Angular 7 - tutorial o bohaterach cz. 2

Tworzenie serwisów Web 2.0

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

13 kwietnia 2019

Plan prezentacji

Spis treści

1 Tutorial

1.1 Trasowanie

7. Trasowanie

Dodamy tutaj komponent do trasowania i nauczymy się przemieszczania między widokami.

Rosną wymagania wobec naszej aplikacji:

- dodatkowy widok - panel z najważniejszymi bohaterami,
- dodanie możliwości nawigowania pomiędzy widokiem panelu a widokiem z listą bohaterów,
- po kliknięciu na bohatera w dowolnym z obu widoków, przechodzimy do widoku szczegółów tego bohatera,
- po kliknięciu na głęboki odnośnik (ang. *deep link*) we wiadomości email, aplikacja otwiera widok ze szczegółami danego bohatera.

W spełnieniu tych wymagań pomoże nam moduł trasowania w Angular (Angular Router).

Diagram nawigacji

Dodajemy trasowanie

Lista bohaterów nie będzie wyświetlana automatycznie ale po kliknięciu na przycisk. Użytkownik może zadecydować o wyświetleniu listy.

Do nawigowania używamy mechanizmu trasowania Angular Router.

- Angular router to zewnętrzny, opcjonalny moduł Angulara `RouterModule`.
- Mechanizm trasowania jest kombinacją wielu:
  - dostawców usług (`RouterModule`),
  - wielu dyrektyw (`RouterOutlet`, `RouterLink`, `RouterLinkActive`)
  - i konfiguracji (`Routes`). Od konfiguracji zaczniemy.

`<base href>`

- W pliku `index.html` na samej górze sekcji `<head>` powinien być element `<base href="...">` (albo skrytp, który go generuje).

index.html

```
<head>
  <base href="/">
  ...
```

- To jest podstawowy element dla działania mechanizmu trasowania Angular.

7.1. Dodatkowy moduł odpowiedzialny za rooting

- Generujemy sobie nowy moduł, zwyczajowo nazywa się on `AppRoutingModule` (`app-routing`). Polecenie:

```
ng generate module app-routing --flat --module=app
```

- opcja `--flat` nie tworzy osobnego folderu, dodaje moduł do `src/app`

- opcja `--module=app` dodaje tworzony moduł do sekcji `imports` modułu `AppModule`.
- Moduł po wygenerowaniu:

src/app/app-rooting.module.ts (po wygenerowaniu)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModule { }
```

Osobny moduł do nawigacji

- Rozbudowa aplikacji zwykle skutkuje znacznym rozrostem liczby tras.
- Konieczne wydaje się przeniesienie konfiguracji tras do osobnego modułu - *modułu nawigacyjnego*.
- Przyjęta konwencja mówi o umieszczaniu w nazwie modułu nawigacyjnego słowa „Routing" poprzedzonego nazwą modułu deklarującego trasy dla komponentów.
- Przykład: `app-routing.module.ts`, `store-routing.module.ts`.

Typowe elementy modułu nawigacyjnego

Typowe elementy modułu nawigacyjnego:

- Konfiguracja tras jest przechowywana w zmiennej. Użycie zmiennych narzuca porządek wewnątrz modułu gdybyśmy chcięli w przyszłości go eksportować.
- Dodajemy `RouterModule.forRoot(routes)` do sekcji `imports`.
- Dodajemy `RouterModule` do sekcji `exports` tak, żeby komponenty w drugim module (towarzyszącym) miały dostęp do deklaracji mechanizmu trasowania takich jak `RouterLink` czy `RouterOutlet`.
- Nie ma sekcji `declarations`. To jest odpowiedzialność modułu towarzyszącego.

7.2. Dostosowanie modułu odpowiedzialnego za rooting

- Moduł odpowiedzialny za trasowanie nie używa komponentów, można usunąć sekcję `declarations: []`.
- Usuwamy też odwołania do `CommonModule`.
- Potrzebujemy za to modułów `Routes` i `RouterModule`, dodajemy więc ich import.
- Do głównego modułu aplikacji eksportujemy `RouterModule`, będzie można korzystać z trasowania.
- Moduł rootingu po zmianach:

src/app/app-rooting.module.ts (po pierwszych zmianach)

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

@NgModule({
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

7.3. Dodawanie tras

- Trasy określają, który widok pokazać przy konkretnym adresie URL (wpisanym, kliknięty, ...)
- Zwykle trasy w Angularze mają dwie właściwości:
  - `path` - łańcuch znaków dopasowywany do adresu URL przeglądarki
  - `component` - komponent, który moduł trasowania ma utworzyć przy danym adresie URL
- Trasa powinna zadziałać dla adresu `localhost:4200/heroes`
- Importujemy potrzebny komponent i definiujemy tablicę z trasami.
- Moduł musi być zainicjowany i nadsłuchiwać na określonych trasach:

src/app/app-rooting.module.ts (po kolejnych zmianach)

```
...
import { HeroesComponent }      from './heroes/heroes.component';

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];

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

7.4. Dodawanie tras i dyrektywa `RouterOutlet`

- Dodajemy moduł `RouterModule` Angulara do sekcji `imports` modułu i konfigurujemy go ścieżkami określonymi w tablicy `routs`
- Wykorzystujemy tutaj metodę `forRoot()`

  src/app/app-rooting.module.ts (po kolejnych zmianach)

  ```
  ...
  imports: [ RouterModule.forRoot(routes) ],
  ...
  ```

- W szablonie głównego komponentu (`AppComponent`) zastępujemy `<app-heroes>` wyjściem trasowania czyli `<router-outlet>`:

  src/app/app.component.html

  ```
  <h1>{{title}}</h1>
  <router-outlet></router-outlet>
  <app-messages></app-messages>
  ```

Metody `forRoot()` i `forChild()`

- Moduł trasowania może być importowany wiele razy: jeden raz dla każdego leniwe wczytywanego pakietu.
- Ponieważ mechanizm trasowania ma do czynienia z globalnym, współdzielonym zasobem - lokalizacją, nie możemy mieć więcej niż jedna aktywna usługa trasowania.
- Stąd mamy dwa sposoby utworzenia tego modułu: `RouterModule.forRoot` and `RouterModule.forChild`.
- `forRoot` tworzy moduł, który zawiera wszystkie dyrektywy, podane ścieżki i usługę trasowania samą w sobie.
- `forChild` tworzy moduł, który zawiera wszystkie dyrektywy i podane ścieżki, ale nie zawiera samej usługi trasowania.
- Wywołujemy metodę `forRoot()` ponieważ mechanizm trasowania jest dostarczony do korzenia aplikacji (do głównego modułu).
- Metoda `forRoot()` dostarcza dostawcę usługi trasowania i dyrektywy potrzebne do obsługi tras, i wykonuje pierwszą nawigację zależną od bieżącej wartości adresu URL.

Wyjście trasowania czyli Router outlet

- Z każdym adresem w przeglądarce, np. `\heroes`, mechanizm trasowania poprzez zdefiniowaną ścieżkę będzie wyświetlał przypisany jej komponent, u nas `HeroesComponent`.
- Trzeba jednak poinformować Angular gdzie ma ten komponent wyświetlać.
- Taką informacją jest element `<router-outlet>`, dodajemy go np. wewnątrz szablonu głównego komponentu.

- `<router-outlet>` - wyjście, wypływ trasowania - jedna z dyrektyw dostarczona w module `RouterModule`.
- Mechanizm trasowania wyświetla każdy komponent bezpośrednio za znacznikiem `<router-outlet>`, zgodnie z tym, jak użytkownik zmienia adresy URL w przeglądarce.

