Angular 7 - tutorial o bohaterach cz. 1

Tworzenie serwisów Web 2.0

dr inż. Robert Perliński
rperlinski@icis.pcz.pl

13 kwietnia 2019

Plan prezentacji

Spis treści

1 Wprowadzenie do tutorialu **1**

2 Tutorial **2**

2.1 Podstawowa struktura aplikacji . . . 2
2.2 Edytor bohaterów . . . 3
2.3 Lista bohaterów . . . 5
2.4 Dwa komponenty przekazujące sobie dane . . . 9
2.5 Usługi . . . 11
2.6 Usługi asynchroniczne i wzorzec obserwator . . . 14
2.7 Pokazywanie komunikatów . . . 18

3 Źródła **20**

1 Wprowadzenie do tutorialu

Wprowadzenie do tutorialu I

- Tutorial o bohaterach wprowadza w najważniejsze elementy Angulara.
- Pozwala zbudować aplikację zarządzającą listą bohaterów.
- Tutorial będzie zawierać wiele funkcji spodziewanych w aplikacji opartej na danych.
- Mamy:
  - wyświetlanie listy bohaterów,
  - edycję danych wybranego bohatera,
  - nawigację pomiędzy różnymi widokami naszych danych.

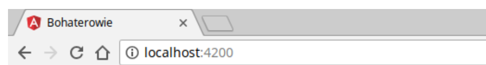
Wprowadzenie do tutorialu II

Umiejętności, których można nauczyc się z tego tutorialu:

- użycie wbudowanych dyrektyw do pokazywania/ukrywania elementów i wyświetlenia listy bohaterów
- utworzenie komponentu Angulara wyświetlającego szczegóły bohatera i wyświetlenie listy bohaterów
- użycie jednokierunkowego wiązania danych dla danych tylko do odczytu
- dodanie pól edycyjnych do aktualizacji modelu za pomocą dwukierunkowego wiązania danych
- wiązanie metod komponentu do zdarzeń użytkownika, jak kliknięcie czy naciśnięcie klawisza na klawiaturze
- zaznaczenie bohatera w liście głównej i jego edycja w komponencie podrzędnym
- formatowanie danych za pomocą potoków
- tworzenie współdzielonych usług w celu zebrania informacji o bohaterach w jednym miejscu
- użycie trasowania do nawigowania pomiędzy różnymi widokami i ich komponentami

Wprowadzenie do tutorialu III

Wygląd aplikacji po skończonej części pierwszej - 3 komponenty i wykorzystanie usługi:



2 Tutorial

2.1 Podstawowa struktura aplikacji

1.1 Utworzenie aplikacji

- Tworzymy, kompilujemy i uruchamiamy aplikację:

```
ng new bohaterowie
cd bohaterowie
ng serve --open
```

- Otwieramy aplikację pod adresem `http://localhost:4200/`.

- Zmieniamy domyślny tytuł aplikacji:

  src/app/app.component.ts (właściwość `title` klasy)

```
title = "Tutorial o bohaterach";
```

- Usuwamy domyślny wygląd głównego komponentu aby zawierał tylko tytuł:

  src/app/app.component.html (szablon)

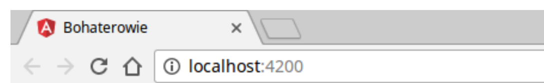
```
<h1>{{title}}</h1>
```

- Ustalamy trochę styli dla całej aplikacji:

1.2 Utworzenie aplikacji

src/styles.css

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: #888;
  font-family: Cambria, Georgia;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```



Tutorial o bohaterach

2.2 Edytor bohaterów

2.1. Utworzenie komponentu `heroes`

- Tworzymy nowy komponent odpowiedzialny za wyświetlanie naszych bohaterów:

```
ng generate component heroes
```

- Zostaną utworzone cztery pliki naszego komponentu.
- Plik komponentu ma strukturę:

  src/app/heroes/heroes.component.ts (wersja początkowa)

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

2.2. Dodanie pola `hero`

- Dodajemy pole `hero` do klasy `HeroesComponent`:

  src/app/heroes/heroes.component.ts (pole hero wewnątrz klasy)

  ```
    hero = 'Jan Paweł II';
  ```

- Dodajemy szablon do klasy `HeroesComponent`

  src/app/heroes/heroes.component.html

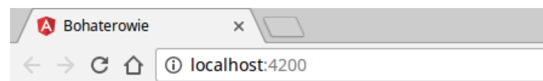
  ```
  {{hero}}
  ```

- Wyświetlamy naszego bohatera pod tytułem całej aplikacji:

  src/app/app.component.html

  ```
  <h1>{{title}}</h1>
  <app-heroes></app-heroes>
  ```

- Kompilujemy i uruchamiamy aplikację po zmianach:



