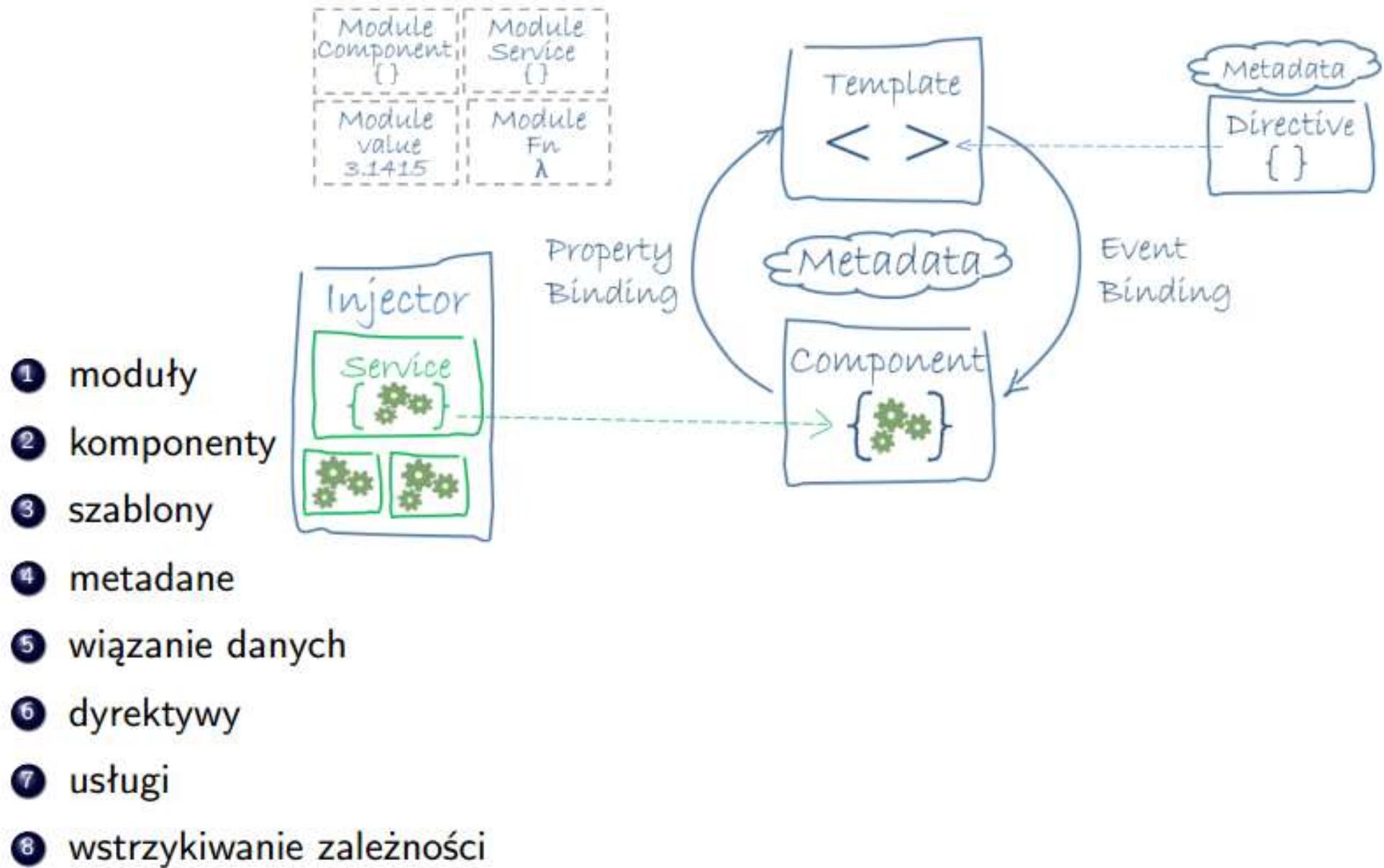


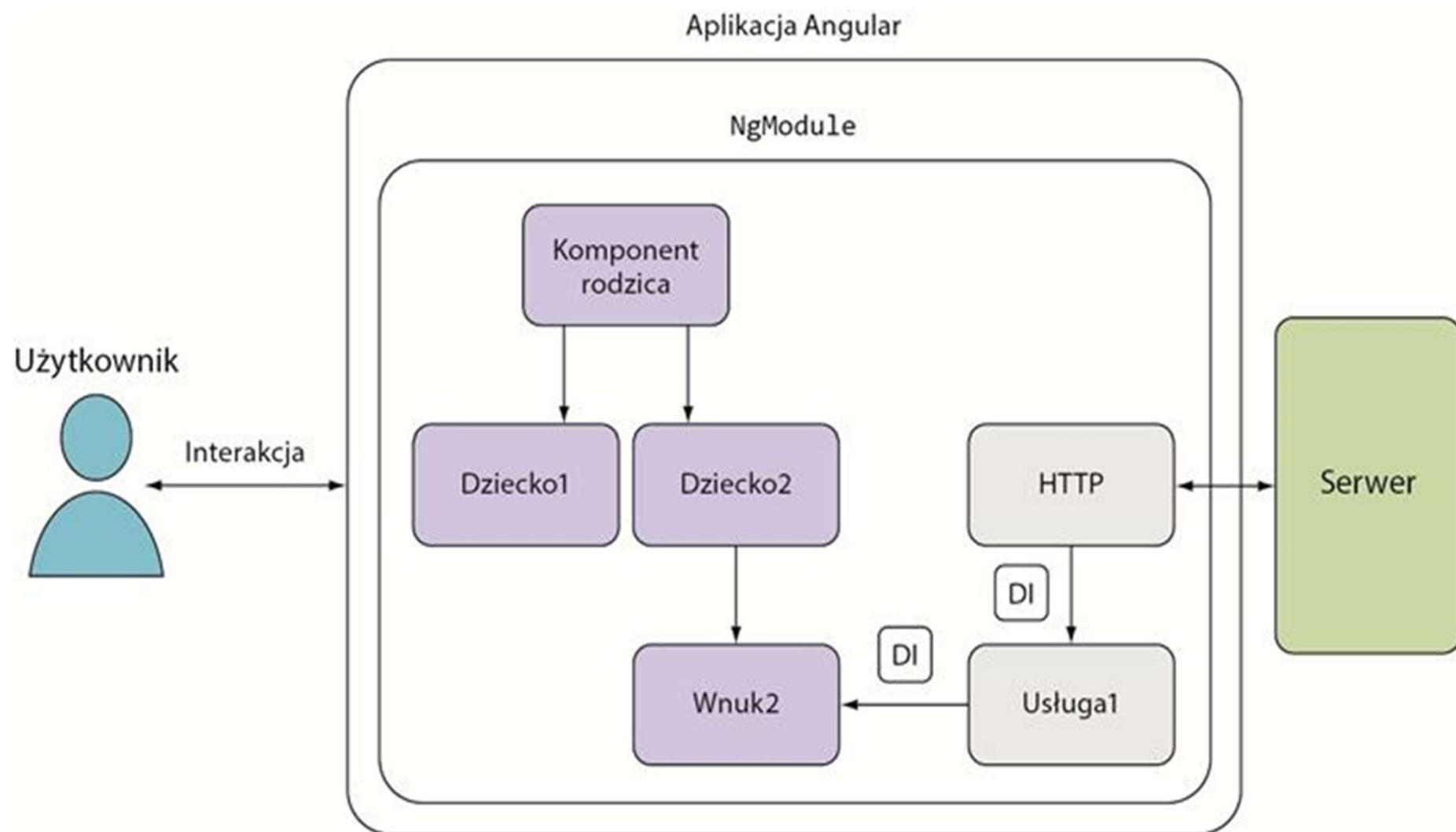
Angular czesc 2

dr inż. Grzegorz Rogus

Architektura Angular - 8 głównych bloków konstrukcyjnych



Architektura Angulara w praktyce



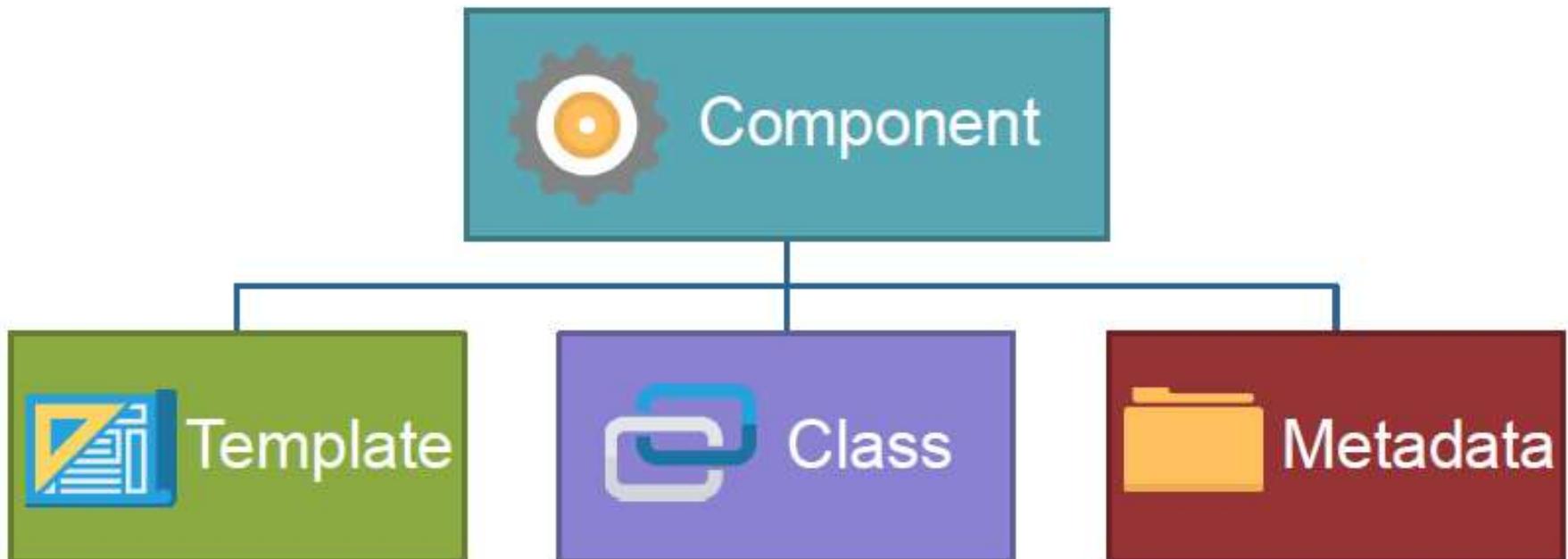
KOMPONENTY



- Podstawowa koncepcja aplikacji w Angular
- Reużywalna blok składowy UI
- Cała aplikacja jest drzewem komponentów

Czym jest komponent w Angular

- Komponenty są podstawowymi blokami konstrukcyjnymi aplikacji Angular.
- Kontrolują jakiś obszar ekranu - widok - poprzez związany z nimi szablon.
- Wewnątrz klasy komponentu definiujemy logikę aplikacji - określamy jak komponent obsługuje widok.
- Klasa komponentu komunikuje się z widokiem poprzez API pól i metod.



Przykład komponentu

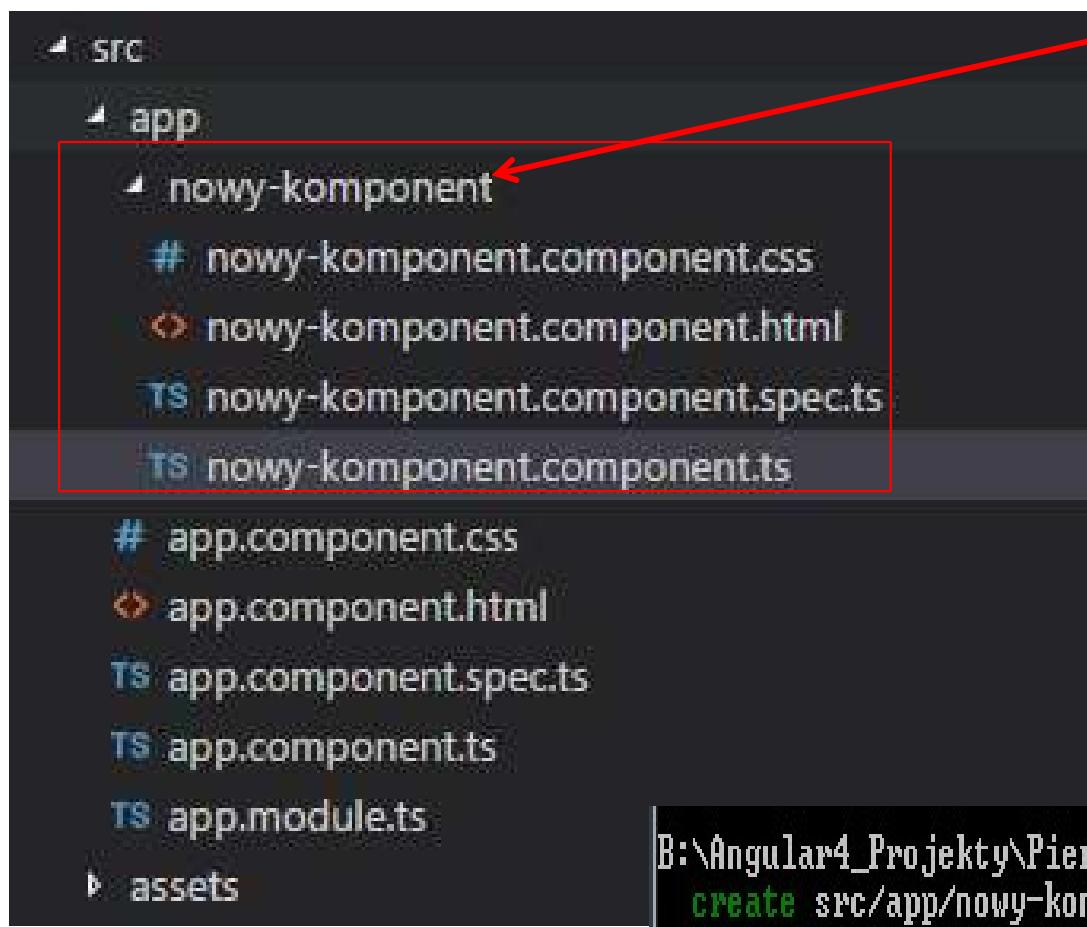
```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  template: `<div>Witam – jestem {{name}} .  
  <button (click)="start()"> test </button></div>`  
})

export class MyComponent {  
  constructor() {  
    this.name = 'Grzegorz'  
  }  
  start() {  
    console.log('to ja', this.name)  
  }  
}
```

1. Tworzymy nowy komponent

- `ng g ComponentName`



Wygenerowane pliki

```
B:\Angular4_Projekty\PierwszyAngular>ng g component NowyKomponent
create src/app/nowy-komponent/nowy-komponent.component.html (33 bytes)
create src/app/nowy-komponent/nowy-komponent.component.spec.ts (678 bytes)
create src/app/nowy-komponent/nowy-komponent.component.ts (300 bytes)
create src/app/nowy-komponent/nowy-komponent.component.css (0 bytes)
update src/app/app.module.ts (488 bytes)
```

2. Rejestracja komponentu

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { NowyKomponentComponent } from './nowy-komponent/nowy-komponent.component';

@NgModule({
  declarations: [
    AppComponent,
    NowyKomponentComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Komponent w Angular

