Dokumentacja

“HEROES”

inż. Piotr Niemczuk

[Wstęp 4](#_Toc74821238)

[Narzędzia 5](#_Toc74821239)

[Angular 5](#_Toc74821240)

[Architektura 5](#_Toc74821241)

[Component Tree 6](#_Toc74821242)

[Data binding 6](#_Toc74821243)

[Interpolation {{}} 6](#_Toc74821244)

[Pipes 6](#_Toc74821245)

[Custom pipes 7](#_Toc74821246)

[Property binding 7](#_Toc74821247)

[Event binding 7](#_Toc74821248)

[Two-way data binding, [(NgModel)] 8](#_Toc74821249)

[Template Controll 8](#_Toc74821250)

[\*ngFor 8](#_Toc74821251)

[\*ngIf 8](#_Toc74821252)

[@Input() 8](#_Toc74821253)

[Dodawanie komponentu ręcznie 9](#_Toc74821254)

[Routing 10](#_Toc74821255)

[Routing w osobnym module 11](#_Toc74821256)

[Nawigacja poprzez komponenty 11](#_Toc74821257)

[URL Params 11](#_Toc74821258)

[HttpClient 11](#_Toc74821259)

[Testy jednostkowe 11](#_Toc74821260)

[describe 11](#_Toc74821261)

[Konfiguracja testów 11](#_Toc74821262)

[Unit test → it function 12](#_Toc74821263)

[Phase I 13](#_Toc74821264)

[Generowanie projektu 13](#_Toc74821265)

[Struktura katalogów src/app 13](#_Toc74821266)

[Lista bohaterów 14](#_Toc74821267)

[Nawigacja 14](#_Toc74821268)

[Tworzenie i Edycja bohatera 14](#_Toc74821269)

[Error Page 15](#_Toc74821270)

[PHP&Symfony 15](#_Toc74821271)

[Ubuntu 20.04: Konfiguracja środowiska 15](#_Toc74821272)

[php-cli 15](#_Toc74821273)

[Composer 15](#_Toc74821274)

[php xml extenstion 16](#_Toc74821275)

[Instalacja Symfony 16](#_Toc74821276)

[Symfony 5: Tworzenie nowego projektu 16](#_Toc74821277)

[Struktura katalogów w projekcie Symfony 5 16](#_Toc74821278)

[Symfony CLI 17](#_Toc74821279)

[Tworzenie projektu 17](#_Toc74821280)

[Uruchamianie aplikacji 17](#_Toc74821281)

[Zatrzymywanie aplikacji 17](#_Toc74821282)

[Sprawdzanie statusu serwera 17](#_Toc74821283)

[Symfony 5 17](#_Toc74821284)

[Routing 17](#_Toc74821285)

[Annotations 18](#_Toc74821286)

[Routing 18](#_Toc74821287)

[Recipes 18](#_Toc74821288)

[Twig Template 18](#_Toc74821289)

[Profiler 18](#_Toc74821290)

[Debug-pack 19](#_Toc74821291)

[API-Platform 19](#_Toc74821292)

[Maker 19](#_Toc74821293)

[Migrations 19](#_Toc74821294)

[Przygotowywanie danych testowych: doctrine-fixtures-bundle 20](#_Toc74821295)

[ManyToOne relations 20](#_Toc74821296)

[Password Encoder: wszystkiwanie zalezności 21](#_Toc74821297)

[Faker 21](#_Toc74821298)

[Disable operations from Api-platform 21](#_Toc74821299)

[Unit Test: PHPUnit 22](#_Toc74821300)

[Phase II 22](#_Toc74821301)

[Added dependencies 22](#_Toc74821302)

[Create Entity 22](#_Toc74821303)

[Phase III 22](#_Toc74821304)

[Docker 22](#_Toc74821305)

[Phase IV 26](#_Toc74821306)

[Phase V 27](#_Toc74821307)

[Azure 27](#_Toc74821308)

[Phase VI 28](#_Toc74821309)

[Phase VII 28](#_Toc74821310)

[Azure DevOps 28](#_Toc74821311)

[Phase VIII 28](#_Toc74821312)

[Napotkane problemy 28](#_Toc74821313)

[Podsumowanie 28](#_Toc74821314)

[Bibliografia 28](#_Toc74821315)

# Wstęp

Wstęp

# Narzędzia

* Windows 10 Pro Build 19041
* Linux subsystem Ubuntu 20.04 LTS WSL2
* PhpStorm 2021.1.1 Build #PS-211.7036.8, built on April 15, 2021
* Symfony CLI version v4.23.5 © 2017-2021 Symfony SAS
* Node version 10.19.0
* Angular CLI version 11.2.10 Node version 10.19.0
* Docker/Docker Desktop Version 3.3.1 (63152)
* Powershell
* GitBash
* [GitHub](https://github.com/Pyoneru/HEROES)
* Trello