7. Bohaterowie wyświetlani przez mechanizm trasowania

Adres w przeglądarce, `localhost:4200/heroes` określa, który komponent wczytać.

7.5. Dodanie linków nawigacyjnych

- Zamiast ręcznie wpisywać odpowiednie adresy URL w przeglądarce, dodamy w szablonie menu nawigacyjne.
- Będzie to znacznik `<a>` ze specjalnym atrybutem.
- Po kliknięciu w dany element, nawigacja przeniesie nas do komponentu związanego z danym adresem.
- W naszym przypadku będzie to komponent `HeroesComponent`.

src/app/app.component.html

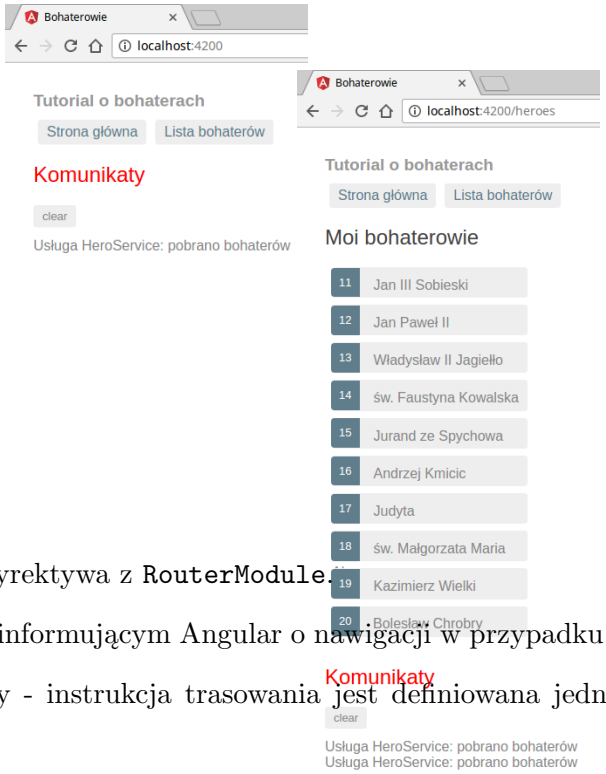
```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/">Strona główna</a>
  <a routerLink="/heroes">Lista bohaterów</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

7.6. Dodanie odpowiednch stylów CSS dla nawigacji

src/app/app.component.css

```
h1 {
  font-size: 1.2em;
  color: #999;
  margin-bottom: 0;
}
h2 {
  font-size: 2em;
  margin-top: 0;
  padding-top: 0;
}
nav a {
  padding: 5px 10px;
  text-decoration: none;
  margin-top: 10px;
  display: inline-block;
  background-color: #eee;
  border-radius: 4px;
}
nav a:visited, a:link {
  color: #607D8B;
}
nav a:hover {
  color: #039be5;
  background-color: #CFD8DC;
}
nav a.active {
  color: #039be5;
}
```

Dwie dyrektywy trasowania w działaniu



- `RouterLink` to kolejna dyrektywa z `RouterModule`.
- Jest związana z napisem informującym Angular o nawigacji w przypadku kliknięcia linku.
- Link nie jest dynamiczny - instrukcja trasowania jest definiowana jednorazowo według podanej ścieżki.
- `/heroes` wkazuje na komponent `HeroesComponent`.

7.7. Panel najważniejszych bohaterów

- Nawigacja ma sens kiedy jest wiele widoków. Tworzymy nowy widok.
- W tym celu tworzymy nowy komponent `DashboardComponent`, z którego i do którego będzie można przechodzić z innych widoków. Polecenie:

```
ng generate component dashboard
```

src/app/dahsboard/dashboard.component.ts

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

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

  constructor(private heroService: HeroService) { }
  ngOnInit() {
    this.getHeroes();
  }
  getHeroes(): void {
    this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes.slice(1, 5));
  }
}
```

Usługa `HeroService` w `DashboardComponent`

- Importujemy klasę `Hero`, usługę `HeroService` oraz `OnInit`.
- Dodajemy tablicę bohaterów, konstruktor z usługą i obsługę `ngOnInit()`.
- Używamy `Array.slice()` żeby wybrać czterech najważniejszych bohaterów.
- Usługa `HeroService` jest singletonem, jedna globalna instancja dostępna dla wszystkich komponentów.
- Wykorzystując wstrzykiwanie zależności możemy wykorzystać `HeroService` w naszym Panelu.

7.8. Szablon panelu najważniejszych bohaterów

- Widok prezentuje tablicę z odnośnikami do najważniejszych bohaterów.
- Widok wyświetla tylko 4 bohaterów - cztery kolumny.

src/app/dahsboard/dashboard.component.html

```
<h3>Najważniejsi, najlepsi bohaterowie</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
```

- Styl linków określono w `dashboard.component.css`.

src/app/dahsboard/dashboard.component.css

```
/* DashboardComponent's private CSS styles */
[class*='col-'] {
  float: left;
  padding-right: 20px;
  padding-bottom: 20px;
}
[class*='col-']:last-of-type {
  padding-right: 0;
}
...
```

7.9. Konfigurujemy ścieżkę do Panelu

W module nawigacyjnym aplikacji (`AppRoutingModule`):

- importujemy `DashboardComponent`,
- konfigurujemy ścieżkę do niego, `/dashboard` zamiast `/` (Strona główna).

src/app/app-rooting.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent }      from './heroes/heroes.component';
import { DashboardComponent }   from './dashboard/dashboard.component';

const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

7.10. Ustawiamy przekierowanie do Panelu

- Domyślnie uruchamiany adres `/` nie wczytuje żadnego komponentu, wyświetla sam tytuł aplikacji.
- Chcemy aby domyślnie wyświetlany był właśnie tworzony Panel, aby była uruchamiana ścieżka `/dashboard`.
- Używamy ścieżki przekierowania (ang. *redirect route*), w tablicy określającej ścieżki dodajemy:

src/app/app-rooting.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent }      from './heroes/heroes.component';
import { DashboardComponent }   from './dashboard/dashboard.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

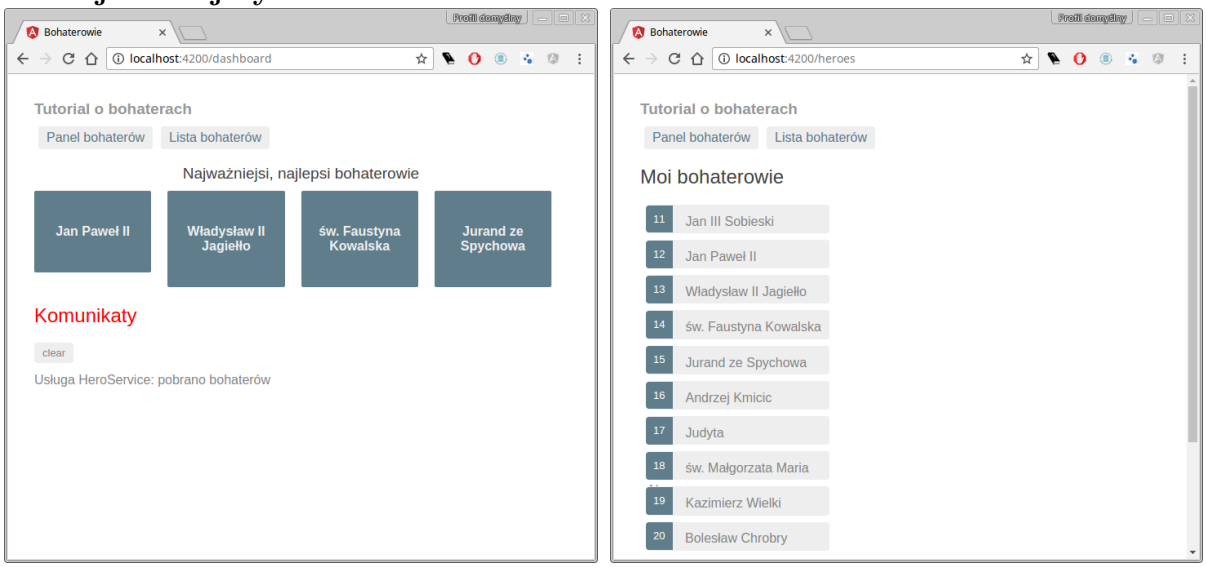
7.11. Poprawiamy nawigację w szablonie `AppComponent`