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

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

  constructor() { }

  ngOnInit() {
  }
}
```

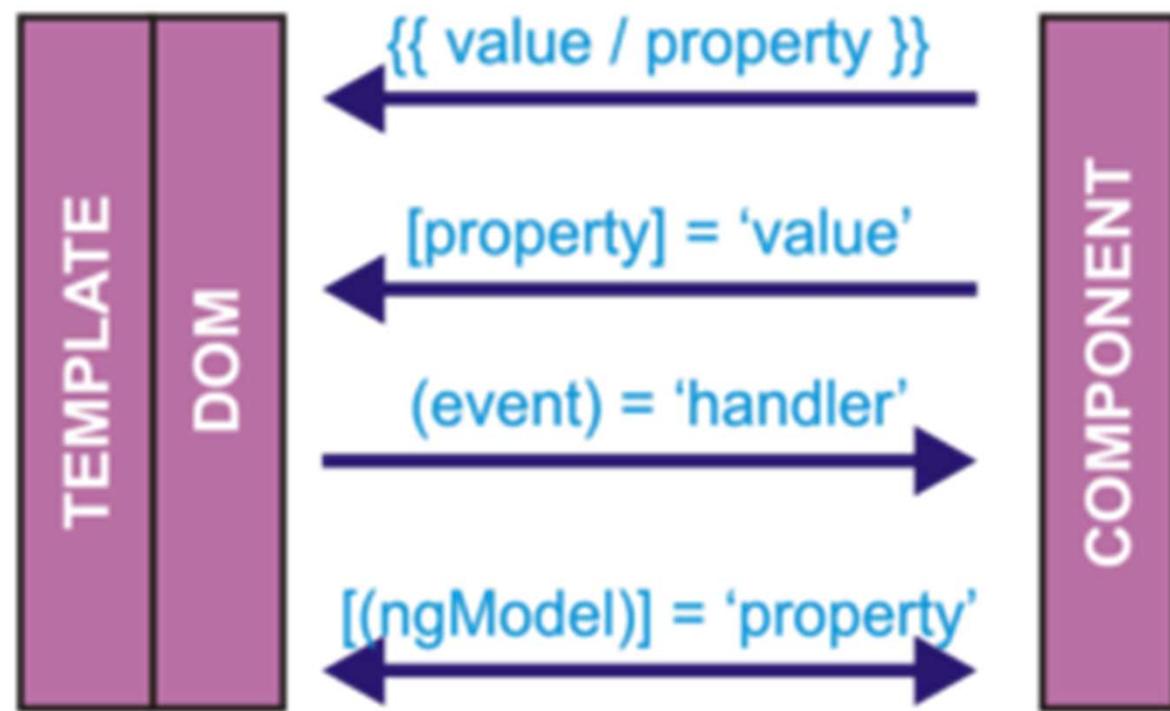
Typy wiązania danych

Bez wykorzystania frameworka sami jesteśmy odpowiedzialni za:

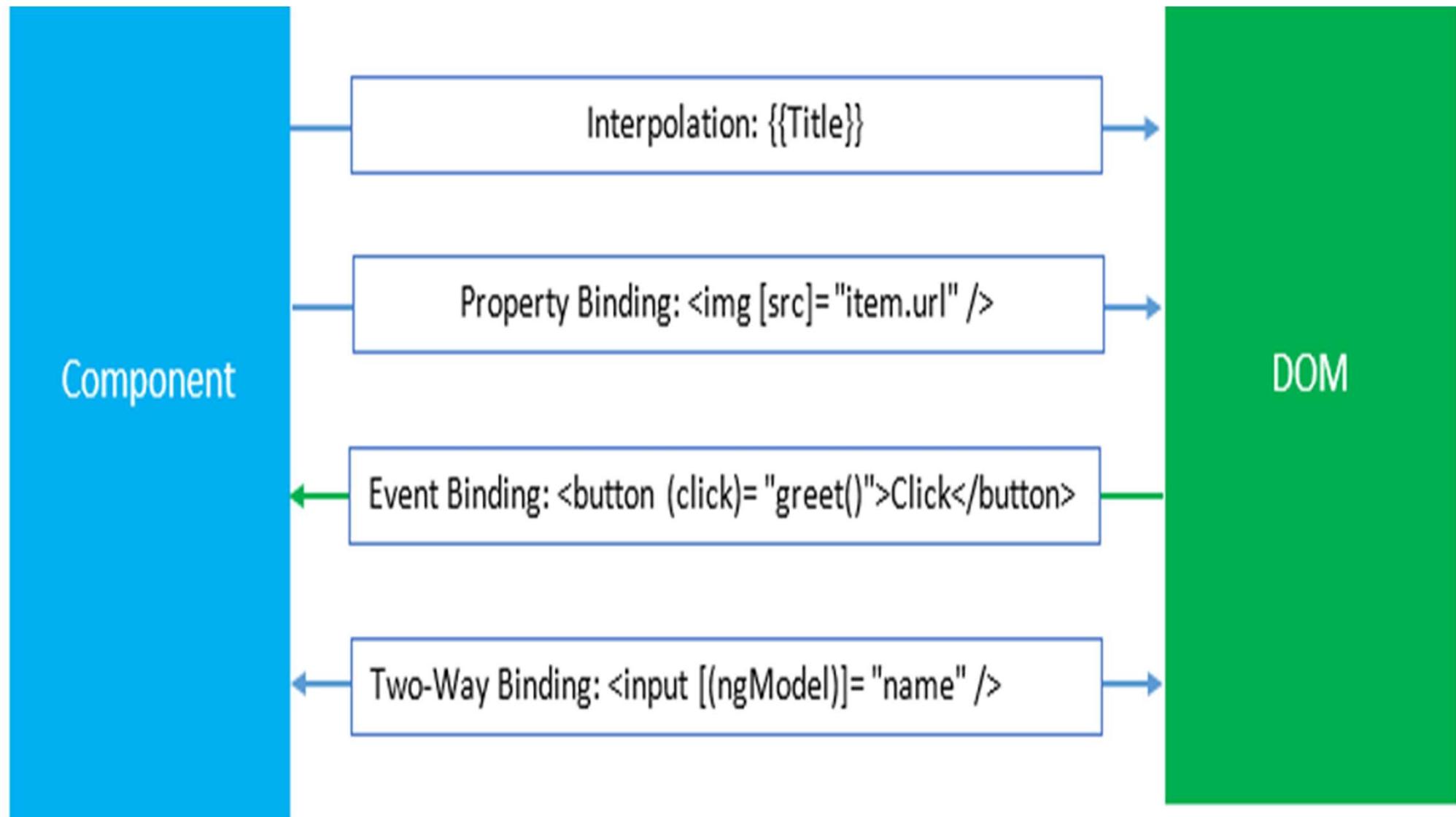
- wstawianie danych do kontrolek HTML
- zamianę reakcji użytkownika w akcje i aktualizacje wartości.

Pisanie takiej logiki wstaw/wyciągnij ręcznie jest uciążliwe, sprzyja błędom i jest koszmarem przy czytaniu kodu.

- Angular wspiera wiązanie danych, mechanizm wiążący część szablonu z częściami komponentu.
- Dodajemy oznaczenia wiązania danych w szablonie HTML, zeby określić jak Angular ma łączyć obie strony.



Wiązanie danych



1-way Binding

Interpolation



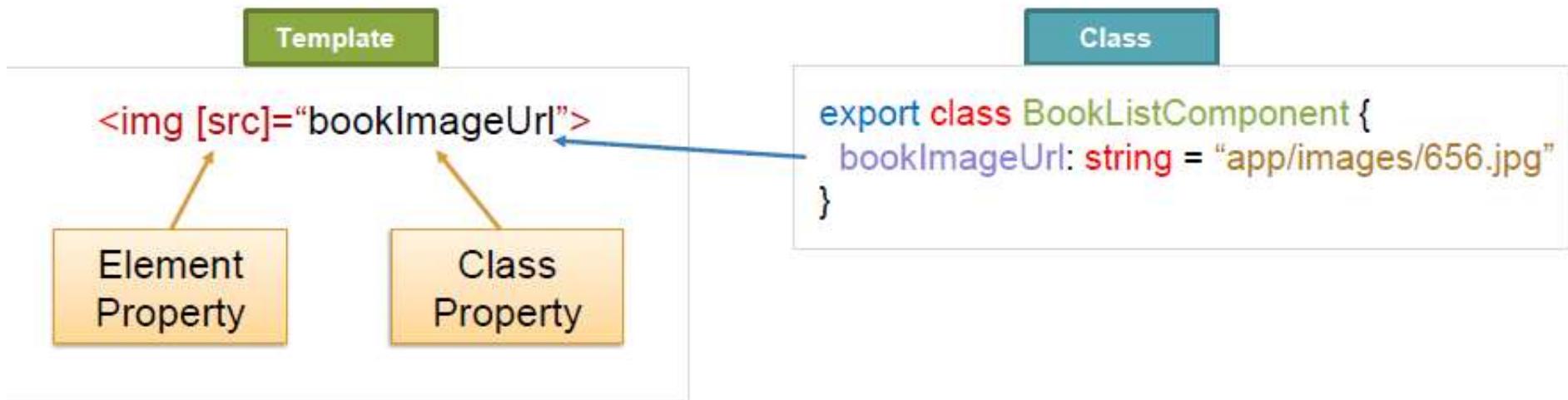
safe navigation operator (?) ,

`<p> Moje imię to: {{item?.name}}</p>` →

gdy item = null wyświetli się pusty tekst –
W consoli nie będzie błędów, związanych z
odwołaniem do obiektu pustego

1-way Binding

Property Binding



Event Binding

Template

```
(click)="hidelImage()"
```



Class

```
export class AppComponent {  
  hidelImage(): void {  
    this.showImage = !this.showImage;  
  }  
}
```

Binding - Widok do modelu danych

Podpięcie zdarzenia:

```
<div class="container">  
  
  <button type="button" (click)="clickListener($event)">Button</button>  
  
</div>
```

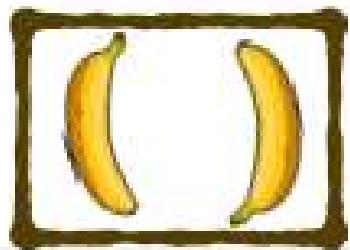
Naciśnięcie przycisku spowoduje wywołanie metody clickListener(\$event) i przekazanie parametru \$event jako argument funkcji. Podpiąć możemy wszystkie dostępne standardowo zdarzenia drzewa DOM

```
<button type="button"  (click) = "clickListener($event)" >    Przycisk  
1</button>  
<button type="button"  on-click = "clickListener($event)"> Przycisk  
2</button>
```

Dwukierunkowe wiązanie danych

- Połączenie komunikacji "z" i "do"
- Wykorzystujemy do tego dyrektywę `ngModel`

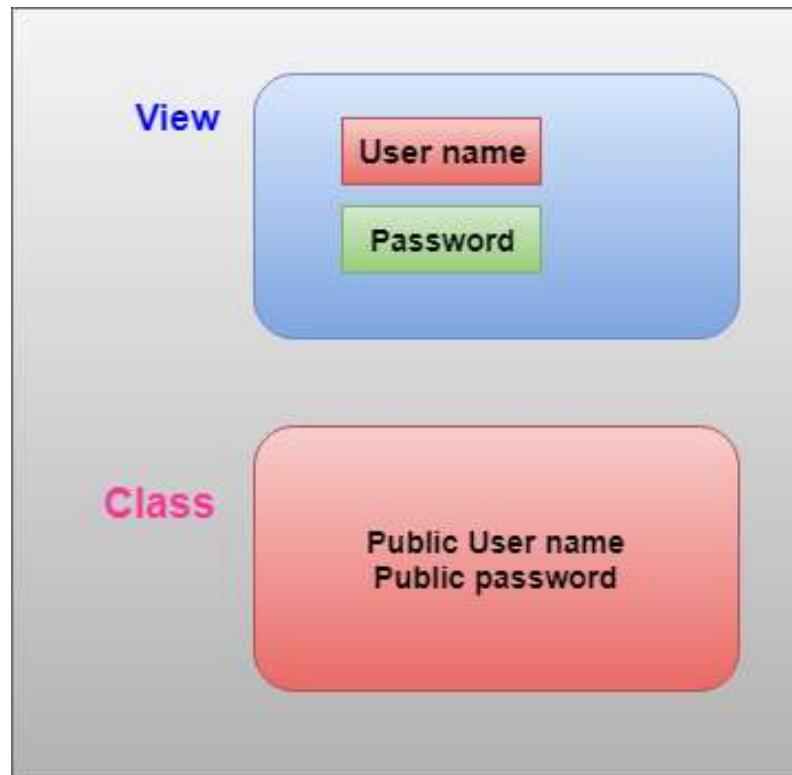
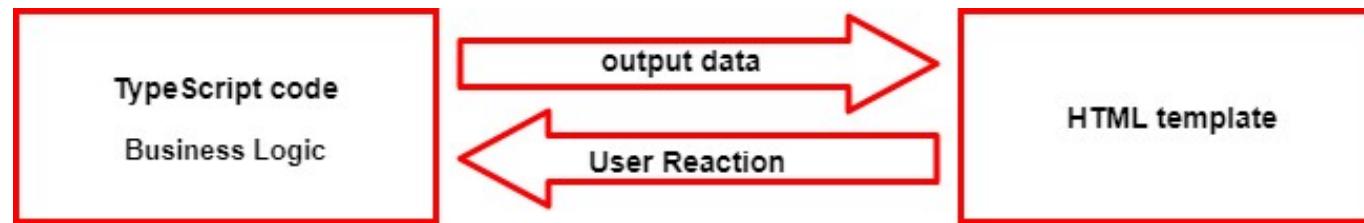
```
<input [ngModel]="name" (ngModelChange)="name=$event">  
  
// Skrócony zapis:  
<input [(ngModel)]="name">
```



[(ngModel)]

Dwukierunkowe wiązanie danych

`[(ngModel)]="[{właściwość Komponentu}]"`



```
<h2>Two-way Binding Przyklad</h2>
<input [(ngModel)]="fullName" />
<br/><br/>
<p> {{fullName}} </p>
```

Cykl życia

constructor

ngOnChanges

Zdarzenie wywoływanie przy każdej zmianie składowych `@Input()`

ngOnInit

Zdarzenie wywoływanie po inicjalizacji składowych `@Input()`,
po pierwszym zdarzeniu `ngOnChanges()`

ngDoCheck

Zdarzenie wywoływanie przy każdorazowej detekcji zmian
składowych komponentu (po wykryciu każdej zmiany)

ngAfterContentInit

Zdarzenie wywoływanie po zainicjowaniu zawartości komponentu

ngAfterContentChecked

po każdym sprawdzeniu zawartości komponentu

ngAfterViewInit

po zainicjowaniu widoków komponentu

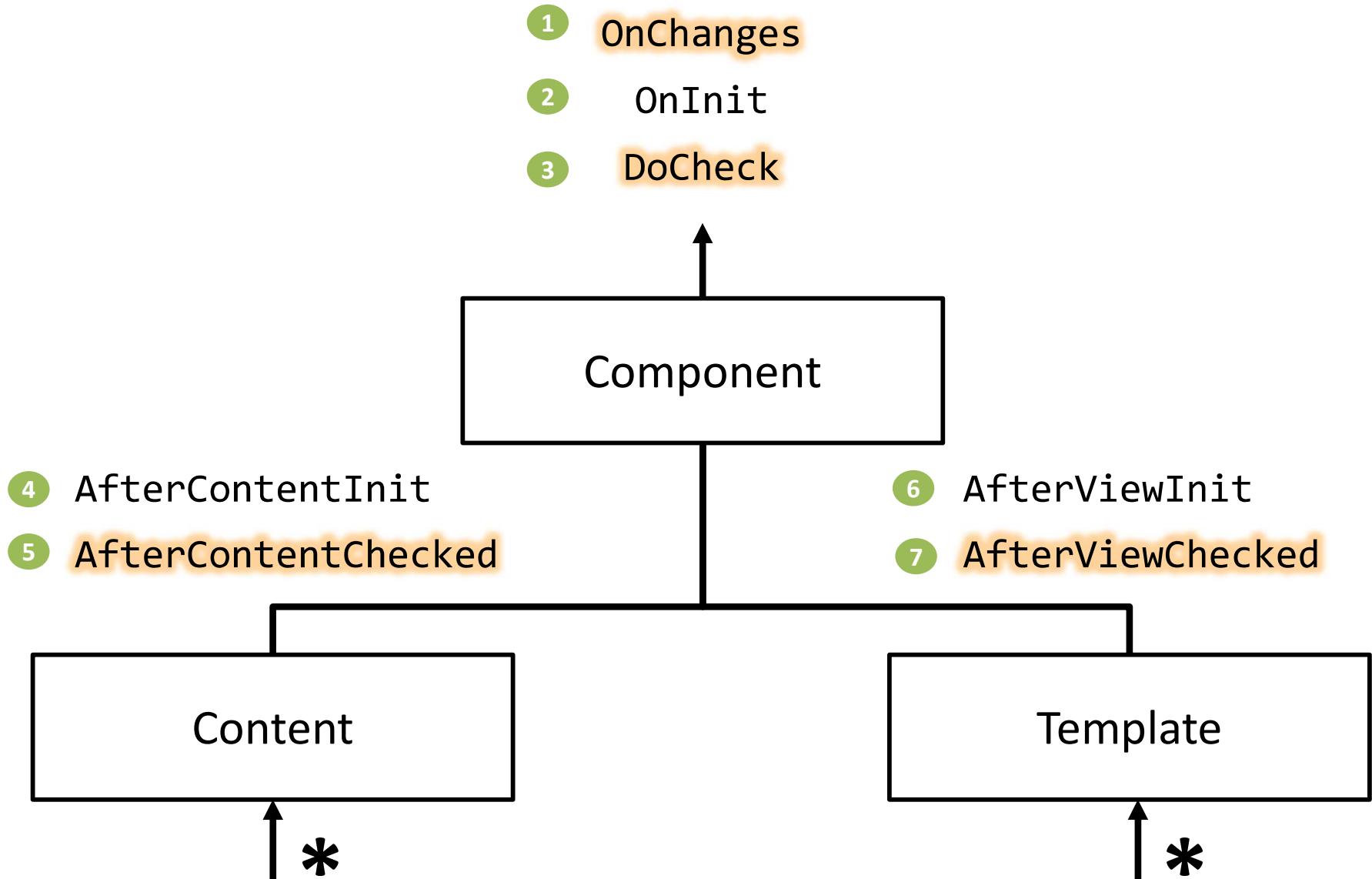
ngAfterViewChecked

po każdym sprawdzeniu widoku (ów) komponentu

ngOnDestroy

tuż przed zniszczeniem komponentu

Kolejność wywoływania



books-list.component.ts

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

```
@Component({  
  selector: 'bs-books-list'  
})
```

```
export class BooksListComponent implements OnInit {
```

```
  ngOnInit(): void {  
    console.log('Inside OnInit');
```

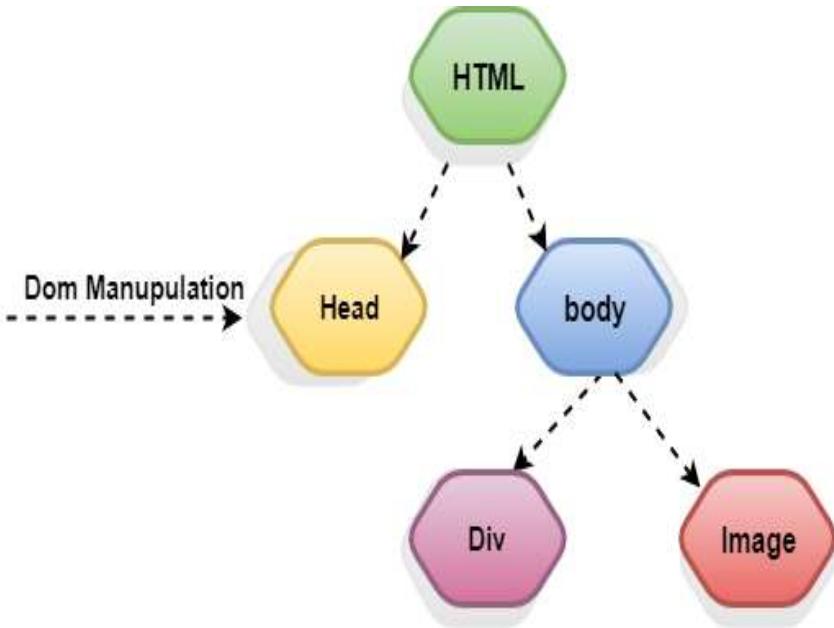
```
}
```

```
}
```



Dyrektywy

Dyrektywa modyfikuje DOM zmieniając jego wygląd lub zachowanie



W Angular wyróżniamy 3 rodzaje dyrektyw:

- 1. Komponenty** - dyrektywy z szablonami
- 2. Dyrektywy atrybutowe** - zmieniają zachowanie komponentu/elementu ale nie wpływają na jego szablon
- 3. Dyrektywy strukturalne** - zmieniają zachowanie komponentu/elementu przez modyfikację jego szablonu

Szablony Angulara są dynamiczne. Podczas ich renderowania Angular przetwarza DOM zgodnie z instrukcjami reprezentowanymi przez dyrektywy.

Typy Dyrektyw

Component

- <bs-app></bs-app>

```
welcome.component.ts  
@Component({  
  selector: 'bs-welcome'  
})
```

Structural

- NgIf, NgFor, NgStyle

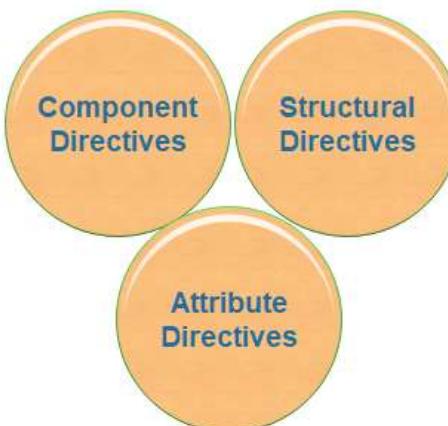
```
app.component.ts  
@Component({  
  selector: 'bs-app',  
  template: <bs-welcome></bs-welcome>  
})
```

Attribute

- <p highlight></p>

Modyfikuje strukturę DOM

Modyfikuje atrybuty elementów DOM



Wyświetlanie lub ukrywanie elementów DOM

```
<p> Wersja z atrybutem hidden</p>
<div [hidden]="ksiazki.length == 0" >
    lista Ksiazek
</div>
<div [hidden]="ksiazki.length > 0">
    Brak ksiazek na liscie
</div>
```

Atrybut [hidden]

```
<div *ngIf="ksiazki.length > 0" >
    lista Ksiazek
</div>

<div *ngIf="ksiazki.length == 0">
    Bark ksiazek na liscie
</div>
```

Dyrektywa *ngIf

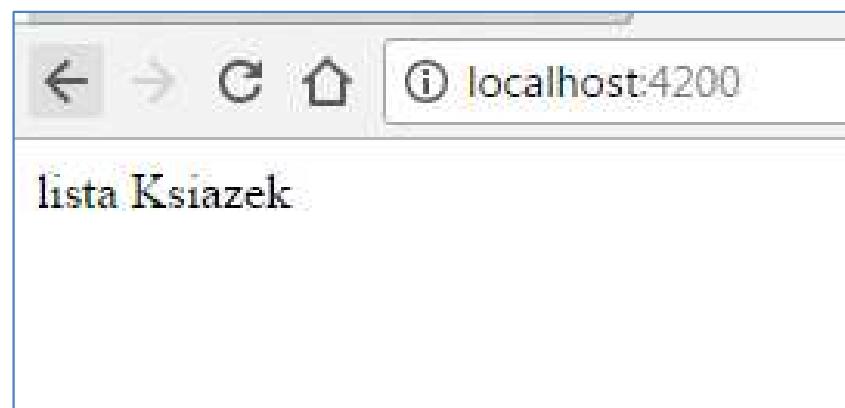
***ngIf="*<condition>*"**

books-list.component.html

showImage –
wartosc boolowska

ngIf w praktyce – sposoby implementacji