# Angular

Opisany tutaj angular bazuje na starej dokumentacji <https://v2.angular.io/docs/>

## Architektura

Architekture Angulara można podzielić na następujące **główne**(nie jedyne) elementy:

* Moudles → Moduły są odpowiedzialne za konfigurację aplikacji. Moduł to klasa oznaczona dekoracją **@NgModule** i w aplikacji musi istnieć co najmniej jeden moduł – **root module**, jest to klasa **AppModule**(konwencja nazewnicza) w pliku *src/app/app.module.ts*.   
  Dekoracja **@NgModule** zawiera kilka sekcji, w **declarations** definiujemy istniejące komponenty(ang. *Components*) w aplikacji. Sekcja **providers** pozwala nam na dodanie serwisów do globalnej kolekcji serwisów, dzięki czemu mamy do nich dostęp w całej aplikacji. Sekcja **bootstrap** wyznacza główny widok aplikacji – **root component**(ta sekcja powinna zawierać tylko 1 element). Sekcja **imports** pozwala nam na dodanie komponentów lub osobnych modułów które będą używane w aplikacji, np. w tek sekcji możemy skonfigurować nasz routing.
* Components → W komponentach piszemy logikę aplikacji, są one połączone z templetami. Komponenty to klasy oznaczone dekoracją **@Component** która wymaga zdefniowania kilku sekcji. Sekcja **selector** określa nazwe znacznika używanego w templetach aby wczytać komponent w wyznaczonym miejscu. Sekcja **templateUrl** przyjmuje ściezkę do templata który ma zostać połączony z komponentem, zamiast **templateUrl** możemy użyć **template** i przekazać bezpośrednio nasz template jako kod html.
* Templates → Templaty definują widok(ang. *View*) naszej aplikacji, jest to kod html z składnią template’u Angulara. Na jego podstawie jest generowany „klasyczny” kod html który jest przesyłany do użytkwnika.
* Services → Serwisy to tak naprawdę zwykłe klasy które zawierają(powinny zawierać) szczegołówą logike aplikacji aby odciążyć/wydzielić logikę z komponentów. Komponenty powinny obsługiwać interakcję z użytkownikiem. Serwisy natomiast winny zawierać np. komunikację z back-endem, bazą danych, transwerowanie danych.

Działanie angulara obrazuje poniży obraz z dokumentacji angulara.

### Component Tree

Figure 1: Architektura Angulara; source: <https://v2.angular.io/docs/ts/latest/guide/architecture.html>

Komponenty tworzą strukturę drzewa gdzie w **AppModule** w sekcji **bootstrap** wyznaczamy korzeń tego drzewa. Następnie w templetach wykorzystujące znaczniki komonetów (**@Component → selector**) budujemy ich hierarchię i zależności między komponentami. Jest to ważna informacja ponieważ pozwala na lepsze zrozumienie działania aplikacji.

## Data binding

Angular dostarcza nam rozwiązania do komunikacji między komponentami, a ich templetami.

### Interpolation {{}}

Interpolacja pobiera wartość z zmiennej, ale nie pozwala na jej modyfikację więc jest to „one-way binding”. W kodzie html aby pobrać wartość z zmiennej, funkcji lub wykonać jakiś „kod” możemy użyć **{{CONTENT}}**. Pod **CONTENT** możemy odwołać się do zminnej z komponentu z którym template jest połączony, możemy również wywołać funkcję lub jakiś kod np. „1 + 1”.

Angular dostarcza nam również możliwość formatowania tych danych w widoku.

#### Pipes

Dzięki pipes możemy dostosować wyświetlaną zawartość bez jej modyfikacji w komponencie. Aby to zrobić w instrukcji **{{}}**  po wartości dodać znak ‘|’(pionowa kreska), a po niej jeden ze zdefiniowanych formatów. Formaty możemy modyfikować poprzez znak „:WARTOŚĆ” (dwukropek i wartość)

|  |  |  |  |
| --- | --- | --- | --- |
| **Formater** | **Opcje** | **Cel** | **Opis** |
| date | Custom format: string | Date | Wyświetla datę w podanym formacie. |
| shortDate | No known | Date | Krótki zapis daty „MM/dd/yyy” |
| fullDate | No known | Date | Długi zapisa daty „MONTH dd, yy” |
| uppercase | No knwon | String | Zamienia wszystkie znaki na duże |
| lowercase | No known | String | Zamienia wszystkie znaki na małe |

Wyrażenia **pipes** możemy łączyć, przykład:

„*{{ birthday | date | uppercase }}*”.