2.3. Utworzenie i wykorzystanie klasy `Hero`

- Tworzymy w katalogu `app` osobną klasę `Hero` zawierającą jego nazwę i id:

  src/app/hero.ts

  ```
  export class Hero {
    id: number;
    name: string;
  }
  ```

- Importujemy klasę `Hero` w komponencie `HeroesComponent`, zmieniamy pole `hero` i inicjalizujemy je:

  src/app/heroes/heroes.component.ts

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

  ...
  export class HeroesComponent implements OnInit {
    hero: Hero = {
      id: 1,
      name: 'Jan Paweł II'
    };

    constructor() { }

    ngOnInit() {
    }

  }
  ```

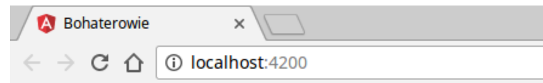
2.4. Zmiana szablonu komponentu `HeroesComponent`

- Zmodyfikowania wymaga także szablon komponentu `HeroesComponent`:

src/app/heroes/heroes.component.html (szablon komponentu `HeroesComponent`)

```
<h2>Informacje o: {{ hero.name | uppercase }}</h2>
<div><span>id: </span>{{hero.id}}</div>
<div><span>nazwa: </span>{{hero.name}}</div>
```

- Duże litery uzyskujemy za pomocą potoku `uppercase`



2.5. Edycja nazwy naszego bohatera

- Edycji nazwy bohatera dokonamy w polu `<input>`.
- Wykorzystamy do tego dwukierunkowe wiązanie danych, `[(ngModel)]`:

src/app/heroes/heroes.component.html (szablon komponentu `HeroesComponent`)

```
<div>
    <label>name:
      <input [(ngModel)]="hero.name" placeholder="name">
    </label>
</div>
```

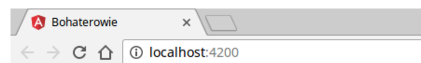
- Dyerektywa `ngModel` należy do opcjonalnego modułu `FormsModule`, należy go samodzielnie dodać:

src/app/app.module.ts (główny moduł aplikacji)

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

import { AppComponent } from './app.component';
import { HeroesComponent } from './heroes/heroes.component';

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



2.3 Lista bohaterów

3.1. Utworzenie tablicy bohaterów

- Docelowo będziemy mieli dane bohoaterów na zewnętrznym serwerze, tutaj stworzymy imitację tego serwera.
- Tworzymy w katalogu `src/app` plik `mock-heroes.ts`:

src/app/mock-heroes.ts

```
import { Hero } from './hero';

export const HEROES: Hero[] = [
  {id:11, name:"Jan III Sobieski"},
  {id:12, name:"Jan Paweł II"},
  {id:13, name:"Władysław II Jagiełło"},
  {id:14, name:"św. Faustyna Kowalska"},
  {id:15, name:"Jurand ze Spychowa"},
  {id:16, name:"Andrzej Kmicic"},
  {id:17, name:"Judyta"},
  {id:18, name:"św. Małgorzata Maria Alacoque"},
  {id:19, name:"Kazimierz Wielki"},
  {id:20, name:"Bolesław Chrobry"}
];
```

- Dane bohaterów eksportujemy.
- Będziemy je wyświetlać w `HeroesComponent`.

3.2. Dodanie listy bohaterów do `HeroesComponent`

- Importujemy tablicę bohaterów (`import HEROES`)
- Dodajemy pole `heroes`, które przechowa tablicę w naszym komponencie:

src/app/heroes/heroes.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HEROES } from '../mock-heroes';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  hero: Hero = {
    id: 1,
    name: "Jan Paweł II"
  };
  heroes = HEROES;

  constructor() { }

  ngOnInit() {
  }

}
```

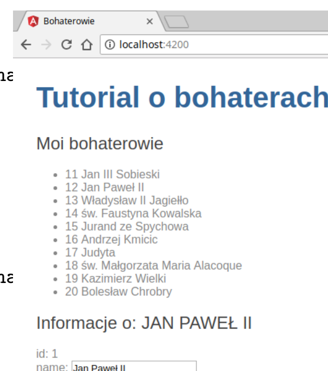
3.3. Lista bohaterów z dyrektywą `ngFor`

- Wzbogacony szablon komponentu `HeroesComponent` o listę bohaterów:

src/app/heroes/heroes.component.html

```
<h2>Moi bohaterowie</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <span class="badge">{{hero.id}}</span> {{hero.na
  </li>
</ul>

<h2>Informacje o: {{ hero.name | uppercase }}</h2>
<div><span>id: </span>{{hero.id}}</div>
<div>
    <label>nazwa:
      <input [(ngModel)]="hero.name" placeholder="na
    </label>
</div>
```



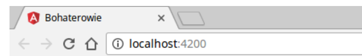
- Dyrektywa `ngFor` odpowiada tutaj za generowanie odpowiedniej liczby znaczników `<li>`.