```
<div *ngIf="ksiazki.length > 0">  
    lista Ksiazek  
</div>  
  
<div *ngIf="ksiazki.length == 0">  
    Brak ksiazek na liscie  
</div>
```



```
<div *ngIf="ksiazki.length > 0; else nobooks">  
    lista Ksiazek  
</div>  
  
<ng-template #nobooks>  
    Brak ksiazek na liscie  
</ng-template>
```

```
export class PierwszyPrzykladDyrektywComponent implements OnInit {  
  ksiazki = ["Pierwsz ksiazka", "Druga ksiazka", "Trzecie ksiazka"];
```

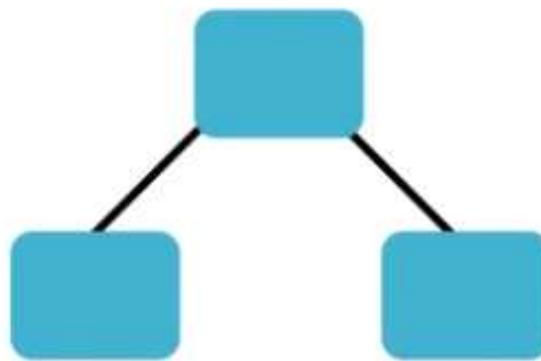
```
<div *ngIf="ksiazki.length > 0; then booksList else nobooks" ></div>  
<ng-template #bookList>  
    lista Ksiazek  
</ng-template>  
<ng-template #nobooks>  
    Brak ksiazek na liscie  
</ng-template>
```

Zarzadzanie modelem DOM – kiedy która technika

[hidden]

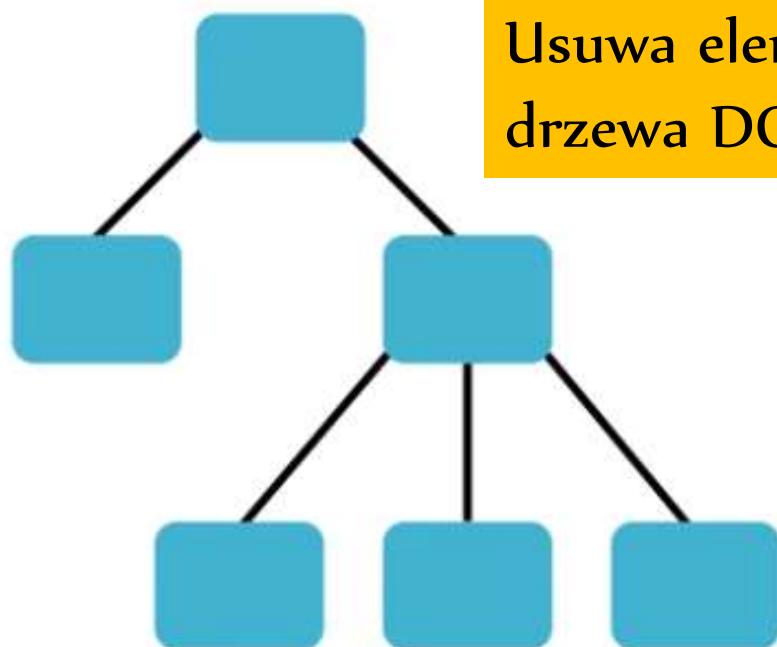
*ngIf

Ukrywa element w DOM



Dla drzewa z małą
ilością elementów

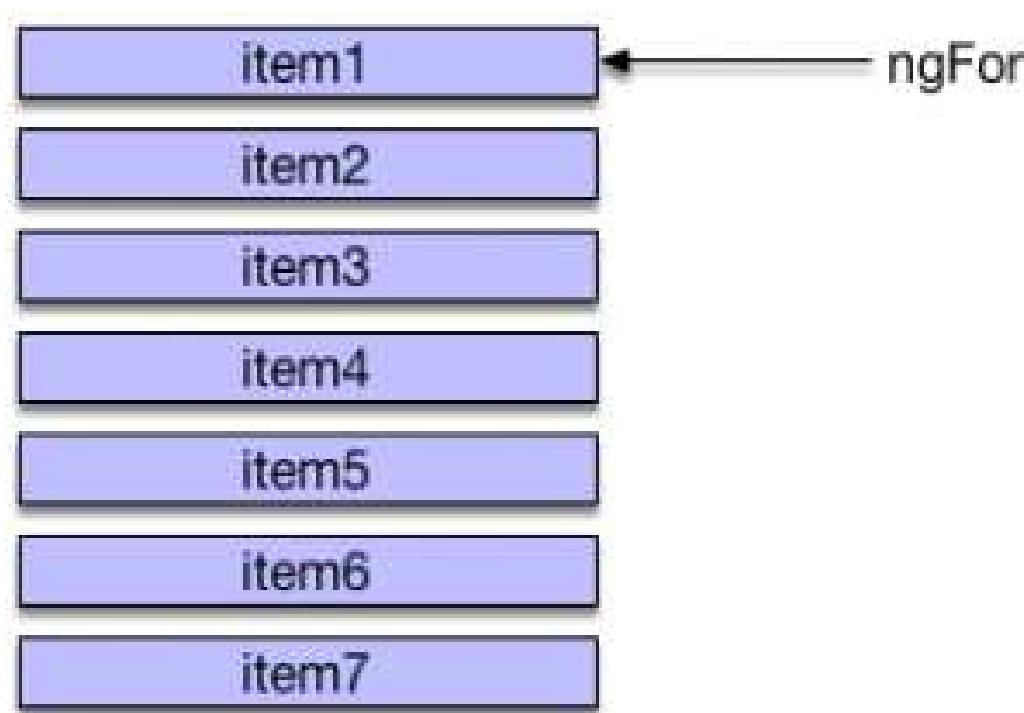
Usuwa element z
drzewa DOM



Dla drzewa z dużą
ilością elementów

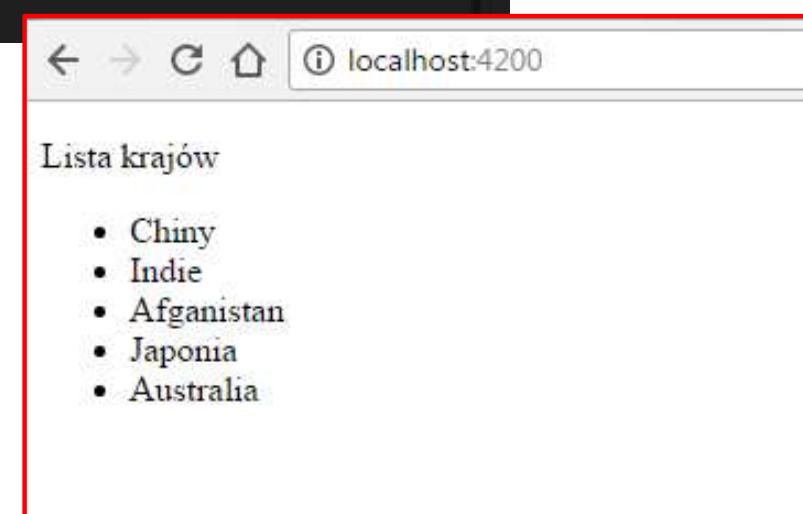
Angular *ngFor Directive

```
<li *ngFor="let item of items ;">....</li>
```



Wyświetlanie zawartości tablicy - ngFor

```
@Component({
  selector: 'app-root',
// templateUrl: './app.component.html',
  template: `
    <p> Lista krajów </p>
    <ul>
      <li *ngFor="let tab of tablica"> {{tab}} </li>
    </ul> `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  tablica = ["Chiny", "Indie", "Afganistan", "Japonia", "Australia"]
}
```



Wypisanie tablicy – realizacja w komponencie

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  name = 'Grzegorz Rogus';

  ksiazki = [{title:"Node.js, MongoDB, AngularJS. Kompendium wiedzy", price:99},
  {title:"Tworzenie gier internetowych. Receptury", price:49},
  {title:"Web 2.0 Architectures. What entrepreneurs and information architects need to know", price:84.92},
  {title:"React dla zaawansowanych", price:45},
  {title:"Spring MVC 4. Projektowanie zaawansowanych aplikacji WWW", price:102}];
```

```
  getKsiazki(){
    return this.ksiazki;
  }
}
```

Komponent w Angular10

lub tak

```
<ul>
  <li *ngFor="let ksiazka of getKsiazki()">
    {{ksiazka.title}}  w cenie  {{ksiazka.price}}
  </li>
</ul>
```

```
<div style="text-align:center">
  <h1>
    Witaj  {{name}}!
  </h1>

</div>

<p>
  Ksiazki warte polecenia na temat technologii Webowych:
</p>
<ul>
  <li *ngFor="let ksiazka of ksiazki">
    {{ksiazka.title}}  w cenie  {{ksiazka.price}}
  </li>
</ul>
```

Witaj Grzegorz Rogus!

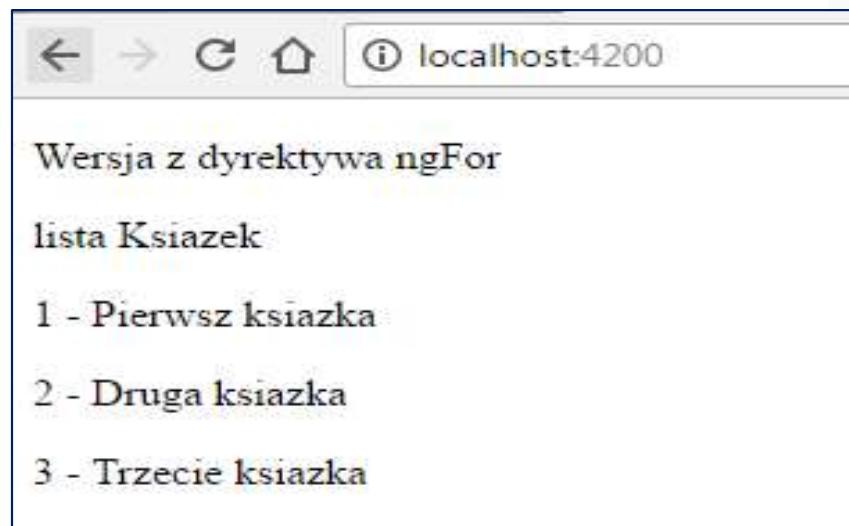
Ksiazki warte polecenia na temat technologii Webowych:

- Node.js, MongoDB, AngularJS. Kompendium wiedzy w cenie 99
- Tworzenie gier internetowych. Receptury w cenie 49
- Web 2.0 Architectures. What entrepreneurs and information architects need to know w cenie 84.92
- React dla zaawansowanych w cenie 45
- Spring MVC 4. Projektowanie zaawansowanych aplikacji WWW w cenie 102

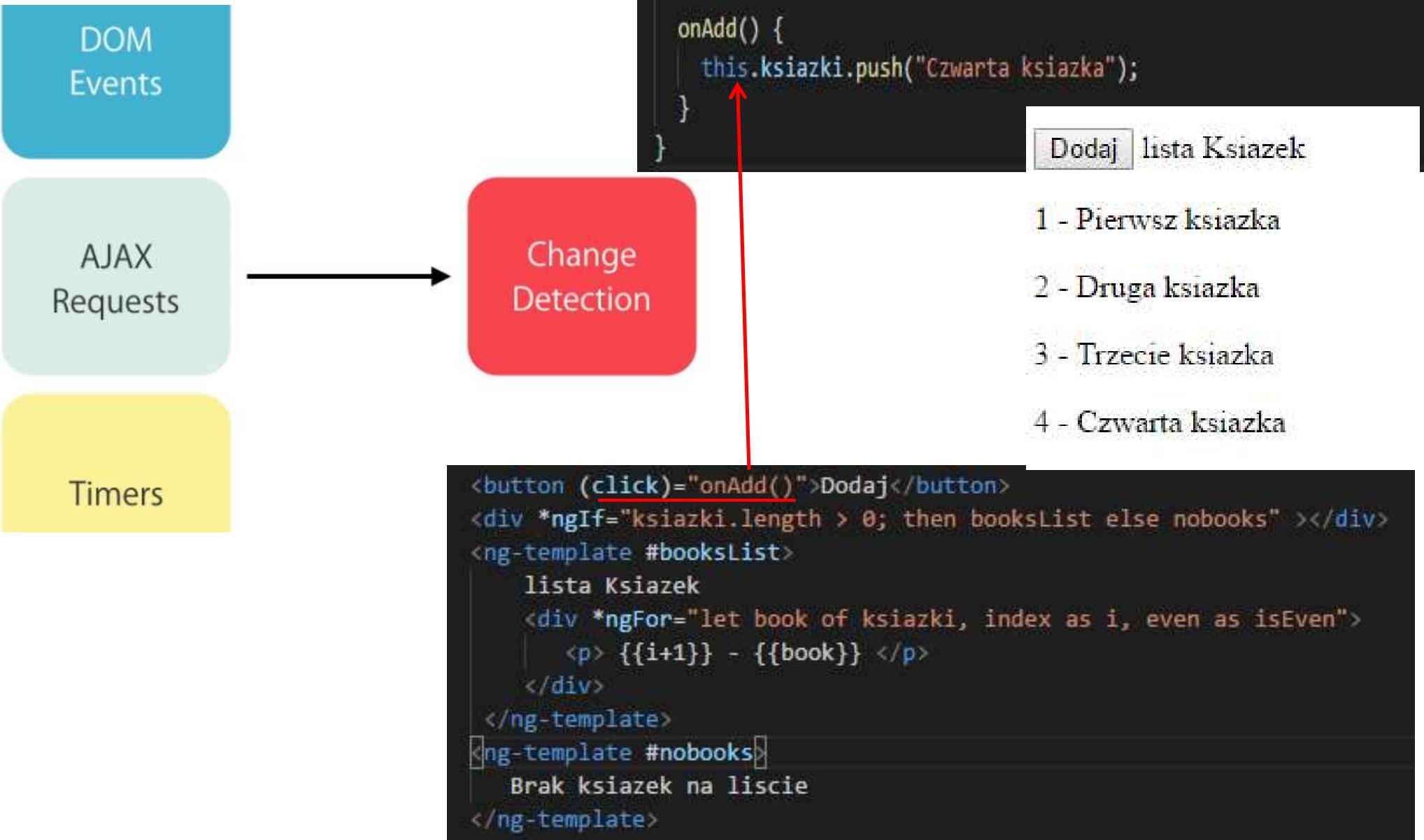
Wyświetlanie zawartości tablicy - ngFor

```
<p> Wersja z dyrektywa ngFor</p>
<div *ngIf="ksiazki.length > 0; then booksList else nobooks" ></div>
<ng-template #booksList>
  lista Ksiazek
  <div *ngFor="let book of ksiazki, index as i">
    <p> {{i+1}} - {{book}} </p>
  </div>
</ng-template>
<ng-template #nobooks>
  Brak ksiazek na liscie
</ng-template>
```

- index: number : The index of the current item in the iterable.
- first: boolean : True when the item is the first item in the iterable.
- last: boolean : True when the item is the last item in the iterable.
- even: boolean : True when the item has an even index in the iterable.
- odd: boolean : True when the item has an odd index in the iterable.



Źródła zmian w systemie



```
export class PierwszyPrzykladDyrektywComponent {
  ksiazki = ["Pierwsz ksiazka", "Druga ksiazka", "Trzecie ksiazka"];

  onAdd() {
    this.ksiazki.push("Czwarta ksiazka");
  }
  onRemove(book) {
    let index = this.ksiazki.indexOf(book);
    this.ksiazki.splice(index, 1);
  }
}
```

Wersja z dyrektywą ngFor

Dodaj lista Ksiazek

1 - Druga ksiazka

Usun

2 - Trzecie ksiazka

Usun

3 - Czwarta ksiazka

Usun

```
<button (click)="onAdd()">Dodaj</button>
<div *ngIf="ksiazki.length > 0; then booksList else nobooks" ></div>
<ng-template #booksList>
  lista Ksiazek
  <div *ngFor="let book of ksiazki, index as i, even as isEven">
    <p> {{i+1}} - {{book}} </p>
    <button (click)="onRemove(book)">Usun</button>
  </div>
</ng-template>
<ng-template #nobooks>
  Brak ksiazek na liscie
</ng-template>
```

Dynamiczna stylizacja komponentów

Dyrektywy atrybutowe

- `ngStyle`:

```
<div [ngStyle]="'font-size': mySize+'px'">...</div>
<div [ngStyle]="'font-size.px': mySize}>...</div>
<div [ngStyle]="myStyle">...</div>
```

- `ngClass`:

```
<div [ngClass]="'first second'">...</div>
<div [ngClass]=["first", "second"]>...</div>
<div [ngClass]={"first: true, second: -1, third: 0}">...
```

Pozwalają na dynamiczną stylizację poszczególnych elementów DOM

ngStyle

Statyczne przypisanie

```
<div [ngStyle] = "{'background-color': 'green'}"></div>
```

```
<div [ngStyle] = "{'background-color': ksiazki.length == 3 ? 'green' : 'red'}">  
</div>
```

Dynamiczne przypisanie

```
<div [ngStyle] = "{'property': 'value'}"></div>
```

tak albo tak

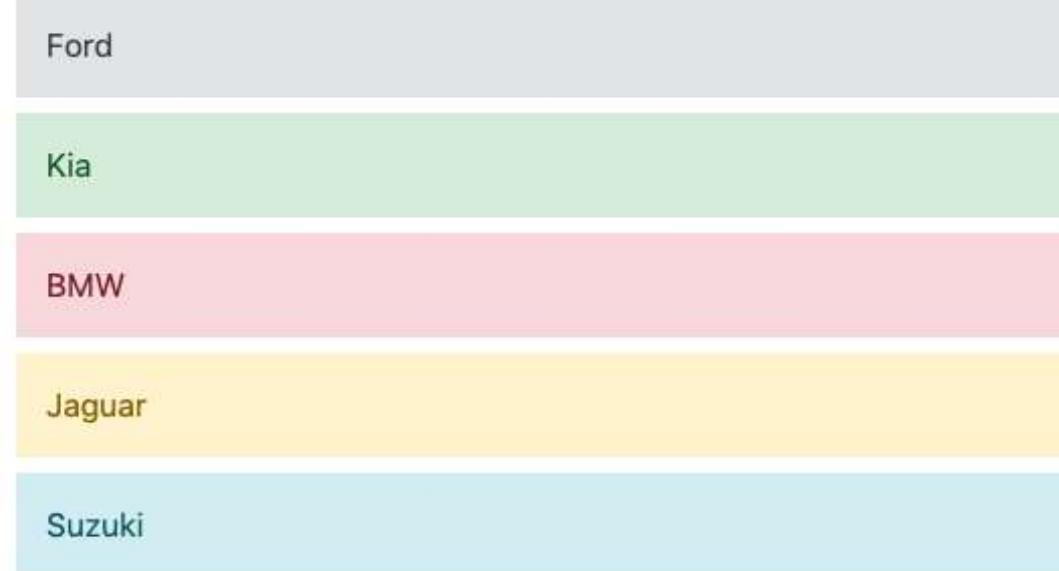
```
<div [style.property] = "{value}"></div>
```

```
<ul *ngFor="let person of ludzie">  
<li [ngStyle] = "{'font-size.px': 24}"  
     [style.color] = "getColor(person.kraj)">  
     {{ person.imie }} ({{ person.kraj }}) </li> </ul>
```

