kopii mają ulec **zmianie** na nową wartość.
- Czyli w przypadku kodu z poprzedniej strony stworzona została kopia dla studenta, którego `id` równa się `this.currentId`, a następnie w kopii zmieniono wartości pól `index`, `firstName`, `lastName`, `department` na analogiczne wartości pobrane z obiektu zdarzenia.

## Ostateczny efekt

- Pamiętać należy, że jest to ciągle **jedna** strona HTML, więc paragrafy dodane za elementem <dialog> (który jest usunięty z kolejki elementów do ustawienia na stronie ) należą do <main>, do jego początku.

University Management

Two-way binding outside: Loco Maroco

- Dariusz Konieczny
- Loco Maroco
- Anna Newmann
- Adam Babacki

### Modify student data

Index: 22222

First name: Loco

Last name: Maroco

Department: W4N

Two-way binding in dialog: Loco Maroco





Cancel Save Initials

Po kliknięciu przycisku "Initials"

- Zmiany, poprzez mechanizm sygnałów, są dokonywane we wszystkich miejscach odczytywania sygnału.

University Management

Two-way binding outside: L M

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki


### Modify student data

Index

First name

Last name

Department



Two-way binding in dialog: L M

# Zmiana danych za pomocą klawiatury





- Zmiana danych w polach za pomocą klawiatury
  - komunikacja od `<input>` do właściwości (sygnałów)
  - a potem od sygnałów do metod `compute()` dla interpolowanych tekstów.

The screenshot displays a web application titled "University Management". On the left, a sidebar lists four users: Dariusz Konieczny, Loco Maroco, Anna Newmann, and Adam Babacki. A red box highlights the text "Two-way binding outside: Jennifer Zola" above the sidebar. The main area features a "Modify student data" dialog with a cyan background. This dialog contains four input fields: "Index" (22222), "First name" (Jennifer), "Last name" (Zola), and "Department" (W4N). The "First name" and "Last name" fields are highlighted with red boxes. Below the fields is a placeholder image of a female student. At the bottom of the dialog are three buttons: "Cancel", "Save", and "Initials". A red box at the bottom of the dialog highlights the text "Two-way binding in dialog: Jennifer Zola".



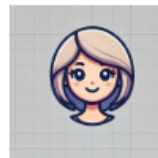
# Po kliknięciu przycisku "Save"

## University Management

-  Dariusz Konieczny
-  Jennifer Zola
-  Anna Newmann
-  Adam Babacki

**Index** 22222  
**First name** Jennifer  
**Last name** Zola  
**Department** W4N

**Foto**



Modify

## Uwaga do działania aplikacji

- Jak było wcześniej wspomniane sprawdzanie poprawności typów wykonywane jest na etapie **kompilacji**, a nie **wykonania**. Zatem wpisanie stringu, który nie jest dopuszczalnym dla np. danej typu Department (w formularzu), nie generuje błędu.


**Modify student data**

Index 12346

First name Anna





Last name Newmann

Department W2222222




Cancel Save Initials

Two-way binding in dialog: Anna Newmann

-  Dariusz Konieczny
-  Loco Maroco
-  Anna Newmann
-  Adam Babacki

Index 12346  
First name Anna  
Last name Newmann  
Department W2222222

Foto 

Modify

## Uwagi końcowe

- Środowisko VS Code z odpowiednimi wtyczkami podpowiada dopuszczalne wartości, np. zdarzenia.
- Podczas wpisywania wyrażenia interpolowanego **podpowiada** pola/właściwości/metody z powiązanej klasy oraz z całej aplikacji.
- W przypadku podania klasy/metody zakończonej klawiszem [Tab], w razie potrzeby uzupełnia kod o **importowanie** tegoż **elementu**:
  - zarówno jako `import` w kodzie klasy, jak i jako `imports` w kodzie dekoratora `@Components`.
- W **trakcie pisania** kompiluje kod i podkreśla błędy, które w "dymkach" wyjaśnia.
- Jeśli zmieniamy kod **w trakcie działania** aplikacji, zmiany po zapisie pliku na dysk są **automatycznie aktualizowane** w aplikacji
  - chyba, że zmieniamy konfigurację itp.
- Przygotowane wykłady nt. Angulara wzorują się na wideo-kursie "Angular - The Complete Guide [2024 Edition]" Maximilian Schwarzmüller, ze strony <https://learning.oreilly.com/>
  - dla studentów PWr dostęp za darmo
  - ok. 70h, ale można oglądać na szybkości 1,5 -> 45h
    - Momentami w kolejnej przykładowej aplikacji powtarza część tłumaczenia, która już była, ale przecież można szybko przesunąć taką część
  - do wersji Angular 18, ale pokazuje też jak to samo uzyskać we wcześniejszych wersjach, również tych korzystających z modułów.
  - lektor mówi po angielsku prostymi wyrazami, wyraźnie, czysto
  - Polecam