3.4. Wygląd komponentu po dodaniu stylów

Cały kod dostępny tutaj: `https://angular.io/tutorial/toh-pt2#final-code-review`

src/app/heroes/heroes.component.css

```
/* HeroesComponent's private CSS styles */
.selected {
  background-color: #CFD8DC !important;
  color: white;
}
.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}
.heroes li {
  cursor: pointer;
  position: relative;
  left: 0;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
}
.heroes li.selected:hover {
  background-color: #BBD8DC !important;
  color: white;
}
...
```



3.5. Komponent nadrzędny i podrzędny

- Po kliknięiu w listę bohaterów aplikacja powinna udostępnić szczegóły tego wybranego i pozwolić na edycję jego nazwy.
- Użyjemy do tego zdarzenia związanego z kliknięciem wybranego bohatera, wiązanie zdarzeń (ang. *event binding)*:

  src/app/heroes/heroes.component.html

  ```
  <li *ngFor="let hero of heroes" (click)="onSelect(hero)">
  ```

- Angular oczekuje na zdarzenie `click` (nawiasy okrągłe), które wykona wyrażenie `onSelect(hero)` będące metodą klasy komponentu.

3.6. Obsługa wyboru bohatera - dodanie pola i metody

- Pole `hero` zastępujemy polem `selectedHero` bez przypisanej wartości.
- Będzie mu przypisany wybrany bohater z listy.
- Do `HeroesComponent` dodajemy też metodę `onSelect()`:

  src/app/heroes/heroes.component.ts

  ```
  import { Component, OnInit } from '@angular/core';
  import { Hero } from '../hero';
  import { HEROES } from '../mock-heroes';

  @Component({
    selector: 'app-heroes',
    templateUrl: './heroes.component.html',
    styleUrls: ['./heroes.component.css']
  })
  export class HeroesComponent implements OnInit {
    heroes = HEROES;
    selectedHero: Hero;
    onSelect(hero: Hero): void {
      this.selectedHero = hero;
    }
    constructor() { }

    ngOnInit() {
    }

  }
  ```

3.7. Szablon komponentu z dostosowanymi zmianami

- W szablonie zmieniamy pole `hero` na `selectedHero`:

src/app/heroes/heroes.component.html

```
<h2>Moi bohaterowie</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes" (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>

<h2>Informacje o: {{ selectedHero.name | uppercase }}</h2>
<div><span>id: </span>{{selectedHero.id}}</div>
<div>
    <label>nazwa:
      <input [(ngModel)]="selectedHero.name" placeholder="name">
    </label>
</div>
```

- Konsola przeglądarki wyświetla błąd:
  `ERROR TypeError: Cannot read property 'name' of undefined`
- Szczegóły bohatera wyświetlamy tylko, jeśli jeden z nich został zaznaczony - dyrektywa `ngIf`.

3.8. Ukrywanie szczegółów bohatera - dyrektywa ngIf

- Informacje o szczegółach bohatera umieszczamy wewnątrz bloku warunkowego - `ngIf`:

src/app/heroes/heroes.component.html

```
<h2>Moi bohaterowie</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
<div *ngIf="selectedHero">
  <h2>Informacje o: {{ selectedHero.name | uppercase }}</h2>
  <div><span>id: </span>{{selectedHero.id}}</div>
  <div>
    <label>nazwa:
      <input [(ngModel)]="selectedHero.name" placeholder="name">
    </label>
  </div>
</div>
```

- Działa wybieranie bohaterów, zaznaczanie i zmiana ich nazw.
- Ustawiamy też styl klasy na `selected` korzystając z wiązania właściwości: `[class.selected]=...`.

Gdzie jesteśmy - obsługa wyboru bohatera

Gdzie jesteśmy? Mamy dwa komponenty:

- `AppComponent`:
  - zawierający tytuł naszej aplikacji
  - wyświetlający komponent `HeroesComponent`
- `HeroesComponent` zawierający:
  - wybranego bohatera,
  - metodę do wyboru bohatera (`onSelect()` i zdarzenie `click`),
  - listę bohaterów (dyrektywa `ngFor`),
- i dwa dodatkowe pliki aplikacji:

- `mock-heroes.ts` - tablica z listą bohaterów,
- `hero.ts` - klasa odpowiedzialna za pojedynczego bohatera.

Funkcjonalność:

- wyświetlanie listy bohaterów,
- możliwć edycji wybranego bohatera.