Angular ngClass Directive

```
<element ng-class=""expression""></element>
```

```
<div class="container">
<div *ngFor="let car of cars" [ngClass] ="{>
  'alert-secondary':car.name==='Ford',
  'alert-success':car.name==='Kia',
  'alert-danger':car.name==='BMW',
  'alert-warning':car.name==='Jaguar',
  'alert-info':car.name==='Suzuki'
}>
  {{car.name}}
</div>
</div>
```



ngClass

books-list.component.html

```
<div [ngClass]={"'redClass': showImage, 'yellowClass': !showImage}">
```

Dodaje lub usuwa klasy CSS dla elementu HTML w zależności od wartości zmiennej

Rotacja klasy elementu (klasa underline jest dodana tylko wtedy gdy `isUnderlined = true`):

```
<div class="container">  
  
  <p [class.underline]="isUnderlined"></p>  
  
</div>
```

ngClass – Zarządzanie wieloma klasami

```
public getClassNames() {  
  
    return {  
  
        'underline': this.isUnderlined,  
  
        'active': this.isActive  
  
    }  
}
```

Natomiast w szablonie wystarczy:

```
<div class="container">  
  
    <p [ngClass]="getclassNames()"></p>  
  
</div>
```

ngStyle ngClass - przykład zastosowania

test dyrektyw ngStyle ngClass

Dodaj

lista Ksiazek 2

- 1 - Pierwsza ksiazka Usun
- 2 - Czwarta ksiazka Usun

test dyrektyw ngStyle ngClass

Dodaj

lista Ksiazek 5

- 1 - Pierwsza ksiazka Usun
- 2 - Druga ksiazka Usun
- 3 - Trzecie ksiazka Usun
- 4 - Czwarta ksiazka Usun
- 5 - Czwarta ksiazka Usun

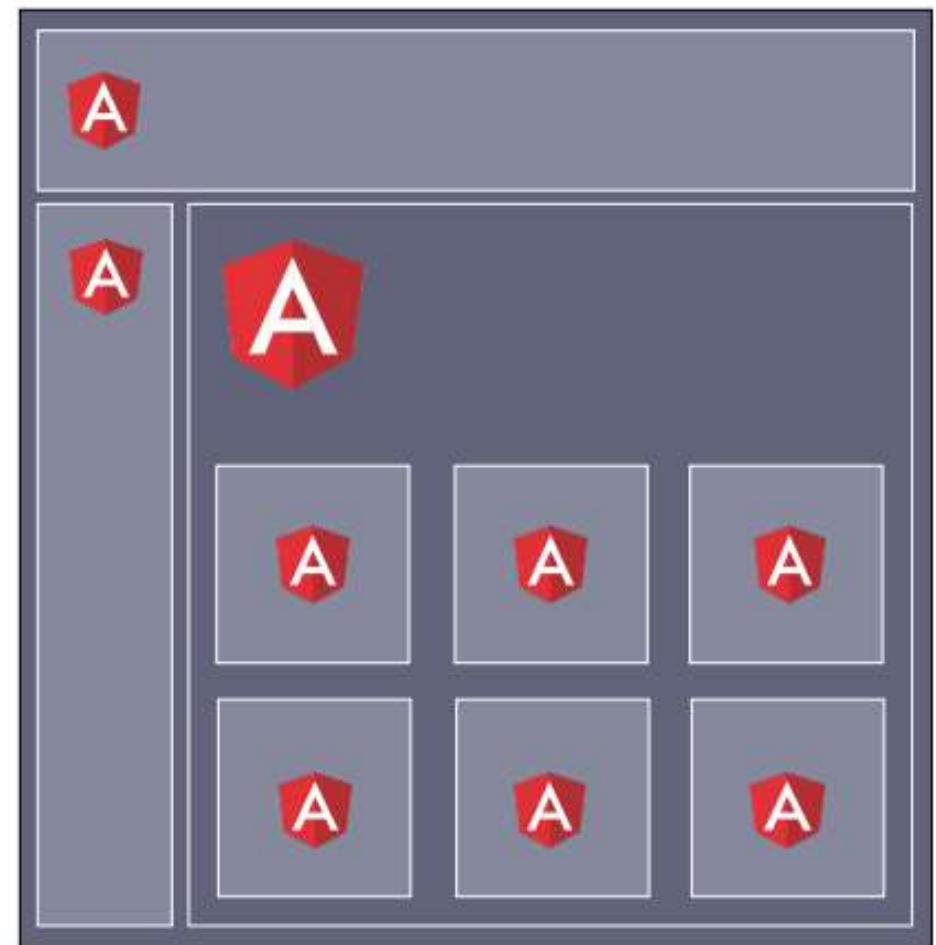
```
getColor(): string {
    return this.ksiazki.length > 3 ? 'red' : 'green';
}

<ng-template #booksList>
    <p [ngStyle] ='{color: getColor()}'> lista Ksiazek {{ksiazki.length}} </p>
    <ul>
        <div *ngFor="let book of ksiazki, index as i, even as isEven">
            <li [ngClass] ='{oddtest:isEven, eventest: !isEven}'>
                {{i+1}} - {{book}} <button (click) = "onRemove(book)"> Usun </button>
            </li>
        </div>
    </ul>
</ng-template>

.oddtest{
    background-color: grey;
}
.eventest {
    background-color: linen;
}
```

Aplikacja składa się z komponentów

W Angular aplikacja zbudowana jest z drzewa komponentów. Każdy komponent może mieć zestaw komponentów „dzieci” oraz rodzica. Naszym głównym komponentem jest jego korzeń, tzw. root component.



Połączenie pomiędzy komponentami

```
@Component({  
  selector: "my-child",  
  template: "This is a child component"  
})  
export class ChildComponent {}
```

POTOMEK

Dodaj selektor
komponentu dziecka w
szablonie rodzica

```
@Component({  
  selector: "app",  
  template: "<my-child></my-child>",  
})  
export class AppComponent {}
```

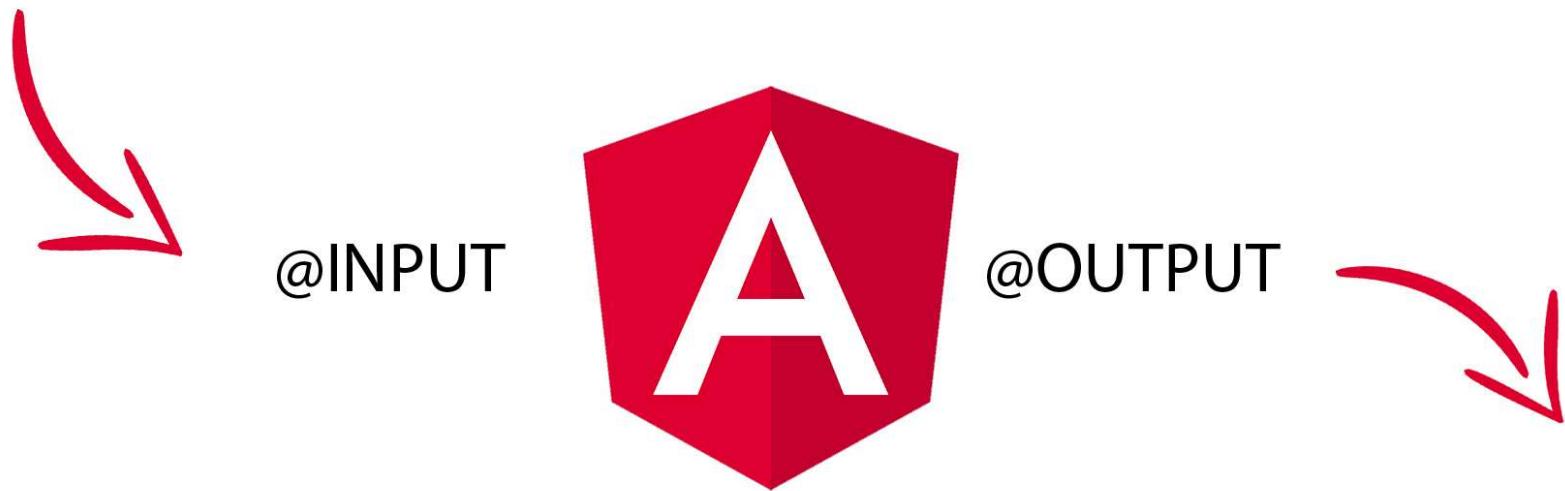
RODZIC

W jaki sposób komponenty się komunikują??

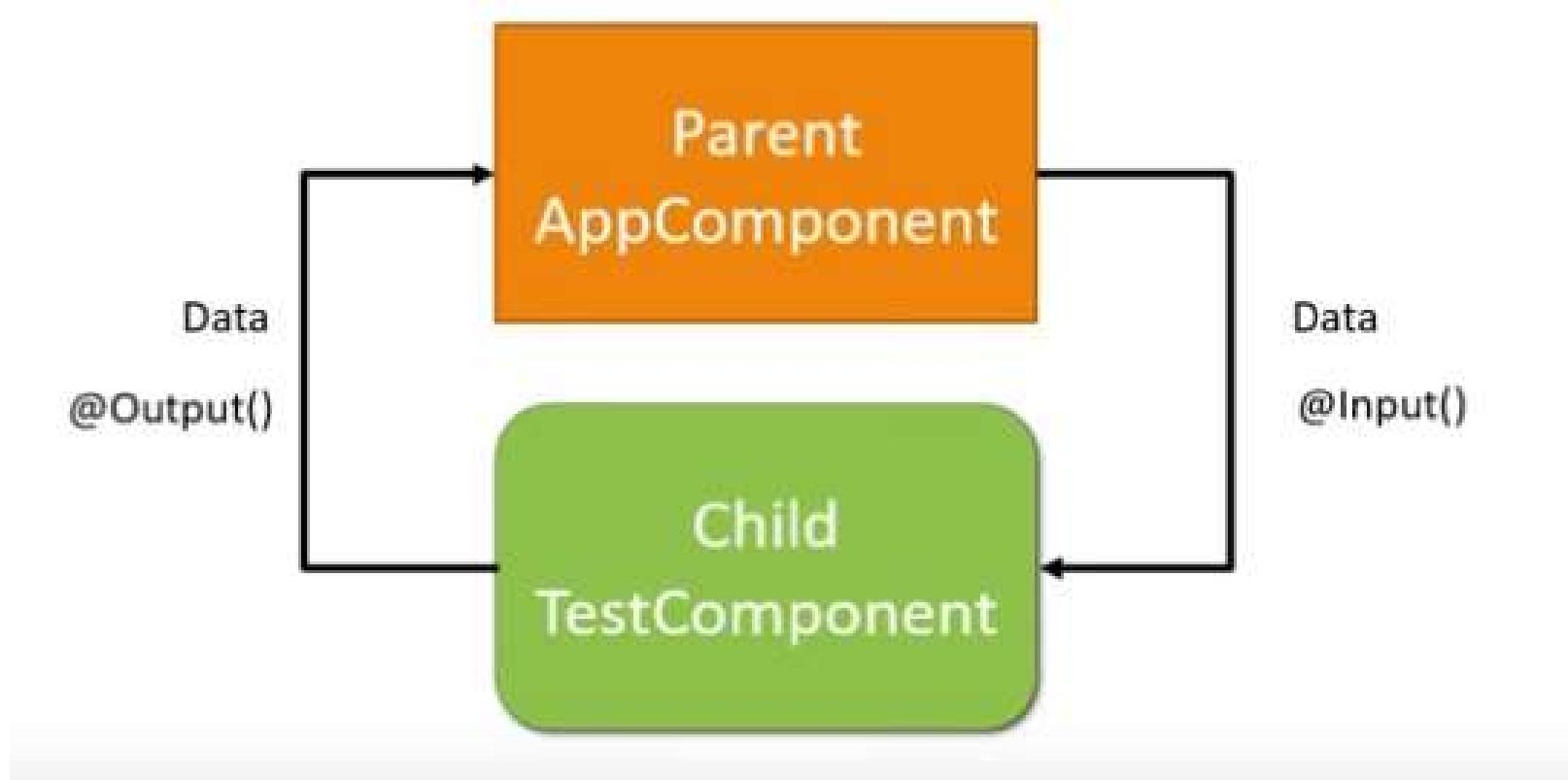
1. Inputs i Outputs (tylko powiązane)

2. Usługi (wszystkie)

Komunikacja miedzy komponentami



Komunikacja między komponentami



Input

Container Component

Nested Component

Input

```
@Input() reviews: number;
```

komponent rodzica ma możliwość przekazania do dziecka danych, które mogą determinować zachowanie komponentu lub w ogóle – pozwolić na jego odpowiednie wyrenderowanie. Jest to możliwe dzięki adnotacji `@Input()`

Komponent: komunikacja "do"

```
@Component({  
  selector: 'test-input',  
  template: '....'  
})  
export class GR_Test {  
  @Input() item: myType;  
}
```

```
// lub:  
@Component({  
  selector: 'test-input',  
  inputs: ['item'],  
  template: '....'  
})  
export class GR_Test {  
  item: myType;  
}
```

```
// przekazywanie przez zmienną:  
<test-input [item]="myItem-GR"></test-input>  
  
// przekazywanie wartości bezpośrednio  
<test-input item="myItem-GR"></test-input>
```

Krok 1. Dodanie nowej własności typu input



```
import {Component, Input} from "@angular/core";  
  
@Component({  
  selector: "my-child",  
  template: "This is a child component with a message: {{ message }}"  
})  
export class ChildComponent {  
  @Input() message: string;  
}
```

A large red arrow pointing upwards, indicating the flow of the code snippet above it.

Krok 2: Powiąż zmienną rodzica z tą właściwością

```
@Component({  
  selector: "app",  
  template: `<my-child [message]=${pozdrowienia}></my-child>`,  
  directives: [ChildComponent]  
})  
export class AppComponent {  
  pozdrowienia = "Pozdrowienia od GR";  
}
```



Input

child_component.ts

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

@Component({
  selector: 'child'
})

export class ChildComponent {
  @Input() reviews: number;
}
```

parent_component.ts

```
export class ParentComponent {
  books: any[] = [
    {
      bookReviews: 15
    }
  ]
}
```

child_component.html

```
<div>
  <p>{{ reviews }}</p>
</div>
```

parent_component.html

```
<div><h1>Parent Title</h1>
<p>body text...</p>
<child [reviews]="book.bookReviews">
</child>
</div>
```

Komponent: komunikacja "z" - zdarzenia

```
// lub:  
  
@Component({  
  selector: 'test-output',  
  template: '...'  
})  
export class Hello {  
  @Output() completed:  
  EventEmitter<boolean> = new EventEmitter<boolean>();  
}
```

```
@Component({  
  selector: 'test-output',  
  outputs: ['completed'],  
  ...  
})
```

```
<test-output (completed)="saveProgress(item)"></test-output>  
  
// alternatywna składnia - dlatego nie prefixujemy zdarzeń "on"  
<test-output on-completed="saveProgress(item)"></test-output>
```

Implementacja

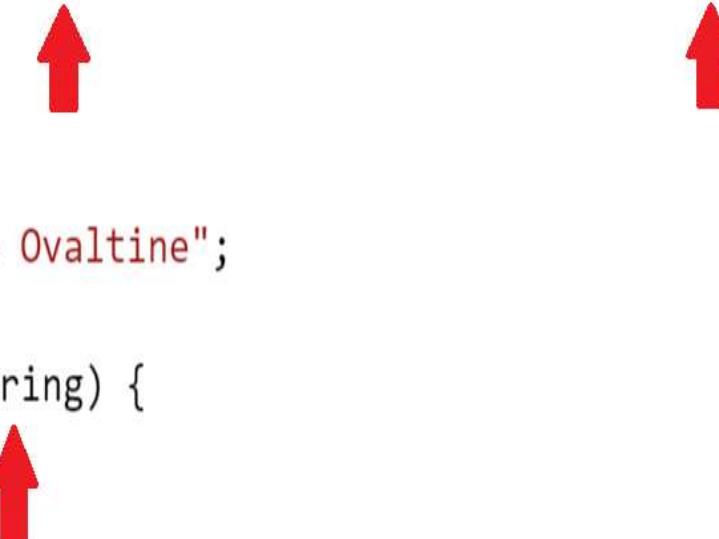
```
import {Component, Input, EventEmitter, Output} from "@angular/core";  
  
@Component({  
    selector: "my-child",  
    template: `This is a child component with a message: {{ message }}`  
})  
export class ChildComponent {  
    @Input() message: string;  
    @Output() signaledIsHungry = new EventEmitter<string>();  
}
```

EventEmitter jest klasą generyczną – szablonem.
parametr jaki będzie przekazywał w postaci argumentu, będzie typu string.

Dodanie funkcji subskrybujacej

```
@Component({
  selector: "app",
  template: `<my-child [message]="marketingPhrase"
                (signaledIsHungry)="didSignalIsHungry($event)"></my-child>`,
  directives: [ChildComponent]
})
export class AppComponent {
  marketingPhrase = "Drink more Ovaltine";

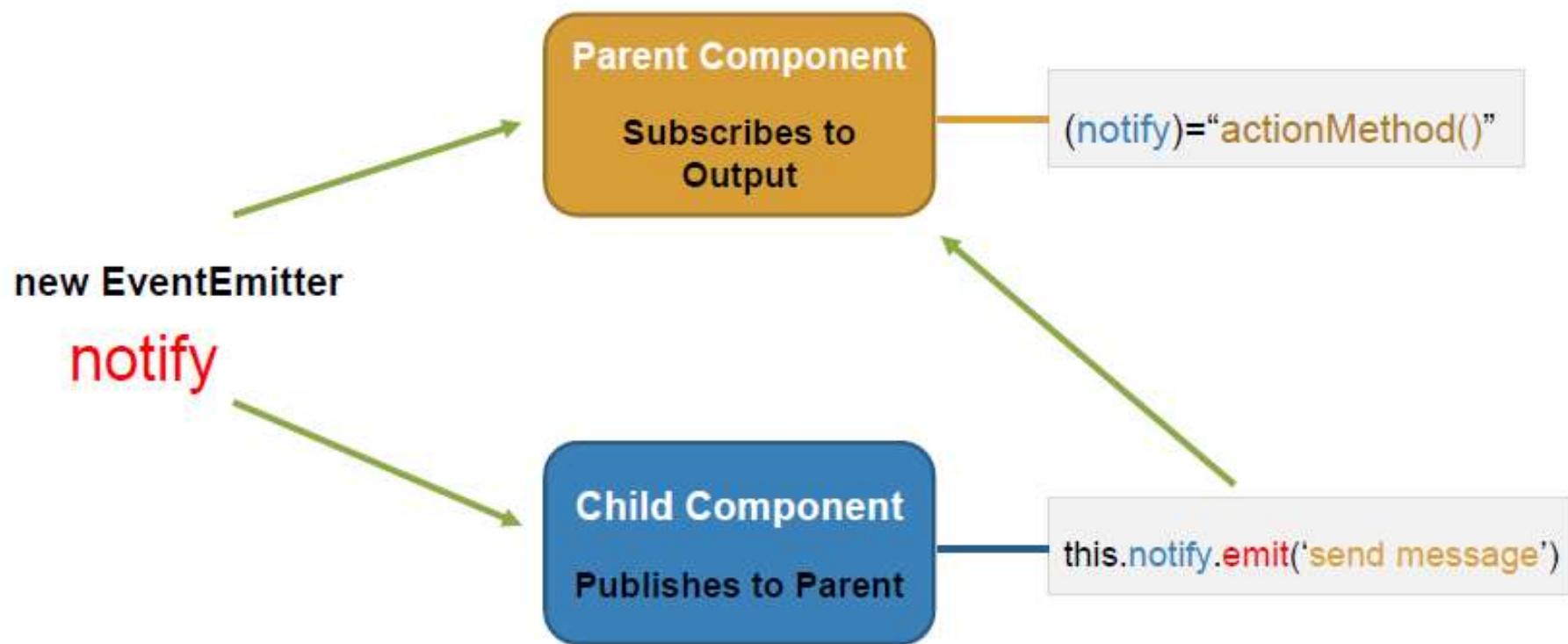
  didSignalIsHungry(message: string) {
    console.log(message);
  }
}
```



W tym kodzie dodaliśmy funkcję `didSignalIsHungry`, która jest wywoływana, gdy zdarzenie `signaledIsHungry` jest emitowane z komponentu dziecięcego.

W komunikacie z komponentem dziecięcym, który emituje zdarzenie, możemy dodać funkcję obiektu, aby ją skorzystać z tego zdarzenia.

Subskrypcja zdarzeń



Output

child_component.ts

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

export class ChildComponent {
  Output() notify: EventEmitter<string> =
    new EventEmitter<string>();

  onClick(): void {
    this.notify.emit('Message from child');
  }
}
```

parent_component.ts

```
export class ParentComponent {

  onNotifyClicked(message: string): void {
    this.showMessage = message;
  }
}
```

parent_component.html

```
<div><h1>Parent Title</h1>
<p>{{ showMessage }}</p>
<child (notify)="onNotifyClicked($event)">
</child>
</div>
```

child_component.html

```
<div
  (click)="onClick()">
<button>Click Me</button> </div>
```