##### Custom pipes

Możemy również tworzyć własny **pipe**. Aby to zrobić należy stworzyć klasę która implementuje interfejs **PipeTransform** z pakietu **@angular/core**. Interfejs ten wymaga zaimplementowania jednej metody → *transform(value: any, ...args: any[]): any*, w której zapiszemy naszą logikę. Nasz **pipe** musi zostać również udekorowany poprzez **@Pipe** z sekcją **name** która odpowiada za nazwę naszego **pipe**  w templetach.

### Property binding

Dzięki **property binding** możemy osiągnąć kilka rzeczy jednak jego składania jest taka sama: *[PROPERTY]=”VALUE”*. Jest on zapisywany jako atrybut jakiegoś znacznika np. *<img [src]=”someoneUrl”>*. **Property binding** pozwala nam na powiązanie właściwości z danymi(zmienna, funkcja) z komponentu, sterowanie komponentem → wykorzystując **property [class.CLASS\_NAME]=”WARUNEK”** możemy określić warunki dla jakich dany komponent przyjmę daną klase. **Property binding** służy również do wymiany danych między komponentami.

### Event binding

Event binding odpowiada za obsługę zdarzeń których można wykonać na danym znaczniku. Składnia wygląda następująco *(EVENT)=”ACTION”.* Akcja jest najcześciej metodą z komponentu.

|  |  |
| --- | --- |
| **Event** | **Opis** |
| click | Wykonuje akcje po naciśnięciu na element. |
| keyup | Za każdym razem kiedy element wykryje naciśnięcie klawisza wykona podaną akcje. Funkcja powinna przyjmować $event aby móc rozpoznać jaką wartość ma dane pole. |
| keyup.enter | W odróżnieniu od poprzednika, tutaj dostajemy tylko wiadomość poprzez $eveny.keyCode jaki przycisk został wciśnięty. |

### Two-way data binding, [(NgModel)]

NgModel jest połączeniem **property binding** oraz **event binding**. Dzięki nim możemy połączyć pole formularzu z zmienną komponentu. Wówczas to jaką wartość przybierze i zmieni pole będzie odzwierciedlone w zmiennej.

## Template Controll

### \*ngFor

Do iterowania po liście w template mamy narzędzie **\*ngFor=”let var of list”**. Aby móc się odnosić do indexu elementu w liście możemy dodać **;let i=index**.

[Dokumentacja NgFor[1].](https://v2.angular.io/docs/ts/latest/api/common/index/NgFor-directive.html)

### \*ngIf

Aby móc sterować czy fragment html ma zostać wyrenderowany możemy użyć **\*ngIf,** przykład:

**<div \*ngIf=”selectedHero”>…</div>.**

Powyższy fragment wyrenderuje się tylko wtedy kiedy **selectedHero** posiada wartość(nie jest null/undefinded)

## @Input()

Aby komponenty mogły się ze sobą kuminikować, można w tym celu użyć **@Input()***.* Pole klasy które będzie odbierało dane z innego komponentu należy oznaczyć adnotacją **@Input()** z pakietu **@angular/core**.

Aby przekazać dane do komponentu użyjemy jego znacznika i atrybutu z nazwą zmiennej oznaczonej adnotacją **@Input()**, przykład:

**<hero-detail [hero]=”selectedHero”></hero-detail>**

W nawiasach przekazujemy wartość która ma być dostarczona do komponentu.

## Dodawanie komponentu ręcznie

Aby dodać nowy komponent(ang. *Component*) do projektu w sposób „ręczny” należy stworzyć plik z rozszreniem **.component.ts** w katalogu **src/app/**. Nazwa komponentu powinna być pisana z małych liter, a wyrazy oddzielone myslnikiem ‘-’. Nazwa klasy natomiast powinna być pisana CamelCase i kończyć się powinna wyrazem **Component***.* Klasa powinna oznaczona adnotacją **@Component**którą należy za importować z **@angular/core***.* Adnotacja musi zawierać co najmniej **selector** i **template/templateUrl**.

Komponent musi również zostać zadeklarowany w pliku **app.module.ts**, w tym celu należy do **@NgModule** w sekcji **declarations** dodać nasz komponent.