2.4 Dwa komponenty przekazujące sobie dane

4.1. Osobny komponent do wyświetlania szczegółów

- Przechowywanie w jednym komponencie obsługi listy bohaterów i wyświetlanie szczegółów o nich narusza **zasadę pojedynczej odpowiedzialności**.
- Dzielimy tę funkcjonalność na dwa osobne komponenty.
- Tworzymy komponent `HeroDetailComponent`: `ng generate component hero-detail`
- Jest to wersja początkowa, niepełna:

src/app/hero-detail/hero-detail.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

4.2. Szablon do komponentu ze szczegółami bohatera

- Do komponentu `HeroDetailComponent` przenosimy kod szablonu odpowiadający za szczegóły wybranego bohatera:

src/app/hero-detail/hero-detail.component.html

```
<div *ngIf="hero">
  <h2>Informacje o: {{ hero.name | uppercase }}</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label>nazwa:
      <input [(ngModel)]="hero.name" placeholder="name">
    </label>
  </div>
</div>
```

- Zmiana `selectedHero` na `hero`, bo mamy osobny komponent.

4.3. Import klasy `Hero` i dekoratora `@Input`

- W obu komponentach korzystamy z klasy `Hero`, trzeba ją zaimportować.
- Z `@angular/core` importujemy dodatkowo dekorator `Input`.
- Do klasy `HeroDetailComponent` dodajemy też pole `hero` z dekoratorem `@Input`:

src/app/hero-detail.component.ts

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

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: ['./hero-detail.component.css']
})
export class HeroDetailComponent {
  @Input() hero: Hero;

  constructor() { }

  ngOnInit() {
  }
}
```

4.4. Pole `hero` jako Input, [...] w nadrzędnym komponencie

- Komponent `HeroDetailComponent` musi wiedzieć, którego bohatera dane ma wyświetlać.
- Informacje o tym posiada komponent nadrzędny: `HeroesComponent`.
- Chcemy wyświetlić szczegóły wybranego bohatera.
- Komponent nadrzędny wiąże pole `selectedHero` z polem `hero` komponentu podrzędnego:
- Pole docelowe, w nawiasach [ ] po lewej stronie znaku = **musi być wejściowym** (input):

  src/app/heroes/heroes.component.ts (fragment szablonu)

  ```
  ...
  <app-hero-detail [hero]="selectedHero"></app-hero-detail>
  ...
  ```

- To jest wiązanie właściowości (ang. *property binding).*

4.5. Deklaracje komponentów w głównym module

- Każdy komponent musi być zadeklarowany w jednym, i tylko jednym module.
- Dzięki Angular-CLI na razie nie musimy tutaj dodawać jawnie żadnych zależności.

src/app/app.module.ts (fragment)

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

import { AppComponent } from './app.component';
import { HeroesComponent } from './heroes/heroes.component';
import { HeroDetailComponent } from './hero-detail/hero-detail.component';

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

2.5 Usługi

5.1. Wykorzystywanie usług

- Korzystanie z usług ułatwia testy jednostkowe.
- Komponenty nie powinny bezpośrednio pobierać czy zapisywać danych. Nie powinny też prezentować fałszywych danych.
- Komponenty skupiają się na prezentowaniu danych i dostęp do danych powinny zostawiać usługom.
- Ten tutorial będzie korzystał z jednej usługi udostępniającej bohaterów.
- Nie będziemy używać operatora `new` ale mechanizmu wstrzykiwania zależności - wstrzykujemy dane do konkstuktora konkretnego komponentu.
- Usługi świetnie dzielą dane pomiędzy klasami, które się nawzejem nie znają.
- Generowanie naszej usługi:

  ```
  ng generate service hero
  ```

5.2. Tworzenie usługi

- Nazwa pliku usługi kończy się na `*.service.ts` - taka konwencja.
- Eksportujemy klasę HeroService - usługa z listą bohaterów.
- Importujemy funkcję `Injectable` i stosujemy ją korzystając z wzorca dekorator.
- Dzięki temu Angular śledzi zależności tej usługi od innych - emituje metadane o naszej usłudze.
- Nawet, jeśli nasza usługa nie ma żadnych zależności, dobrze jest od początku dodawać dekorator `@Injectable()` - usługa będzie wczytywana poprzez wstrzykiwanie zależności.

src/app/hero.service.ts

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

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

  constructor() { }
}
```

5.3. Dostarczenie usługi `HeroService`

- Usługa `HeroService` musi być dostępna dla wstrzykiwania zależności zanim ten może jej użyć np. w `HeroesComponent`.
- Należy zarejestrować dostawcę usług (ang. *provider*).
- Dostawca może stworzyć albo dostarczyć usługę.
- Tutaj dostarczana jest instancja klasy `HeroService`.
- Klasa `HeroService` musi być zarejestrowana jako dostawca tej usługi.
- Robimy to za pomocą mechanizmu wstrzykiwania zależności czyli odpowiada za to dekorator `@Injectable`