Usługi

Service

Klasa do specyficznych celów:

- dzielenia danych
- Implementacja logiki aplikacyjnej/biznesowej
- Zewnętrzna interakcja – odczyty danych z serwera

Usługi

- Klasy implementujące funkcje potrzebne w aplikacji
 - Pojedyncza usługa powinna odpowiadać za konkretną funkcjonalność
- Podział logiki między klasy komponentów i usług
 - Klasa komponentu powinna zawierać logikę specyficzną dla widoku
 - Właściwości i metody do wiązania danych z widokiem
 - Pośredniczenie między widokiem a logiką biznesową (modelem)
 - Klasy usług nie powinny zależeć od widoków
 - Komunikacja z backendem, walidacja, obsługa logu itd.
- Usługi są udostępniane komponentom i innym usługom przez wstrzykiwanie zależności (ang. dependency injection)
 - Dekorator @Injectable oznacza klasy będące usługami oraz klasy i komponenty zależne od usługi
 - Wstrzykiwanie realizowane przez typowany parametr konstruktora
 - Dostawcy usług (ang. providers), typowo klasy usług, rejestrowani na poziomie modułów lub komponentów

Implementacja usług - uwagi

- Klasy komponentów powinny być szczupłe, bez nadmiaru (kodu, funkcjonalności). Nie pobierają danych z serwera, walidują danych wejściowych od użytkownika czy zapisują logi bezpośrednio do konsoli. Takie zadania są delegowane do usług.
- **Zadaniem komponentu jest udostępnienie użytkownikowi danej funkcjonalności i nic ponad to.**
- Komponent pośredniczy pomiędzy widokiem (renderowanym na bazie szablonu) i logiką aplikacji (która często zawiera jakąś wiedzę o modelu).
- Dobry komponent prezentuje właściwości i metody do wiązania danych. Wszystkie nietrywialne zadania są delegowane do usług.
- Angular pomaga nam przestrzegać tych wytycznych poprzez ułatwianie nam podzielenia logiki aplikacji na usługi i uczynienie tych usług dostępnymi w komponentach poprzez *wstrzykiwanie zależności*

Usługi - service

- Generacja automatyczna ng –g s nazwaUsługi

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

@Injectable()
export class CountryService {

    constructor() { }

    getCountry() {
        return ["Polska", "niemcy", "Rosja", "czechy"];
    }
}

import { CountryService } from './country.service';
import { Component, OnInit } from '@angular/core';

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

    buttonStatus = true;
    kraje;

    constructor() {
        let service = new CountryService();
        this.kraje = service.getCountry();
    }
}
```

Usługa z DI i bez - porównanie

Without DI

```
class Car{
    engine;
    tires;
    constructor()
    {
        this.engine = new Engine();
        this.tires = new Tires();
    }
}
```

With DI

```
class Car{
    engine;
    tires;
    constructor(engine, tires)
    {
        this.engine = engine;
        this.tires = tires;
    }
}
```

```
class Engine{
    constructor(){}
}
class Tires{
    constructor(){}
}
```

```
class Car{
    engine;
    tires;
    constructor()
    {
        this.engine = new Engine();
        this.tires = new Tires();
    }
}
```

Wstrzykiwanie zależności

Klasę rozszerzoną dekoratorem (*np. @Injectable*)

```
@Injectable()
class TodosService {
    constructor() {}
}
```

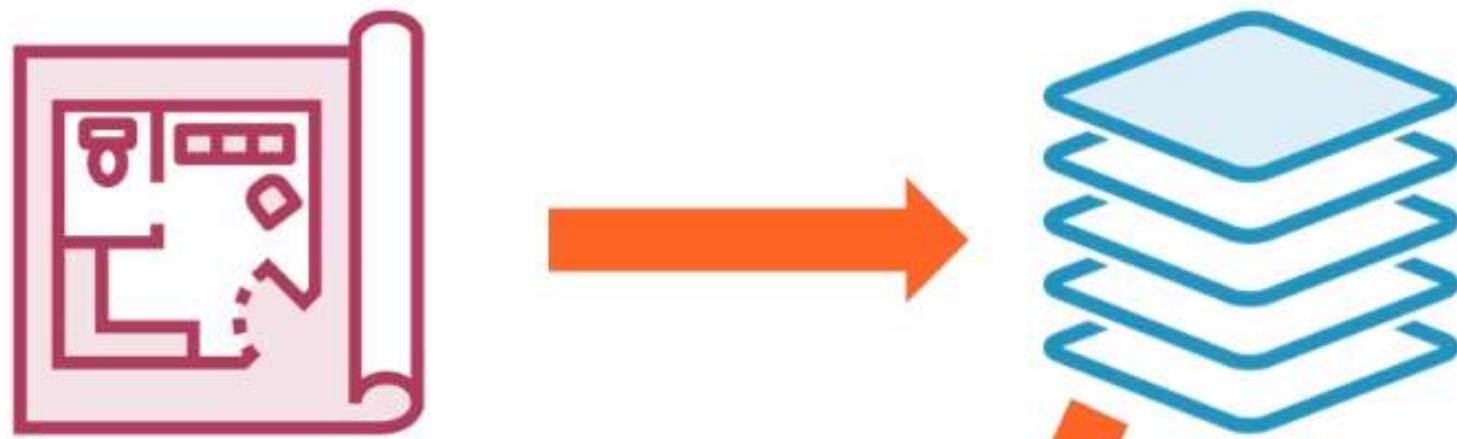
... i zarejestrowaną jako *provider*

```
@NgModule({
  providers: [
    TodoService,
```

... możemy wstrzyknąć do konstruktorów innych klas

```
class TodoItemComponent {
  constructor(ts: TodosService) {} // skrócony zapis
// constructor(@Inject(TodosService) ts) {} // pełen zapis
}
```

Wstrzykiwanie serwisu do Komponentu



Provider

Injector

```
export class DashboardComponent {  
  constructor(private dataService: DataService) { }  
}
```

Wstrzykiwanie zależności Dependency Injection

```
constructor() {  
    let service = new CoursesService();  
    this.courses = service.getCourses();  
}
```

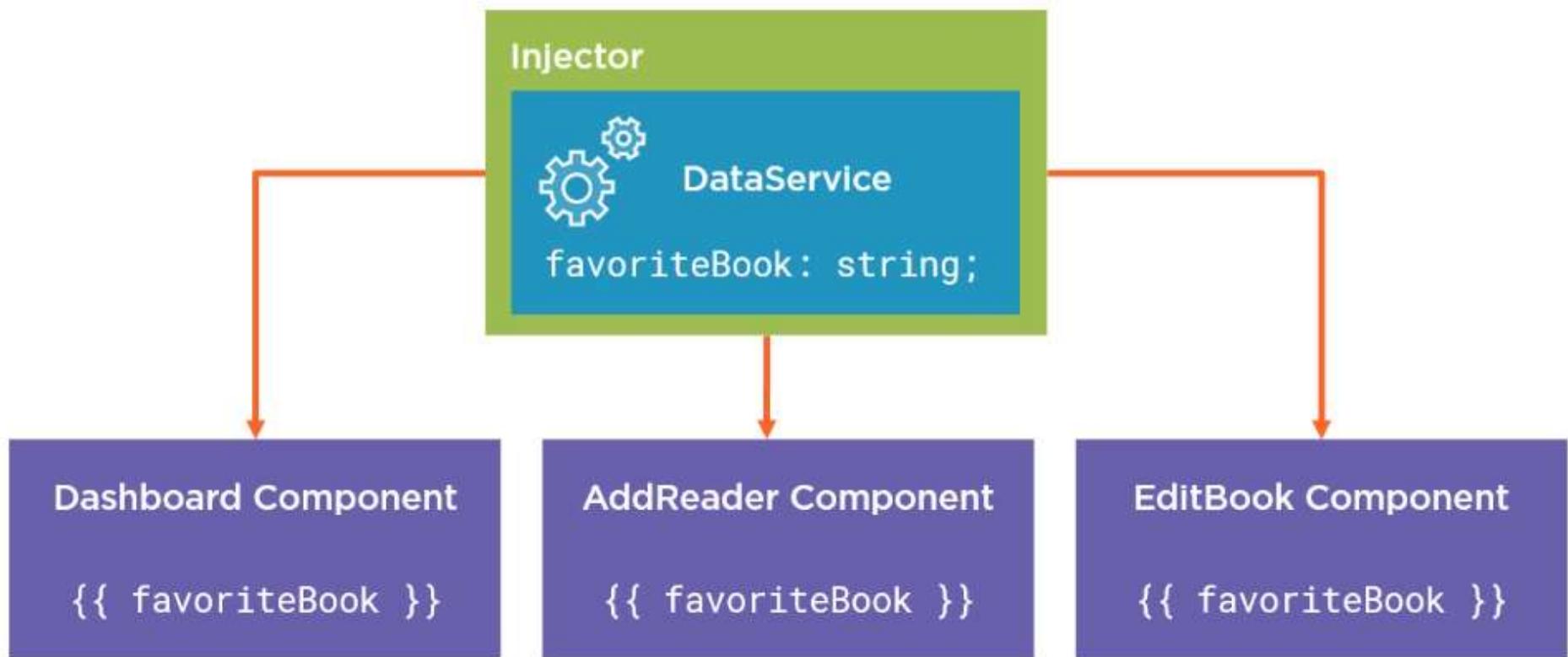
ZLE

VS

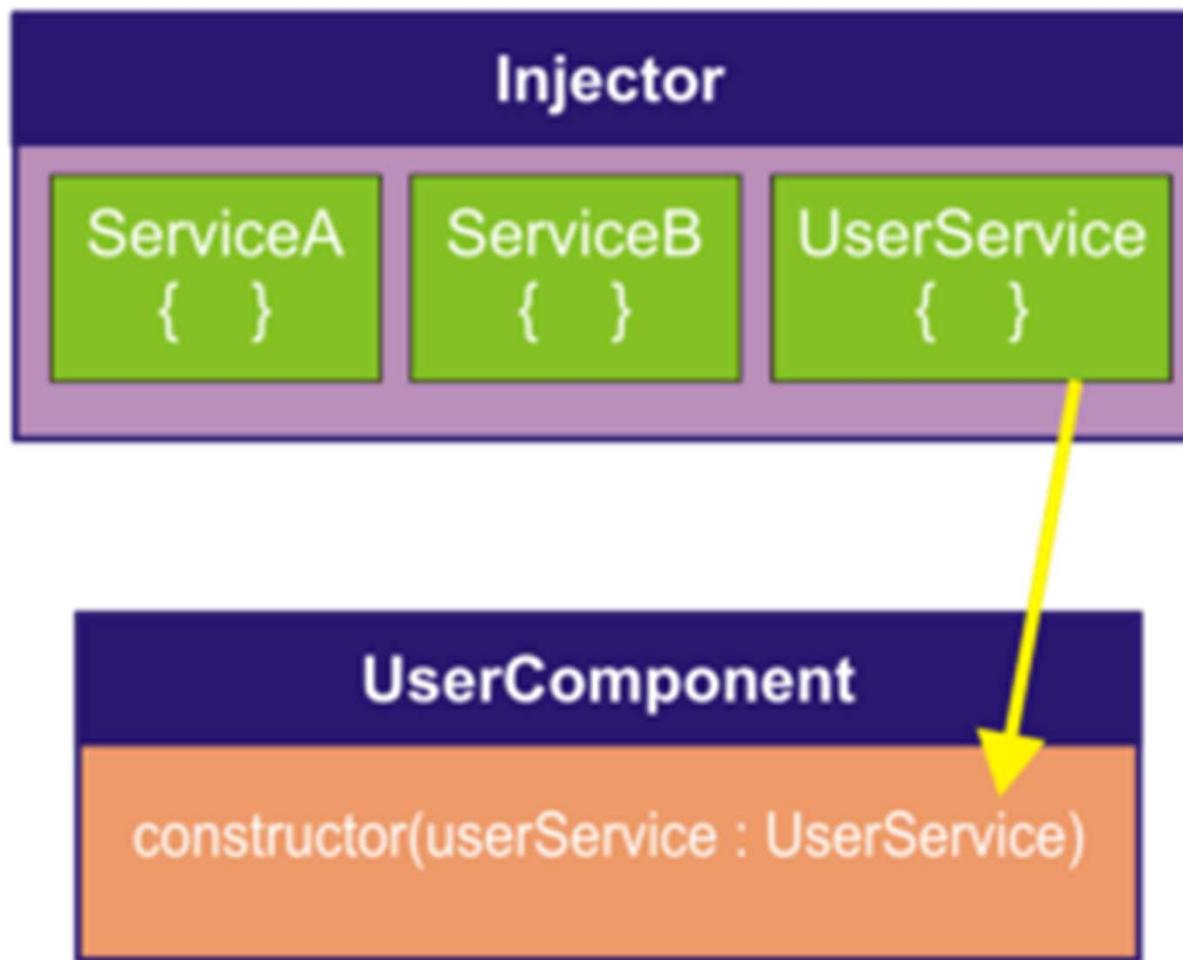
```
constructor(service: CoursesService) {  
    this.courses = service.getCourses();  
}
```

DOBRZE

Wymiana danych poprzez usługę



Usługa – wstrzykiwanie zależności





Usługi Service

Klasa do specyficznych celów:

- dzielenia danych
- implementacja logiki aplikacyjnej/biznesowej
- zewnętrzna interakcja – odczyty danych z serwera

Dlaczego Usługi?

- Usługi najlepiej używać do obsługi CRUD na danych
- Pozwalając na separację danych od logiki przetwarzania danych!



Użycie zewnętrznej Usługi - service

```
@Injectable()
export class KsiazkiService {

  constructor() { }

  table = [ {title:"Node.js, MongoDB, AngularJS. Kompendium wiedzy", price:99},
            {title:"Tworzenie gier internetowych. Receptury", price:49},
            {title:"Web 2.0 Architectures. What entrepreneurs and information architects need to know", price:84.92},
            {title:"React dla zaawansowanych", price:45},
            {title:"Spring MVC 4. Projektowanie zaawansowanych aplikacji WWW", price:102}
          ];

  getKsiazki(){
    return this.table;
  }
}
```

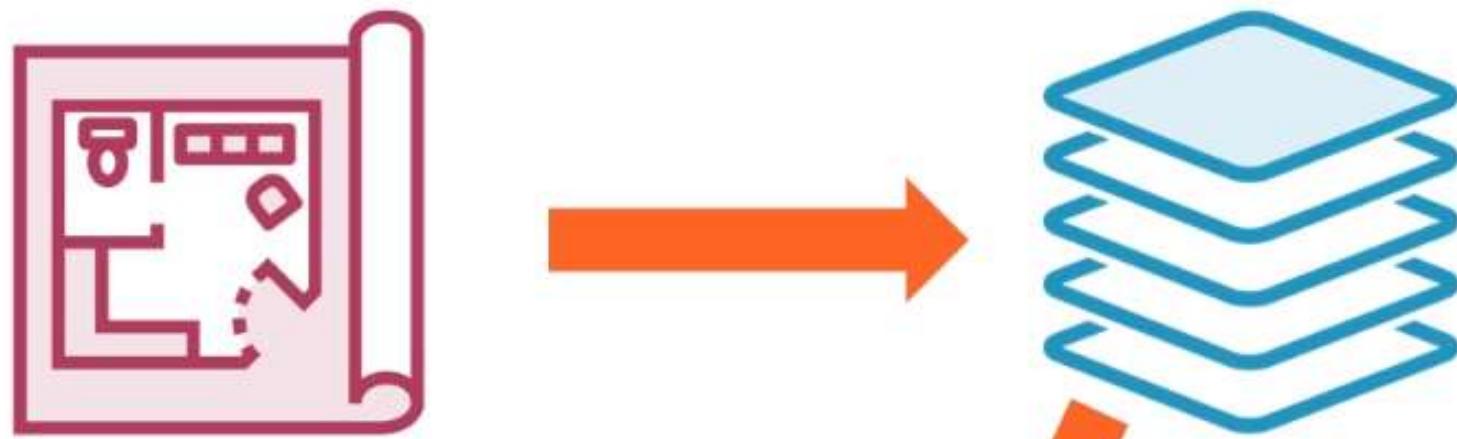
Rejestracja w module

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { KsiazkiService } from './ksiazki.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [
    KsiazkiService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Wstrzykiwanie serwisu do Komponentu



Provider

Injector

```
export class DashboardComponent {  
  constructor(private dataService: DataService) {}  
}
```

Wstrzykiwanie do konstruktora – Dependency Injection

```
import { Component } from '@angular/core';
import { KsiazkiService } from './ksiazki.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  name = 'Grzegorz Rogus';

  ksiazki;

  constructor(service: KsiazkiService ) {
    this.ksiazki = service.getKsiazki();
  }
}
```

```
<div style="text-align:center">
  <h1>
    Witaj {{name}}!
  </h1>
</div>

<p>
  Ksiazki warte polecenia na temat technologii Webowych:
</p>
<ul>
  <li *ngFor=" let ksiazka of ksiazki">
    {{ksiazka.title}} w cenie {{ksiazka.price}}
  </li>
</ul>
```

Model dziedziny i Mock data

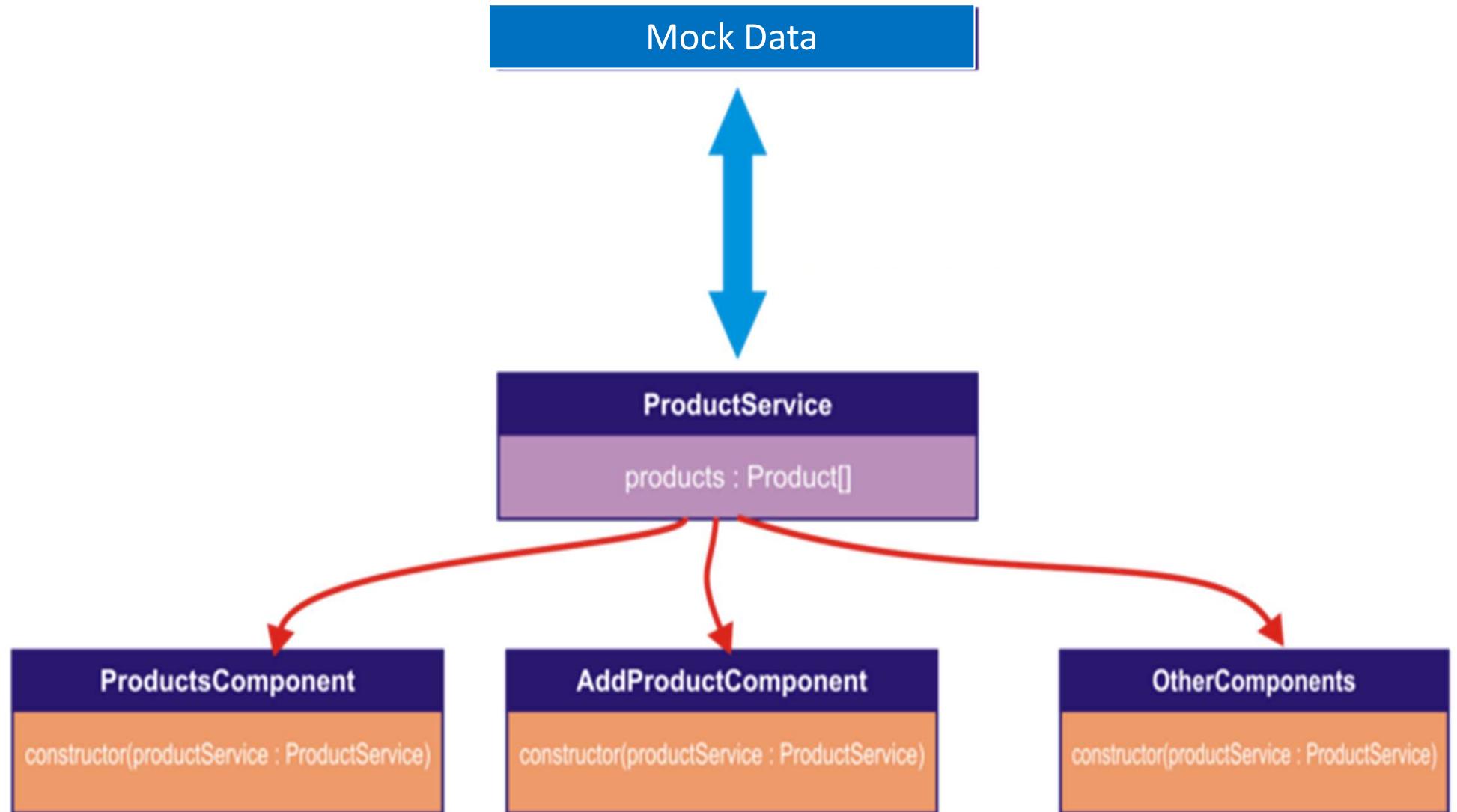
Dobrą praktyką projektową jest wyizolowanie struktury danych oraz ewentualnych danych od komponentu.