- Dodajemy link nawigacyjny wskazujący na Panel w szablonie naszego głównego komponentu.
- Można usunąć link wskazujący na stronę główną.
- Wpisanie w przeglądarce adresu `/` przekieruje nas na `/dashboard` i wyświetli Panel najważniejszych bohaterów.

src/app/app.component.html

```
<h1>{{title}}</h1>
<nav>
  <!-- <a routerLink="/">Strona główna</a> -->
<a routerLink="/dashboard">Panel bohaterów</a>
  <a routerLink="/heroes">Lista bohaterów</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Panel najważniejszych bohaterów w działaniu



Nawigowanie do szczegółów wybranego bohatera

- Szczegółowe dane o wybranym bohaterze wyświetlają się na dole komponentu `HeroesComponent`.
- Odbywa się to poprzez komponent podrzędny `HeroDetailComponent`.
- Chcemy mieć również możliwość dostać się do szczegółów bohareta poprzez:
  - Panel, klikając dowolnego bohatera,
  - listę bohaterów, kllikając wybranego bohatera,
  - głębokie dowiązanie, adres URL wpisany w przeglądarce.
- Nie wystarczy tutaj wiązanie między komponentem nadrzędnym i podrzędnym:

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

- Potrzebujemy adresu URL z parametrem `id` wybranego bohatera.

7.12. Wybór konkretnego bohatera

- Moduł z trasowaniem musi importować `HeroDetailComponent`.
- Wybór bohatera określimy w w ścieżce z prametrem: `detail/:id`
- `:id` to numer bohatera, którego szczegóły chcemy wyświelić i edytować.

src/app/app-routing.module.ts (moduł z trasowaniem)

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent }      from './heroes/heroes.component';
import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

7.13. Odnośniki do bohatera w Panelu

- Komponent `DashboardComponent` powinien zawierać linki do poszczególnych bohaterów.
- Należy więc delikatnie zmodyfikować odnośnik.
- Używamy `*ngFor` i dyrektywy `routerLink`.

src/app/dashboard/dashboard.component.html

```
<h3>Najważniejsi, najlepsi bohaterowie</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4" routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
```

7.14. Odnośniki do bohatera w liście bohaterów

- Bardzo podobne zmiany wprowadzamy w `HeroesComponent`.
- Każdy element listy ma być odnośnikiem do szczegółów wybranego bohatera.
- Tutaj również używamy `*ngFor` i dyrektywy `routerLink`.

src/app/heroes/heroes.component.html

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

- Z klasy `HeroesComponent` można usunąć pole `selectedHero` oraz metodę `onSelect()`

7.15. Zmiany w komponencie `HeroDetailComponent` I

- Wcześniej dane bohatera potrzebne do wyświetlania szczegółów były w polu `hero`.
- Dane przekazywał komponent nadrzędny (pole `@Input`).
- Teraz dane bohatera chcemy wczytywać z usługi `HeroService`.
- Dodajemy odpowiednie importy i potrzebne usługi:

src/app/hero-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';
import { HeroService }  from '../hero.service';
...

export class HeroDetailComponent implements OnInit {
  hero: Hero;

  constructor(
    private route: ActivatedRoute,
    private heroService: HeroService,
    private location: Location
  ) {}
  ...
}
```

- Usługa `location` pozwala na interakcję Angulara z przeglądarką.

Zmiany w komponencie `HeroDetailComponent` - opis

- Szablon komponentu nie uległ zmianie. Będziemy jednak inaczej pobierać naszego bohatera.
- Nie pobieramy go już przez wiązanie danych komponentu nadrzędnego.
- Pobieramy dane bezpośrednio z usługi `HeroService` z id podanym w adresie URL.
- Wstrzykujemy usługi do konstruktora.

7.16. Zmiany w komponencie `HeroDetailComponent` II

- Musimy jeszcze wydobyć parametr `id` z adresu URL.
- Napiszemy w tym celu metodę `getHero()` i wywołamy ją w `ngOnInit()`:

src/app/hero-detail.component.ts

```
...

export class HeroDetailComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.getHero();
  }
  getHero(): void {
    const id = +this.route.snapshot.paramMap.get('id');
    this.heroService.getHero(id)
      .subscribe(hero => this.hero = hero);
  }
}
```

- `route.snapshot` to statyczny obraz informacji o trasowaniu zaraz po utworzeniu komponentu.
- `paramMap` to słownik zawierający dostępne prametry routingu.
- Parametry te są zawsze napisami (string). Operator + konwertuje je na liczbę.

7.17. Zmiany w usłudze `HeroService`

- W usłudze `HeroService` dodajemy metodę `getHero(id)`.
- Pobranie danych jest asynchroniczne i korzysta z wzorca obserwator.
- Bardzo łatwo będzie przejść do wykorzystanie usługi Http bez zmiany metod w `HeroService`.

src/app/hero.service.ts

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

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

  getHeroes(): Observable<Hero[]> {
    this.messageService.add('Usługa HeroService: pobrano bohaterów');
    return of(HEROES);
  }
  getHero(id: number): Observable<Hero> {
    this.messageService.add(`Usługa HeroService: pobrano bohatera o id=${id}`);
    return of(HEROES.find(hero => hero.id === id));
  }
}
```

7.18. Przycisk cofania I

- Użytkownik ma dostępne różne opcje nawigacji: przyciski nawigacyjne na górze aplikaji, adres URL.
- Dodamy trzecią opcję: przycisk powrotu do poprzedniej strony.
- Utworzymy metodę `goBack()`, która będzie korzystać z historii przeglądarki dzięki wstrzykniętej usłudze `Location`.

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

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

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

  goBack(): void {
    this.location.back();
  }
}
```