Przykład:

nazwa pliku: *hero-details.component.ts*

plik:

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

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

*@Component({*

*selector: 'hero-detail',*

*templateUrl: './hero-detail.component.html',*

*})*

*export class HeroDetailComponent{*

*@Input() hero: Hero;*

*}*

app.module.ts:

*…*

*import {HeroDetailComponent} from './hero-detail.component';*

*…*

*@NgModule({*

*declarations: [*

*AppComponent,*

*HeroDetailComponent*

*],*

*...*

## Routing

Routing to narzędzie pozwalające na sterowanie aplikacją poprzez adres URL. Aby móc korzystać z tego narzędzia należy je **zaimportować**(ang.*imports)* w **AppModule** lub w osobnym module. Aby to osiągnąć należy dodać do sekcji **imports** w **@NgModule** instrukcję **RouterModule.forRoot(ROUTES)**  z pakietu **@angular/router**, gdzie pod **ROUTES** znajduje się tablica **Routes** z naszą konfiguracją.

Podstawowa konfiguracja zawiera **path** określającą ścieżkę i **component** określający komponent który zostanie do niej przypisany. Przykład:

*const myRoutes: Routes = [{ path: ‘api/bowling’, component: BowlingListComponent}];*

Poniższa tablica zawiera kilka opcji do konfiguracji.

|  |  |  |
| --- | --- | --- |
| **Opcja** | **Wartość** | **Opis** |
| path | string | Względny adres URL po którym nastąpić ma akcja. Path może być parametryzowany poprzez „:value” , gdzie value będzie się zmieniać i zostanie on przekazany dalej. |
| component | Klasa komponentu | Powiązanie URL z komponentem. |
| redirectTo | string | Przekieruje z URL podanego w **path** do nowej ścieżki |
| path: ‘\*\*’ | Specjalny znak | Ten zapis oznacza przekierowanie akcji gdy URL nie zostanie odnaleziony. |
| data | object | Pozwala na przekazanie danych do komponentu. |

Aby móc korzystać z routing’u wewnątrz komponentów/templateów, należy skorzystać z:

* atrybutu **routerLink** → atrybut ten przy znaczniku ‘a’ zmieni naszą ścieżkę url.
* Tagu **router-outlet** → aby podmieniać zawartość komponentu w zależności od adresu URL należy użyć **<router-outlet></router-outlet>** pod który zostanie podstawiony komponent.

### Routing w osobnym module

Całą konfigurację możemy wyodrębnić do osobnego modułu postępując analogicznie jak z **AppModule** z wyjątkiem aby w wyodręnionym module wyexportować w sekcji **exports** ten moduł, jednak należy pamięć że w sekcji **imports** w **AppModule** należy go dołączyć.

### Nawigacja poprzez komponenty

Aby móc nawigować w komponencie między URL należy wstrzyknąć w ten komponent *Router* z pakietu *@angular/core*, następnie za pomocą metody *navigate* podać argument tablice, która posłuży do wygenerowania adresu URL. Metoda navigate zwraca obiekt **Promise**.

### URL Params

Aby odczytać wartość z URL w komponencie należy wstrzyknąć w niego *ActivatedRoute* z pakietu *@angular/router*. Następnie subskrybujemy właściwość *params* aby uzyskać do nich dostęp.  
! Nie rozumiem(jeszcze) dlaczego gdy wykonałem na *Obserable* toPromise to funckja nie wykonywała się wcale.

## HttpClient

Aby móc korzystać z protokołu HTTP w aplikacji angular należy go zaimportować w **AppModule**, dodając **HttpModule** **or**a**z Jso**np**Module**(w przypadku JSON)z pakietu **@angular/http.**

Od tego momentu możemy wstrzykiwać obiekt **Http** który dosatczna nam metody odpowiadające za czasowniki HTTP: GET, POST, PUT, PATCH, DELETE. Metody są pisane z małych liter i zwracją obiekt **Observable.**

Aby pobrać dane należy wywołać metodą **subscribe** która przyjmie funkcje/wyrażenie lambda do ich obsługi. W przypadku błędu możemy wykorzystać kolejne wyrażenie lambda jako drugi argument do jego obsługi.

## Testy jednostkowe

Testy w angularze tworzyłem z wykorzystaniem **Jasmine.**

Testy są tworzone z roszerzniem **.spec.ts**. Przed rozszerzeniem powinniśmy dodać nazwę komponentu który test ten będzie testować.

### describe

Testy tworzymy wewnątrz funkcji *describe.* Jako **1 argument** przyjmuje ona nazwę pod którą zostaną zebrane testy, a jako **2 argument** przyjmuje ona funkcje anonimową. Funkcję *describe* możemy zagnieżdać w sobie.