```
export interfejs Product {  
    id: number;  
    modelName: string;  
    color: string;  
    productType: string;  
    brand: string;  
    price: number;  
}
```

product.ts

```
import { Product } from '../models/product';  
  
export class MockData {  
    public static Products: Product[] = [  
        {  
            'id': 11,  
            'modelName': 'F5 Youth',  
            'color': 'Gold',  
            'productType': 'Mobile',  
            'brand': 'OPPO',  
            'price': 16990  
        },  
        {  
            'id': 12,  
            'modelName': 'Inspiron',  
            'color': 'Gray',  
            'productType': 'Laptop',  
            'brand': 'DELL',  
            'price': 59990  
        }  
    ];  
}
```

mock-product-data.ts



Lista komentarzy – implementacja usługi

- Przenosimy dane z pliku component-list do pliku mock.ts (symulującego źródło danych).
- Następnie tworzymy serwis udostępniający dane pochodzące z mock

Dlaczego usługa obsługi danych?

- Użytkownik usługi nie wie z jakiego źródła są dane.
- Dane mogą pochodzić z Web Serwisu, z lokalnego pliku albo być imitowane.
- To jest piękno korzystania z usług!
- Usługa odpowiada za dostęp do danych.
- W każdej chwili można zmienić sposób dostępu - zmiany są tylko w tej jednej usłudze.

Lista komentarzy

Realizacja usługi udostępniającej dane

```
import { Comment } from './comment';
export const KomentarzeDane: Comment[] = [
  {imie: "Grzegorz", komentarz: "Pierwszy komentarz", hiden: true },
  {imie: "Anna", komentarz: "Super strona", hiden: false },
  {imie: "Alicja", komentarz: "Fajny film wczoraj widziałam", hiden: true },
];
```

```
import { Injectable } from '@angular/core';
import { KomentarzeDane } from './mock';
@Injectable()
export class KomentarzeService {
  getComments() {
    // return komentarze;
    return Promise.resolve(KomentarzeDane);
  }
}
```

Komunikacja asynchroniczna

```
constructor( service: KomentarzeService ) {

  this.comments = service.getComments();
}
```



Routing

Nawigacja po aplikacji

Single Page Application

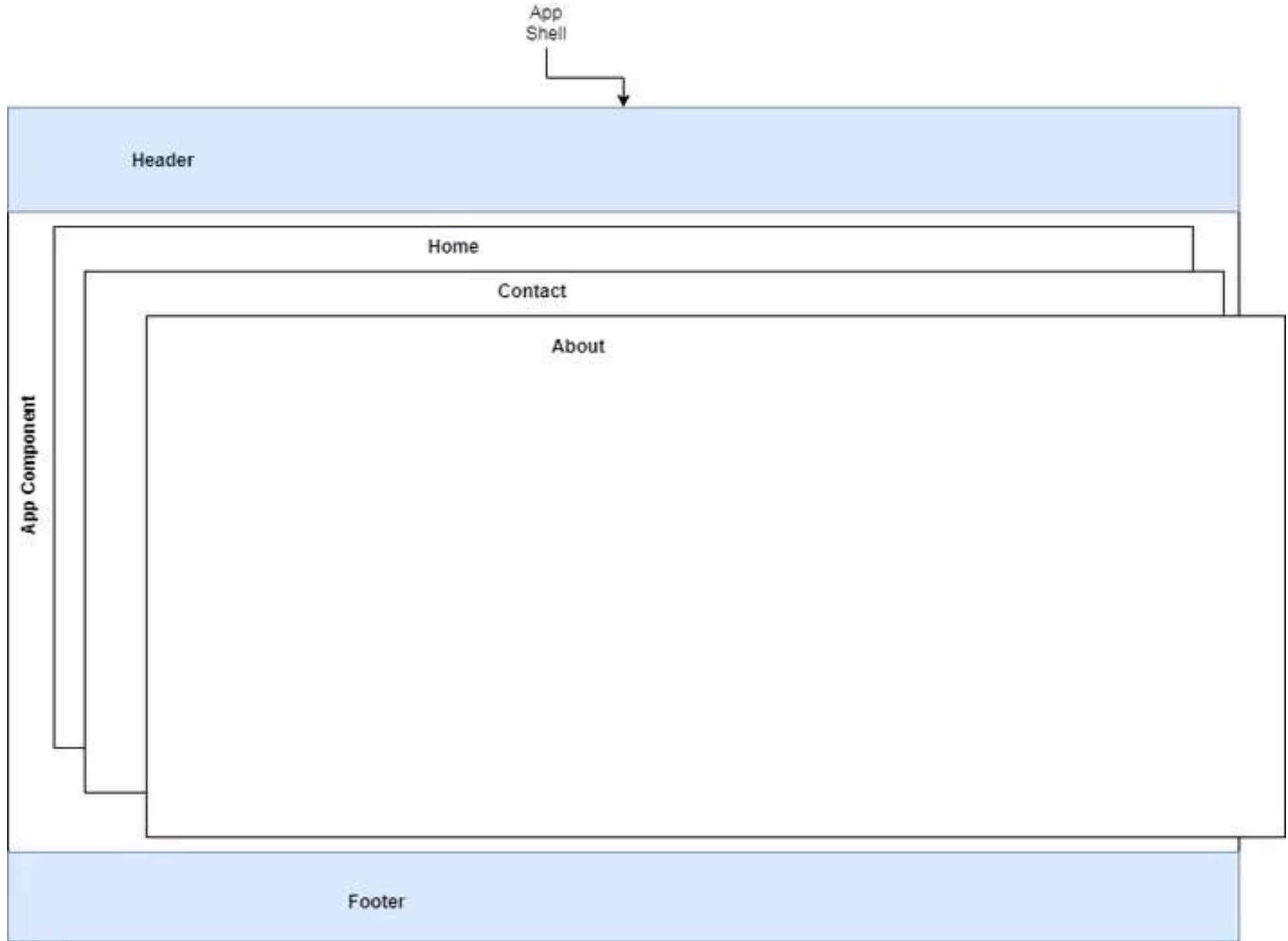


No page refresh on request

Traditional Web Application



Whole page refresh on request





Routing

Routing pozwala zawrzeć pewne aspekty stanu aplikacji w adresie URL.

Dla aplikacji front-end jest to opcjonalne - możemy zbudować pełną aplikację bez zmiany adresu URL. Dodanie routingu pozwala jednak użytkownikowi przejść od razu do pewnych funkcji aplikacji.

Dzięki temu aplikacja jest łatwiej przenośna i dostępna dla zakładek oraz umożliwi użytkownikom dzielenie się linkami z innymi.

Routing ułatwia:

- Utrzymanie stanu aplikacji
- Wdrażanie aplikacji modułowych
- Stosowanie ról w aplikacji (niektóre role mają dostęp do określonych adresów URL)



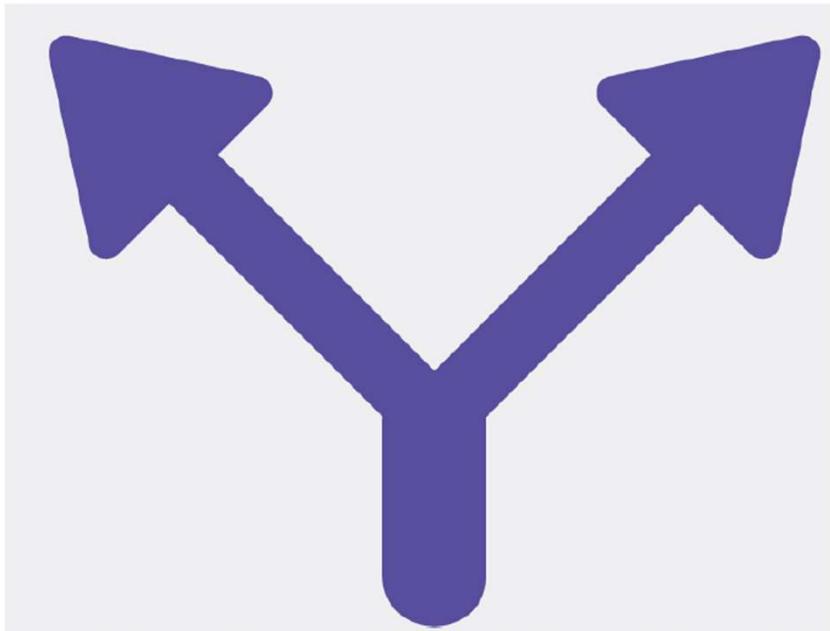
Routing

- Zadaniem routera w ramach frameworka Angular jest nawigacja między widokami
- Router Angulara bazuje na modelu nawigacji przeglądarki
 - URL wprowadzony w pasku adresu prowadzi do wskazanego widoku
 - Kliknięcie linku w aktualnym widoku powoduje przejście do innego
 - Adres URL może zawierać parametry dla widoku
 - Przyciski Back i Forward w przeglądarce nawigują po historii
- Obszar na stronie, w którym wyświetlane mają być różne komponenty zależnie od stanu routera, wskazuje się znacznikiem
`<router-outlet></router-outlet>`



Routing

Składowe routingu



1. `<base href="/">`
2. `import RouterModule`
3. Konfiguracja ścieżek
4. `<router-outlet>`

Konfiguracja routingu

CLI -> Would you like to add Angular routing? (y/N)

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

src/app/app.component.html

```
<router-outlet></router-outlet>
```

Definiowanie tablicy routes

```
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

Definiowanie połączeń między trasami

```
<a routerLink="/component-one">Component One</a>
```

Nawigacja programowo

```
this.router.navigate(['/component-one']);
```

Deklaracja parametrów trasy

```
export const routes: Routes = [  
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },  
  { path: 'product-list', component: ProductList },  
  { path: 'product-details/:id', component: ProductDetails }  
];
```

localhost:4200/szczegóły produktu/5

Powiązanie tras z parametrami

```
<a *ngFor="let product of products"  
  [routerLink]=["'/product-details', product.id]">  
  {{ product.name }}  
</a>
```

Kontrolowanie dostępu do lub z Route

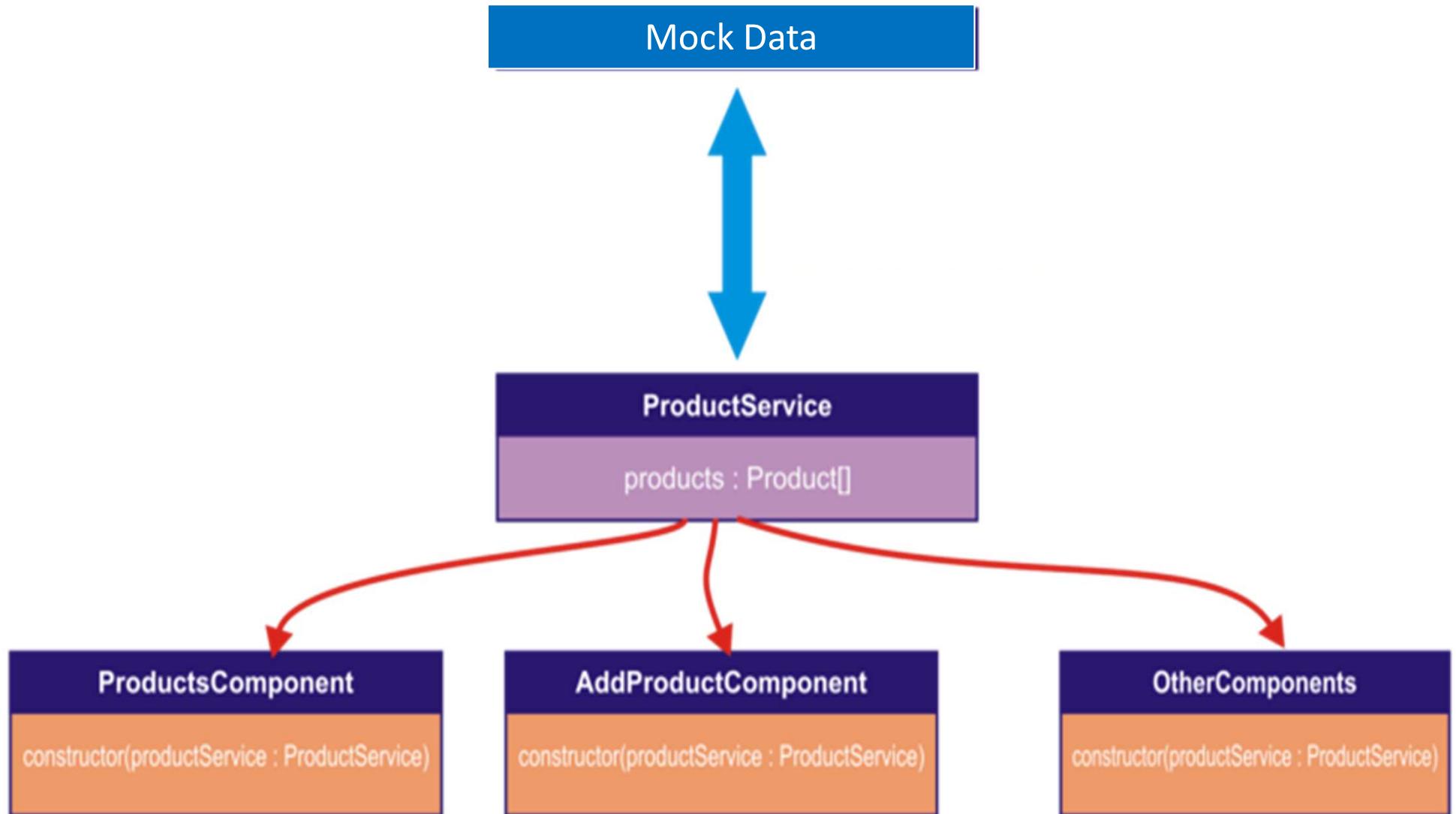
Niektóre trasy mają być dostępne tylko po zalogowaniu się użytkownika lub zaakceptowaniu Warunków.

```
const routes: Routes = [
  { path: 'home', component: HomePage },
  {
    path: 'accounts',
    component: AccountPage,
    canActivate: [LoginRouteGuard],
    canDeactivate: [SaveFormsGuard]
  }
];
```

```
import { CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
import { LoginService } from './login-service';

@Injectable()
export class LoginRouteGuard implements CanActivate {
  constructor(private loginService: LoginService) {}

  canActivate() {
    return this.loginService.isLoggedIn();
  }
}
```



Uzycie servisu

- ng g service product

```
import { MockData } from './mock-data/mock-product-data';
import { Injectable } from '@angular/core';
import { Product } from '../models/product';

@Injectable()
export class ProductService {
  products: Product[] = [];

  constructor() {
    this.products = MockData.Products;
  }

  getProducts(): Product[] {
    return this.products;
  }

  removeProduct(product: Product) {
    let index = this.products.indexOf(product);
    if (index !== -1) {
      this.products.splice(index, 1);
    }
  }

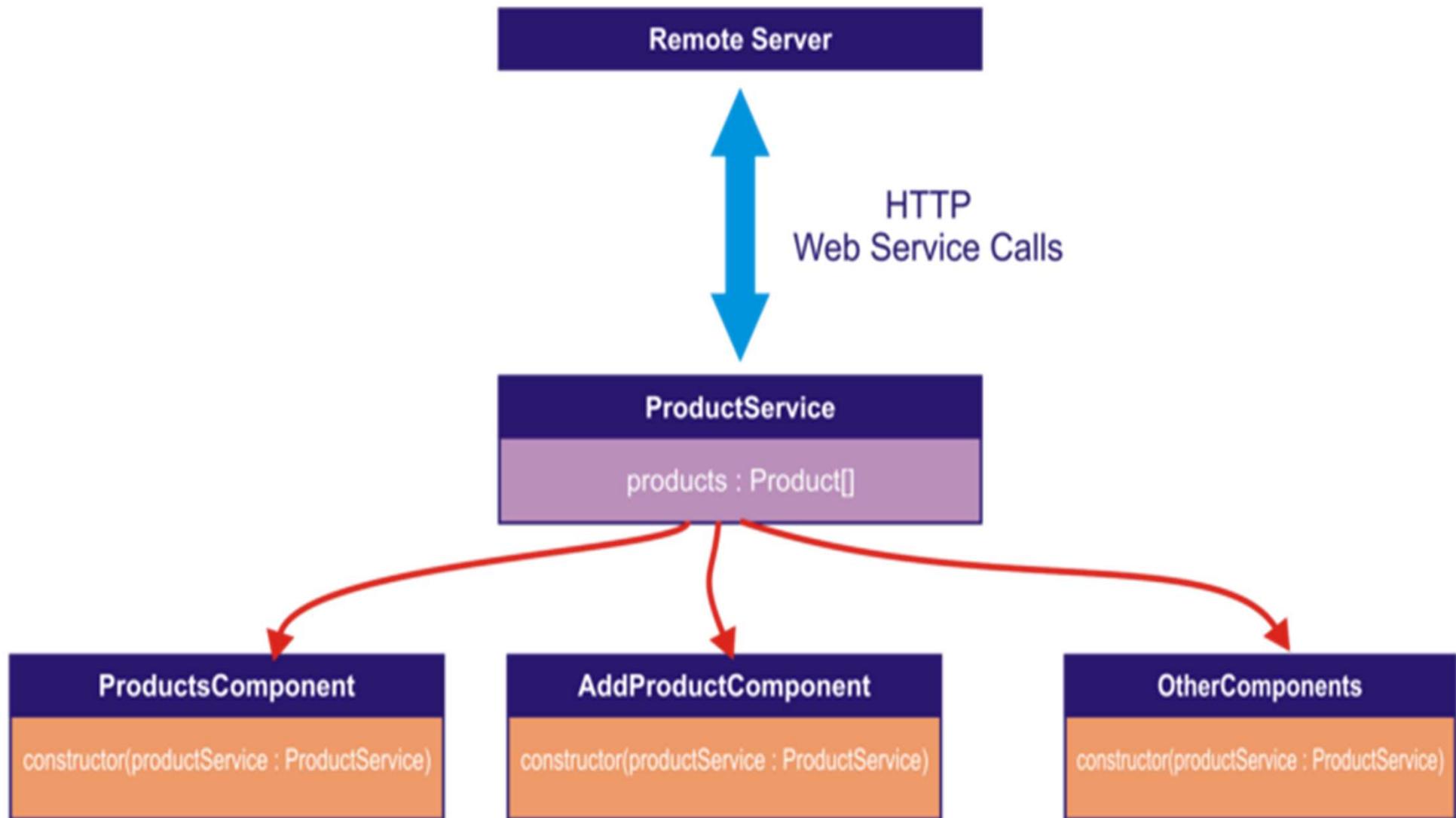
  getProduct(id: number): Product {
    return this.products.find( p => p.id === id));
  }

  addProduct(product: Product) {
    this.products.push(product);
  }
}
```

Wywołanie usługi – przeniesienie danych do usługi

```
export class ProductsComponent implements OnInit {  
  products: Product[] = [];  
  
  constructor(public productService: ProductService) {  
    //  this.products = productService.getProducts();  
  }  
  
  ngOnInit() {  
    this.products = productService.getProducts();  
  }  
  
  deleteProduct(product: Product) {  
    this.productService.removeProduct(product);  
    this.products = this.productService.getProducts();  
  }  
}
```





HTTP



Request - http

Response - Observable



Usługi asynchronous

- Usługi asynchronous zwracające jako wynik obiekty typu Observable lub Promise.
- Należy więc przekształci zwracana wartość w Observable lub promise

```
import { Observable } from 'rxjs/Observable';
```

```
import { of } from 'rxjs/observable/of';
```

```
getProducts(): Observable<Product[]> {
```

```
    return of(this.products);
```

```
}
```

of(this.products) emituje pojedynczą wartość pochodząca z tablicy produktów.