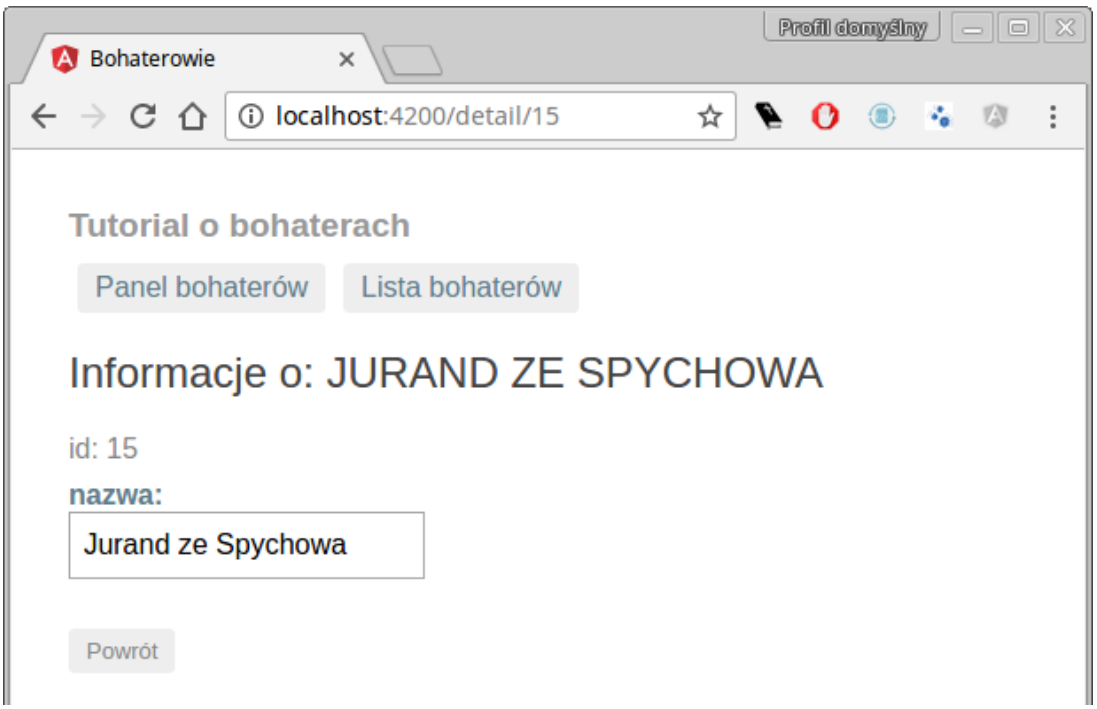
7.19. Przycisk cofania II

- W szablonie komponentu dodajmy przycisk.
- Zdarzenie `click` jest związane z metodą `goBack()`.

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>
  <button (click)="goBack()">Powrót</button>
</div>
```

Szczegóły bohatera - id w adresie URL



1.2 Usługa HTTP

8. Usługa HTTP

Dodajemy do projektu usługę HTTP (`HttpClient`):

- usługa `HeroService` pobiera dane poprzez żądania HTTP,
- użytkownik może dodać, usunąć albo zmodyfikować bohatera i zapisać zmiany poprzez protokół HTTP,
- użytkownik może wyszukiwać bohatera po jego nazwie.

8.1. Dodanie usługi 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`
  - zaimportować `HttpClientModule` z modułu `@anguar/common/http`
  - dodać go do tablicy `@NgModule.imports`

src/app/app.module.ts

```
...
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  declarations: [
    AppComponent, HeroesComponent, HeroDetailComponent, MessagesComponent, DashboardComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    HttpClientModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Moduł `In-memory Web API`

- Zamiast korzystać z rzeczywistego serwera moża go tylko symulować.
- `In-memory Web API` przechwytuje żądania z usługi `HttpClient`, dane są przechowywane tylko w pamięci operacyjnej.
- Jest to bardzo wygodne podczas wstępnych etapów pracy nad aplikacją albo podczas pisania tutoriala.

8.2. Moduł `In-memory Web API` I

Jak wykorzystać `In-memory Web API`?

- Instalujemy paczkę z repozytoriów `npm`:

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

- importujemy dwie klasy:

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';
```

- Dodajemy `HttpClientInMemoryWebApiModule` do tablicy `@NgModule.imports` (po wcześniejszym zaimportowaniu `HttpClient`)

```
    HttpClientModule,

    // Moduł HttpClientInMemoryWebApiModule przechwytuje żądania HTTP
    // i zwraca symulowane odpowiedzi serwera.
    // Przy działającym rzeczywistym serwerze wystarczy usunąć.
    HttpClientInMemoryWebApiModule.forRoot(
      InMemoryDataService, { dataEncapsulation: false }
    )
```

- Metoda konfiguracyjna `forRoot()` przyjmuje klasę `InMemoryDataService`, która uruchamia bazę danych w pamięci.

8.2. Moduł `In-memory Web API` II

- Tworzymy plik `src/app/in-memory-data.service.ts`:

src/app/in-memory-data.service.ts

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

export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      {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"}
    ];
    return {heroes};
  }
}
```

- Plik ten zastąpi `mock-heroes.ts` (można go już bezpiecznie usunąć).
- Wyłączenie `In-memory Web API` pozwoli na pracę usługi HTTP bezpośrednio na serwerze.

8.3. Wykorzystanie usługi HTTP w `HeroService`

- importujemy potrzebne klasy z modułu `@angular/common/http`:

src/app/hero.service.ts

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

- wstrzykujemy `HttpClient` do konstruktora klasy `HeroService`:

src/app/hero.service.ts

```
  constructor(
    private http: HttpClient,
    private messageService: MessageService) { }
```

- używamy usługi `MessageService` do logowania komunikatów:

src/app/hero.service.ts

```
  /** Zapisujemy komunikat usługi HeroService korzystając z MessageService */
  private log(message: string) {
    this.messageService.add('HeroService: ' + message);
  }
```

- definiujemy `heroesUrl` z adresem servera z danymi bohaterów:

src/app/hero.service.ts

```
  private heroesUrl = 'api/heroes';  // Adres URL serwerowego API
```

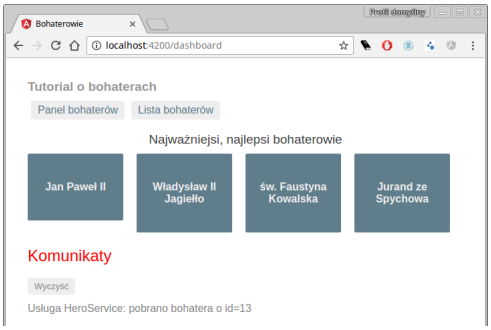
8.4. Pobieramy bohaterów przy użyciu `HttpClient`

- zamieniamy pobranie bohaterów korzystające z `RxJs` na usługę `HttpClient`:

src/app/hero.service.ts (getHeroes with RxJs 'of()')

```
// getHeroes(): Observable<Hero[]> {
//   return of(HEROES);
// }
/** Pobieranie bohaterów (metoda GET) z serwera */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
}
```

- odświeżamy przeglądarkę - dane wczytywane z servera poprzez API HTTP
- zastąpiliśmy `of` poprzez `http.get` - apllikacja działa bez zmian ponieważ obie funkcje zwracają `Observable<Hero[]>`



Dane zwracane z `HttpClient` I

- Wszystkie metody `HttpClient` zwracają jakiś obserwowany obiekt (`Observable`).
- Protokół HTTP dla pojedynczego zapytania zwraca pojedynczą odpowiedź.
- Obiekt obserwowany może zwrócić wiele wartości w czasie.
- Obiekt obserwowany z `HttpClient` zawsze zwraca pojedynczą wartość i kończy działanie.
- Metoda `HttpClient.get` u nas zwraca `Observable<Hero[]>` (obserwowanie obiektu z tablicą bohaterów).
- Zwraca po prostu pojedynczą tablicę bohaterów.

Dane zwracane z `HttpClient` II

- Domyślnie `HttpClient.get` zwraca zawartość odpowiedzi jako obiekt JSON bez określonego typu.
- Zastosowanie opcjonalnego określenia typu (np. `<Hero[]>`) zwraca obiekt określonego typu.
- Struktura zwracanych danych JSON jest zależna od serwera danego API.
- Nasz tutorial o bohaterach zwraca dane bohaterów jako tablicę.
- Inne interfejsy API mogą pochować potrzebne nam dane wewnątrz obiektu.
- Być może będzie trzeba wydobyć te dane przetwarzając wynik `Observable` za pomocą operatora `map` w `RxJs`.