### Konfiguracja testów

Jasmine dostarcza nam kilka funkcji które przyjmują **callback**, którymi możemy skonfigurować dane do testów.

|  |  |
| --- | --- |
| **Funkcja** | **Opis** |
| beforeEach | Funkcja wykona się **przed** każdym uruchomieniem testu. |
| beforeAll | Funkcja wykona się **przed** uruchomieniem testów. |
| afterEach | Funkcja wykona się **po** każdym wykonaniu testu. |
| afterAll | Funkcja wykona się **po** wykonaniu testów. |

Do dyspzycji mamy również*TestBed*dzięki któremu możemy zasymulować działanie aplikacji. *TestBed* dostarcza nam kilka użytecznych metod.

|  |  |  |
| --- | --- | --- |
| **Metoda** | **Argumenty** | **Opis** |
| configureTestingModule | TestModuleMetadata | Pozawala na skonfigurowanie zależności, analogicznie jakbyśmy to zrobili w NgModule.  (imports, providers) |
| compileComponent | None | „Kompiluje komponenty”, należy wykonać po **configureTestingModule** |
| inject | Nazwa klasy? | Wstrzykuje(pobiera) obiekt wraz z jego zależnościami. |
| createComponent | Component Class(klasa oznaczona dekoracją @Component) | Tworzy komponent wraz z jego zaleznościami. |

### Unit test → it function

Testy jednostkowe tworzymy za pomocą funkcji *it*. Pamiętaj że funkcja *it* musi zostać wywołana wewnątrz *describe*. Jako **1 argument** przyjmuje specyficzną nazwę/opis testu, a jako **2 argument** przyjmuje *callback*w którym zapiszemy logikę naszego testu. Należy pamietać że jeżeli testujemy kod **asynchroniczny** to *callback* może dam dostarczyć funkcje **done** którą należy wykonać po zapisaniu asercji aby test mógł przejść poprawnie.

# Phase I

## Generowanie projektu

Wybrane opcje podczas generowania:

* strict : no
* routing: yes
* stylesheet: CSS

Nazwa aplikacji: *frontend-app.*

## Struktura katalogów src/app

|  |  |  |
| --- | --- | --- |
| **Katalog** | **Typ** | **Opis** |
| edit-hero | Component | Komponent odpowiadający za tworzenie nowego lub edytowanie istniejącego bohatera |
| hero-list | Component | Komponent odpowiadający wyświetlanie listy bohaterów i ich usuwanie |
| hero-routes | Module | Konfiguracja Routingu |
| model | Model | Klasa reprezentująca model bohatera |
| nav | Component | Komponent odpowiadający za nawigację po aplikacji. Zawiera przejście do listy bohaterów i dodawania nowego bohatera |
| pipe | Pipe | Własny pipe do wyświetlania ID. Jeżeli id jest **undefined, null** lub **mniejsze od 0** to wyświetl specjalny znak(‘-’) zamiast tej wartości. |
| services | Service | Katalog ten zawiera interfejst do obsługi poleceń CRUD i implementację TestService, która po wykonaniu backendu zostanie zastąpiona implementacją do jej komunikacji. |

## Lista bohaterów

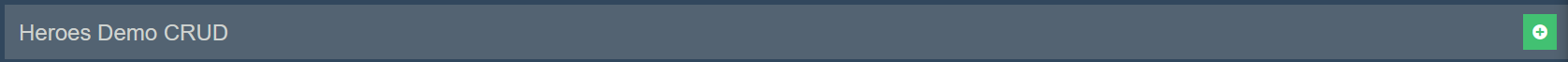
URL: /



* Komórka z imieniem bohatera → przenosi do komponentu z aktualizacją tego elementu.(URL /*hero/id*)
* Ikona kosza → usuwa element z listy.

## Nawigacja

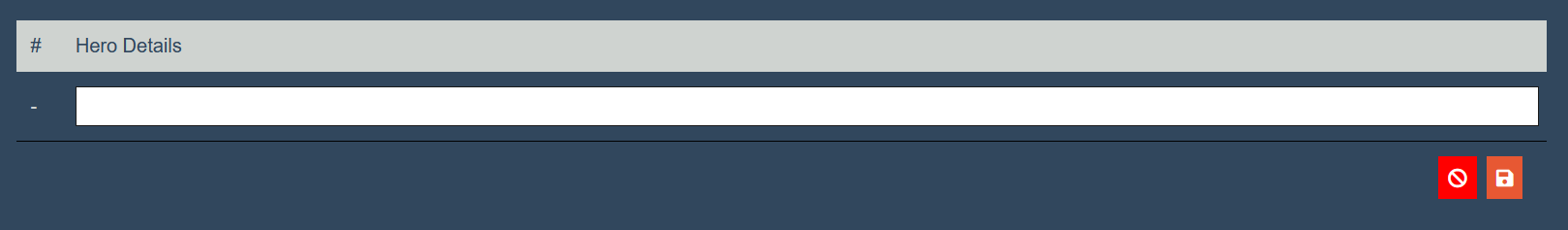
Wyświetlana zawsze.