Usługi asynchroniczne

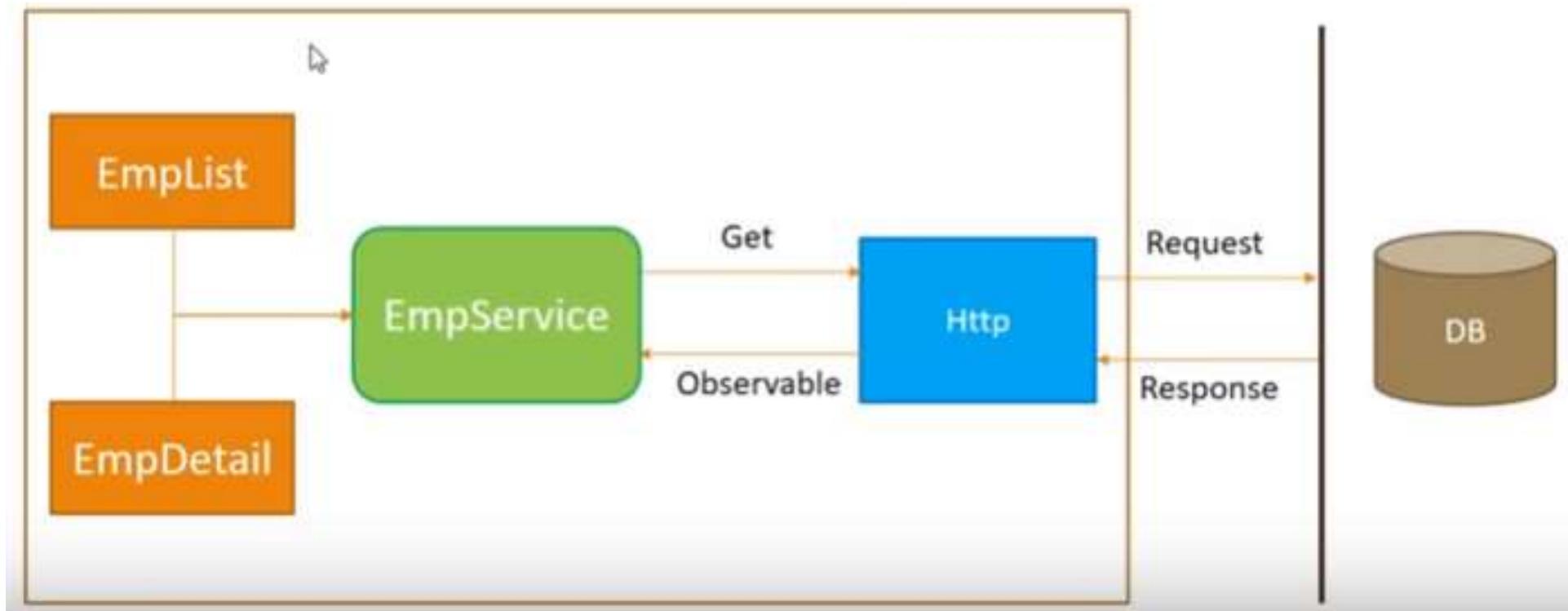
- Zmiany w metodach usługi ProductService skutkują błędem w ProductsComponent. -> powód
ProductService.getProducts() zwraca teraz Observable<Product[]>

```
ngOnInit() {  
    this.products = this.productService.getProducts();  
}
```

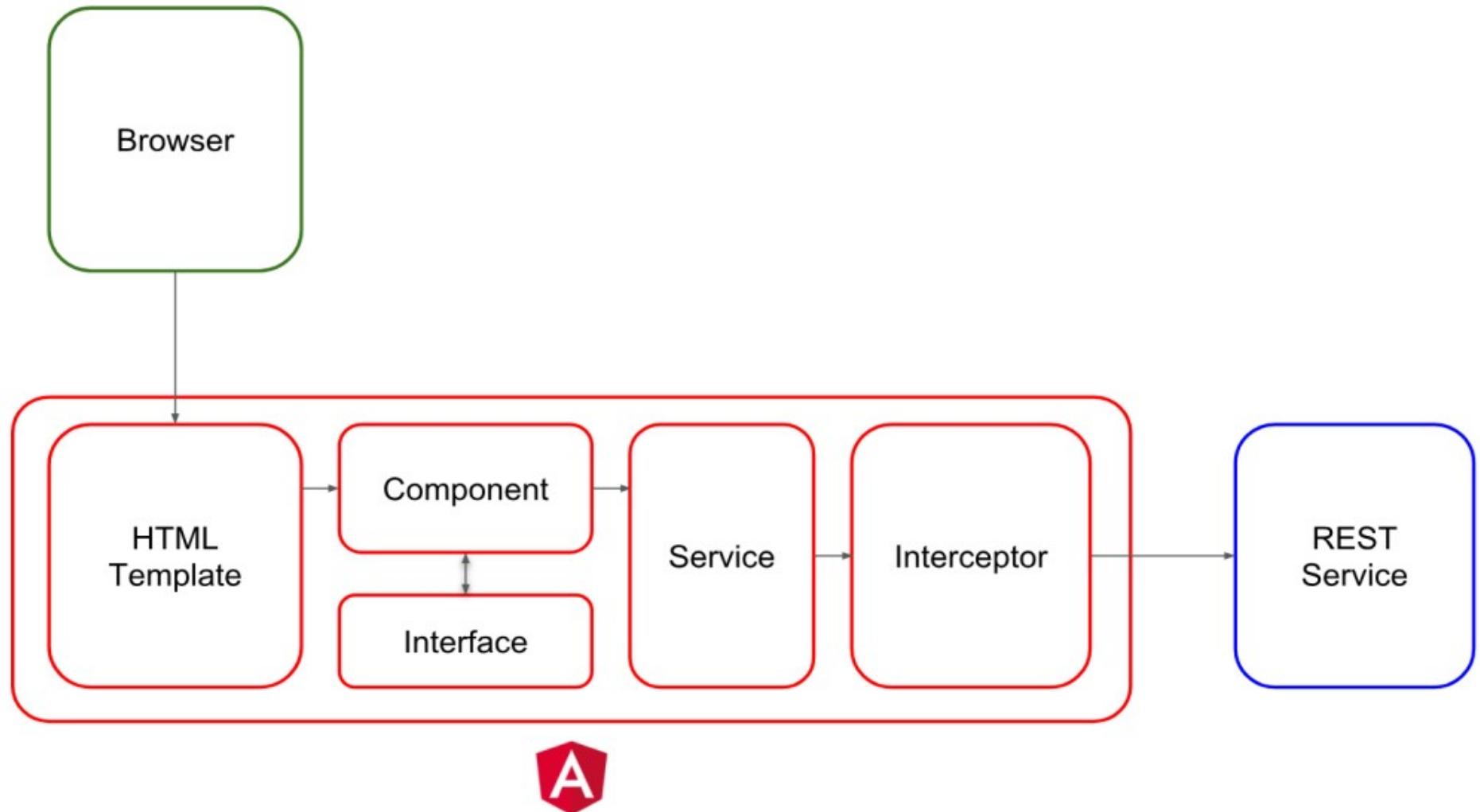
Stara wersja

```
ngOnInit() {  
    this.productService.getProducts().subscribe(  
        products => this.products = products  
    );  
}
```

nowa wersja



Architektura rest API w Angularze



Usługa HTTP

Dodajemy do projektu usługę HTTP (`HttpClient`):

- Angular korzysta z usługi `HttpClient` do komunikacji ze zdalnym serwerem poprzez protokół HTTP
- Aby udostępnić usługę HTTP w całej aplikacji należy:
 - otworzyć główny moduł, `AppModule`
 - zimportować `HttpClientModule` z modułu `@angular/common/http`
 - dodać go do tablicy `@NgModule.imports`
- Zamiast korzystać z rzeczywistego serwera można go tylko symulować.
- In-memory Web API przechytuje żądania z usługi `HttpClient`, dane są przechowywane tylko w pamięci operacyjnej.

Symulacja danych z serwera

```
npm install angular-in-memory-web-api –save
```

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryProductService extends InMemoryDbService {
  createDb() {
    const products = [
      {
        'id': 11,
        'name': 'test',
        ....
      },
      {
        ....
      }
    ];
    return { products };
  }
}
```

fakeserver.service.ts

```
import { InMemoryProductService } from './in-memory-product.service';
import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
...
@NgModule({
  ...
  imports: [
    ...
    HttpClientModule,
    HttpClientInMemoryWebApiModule.forRoot(InMemoryProductService, { delay : 2000})
  ],
  ...
})
export class AppModule { }
```

AppModule

httpClient w akcji

Odwołania do serwerowego API w klasie usługi

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Student } from './student';

const httpOptions = { headers: new HttpHeaders({ 'Content-Type': 'application/json' }) };

@Injectable()
export class StudentService {

  private studentsApiUrl = 'http://localhost:5000/api/student';

  constructor(private http: HttpClient) { }

  getStudents(): Observable<Student[]> {
    return this.http.get<Student[]>(this.studentsApiUrl);
  }

  updateStudent(student: Student): Observable<any> {
    const url = `${this.studentsApiUrl}/${student.id}`;
    return this.http.put(url, student, httpOptions);
  }
}
```

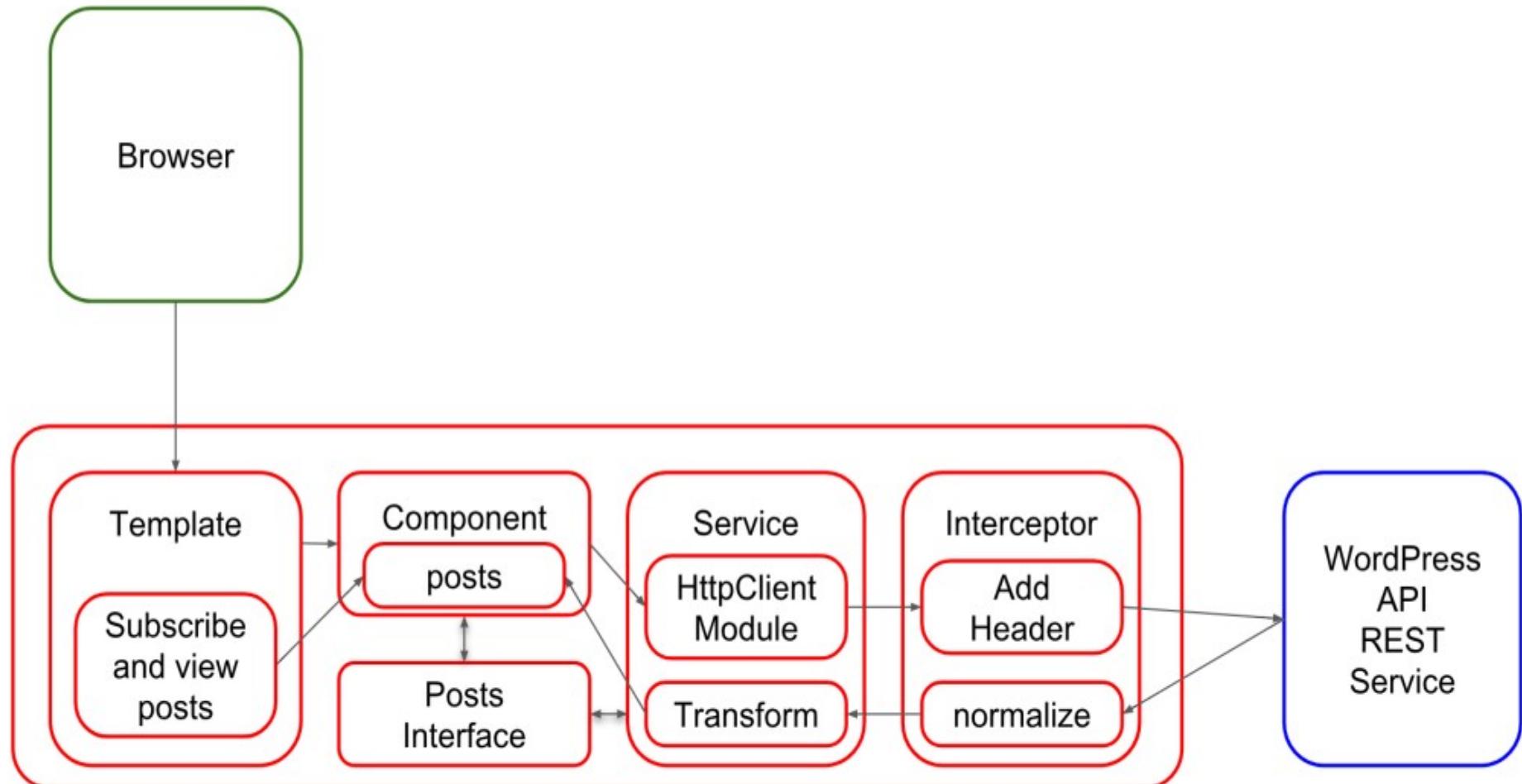
httpClient w akcji

Wykorzystanie usługi dostępowej do API w klasach komponentów

```
...
students: Student[];
...
constructor(private studentService: StudentService) { }
...
getStudents(): void {
  this.studentService.getStudents()
  .subscribe(students => this.students = students);
}
```

```
...
student: Student;
...
constructor(private studentService: StudentService) { }
...
save(): void {
  this.studentService.updateStudent(this.student)
  .subscribe(() => this.goBack());
}
...
```

Szczegóły



HttpClient – współpraca z serwerem danych

Konfiguracja

1. Import modułu httpClient

```
import { HttpClient Module} from  
'@angular/common/http';
```

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule {  
}
```

AppModule

2. Import HttpClient w serwisie

```
import { HttpClient, HttpHeaders } from '@angular/common/http';  
  
export class ProductService{  
  ...  
  constructor(private httpClient: HttpClient) {  
    ...  
  }  
  ...  
}
```

ProductService.service.ts

HttpClient – współpraca z serwerem danych odbiór danych – get()

3. Odczyt danych

```
export class ProductService{  
  productsUrl = 'api/products';  
  
  constructor(private httpClient: HttpClient) {  
  }  
  
  getProducts(): Observable<Product[]> {  
    return this.httpClient.get<Product[]>(this.productsUrl);  
  }  
  ...  
  ...  
}
```

ProductService.service.ts

HttpClient.get zwraca response jako untyped JSON object.
Zastosowanie specyfikatora typu np. <Product[]>, daje obiekt zrzutowany.

HttpClient – współpraca z serwerem danych

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get('/api/customers');  
}
```

1 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get(`/api/customers/${id}`);  
}
```

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get <Customer[]> ('/api/customers');  
}
```

2 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get <Customer> (`/api/customers/${id}`);  
}
```

```
postCustomer(customer): Promise {  
    return this.http.post('/api/customers', customer)  
        .toPromise()  
        .then((data) => data);  
}
```

3 wersja

```
getOrders(): Promise<Order[]> {
  return this.http.get<Order[]>('/api/orders')
    .toPromise()
    .then((response) => response);
}

getOrder(id): Observable<Order> {
  return this.http.get<Order>(`/api/orders/${id}`);
}

getOrdersByCustomer(customerId): Promise<Order[]> {
  return this.http.get<Order[]>(`/api/customers/${customerId}/orders`)
    .toPromise()
    .then((response) => response);
}

postOrder(order): Observable<Order> {
  return this.http.post<Order>('/api/orders', order);
}
```

Obsługa POST i DELETE

```
addProduct(product: Product): Observable<Product> {
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };
  return this.httpClient.post(this.productsUrl, product, httpOptions).pipe(
    tap(p => console.log(`added product with id=${p.id}`)),
    catchError(this.handleError('addProduct'))
  );
}
```