8.5. Obsługa błędów I

- Nie zawsze dane z serwera uda się odczytać.
- Metoda `HeroService.getHeroes()` powinna wyłapywać błędy i odpowiednio reagować.
- Aby przechwycić błędy wykorzystujemy metodę `pipe()` obiektów obserwowanych przekierowującą wyniki z `http.get()` do operatora `catchError()` z `RxJs`.
- Importujemy `catchError` z `rxjs/operators` razem z `map` i `tap`:

src/app/hero.service.ts

```
import { catchError, map, tap } from 'rxjs/operators';
```

- Rozszerzamy obsługę zwracanych wyników metodą `.pipe()` dodając operator `catchError()`:

src/app/hero.service.ts

```
/** Pobieranie bohaterów (metoda GET) z serwera */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      catchError(this.handleError('getHeroes', []))
    );
}
```

8.5. Obsługa błędów II

- Operator `catchError()` obiekt obserwowany, które zwrócił błąd.
- Przekazuje błąd do metody `handleError()` - obsługa błędów, która podejmuje żądane przez nas działania.
- Metoda `handleError()` zgłasza błąd i zwraca nieszkodliwy wynik, żeby aplikacja mogła działać dalej.

src/app/hero.service.ts

```
/**
 * Obsługa operacji Http zakończonej błędem.
 * Pozwala na dalszą pracę aplikacji.
 * @param operation - nazwa operacji, która zakończyła się błędem
 * @param result - opcjonalna wartość zwracana jako obiekt obserwowany
 */
private handleError<T> (operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: wysłanie błędu do zdalnej obsługi rejestrowania błędów
    console.error(error); // wyświetlanie błędów w konsoli

     // TODO: lepiej wykonana transformacja błędów dla większej wygody użytkowników
    this.log(`${operation} zakończona błędem: ${error.message}`);

     // Pozwalamy na dalszą pracę aplikacji zwracając pusty wynik.
    return of(result as T);
  };
}
```

8.5. Obsługa błędów III

- Metoda `handleError()` jest wykorzystywana przez wiele metod usługi `HeroService` - została więc uogólniona.
- Zamiast bezpośrednio obsługiwać błędy zwraca ona do `catchError` funkcję obsługi błędów konfigurowaną:
  - nazwą operacji, która zakończyła się błędem
  - bezpieczną zwracaną wartością.
- Jest zwracany błąd do konsoli (`console.error`), ładny komunikat dla użytkownika (`this.log`) i zwracane są bezpieczne dane.
- Każda metoda zwraca obiekt obserwowany innego typu.
- Metoda `handleError()` wykorzystuje parametr typowany aby zwracana wartość była odpowiedniego typu.

8.6. Operator `tap` z `RxJs`

- Metody usługi `HeroService` będą śledzić przepływ między oberwowanymi obiektami i wyświetlać komunikaty w pasku na dole aplikacji (metoda `log()`.
- Korzystamy tutaj z operatora `tap`, który podgląda wartości obserwowanych obiektów, coś z nimi robi i przekazuje je dalej.
- Wywołanie zwrotne operatora `tap` nie zmienia wartości obserwowanych obiektów.
- Końcowa wersja metody `getHeroes()` z operatorem `tap`:

src/app/hero.service.ts

```
/** Pobieranie bohaterów (metoda GET) z serwera */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      tap(heroes => this.log(‘pobrano bohaterów‘)),
      catchError(this.handleError('getHeroes', []))
    );
}
```

8.7. Pobranie bohatera po id

- Większość internetowych API pozwala na pobieranie obiektów po ich ID w formie `api/hero/:id` 9np. `api/hero/12`.
- Wersja metody `getHero()` korzystającej z usługi http:

src/app/hero.service.ts

```
/** Pobierz metodą GET bohatera po id. Zwraca błąd 404 jeśli id nie istnieje */
getHero(id: number): Observable<Hero> {
  const url = ‘${this.heroesUrl}/${id}‘;
  return this.http.get<Hero>(url).pipe(
    tap(_ => this.log(‘pobrano bohatera id=${id}‘)),
    catchError(this.handleError<Hero>(‘getHero id=${id}‘))
  );
}
```

- Trzy najważniejsze różnice:
  - tworzymy adres url korzystając z potrzebnego nam id bohatera,
  - serwer zwraca pojedynczego bohatera a nie ich tablicę,
  - `getHero` zwraca `Observable<Hero>` (obserwowanie obiektu pojedynczego bohatera), a nie całej tablicy.

8.8. Aktualizacja bohaterów I

- Edycja nazwy bohatera w widoku. Aktyalizacja nagłówka w trakcie pisania.
- Wciśnięcie przycisku ”Powrót”powoduje utratę zmian.
- Zapis danych na stałe wymaga wysłania ich do serwera.
- Na końcu szablonu komponentu `HeroDetailComponent` dodajemy przycisk do zapisu, metoda `save()`:

src/app/hero-detail/hero-detail.component.html (save)

```
<button (click)="save()">Zapisz</button>
```

- Dodajemy metodę `save()` do klasy komponentu:

src/app/hero-detail/hero-detail.component.ts (save)

```
save(): void {
   this.heroService.updateHero(this.hero)
     .subscribe(() => this.goBack());
}
```

- Metoda `save()` zapisuje zmiany na stałe korzystając z metody `updateHero()` po czym wraca do poprzedniego widoku.

8.8. Aktualizacja bohaterów II

- Metoda `updateHero()` jest podobna do `getHeroes()` ale wykorzystuje `http.put()`.
- Przyjmuje trzy parametry: adres URL, dane do aktualizacji (w naszym przypadku dane zmodyfikowanego bohatera) oraz opcje.

src/app/hero.service.ts

```
/** Aktualizacja bohatera na serwerze (metoda PUT) */
updateHero (hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, httpOptions).pipe(
    tap(_ => this.log(`aktualizacja bohatera o id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}
```

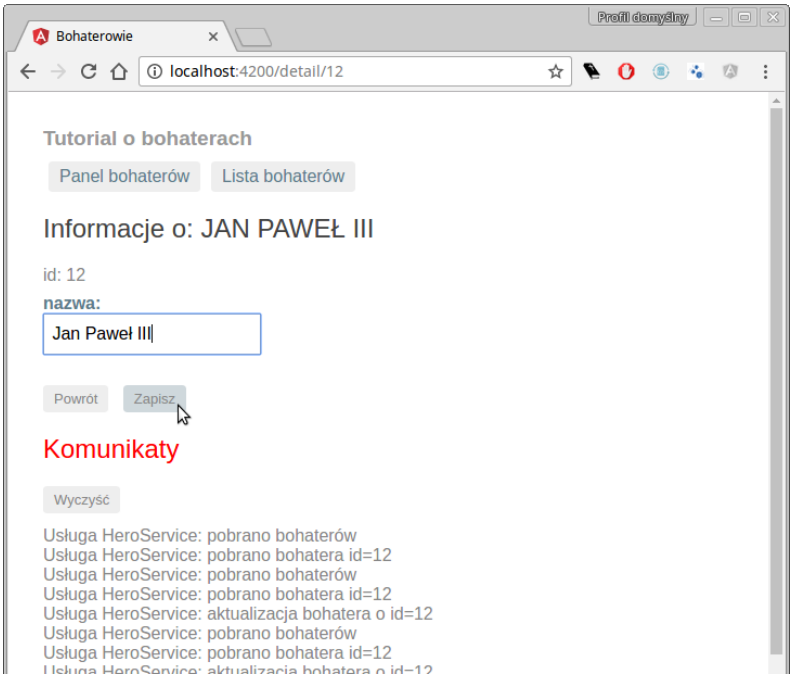
- API oczekuje specjalnego nagłówka protokołu HTTP przy żądaniu zapisu danych na serwerze.
- Nagłówek zdefiniowany na stałę wewnątrz obketu `httpOptions`:

src/app/hero.service.ts

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

8.8. Aktualizacja bohaterów III

Odświeżamy widok w przeglądarce, zmieniamy nazwę bohatera, zapisujemy i sprawdzamy czy działa...