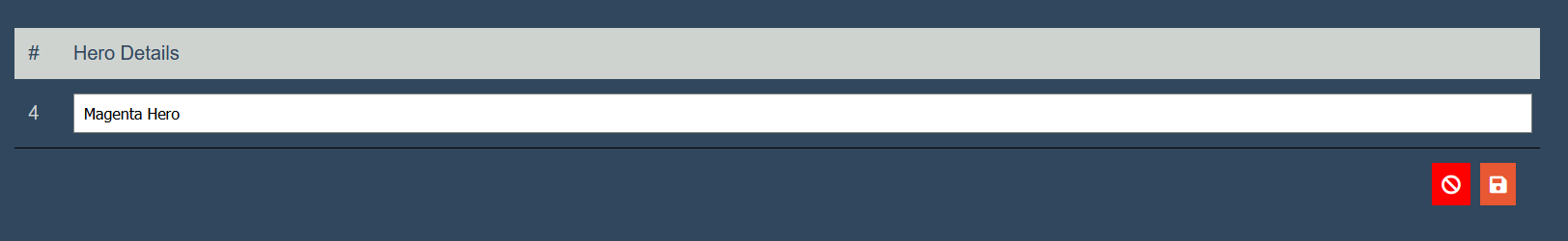
* Heroes Demo CRUD → przenosi do listy bohaterów
* Ikona plusa → przenosi do komponentu z tworzeniem nowego bohatera(URL */hero*).

## Tworzenie i Edycja bohatera

Tworzenie i Edycja to jeden komponent. W metodzie *ngOnInit* jest sprawdzany czy istnieje parametr ‘id’, jeżeli tak to próbuje go pobrać, następnie ustawić. Jeżeli pobieranie się nie powiedzie, przekieruje do listy bohaterów. Gdy parametr ‘id’ nie zostanie odnaleziony, tworzy nowy, pusty obiekt Hero.

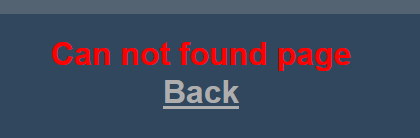
Tworzenie:

Edycja:

Podobnie do metody *ngOnInit*, metoda *saveHero* sprawdza czy obiekt *hero* posiada ID, jeżeli tak, to wykonuje aktualizację, w przeciwnym razie tworzy nowy obiekt.

## Error Page

W przypadku nieprawidłowego adresu URL zostanie wyświetlona poniższa informacja wraz z możliwością przejścia do listy bohaterów.



# PHP&Symfony

## Ubuntu 20.04: Konfiguracja środowiska

### php-cli

Pierwszą rzeczą jaką musimy zainstalować jest **php-cli** (oraz **unzip**  jeżeli nie posiadamy). Instalujemy potrzebne paczki za pomoc apt install:

*$ sudo apt install php-cli unzip*

### Composer

Następnie potrzebować będziemy Composer. Na początku upewnijmy się że jesteśmy w **home directory.**

*$ cd ~*

Pobieramy instalator:

*$ curl -sS https://getcomposer.org/installer -o composer-setup.php*

Weryfikacja pobranego instalatora:

*$ HASH=`curl -sS* [*https://composer.github.io/installer.sig*](https://composer.github.io/installer.sig)*`*

*$ echo $HASH*

Ouput powinien wyglądać podobnie do poniższego:

e0012edf3e80b6978849f5eff0d4b4e4c79ff1609dd1e613307e16318854d24ae64f26d17af3ef0bf7cfb710ca74755a

Weryfikacja czy skryp jest bezpieczny:

*$ php -r "if (hash\_file('SHA384', 'composer-setup.php') === '$HASH') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php'); } echo PHP\_EOL;"*

Output:

Installer verified

Na koniec instalacja composera globalnie:

*$ sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer*

Powyższe instrukcje pochodzącą z [poradnika[3]](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-composer-on-ubuntu-20-04) autorstwa Erika Heidi

### php xml extenstion

Do poprawnego używania symfony potrzebujemy jeszcze rozszerzenia xml do php, zainstalować je możemy następującą komendą:

*$ sudo apt-get install pgp-xml*

### Instalacja Symfony

Instalacja symfony:

*$ sudo wget https://get.symfony.com/cli/installer -O - | bash*

!Pamiętam aby skonfigurować symfony globalnie, po jego zainstalowaniu pokażą się komendy którymi można to zrobić.