- Domyślnie Angular-CLI rejestruje dostawcę z mechanizmem wstrzykiwania na poziomie całej appliakcji (`root`).
- Dostarczanie usług na poziomie całej aplikacji tworzy pojedynczą, współdzieloną usługę `HeroService` i wstrzykuje ją, gdzie tylko jest potrzebna.

  src/app/hero.service.ts (fragment)

  ```
  @Injectable({
    providedIn: 'root'
  })
  ```

5.4. Metoda dostępowa `getHeroes()`

- 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.

src/app/hero.service.ts

```
import { Injectable } from '@angular/core';
import { Hero } from './hero';
import { HEROES } from './mock-heroes';

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

  constructor() { }
  getHeroes(): Hero[] {
    return HEROES;
  }
}
```

5.5. Wykorzystanie usługi

- Importujemy potrzebną nam usługę - możemy się do niej odwołać w kodzie.
- Jak Angular utworzy tę usługę?
- Nie tworzymy jej samodzielnie, nie korzystamy z `new` ponieważ:
  - nasz komponent musiałby wiedzieć jak utworzyc usługę; zmiana konstruktora usługi wymagałaby zmiany jego wywołania w wielu miejscach...
  - samodzielne tworzenie usługi w każdym komponencie utworzy wiele jej instancji ... co jeśli będzie potrzebna tylko jedna współdzielona?
  - wiąże nas to z konkretną implementacją `HeroService` - ciężko przejść na inne scenariusze, zmieniać dane do testowania...
- Dlatego też stosujemy **wstrzykiwanie zależności**.

5.6. Wykorzystanie naszej usługi w `HeroesComponent`

- Usuwamy importowanie bohaterów bezpośrednio z `mock-heroes.ts` - mamy teraz usługę, którą importujemy.
- Definicję pola `heroes` zastępujemy pustą tablicą.
- Wstrzykujemy usługę do klasy komponentu:

  src/app/heroes/heroes.component.ts

  ```
  import { HeroService } from '../hero.service';
  ...
  export class HeroesComponent implements OnInit {
    heroes: Hero[];
    constructor(private heroService: HeroService) { }
    ...
  }
  ```

- Zamiast wykorzystania operatora `new` do komponentu dodamy konstruktor z parametrem potrzebnej usługi:
  - parametr tworzy automatycznie prywatne pole `heroService`,
  - pole `heroService` zostanie rozpoznane jako klasa `HeroService`, która jest tworzona przez wstrzykiwanie zależności,
  - Angular będzie wiedział, aby dostarczyć instancję usługi, przy tworzeniu komponentu.

5.7. Wstawki programowe - `ngOnInit`

- Pobieranie danych z usługi powinno się odbywać automatycznie.
- Gdzie dodać kod odczytu danych z usługi?
- Nie dodajemy logiki do konstruktora, szczególnie takiej, która łączy się z serwerem.
- Wczytywanie danych powinno odbywać się już na utworzonym komponencie.
- Wykorzystamy wstawki programowe dla cyklu życia komponentu - **ngOnInit**.
- Angular sam wywoła odpowiednią metodę przy określonym stanie komponentu.

5.8. Wykorzystanie usługi w `ngOnInit`

- Wstawka programowa `OnInit` pozwala na użycie metody `ngOnInit` przy inicjalizacji komponentu - u nas mamy wczytywanie danych z usługi.
- Usługa `heroService` powstaje dzięki konstruktorowi - pole tworzone przez mechanizm wstrzykiwania zależności.

src/app/heroes/heroes.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';

...
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  onSelect(hero: Hero): void {
    this.selectedHero = hero;
  }
  constructor(private heroService: HeroService) { }