8.9. Dodanie nowego bohatera I

- Dodanie bohatera wymaga tylko jego nazwy.
- Można użyć elementu `<input>` razem z przyciskiem.
- Dodajemy poniższy kod do szablonu bohaterów poniżej nagłówka:

src/app/heroes/heroes.component.html (add)

```
<div>
  <label>Nazwa bohatera:
    <input #heroName />
  </label>
  <!-- (click) przekazuje wprowadzone dane do metody add() a następnie czyści pole input -->
  <button (click)="add(heroName.value); heroName.value=''">Dodaj</button>
</div>
```

- Dla niepustej nazwy metoda `add()` tworzy obiekt bohatera (bez pola id) i przekazuje go do metody `addHero()` usługi `HeroServices`:

src/app/heroes/heroes.component.ts (add)

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.addHero({ name } as Hero)
    .subscribe(hero => {
      this.heroes.push(hero);
    });
}
```

8.9. Dodanie nowego bohatera II

- Jeśli metoda `addHero()` pomyślnie zapisze dane na serwerze, to wywołanie zwrotne `subscribe` otrzyma nowego bohatera i doda go do wyświetlanej tablicy bohaterów.
- Dodajemy metodę `addHero()` do klasy `HeroService`:

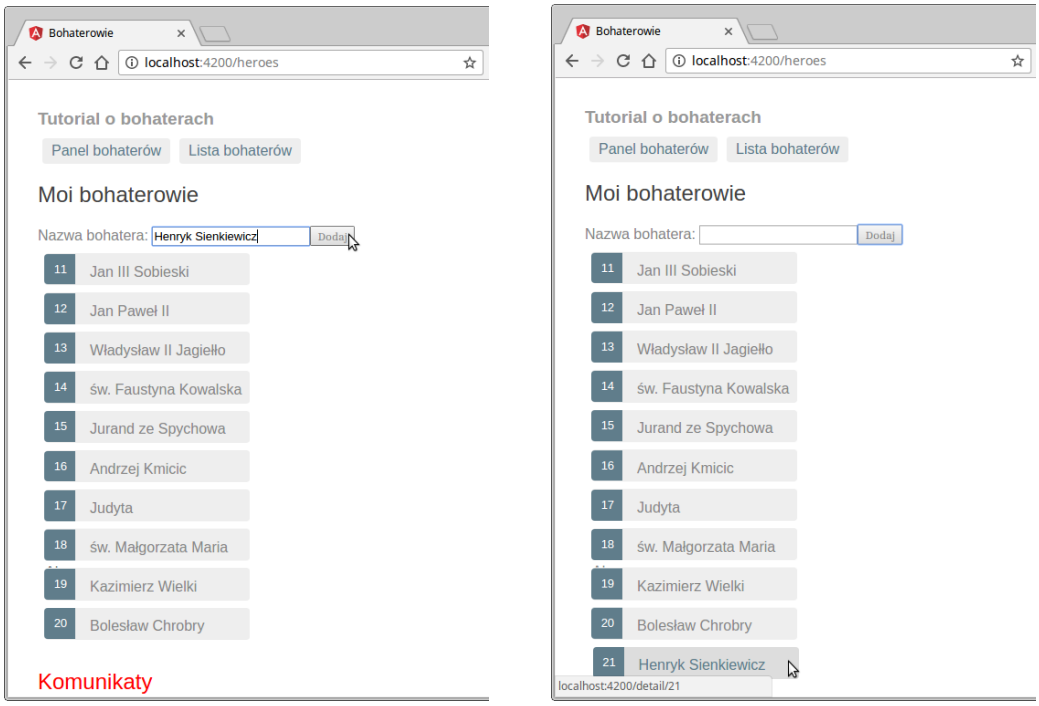
src/app/hero.service.ts (addHero)

```
/** Dodanie nowgo bohatera do serwera (metoda POST) */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(
    tap((hero: Hero) => this.log(`dodano bohatera o id=${hero.id}`)),
    catchError(this.handleError<Hero>('addHero'))
  );
}
```

- Metoda `addHero()` różni się od `updateHero()` na dwa sposoby:
  - wywołuje `Http.post()` zamiast `put()`,
  - oczekuje, że serwer sam wygeneruje nowe id dla nowego bohatera i zwróci obserwowalny obiekt `Observable<Hero>`

8.9. Dodanie nowego bohatera III

Odświeżamy widok w przeglądarce i dopisujemy bohatera...



8.10. Usuwanie bohatera I

- Każdy bohater na liście powinien mieć przycisk usuwania.
- W szablonie bohaterów dodajemy przycisk usuwania:

src/app/heroes/heroes.component.html (lista bohaterów)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
    <button class="delete" title="delete hero" (click)="delete(hero)">x</button>
  </li>
</ul>
```

- Ładny wygląd i pozycja przycisku są ustalane za pomocą kodu CSS, zobacz `https://angular.io/tutorial/toh-pt6#heroescomponent`
- Obsługa usuwania w klasie komponentu:

src/app/heroes/heroes.component.ts (lista bohaterów)

```
delete(hero: Hero): void {
  this.heroes = this.heroes.filter(h => h !== hero);
  this.heroService.deleteHero(hero).subscribe();
}
```

8.10. Usuwanie bohatera II

- Usuwanie bohaterów z komponentu `HeroesComponent` jest oddelegowane od usługi ale aktualizacja własnej listy bohaterów należy już tylko do komponentu.
- Metoda `delete()` natychmiast usunie bohatera z listy zakładając, że usunięcie danych na serwerze się powiedzie.
- Ciągle korzystamy z metody `subscribe()` chociaż nie mamy co zrobić ze zwróconym usuniętym bohaterem.
- Pominięcie metody `subscribe()` nie pozwoli na wysłanie żądania usunięcia do serwera.
- Obiekt obserwowany (`Observable`) nic nie zrobi dopóki ktoś się do niego nie podpisze (`subscribe()`).