## Symfony 5: Tworzenie nowego projektu

Aby stworzyć nowy projekt w Symfony 5 należy wykonać polecenie:

*$ symfony new PROJECT\_NAME*

Żeby sprawdzić czy posiadamy wszystkie zależności możemy wykonać polecenie:

*$ symfony check:req*

!Jeżeli dostaniemy błąd **no PHP binaries detected** należy wykonać polecenie:

*$ symfony local:php:refresh*

## Struktura katalogów w projekcie Symfony 5

|  |  |
| --- | --- |
| **Katalog** | **Opis** |
| config | Konfiguracja projektu |
| src | Kod źródłowy naszej aplikacji |
| vendor | Zawiera zainstalowane zalezności? |
| var | Cache? |
| bin |  |
| public | Zawiera index.php, punkt wejściowy naszej aplikacji. Wszystkie pliku znajdujące się w tym katalogu są dostępne na zewnątrz. |

## Symfony CLI

### Tworzenie projektu

*$ symfony new PROJECT\_NAME*

### Uruchamianie aplikacji

*$ symfony serve*

z flagą **-d**  możemy uruchomić aplikacje w tle(nie zatrzymuje nam konsoli)

### Zatrzymywanie aplikacji

*$ symfony server:stop*

### Sprawdzanie statusu serwera

*$ symfony server:status*

## Symfony 5

### Routing

Routing możemy skonfigurować a pliku *config/routes.yaml*. Przykłada konfiguracja zostanie wygenerowana wraz z projektem.

*index:*

*path: /*

*controller: App\Controller\DefaultController::index*

Rządanie wywołane po adresem *‘/’* zostanie przekazane do metody *index* w kontrolorze *DefaultController.*

Metoda kontrolera musi zwracać obiekt **Respone** z pakietu **Symfony\Component\HttpFoundation.**

### Annotations

Aby móc korzystać z adnotacji musimy dodać zależność:

*$ composer require annotations*

#### Routing

Po dodaniu zależności, możemy konfigurować routing poprzez adnotację nad metodami.

*/\*\**

*\* @Route(„/”)*

*\*/*

Wykorzystując **wildcard** możemy przekazać wartość poprzez URL. W adresie URL między znkami ‘{}’, podajemy nazwę którą odbierami poprzez argumenty metody(nazwa między {} i argumentem metody musi być taka sama).

*/\*\**

*\* @Route(„/hello/{name}”)*

*\*/*

*public function hello($name)…*

### Recipes

Symfony 5 posiada system ‘receptur’, które wykonują zadania przy instalowaniu nowych zależności. Recipes mogą modyfikować i dodawać nowe pliku do projektu dzięki czemu nie musimy tego robić ręczenie. Jeżeli jakaś zależność wymaga pliku konfiguracyjnego to ‘receptura’ stworzy i wypłeni go za nas początkową konfiguracją.

### Twig Template

Install:

*$ composer require twig*

Twig documentation[4]:

[Home - Twig - The flexible, fast, and secure PHP template engine (symfony.com)](https://twig.symfony.com/)

Aby korzystać z Twig template w kontrolerze należy rozszerzyć go o *AbstractController* z pakietu *Symfony/Bundle/FrameworkBundle/Controller*. *AbstractController*  dostarcza nam m.in. metodę *render* która jako **1 argument** przyjmuje względną scieżkę do pliku z template, a jako **2 argument**  przyjmuje tablice danych jaka zostanie do niej przekazana.

Folder w którym znajdują się wszystkie template’y jest skonfigurowany w *config/packages/twig.yaml.*

### Profiler

Dostarcza wiele przydatnych informacji o działaniu aplikacji w postaci ‘zakładki’ na stronie.

Install:

*$ composer require profiler –dev*

--dev → only for development

### Debug-pack

Install:

*$ composer require debug-pack*

Gdy chcemy zobaczyć jak wyglądają obiekty, możemy użyć funkcji *dump* wówczas zostaną one wyświetolne na stronie. Wraz z **Profiler** będą one w odpowiedniej zakładce.

**-pack** jest używany do określenia skrótu do pobrania kilku bibliotek.

### API-Platform

Install:

*$ composer require api*

Create entity:

*$ php bin/console make:entity*

### Maker

Narzędzie do generowania komponentów.

Install:

*$ composer require maker –dev*

Pozwala to nam od teraz generować komponenty, np.:

*$ php bin/console make:entity*.