  getHeroes(): void {
    this.heroes = this.heroService.getHeroes();
  }
  ngOnInit() {
    this.getHeroes();
  }
}
```

Gdzie jesteśmy - usługa `HeroService`

Gdzie jesteśmy?

- Mamy trzy komponenty: (`AppComponent`, `HeroesComponent` i `HeroDetailComponent`).
- Mamy usługę `HeroService`.
- Funkcjonalność dokładnie taka jak wcześniej ale z użyciem usługi i mechanizmu wstrzykiwania zależności.

Następnie trzeba zrobić asynchroniczne odbieranie danych - konieczne w rzeczywistej aplikacji.

2.6 Usługi asynchroniczne i wzorzec obserwator

Wzorzec projektowy obserwator I

- Obserwator (ang. *observer*) – wzorzec projektowy należący do grupy wzorców czynnościowych.
- Używany jest do powiadamiania zainteresowanych obiektów o zmianie stanu pewnego innego obiektu.
- We wzorcu obserwator wyróżniamy dwa podstawowe typy obiektów:
  - **obserwowany** (ang. *observable*, *subject*) - obiekt, o którym chcemy uzyskiwać informacje,
  - **obserwator** (ang. *observer*, *listener*) - obiekty oczekujące na powiadomienie o zmianie stanu obiektu obserwowanego.
- Kiedy stan obiektu obserwowanego się zmienia, wywołuje on metodę `powiadomObserwatorow()`, która wysyła powiadomienia do wszystkich zarejestrowanych obserwatorów.

https://pl.wikipedia.org/wiki/Obserwator_(wzorzec_projektowy)

Wzorzec projektowy obserwator II

- Obserwatorzy często potrzebują informacji o tym, co zostało zmienione w obserwowanym obiekcie.
- Istnieją dwie podstawowe strategie wyciągania informacji o zmianach:
  - **strategia wyciągania** (ang. *pull model*) - obiekt obserwowany przekazuje w argumencie referencję do siebie samego, pozwalając obserwatorom na samodzielne wyciągnięcie niezbędnych informacji,
  - **strategia wpychania** (ang. *push model*) - obiekt obserwowany przygotowuje listę zmian w określonym formacie (najczęściej jako dodatkowy obiekt z określonymi właściwościami) i przekazuje ją jako parametr wywołania obserwatorom.
- W drugim przypadku wymagane jest opracowanie wystarczająco ogólnego formatu opisu zmian, aby był on niezależny od konkretnej sytuacji.

https://pl.wikipedia.org/wiki/Obserwator_(wzorzec_projektowy)

Mechanizm trasowania jako wzorzec obserwator w RxJS

- Angular używa RxJS, biblioteki pozwalającej na korzystanie z programowania reaktywnego, w której założeniem jest, że wszystko jest obserwowalnym strumieniem.
- Obiekt obserwator (ang. *Observer*) dostarcza wsparcia dla iteracji w strategi wpychania po obserwowanej sekwencji.
- Obserwator i intefejsy obiektów biblioteki RxJS dostarczają ogólny mechanizm dla powiadomień w strategi wpychania, znanych też jako wzorzec projektowy obserwator.
- Obiekt obserwowany to obiekt, który przesyła powiadomienia (dostawca danych/powiadomień); obserwujący reprezentuje klasę, która je odbiera (obserwator). U nas jest to komponent HeroesComponent, później też mechanizm trasowania.

`https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md` `https://angular.io/guide/rx-library` `https://typeofweb.com/2017/05/19/observable-rxjs/`

`subscribe`

- Usługa `ActivatedRoute` dostarcza obserwowany typ Params (z wzorca projektowego), którym można przypisać jakiś parametr z adresu URL do obserwowanych, śledzonych obiektów - będzie można po ich zmianie podjąć odpowiednie czynności.
- Powód, dla którego właściwość `params` w `ActivatedRoute` jest typem obserwowanym, jest taki, że mechanizm trasowania (router) może nie tworzyć komponentu na nowo, kiedy jest to ten sam komponent, np. ten sam adres URL albo inny adres ale wymagający tego samego komponentu.
- Nawet jak jest inny parametr to mechanizm trasowania może nie tworzyć komponentu na nowo, jeśli to jest ciągle ten sam komponent.
- Standardowe wykorzystanie typu obserwowalnego z RxJS:

```
ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
       this.id = +params['id'];
       // ... pozostały kod obsługujący zmienę parametru
    });
  }
```

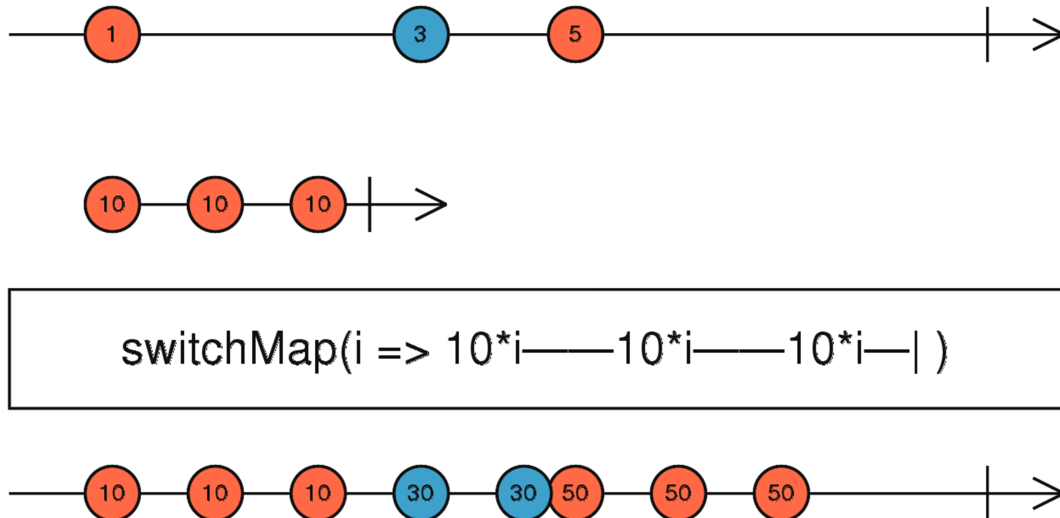
`switchMap`

- Mapuje każdą wartość (`value`) do obiektów obserwowanych, które są później scalane w jeden wyjściowy obiekt obserwowany (u nas jest to funkcja podpisywana w metodzie `subscribe()`).
- Emitowane są tylko te powiadomienia (wartości), które pochodzą z najbardziej aktualnych danych obiektu obserwowanego.
- `map` tworzy obiekty obserwowane, `switch` jest obserwatorem, który nimi zarządza, przekazując dalej (podpisując) najbardziej aktualne zmiany w obiekcie obserwowanym (parametr id).
- Tworzy wewnętrzną listę obiektów obserwowanych, do których przekazana w parametrze funkcja jest podpisana.