8.10. Usuwanie bohatera III

- Dodajemy metodę `deleteHero()` do usługi `HeroService`:

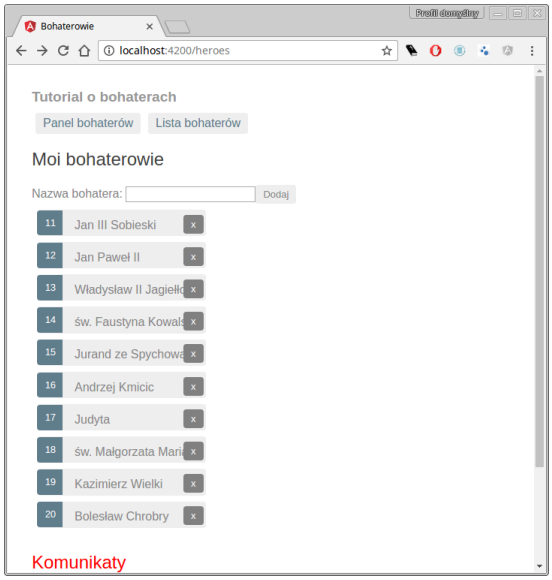
src/app/hero.service.ts (delete)

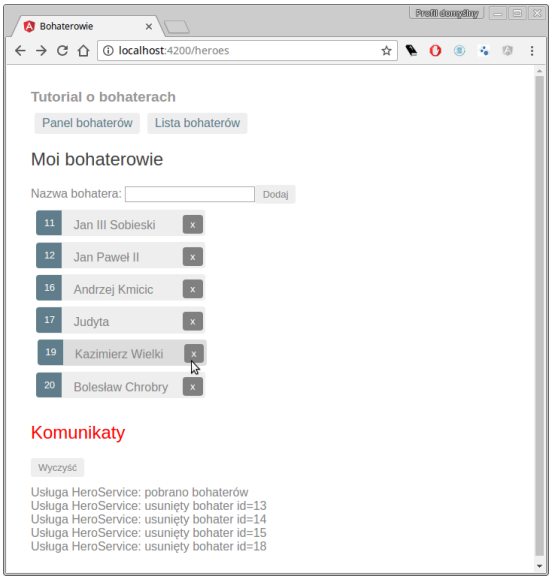
```
/** Usuwanie bohatera z serwera (metoda DELETE) */
deleteHero (hero: Hero | number): Observable<Hero> {
  const id = typeof hero === 'number' ? hero : hero.id;
  const url = `${this.heroesUrl}/${id}`;

  return this.http.delete<Hero>(url, httpOptions).pipe(
    tap(_ => this.log(`usunięty bohater id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero'))
  );
}
```

- Wywołujemy tutaj `HttpClient.delete`.
- Adres URL to zasoby naszych bohaterów razem z numerem id bohatera do usunięcia.
- Nie przesyłamy żadnych danych jak to robiliśmy w `post` czy `put`.
- Przesyłamy jednak opcje, nagłówek HTTP w obiekcie `httpOptions`.

8.10. Usuwanie bohatera IV

Odświeżamy widok w przeglądarce i usuwamy kilku bohaterów...





1.3 Wyszukiwanie po nazwie

8.11. Wyszukiwanie bohatera po nazwie I

- Skorzystamy tutaj z strumieniowego przetwarzania danych dla obiektów obserwowanych.
- Pozwoli to zminimalizować liczbę podobny żądań HTTP i ekonomicznie wykorzystać przepustowość sieci.
- Dodamy funkcję wyszukiwania bohaterów do panelu (doc. Dashboard).
- Po wpisaniu nazwy bohatera przez użytkownika wykonamy wielokrotne zapytania HTTP dla bohaterów zgodnych z wpisaną nazwą.
- Celem jest wykonanie tylko tylu żądań ile jest naprawdę konieczne.

8.11. Wyszukiwanie bohatera po nazwie II

- Zaczynamy od dodania metody `searchHeroes()` do usługi `HeroService`:

src/app/hero.service.ts

```
/* Pobranie bohaterów, który imie zawiera szukaną frazę (metoda GET) */
searchHeroes(term: string): Observable<Hero[]> {
  if (!term.trim()) {
    // if not search term, return empty hero array.
    // jeśli nie podanej frazy to zwracamy pustą tablicę bohaterów
    return of([]);
  }
  return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(
    tap(_ => this.log(`znaleziono bohaterów pasujących do "${term}"`)),
    catchError(this.handleError<Hero[]>('searchHeroes', []))
  );
}
```

- Metoda zwraca natychmiast pustą tablicę jeśli nie podano szukanej frazy.
- Reszta przypomina metodę `getHeroes()`.
- Jedyna istotna różnica to adres URL zawierający parametr, frazę do wyszukania.

8.11. Wyszukiwanie bohatera po nazwie III

- W szablonie z panelem naszych bohaterów na dole dodajemy selektor komponentu `HeroSearchComponent`

src/app/dashboard/dashboard.component.html

```
<h3>Najważniejsi, najlepsi bohaterowie</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4" routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>

<app-hero-search></app-hero-search>
```

- Aplikacja chwolowo nie działa więc tworzmy szybko nowy komponent:

  `ng generate component hero-search`

- Powstaną trzy pliki nowego komponentu, zostaną dodane odpowiednie importy w głównym module i aplikacja się uruchomi.

8.11. Wyszukiwanie bohatera po nazwie IV

- Zastępujemy wygenerowany szablon komponentu poniższym:

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

```
<div id="search-component">
  <h4>Wyszukiwanie bohaterów</h4>

  <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

  <ul class="search-result">
    <li *ngFor="let hero of heroes$ | async" >
      <a routerLink="/detail/{{hero.id}}">
        {{hero.name}}
      </a>
    </li>
  </ul>
</div>
```

- Dodajemy style do komponentu dostępnte pod adresem `https://angular.io/tutorial/toh-pt6#herosearchcomponent`.
- Podczas wpisywania nazwy w pole tekstowe, zdarzenie `keyup` wywoła metodę `search()` komponentu z nową wartością pola tekstowego.

`AsyncPipe`

- Pętla `*ngFor` iteruje po obiektach bohaterów...
- Jednak pętla `*ngFor` iteruje po liście nazwanej `heroes$`
- Symbol `$` to konwencja oznaczania że `heroes$` to `Observable` a nie tablica.
- Pętla `*ngFor` nie może nic zrobić z obiektem `Observable`.
- Mamy tutaj jednak wykorzystany filtr `AsyncPipe`, ktory automatycznie jest przypisany (`subscribe()`) do obiektu `Observable`, obserwuje zmiany na liście znalezionych bohaterów.

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

```
...
<ul class="search-result">
  <li *ngFor="let hero of heroes$ | async" >
    <a routerLink="/detail/{{hero.id}}">
      {{hero.name}}
    </a>
  </li>
</ul>
...
```

8.11. Wyszukiwanie bohatera po nazwie V - zawartość HeroSearchComponent

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

```
import { Component, OnInit } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-hero-search',
  templateUrl: './hero-search.component.html',
  styleUrls: [ './hero-search.component.css' ]
})
export class HeroSearchComponent implements OnInit {
  heroes$: Observable<Hero[]>;
  private searchTerms = new Subject<string>();
  constructor(private heroService: HeroService) {}

  // wrzucamy szukaną frazę do strumienia obiektów obserwowanych
  search(term: string): void { this.searchTerms.next(term); }

  ngOnInit(): void {
    this.heroes$ = this.searchTerms.pipe(
      // czekamy 300ms po każdym naciśnięciu klawisza zanim rozważymy wpisaną frazę
      debounceTime(300),
      // ignorujemy nową frazę jeśli jest taka sama jak poprzednia
      distinctUntilChanged(),
      // przełączamy się do szukania przy każdej zmianie szukanej frazy
      switchMap((term: string) => this.heroService.searchHeroes(term)),
    );
  }
}
```

Definicja `serchTerms` I

- Warto zauważyć deklarację listy bohaterów jako obiekt obserwowany:

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

  ```
  heroes$: Observable<Hero[]>;
  private searchTerms = new Subject<string>();
  constructor(private heroService: HeroService) {}

  // wrzucamy szukaną frazę do strumienia obiektów obserwowanych
  search(term: string): void { this.searchTerms.next(term); }
  ...
  ```

- Właściwość `searchTerms` jest zadeklarowana jako RxJS `Subject`.
- Obiekt będący `Subject` jest jednocześnie źródłem obserwowanych danych i sam może być obserwowany, jest też `Observable`.
- Można się zapisać do obserwowania obiektu `Subject` jak do każdego innego `Observable`.
- Można również dodawać wartości do takiego obiektu `Observable` wywołując jego metodę `next(wartość)`.
- Tak właśnie korzysta z niego metoda `search()`.