```
removeProduct(product: Product | number): Observable<Product> {
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };
  const id = typeof product === 'number' ? product : product.id;
  const url = `${this.productsUrl}/${id}`;
  return this.httpClient.delete<Product>(url, httpOptions).pipe(
    tap(_ => console.log(`deleted Product id=${id}`)),
    catchError(this.handleError<Product>('deleteProduct'))
  );
}
```

Programowanie Reaktywne

- Asynchroniczne strumienie danych
- Wszystko jest strumieniem danych

zdarzenia generują dane

Dane z serwera

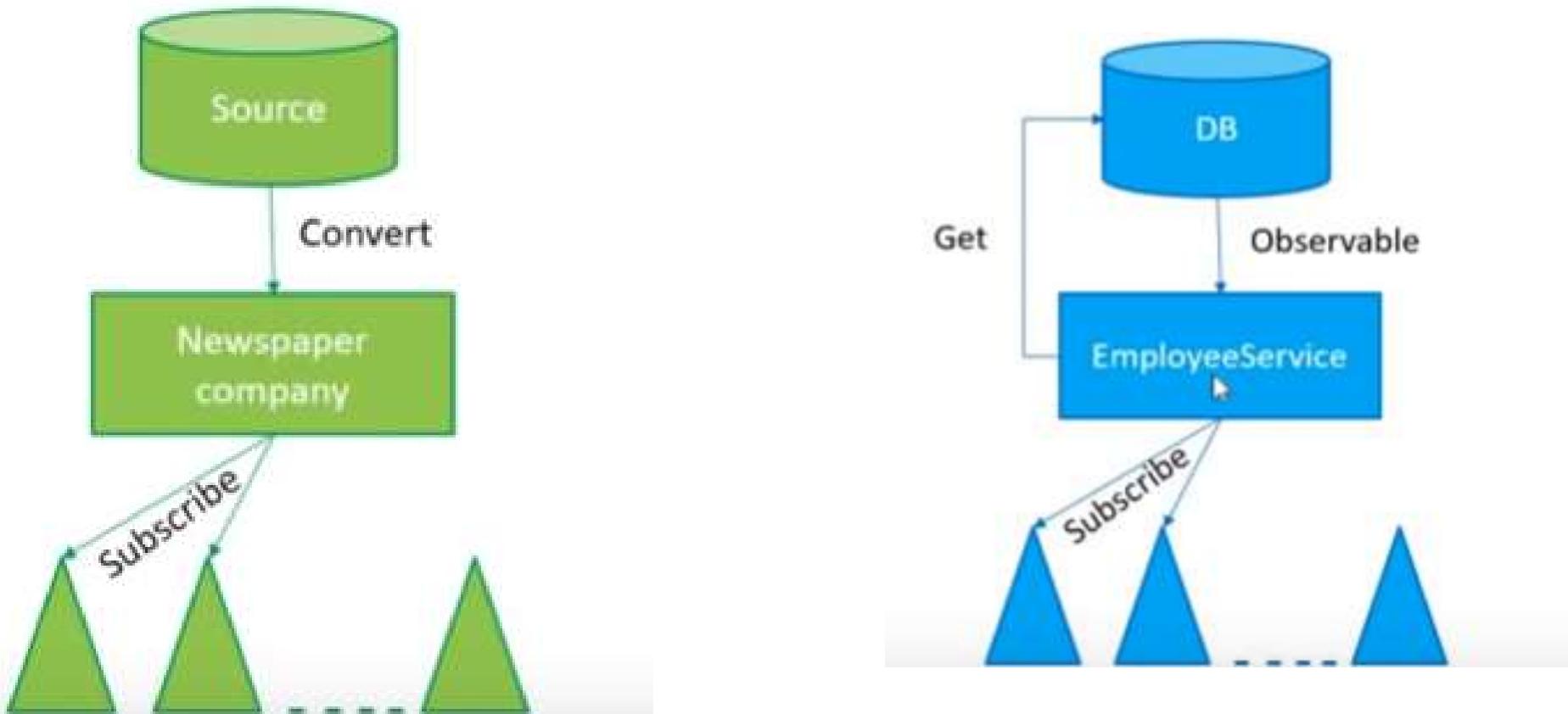
Dane z formularzy

PROGRAMOWANIE REAKTYWNE

- **Programowanie reaktywne jest to programowanie asynchroniczne oparte na obserwacji strumienia danych.**
- Generyczna implementacja wzorca projektowego Obserwator
- Mamy strumienie danych (Observable), które za pomocą wbudowanych operatorów przekształcamy w strumień o interesujących nas własnościach (łączenie, map, flatMap, fold, filter, opóźnienie, wątek ...)
- Ktoś kiedyś zasubskrybuje się na nasz strumień i będzie dostawać powiadomienia o pojawieniu się nowych wartości

Observables

Reprezentuje ideę przyszłego zbioru wartości lub wydarzeń (strumień danych/wydarzeń)



Obiekty typu Observable reprezentują strumień wartości, które zostaną wyemitowane w późniejszym czasie

Źródła danych

	Pojedyncza wartość	Wiele wartości
Synchronicznie	<pre>const value = 42; Console.log(value);</pre>	<pre>var values = [1,2,3,4]; values.forEach(value => { console.log(value); });</pre>
Asynchronicznie	<pre>const asyncValue = Promise.resolve(42); asyncValue.then(value => { console.log(value); });</pre>	<pre>let val\$= Observable.of(1,2,3,4); val\$.subscribe(val\$ => { console.log(val\$); });</pre>

Zastosowania

- Reagowanie na akcje użytkownika:
kliknięcia, ruch myszki, klawiatura itp.
- Otrzymywanie i reagowanie na dane, również w czasie rzeczywistym, np. po web socketach
- Zdarzenia wykonywane po czasie np. setTimeout,
setInterval
- Wszystko co asynchronicznie zwraca od 0 do
nieskończonej ilości wartości

Najważniejsze pojęcia w nowym podejściu

- **Observable** - reprezentuje sekwencję wartości w czasie, która może być obserwowana
- **Observer** - obiekt który otrzymuje dane z Observable
- **Subscription** - wynik subskrybowania, służy głównie do zamykania/zatrzymywania subskrypcji
- **Operators** - zbiór metod dla Observable, najczęściej zwracające nowe Observable (np. *map*, *filter*)
- **Subject** - to samo co Observable, ale nieco bardziej zaawansowane :)

Observable

- Reprezentuje sekwencje wartości w czasie, które są przez nią **emitowane**
- Można ją **zasubskrybować** aby otrzymywać te wartości
- **Leniwe** - kod wykona się i zaczynie produkować wartości dopiero kiedy ktoś słucha (czyli subskrybuje)
- Może wyemitować błąd zamiast wartości (jak promise) lub sygnał "zakończenia" strumienia. W obu wypadkach taki observable jest uważany za zakończony i nie wyśle więcej żadnej wartości

Tworzenie obserwowanych

- Wiele różnych sposobów!

```
const clock$ = new Observable( observer => { setInterval(()  
    => { observer.next('tick'); }, 1000); });
```

```
const colors$ = Observable.from(['red', 'green', 'blue']);
```

```
const colors$ = Observable.of('red', 'green', 'blue');
```

RxJS dodaje dużo nowych metod jak np:

- fromEvent
- fromPromise
- bindCallback
- timer

Podstawy Observable

```
var range = Rx.Observable.range(1, 3); // 1, 2, 3
```

```
var range = range.subscribe(  
    function(value) {},  
    function(error) {},  
    function() {}  
);
```

The code shows the subscribe method being called on the range Observable. Three red arrows point from the word "optional" to the three empty function bodies: one arrow points to the first function (value), another to the second (error), and a third to the third (void). This indicates that the error and void handlers are optional parameters.

Podłączenie się do strumienia jest banalnie proste.
Observable posiada metodę **subscribe**, do której
parametry możemy przekazać na dwa sposoby:

Podstawy Observable

```
var obs = ...;
```

```
// query?
```

gdy sukcesem odbierzemy
wartość ze strumienia

```
var sub = obs.Subscribe(  
    onNext : v => DoSomething(v),  
    onError : e => HandleError(e),  
    onCompleted : () => HandleDone);
```

gdy w strumieniu wystąpi error

gdy observer otrzyma ostatnią wartość ze
strumienia (wypstryka się z danych)

Nasłuchiwanie na zmiany

.subscribe() przyjmuje dwa rodzaje argumentów:

1. Listę funkcji (callbacków)

```
observable$.subscribe(  
    value => console.log('Nowa wartość:', value),  
    err => console.error(err),  
    () => console.log('Koniec nadawania') );
```

2. Obiekt typu **Observer** (z metodami)

```
observable$.subscribe( {  
    next: value => console.log('Nowa wartość:', value),  
    error: err => console.error(err),  
    complete: () => console.log('Koniec nadawania'),  
});
```

Observer

- Specjalny obiekt służący do "konsumowania" wartości które dostarcza Observable.
- Najprościej mówiąc jest to obiekt z trzema metodami - callbackami dla każdego rodzaju notyfikacji którą może dostarczyć Observable:

```
const observer = {  
    next: value => console.log('Nowa wartość:', value),  
    error: err => console.error(err),  
    complete: () => console.log('Koniec nadawania'),  
};
```

Observable nie są takie jak myślisz!

- Lazy computation i nie zawsze asynchroniczna, czyli bardziej jak funkcja niż promise!

```
const hello$ = new Observable(observer => {
    console.log('2. Drugi');
    observer.next('3. Trzeci');
    observer.next('4. Czwarty');
});

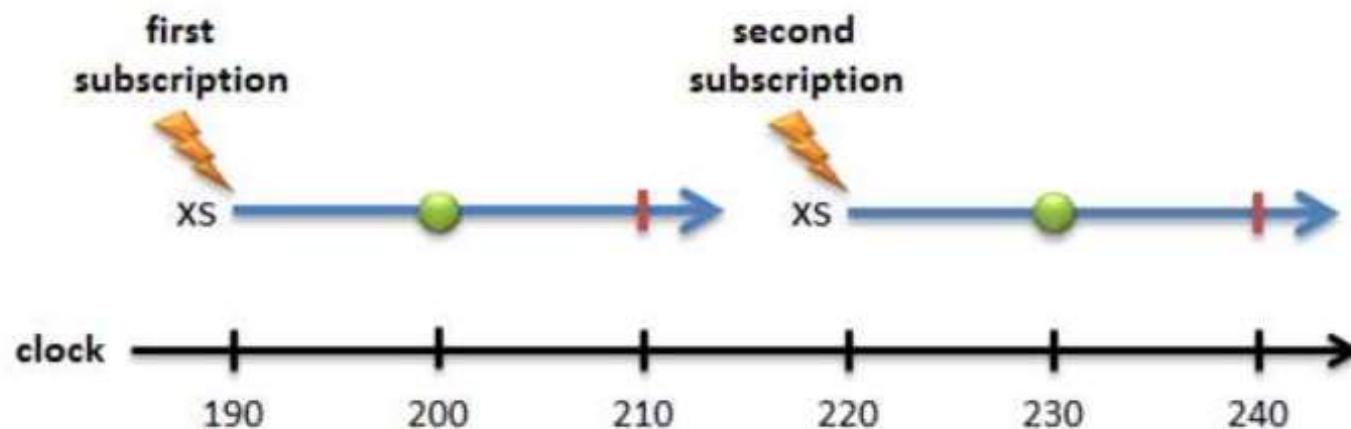
console.log('1. Pierwszy');

hello$.subscribe(x => console.log(x));

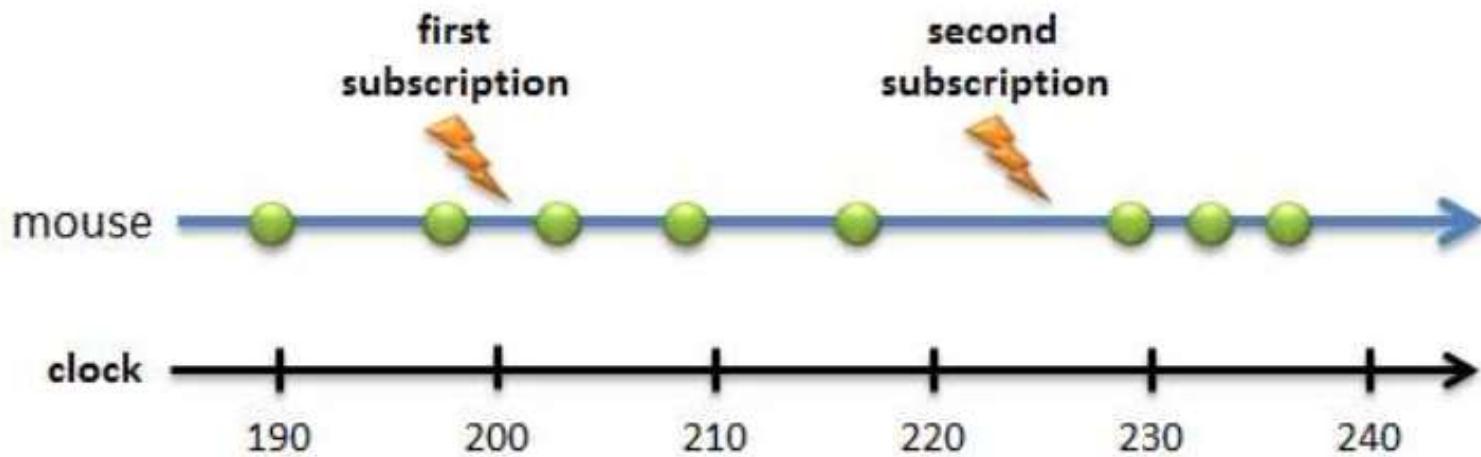
console.log('5. Piaty');
```

```
// Wynik:  
// 1. Pierwszy  
// 2. Drugi  
// 3. Trzeci  
// 4. Czwarty  
// 5. Piaty
```

Cold Observables



Hot Observables



Typy Observable: Hot i Cold

Observables domyślnie są **zimne**:

- każda subskrypcja powoduje wywołanie konstruktora
- odsubskrybowanie skutkuje zamknięciem tej instancji obserwowanej

Mogą być też **gorące**:

- Przestają być leniwe
- Mogą produkować wartości nawet kiedy nikt nie słucha
- Wywołanie *subscribe* wiele razy nie skutkuje wywołaniem konstruktora wielokrotnie
- Mogą współdzielić przesyłane wydarzenia

Subskrypcja

- Specjalny obiekt reprezentujący zasób, najczęściej konkretne wykonanie danego Observable.
- Obiekt praktycznie ma tylko jedną metodę: **unsubscribe()**

```
const clock$ = new Observable( observer => {
    const intervalId = setInterval( () => {
        observer.next('tick');
    }, 1000);

    return () => clearInterval(intervalId);
});

const subscription = clock$.subscribe(val => console.log(val));

subscription.unsubscribe();
```

Observable i RxJS

RxJS

- Część większej rodziny Rx* jak np. RxJava, Rx.NET, RxScala...
- Oryginalnie stworzona i rozwijana przez Microsoft (do v.4), obecnie (od v.5) przepisana i rozwijana m.in przez Google i Netflixa.
- Zawiera implementację Observable i innych typów, ale przede wszystkim posiada bogatą kolekcję **Operatorów**.

Operators

- Operatory służą do komponowania obserwowanych i łatwego deklaratywnego zarządzania asynchronicznym kodem.
- Najczęściej występują w formie metod dostępnych na instancji obiektu Observable.
- Użycie operatora nigdy nie zmienia (nie mutuje) oryginalnego Observable ani wartości jakie przez niego przechodzą. Zamiast tego zawsze tworzy nowy Observable na bazie istniejącego.

Creation Operators	Transform Operators	Filtering Operators
• create	• map	• filter
• of	• mergeMap	• take
• empty	• pluck	• last
• fromEvent	• window	• debounce
• fromPromise	• buffer	• throttle
• interval	• scan	• distinctUntilChanged
• ...	• ...	• ...

Combination Operators	Multicasting Operators	Error Handling Operators
• combine	• cache	• catch
• concat	• multicast	• retry
• merge	• publish	• retryWhen
• race	• share	
• zip	• ...	
• ...		

Przykładowe operatory

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
```

```
const doubleNumbers$ = numbers$.map(num => num * 2);
```

```
doubleNumbers$.subscribe(num => console.log(num));
```

// Output:
// 2
// 4
// 6
// 8
// 10

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
```

```
const smallNumbers$ = numbers$.filter(num => num < 4);
```

```
smallNumbers$.subscribe(num => console.log(num));
```

// Output:
// 1
// 2
// 3

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
```

```
numbers$  
.filter(num => num > 2)  
.map(num => num * 2)  
.subscribe(num => console.log(num));
```

// Output:
// 6
// 8
// 10

Tworzenie Observables

```
Rx.Observable.fromArray([1, 2, 3]);
```

```
Rx.Observable.fromEvent(input, 'click');
```

```
Rx.Observable.fromEvent(eventEmitter, 'data', fn);
```

```
Rx.Observable.fromCallback(fs.exists);
```

```
Rx.Observable.fromNodeCallback(fs.exists);
```

```
Rx.Observable.fromPromise(somePromise);
```

```
Rx.Observable.fromIterable(function*() {yield 20});
```

Własne operatory

```
const echo = (input$) => {
  return new Observable(observer => {
    input$.subscribe({
      next: val => {
        observer.next(val);
        observer.next(val);
      },
      error: err => observer.error(err),
      complete: () => observer.complete()
    });
  });
}
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
```

```
const echoNumbers$ = echo(numbers$);
```

```
echoNumbers$.subscribe(num => console.log(num));
```

// Output:
// 1
// 1
// 2
// 2
// 3
// 3
// 4
// 4
// 5
// 5

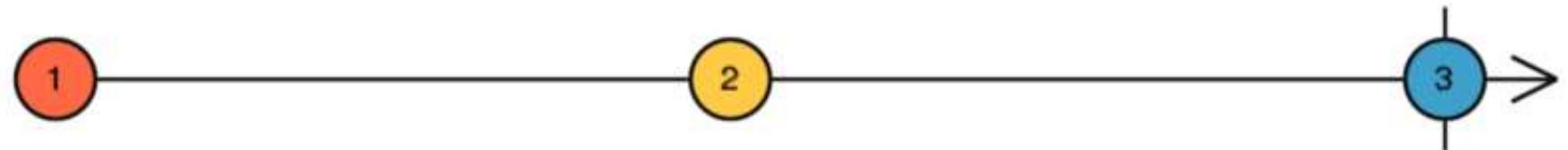
```
var range = Rx.Observable.range(1, 10) // 1, 2, 3 ...
    .filter(function(value) { return value % 2 === 0; })
    .map(function(value) { return "<span>" + value + "</span>"; })
    .takeLast(1);

var subscription = range.subscribe(
    function(value) { console.log("last even value: " + value); });
// "last even value: <span>10</span>"
```

Przykłady tworzenie

```
var bar = Rx.Observable.create(function (observer) {  
    try {  
        console.log('Hello');  
        observer.next(42);  
        observer.next(100);  
        observer.next(200);  
        setTimeout(function () {  
            observer.next(300);  
            observer.complete();  
            observer.next(400);  
        }, 1000);  
    } catch (err) {  
        observer.error(err);  
    }  
});  
  
bar.subscribe(  
    function nextValueHandler(x) {  
        console.log(x);  
    },  
    function errorHandler(err) {  
        console.log('Wystapil blad: ' + err);  
    },  
    function completeHandler() {  
        console.log('OK - zrobione');  
    }  
);
```

of(1, 2, 3)



```
var numbers = Rx.Observable.of(10, 20, 30);
numbers.subscribe(x => {
  console.log(x);
}, err => {
  console.log(err);
}, () => {
  console.log('done');
});
```

```
//output
/*
10
20
30
done
```

```
fromEvent(element, 'click')
```



```
var clicks = Rx.Observable.fromEvent(document, 'click');  
clicks.subscribe(ev => console.log(ev));
```

```
var result = Rx.Observable.fromPromise(fetch('http://myserver.com/'));  
result.subscribe(x => console.log(x), e => console.error(e));
```

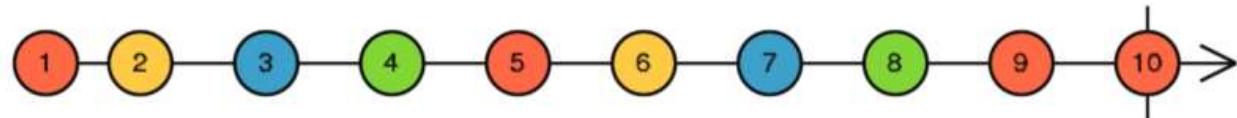
```
timer(3000, 1000)
```



```
var numbers = Rx.Observable.timer(5000);  
numbers.subscribe(x => console.log(x));  
//emits 0 po 5 s. i koniec .
```

```
var numbers = Rx.Observable.timer(3000, 1000);  
numbers.subscribe(x => console.log(x));  
// emituje 0 po 3 s. a nastepnie kolejne numery co sekunde
```

```
range(1, 10)
```



```
var numbers = Rx.Observable.range(1, 10);  
numbers.subscribe(x => console.log(x));
```

/ output

1

2(po 1 sec)

3(po 1 sec)

4(po 1 sec)

5(po 1 sec)

...

*/

```
interval(1000)
```



```
var numbers = Rx.Observable.interval(1000);  
numbers.subscribe(x => console.log(x));
```

Inne



debounce



map($x \Rightarrow 10 * x$)



merge



distinctUntilChanged