```
public switchMap(
  project: function(value: T, ...): ObservableInput,
  resultSelector: function(...): any): Observable
```

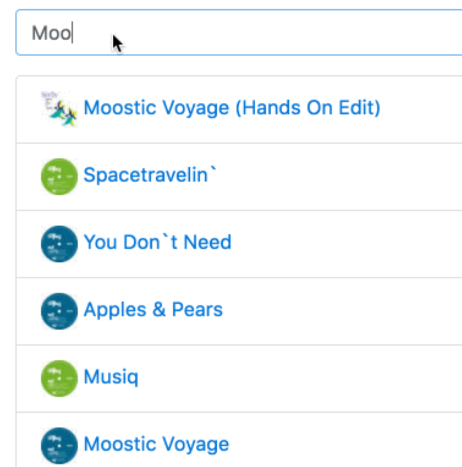
`http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap`

`switchMap`



http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap

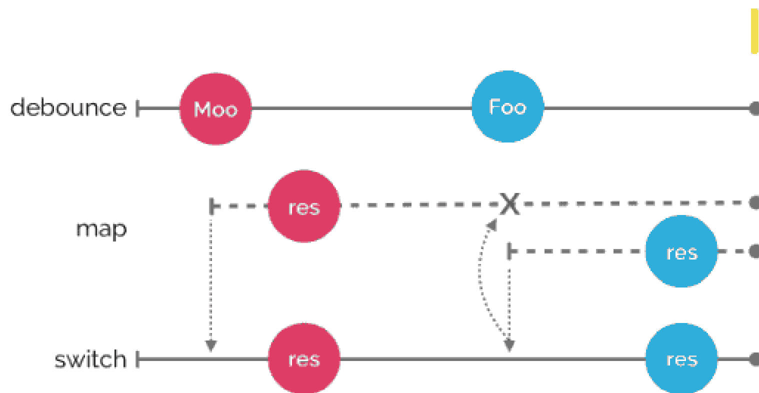
Inny przykład switchMap




```
ngOnInit() {
  this.searchField = new FormControl();
  this.searchField.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .switchMap( term => this.itunes.search(term))
    .subscribe( value => console.log(value));
}
```

https://codecraft.tv/courses/angular/http/http-with-observables/

Inny przykład switchMap



5.9. Usługi asynchroniczne i wzorzec obserwator

- Nasza usługa z tablicą bohaterów działa synchronicznie...
- Normalnie dane wczytywane są asynchronicznie, z użyciem obietnic.
- Ponieważ nasz projekt ma działać z `HttpClient.get()`, a usługa `HttpClient` korzysta z wzorca obserwator, to nasza usługa też będzie korzystała z tego wzorca.
- Można wykorzystać również obietnice i zwykłą funkcję wywoływaną asynchronicznie.

src/app/hero.service.ts

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Hero } from './hero';
import { HEROES } from './mock-heroes';

@Injectable({
  providedIn: 'root'
})
export class HeroService {
  constructor() { }
  // getHeroes(): Hero[] { return HEROES; }
  getHeroes(): Observable<Hero[]> {
    return of(HEROES);
  }
}
```

5.10. Usługi asynchroniczne i wzorzec obserwator

Komponent `HeroesComponent` będzie korzystał z danych poprzez wykorzystanie metody `subscribe()`. Poniższy przykład zawiera też wyrażenia lambda:

src/app/heroes/heroes.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';     // import { HEROES } from '../mock-heroes';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];     // heroes = HEROES;
  selectedHero: Hero;
  onSelect(hero: Hero): void {
    this.selectedHero = hero;
  }
  constructor(private heroService: HeroService) { }
```