Definicja `serchTerms` II

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

```
heroes$: Observable<Hero[]>;
private searchTerms = new Subject<string>();
constructor(private heroService: HeroService) {}

// wrzucamy szukaną frazę do strumienia obiektów obserwowanych
search(term: string): void { this.searchTerms.next(term); }
...
```

- Metoda `search()` jest wywoływana przez wiązanie zdarzeń w polu `<input>` zdarzenia `keyup`

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

  ```
  <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
  ```

- Za każdym razem kiedy użytkownik wpisze coś w polu tekstowym, wiązanie zdarzenia wywołuje metodę `search()` z nową wartością pola tekstowego, "szukana fraza".
- Obiekt `searchTerm` staje się obiektem obserwowanym, który emituje, wysyła ciągły strumień wyszukiwanych haseł.

Łańcuchy operatorów RxJS I

- Przekazywanie nowef frazy do wyszukiwania bezpośrednio do `searchHeroes()` po każdym naciśnięciu klawisza spowoduje nadmierną liczbę żądań HTTP, niepotrzebne wyczerpywanie zasobów serwera i wykorzystywania zasobów dostępnych w ramach zakresu transmisji danych sieci komórkowej.
- Zamiast tego metoda `ngOnInit()` przetwarza potokowo frazy dodane do obserwowania za pomocą sekwencji operatorów RxJS, które zmniejszają liczbę wywołań metody `searchHeroes()`.
- Ostatecznie zwracane są na czas wyniki wyszukiwania bohaterów (każdy wynik to tablica - `Hero[]`).

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

```
this.heroes$ = this.searchTerms.pipe(
  // czekamy 300ms po każdym naciśnięciu klawisza zanim rozważymy wpisaną frazę
  debounceTime(300),

  // ignorujemy nową frazę jeśli jest taka sama jak poprzednia
  distinctUntilChanged(),

  // przełączamy się do szukania przy każdej zmianie szukanej frazy
  switchMap((term: string) => this.heroService.searchHeroes(term)),
);
```

Łańcuchy operatorów RxJS II

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

```
this.heroes$ = this.searchTerms.pipe(
  // czekamy 300ms po każdym naciśnięciu klawisza zanim rozważymy wpisaną frazę
  debounceTime(300),

  // ignorujemy nową frazę jeśli jest taka sama jak poprzednia
  distinctUntilChanged(),

  // przełączamy się do szukania przy każdej zmianie szukanej frazy
  switchMap((term: string) => this.heroService.searchHeroes(term)),
);
```

- `debounceTime(300)` czeka, aż przypływ nowych zdarzeń, nowych fraz zatrzyma się na 300 milisekund zanim przekaże frazę do wyszukania. Nigdy nowe frazy nie będą wyszukiwane częściej niż co 300 ms. Taki czas odbicia, przekazania dalej. distinctUntilChanged() ensures that a request is sent only if the filter text changed.
- `distinctUntilChanged()` zapewnia, że żądanie zostało przesłane tylko jeśli filtrowany tekst się zmienił.
- `switchMap()` wywołuje usługę wyszukiwania dla każdego wyszukiwanego hasła, które przechodzi przez `debounce` i `distinctUntilChanged`. Anuluje i odrzuca poprzednie wyszukiwane wyniki (zwracane jako `Observable`), zwracając jedynie najbardziej aktualne wyniki (zwracane jako `Observable`).

Łańcuchy operatorów RxJS III

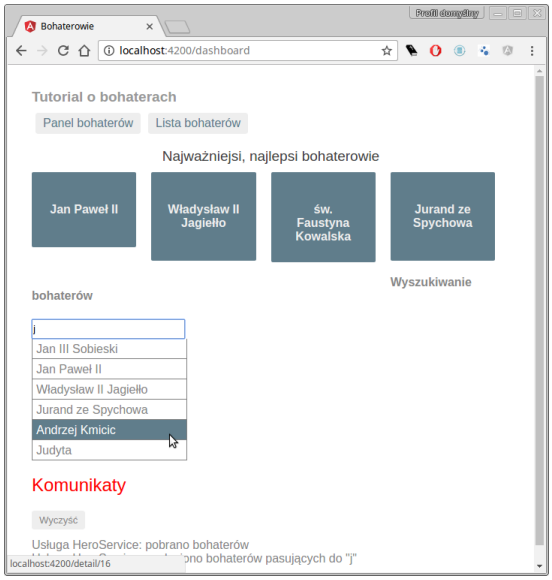
src/app/hero-search/hero-search.component.ts

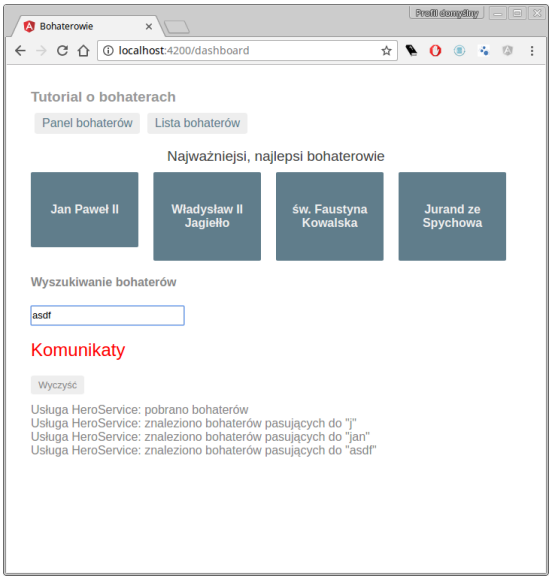
```
this.heroes$ = this.searchTerms.pipe(
  // czekamy 300ms po każdym naciśnięciu klawisza zanim rozważymy wpisaną frazę
  debounceTime(300),

  // ignorujemy nową frazę jeśli jest taka sama jak poprzednia
  distinctUntilChanged(),

  // przełączamy się do szukania przy każdej zmianie szukanej frazy
  switchMap((term: string) => this.heroService.searchHeroes(term)),
);
```

- Za pomocą operatora switchMap każde zdarzenie naciśnięcia klawisza może wywołać metodę `HttpClient.get()`
- Nawet przy 300 ms przerwy między żądaniami, można mieć wiele żądań HTTP wywołanych jednocześnie, które mogą zwracać dane w losowej kolejności, innej niż kolejność wywołania.
- `switchMap()` zachowuje oryginalną kolejność żądań, zwracając tylko obserwowany obiekt z ostatniego wywołania metody HTTP. Wyniki z poprzednich żądań są anulowane i odrzucane.
- Anulowanie poprzedniej operacji `searchHeroes()` (zwracającej `Observable`) faktycznie nie powoduje przerwania oczekującego żądania HTTP. Niepotrzebne wyniki są po prostu wyrzucane, zanim dotrą do kodu aplikacji.

8.11. Wyszukiwanie bohatera po nazwie VI

Sprawdzamy działanie naszego wyszukiwania...





2 Źródła

Źródła

- `https://angular.io/`