```
  // getHeroes(): void { this.heroes = this.heroService.getHeroes(); }
  getHeroes(): void {
    this.heroService.getHeroes()
        .subscribe(heroes => this.heroes = heroes);
  }
  ngOnInit() {
    this.getHeroes();
  }
}
```

5.11. Usługi asynchroniczne i wzorzec obserwator

- Najważniejsza różnica to wykorzystanie `Observable.subscribe()`.
- Wcześniej przypisywaliśmy całą tablicę bohaterów synchronicznie.
- Nie zadziała to przy rzeczywistej usłudze, która zwraca dane z opóźnieniem.
- Nowa wersja czeka, dopóki obserwowana usługa nie wemituje tablicy z bohaterami.
- Obserwujący usługę komponent czeka na dane bohaterów.
- Metoda `subscribe()` przekazuje otrzymaną tablicę do wywołania zwrotnego (tutaj wyrażenie lambda, funkcja strzałkowa), które przypisuje zawartość bohaterów do lokalnego pola komponentu.

2.7 Pokazywanie komunikatów

6.1. Pokazywanie komunikatów - nowy komponent

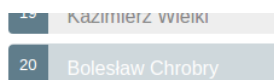
- Dodamy do aplikacji dodatkowy komponent `MessagesComponent` odpowiedzialny za wyświetlanie komunikatów. Polecenie:

  ```
  ng generate component messages
  ```

- Komponent, komunikaty będziemy wyświetlać na dole aplikacji, w jej głównym komponencie:

  src/app/app.component.html

  ```
  <h1>{{title}}</h1>
  <app-heroes></app-heroes>
  <app-messages></app-messages>
  ```



Informacje o: BOLESŁAW CHROBRY

id: 20
nazwa: Bolesław Chrobry

messages works!

6.2. Pokazywanie komunikatów - nowa usługa

- Dodajemy do aplikacji usługę odpowiedzialną za przechowywanie i obsługę komunikatów. Polecenie:

  ```
  ng generate service message
  ```

- Ustawiamy następującą zawartość usługi:

  src/app/message.service.ts

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

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

- Usługa przechowuje w pamięci dodane komunikaty (metoda `add()`) i pozwala na ich usunięcie (metoda `clear()`).

6.3. Usługa w usłudze

- Importujemy usługę `MessageService` wewnątrz usługi `HeroService`
- W konstruktorze deklarujemy prywatne pole `messageService` - usługa będzie utworzona za pomocą mechanizmu wstrzykiwania zależności. Typowe zastosowanie usługi w usłudze.
- Wysyłamy/dodajemy komunikat do usługi, kiedy bohaterowie zostaną pobrani.

src/app/hero.service.ts

```
...
import { MessageService } from './message.service';

@Injectable({
  providedIn: 'root'
})
export class HeroService {
  constructor(private messageService: MessageService) { }

  // TODO: wysłanie komunikatu _po_ pobraniu bohaterów
  getHeroes(): Observable<Hero[]> {
    this.messageService.add('Usługa HeroService: pobrano bohaterów');
    return of(HEROES);
  }
}
```

6.4. Wyświetlenie komunikatów z `MessageService`

- `MessagesComponent` powinien wyświetlać wszystkie komunikaty z usługi.
- Importujemy więc usługę - będzie potrzebna :)
- Tworzymy publiczne pole `messageService`. Tylko do takich można robić wiązania w szablonie.
- Usługa, reprezentowana w pamięci jako singleton, będzie wstrzyknięta do komponentu.

src/app/messages/messages.component.ts

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

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

  constructor(public messageService: MessageService) { }

  ngOnInit() {
  }
}
```

6.5. Szablon komponentu `MessageComponent`

- Szablon wykorzystuje wiązanie bezpośrednio do usługi `MessageService`.
- Jest lista elementów `<div>`, jest przycisk do usuwania komunikatów z usługi.
- Komponent jest widoczny tylko, jeśli przechowuje jakieś komunikaty.

src/app/messages/messages.component.html

```
<div *ngIf="messageService.messages.length">

  <h2>Komunikaty</h2>
  <button class="clear"
          (click)="messageService.clear()">clear</button>
  <div *ngFor='let message of messageService.messages'> {{message}} </div>

</div>
```

- Przyda się też dodanie styli do tego komponentu, całość kodu tutaj: `https://angular.io/tutorial/toh-pt4#final-code-review`

src/app/messages/messages.component.html

```
/* MessagesComponent's private CSS styles */
h2 {
  color: red;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
...
```

6. Wygląd komponentu z jednym komunikatem

Koniec części dotyczącej usług.

3 Źródła

Źródła

- `https://angular.io/`
- `https://angular.io/tutorial`
- `https://angular.io/guide/rx-library`
- `https://github.com/reactivex/rxjs`
- `http://reactivex.io/rxjs/manual/index.html`
- `https://pl.wikipedia.org/wiki/Obserwator_(wzorzec_projektowy)`
- `https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html`
- `https://codecraft.tv/courses/angular/http/http-with-observables/`
- `https://typeofweb.com/2017/05/19/observable-rxjs/`