### Migrations

Install:

*$ composer require migrations*

Mysql driver install:

*$ sudo apt install php-mysql*

Create database:

*$ php bin/console doctrine:database:create*

Create migration

*$ php bin/console make:migration*

Make migrations

*$ php bin/console doctrine:migrations:migrate*

**Restart aplikacji może pomóc rozwiązać problemy.**

### Przygotowywanie danych testowych: doctrine-fixtures-bundle

Install:

*$ composer require doctrine/doctrine-fixtures-bundle –dev*

or

*$ composer require orm-fixutres*.

Następnie w katalogu **DataFixutures** możemy stworzyć klasę rozszerzającą *Fixture* z pakietu *Doctrine/Bundle/FixutresBundle.*

W metodzie **load** definujemy operacje które zainicują naszą baze danych.

Możemy użyć metod set/get Refference aby przechować referencję do obiektu którego chcielibysmy później użyć np. jako relację.

Aby wykonać inicjacje należy wywołać polecenie:

*$ php bin/console doctrine:fixtures:load*

### ManyToOne relations

Relacje możemy zdefnionować za pomocą adnotacji *ORM/ManyToOne, ORM/OneToMany, JoinColumn.*

Właściwości adnotacji:

|  |  |  |
| --- | --- | --- |
| **Adnotacja** | **Właściwość** | **Opis** |
| ManyToOne/OneToMany | targetEntity | Encja z którą pole ma zostać połączone. |
| ManyToOne/OneToMany | mappedBy, inversedBy | „Here we see the mappedBy and inversedBy annotations for the first time. They are used to tell Doctrine which property on the other side refers to the object.„ |
| JoinColumn | nullable | Czy klucz obcy może być nullem(tzn. Czy tabela musi posiadać klucz obcy) |
| JoinColumn | name | Nazwa kolumny |

Więcej o relacjach w [dokumetnacji[5]](https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/reference/association-mapping.html)

### Password Encoder: wszystkiwanie zalezności

Jeżeli mamy dodaną zależność security to możemy wstrzyknąć zależność **password encoder’a** w pliku security.yaml dodając, np.:

*encoders:  
 App\Entity\User: bcryp*

do *security:*

### Faker

Install:

*$ composer require fzaninotto/faker –dev*

Create instance of Faker factory:

*\Faker\Factory::create();*

Generate text:

*$faker→realText(maxNbChars = 200);*

Generate DateTime:

*$faker→dateTime*

*$faker→dateTimeThisYear*

Generate slug:

*$faker→slug*

### Disable operations from Api-platform

Wyłączyć pewne operacje z CRUD możemy poprzez zdefiniowanie jakie operacje powinny być dostępne. Więc tak naprawdę definiujemy jakie operacje są dostępne, a nie jakie chcemy wyłączyć.

Aby to osiągnąć musimy zmodyfikować właściwości **itemOperations={}** i **collectionOperations={}** w adotnacji *@ApiResource* nad daną encją.

* Item → operacje na pojedynczym elemencie
* collection → operacje na kolekcji

Między {} podajamy w cudzysłowie czasowniki HTML jakie będą dostępne, przykład:

*@ApiResource(*

*itemOperations={„get”},*

*collectionOperations={„get”}*

*)*

Powyższy przykład pozwoli jedynie na pobieranie pojedynczego elementu lub całej kolekcji encji, ale nie pozwoli na dodawnie nowego, edytowanie istniejącego czy usuwanie.

### Unit Test: PHPUnit

Install

*$ composer require symfony/phpunit-bridge –dev*

Created tests should be inside **tests** folder and every test class extends by *TestCase* from the *PHPUnit\Framework*.

# Phase II

Symfony version: 5.2.7

Composer version: 2.0.13

## Added dependencies

*$ composer require api*  | API-Platform

*$ composer require maker* | Maker Bundle

## Create Entity

Created only one entity named **Hero** with **name**: string/255/no nullable

# Phase III

Phase III

# Docker

CLI

**Run new container**

if image will not be found locally then docker try find and download from dockerhub

*$ docker run IMAGE*

*Example: $ docker run ubuntu*

**Flags:**

**--interactive** -> get input from user to ran docker container

**--tty** -> allocated virtual terminal(for interactive programs)

Short for interactive and tty: **-it**

**Run new container and execute command(if possible)**

*$ docker run IMAGE COMMAND*

*Example: $ docker run ubuntu ls -l*

**List of ALL created docker containers**

*$ docker container ls -a*

*$ docker ps -a*

**List of current running containers**

*$ docker container ls*

*$ docker ps*

**Run(start) existed container**

*$ docker start CONTAINER*

**Remove existed container**

*$ docker rm CONTAINER*

**Execute command inside container(work only if container is running)**

*$ docker exec CONTAINER COMMAND*

*Example : $ docker exec 232c uname -a*

**Update container changes to new image**

*$ docker commit BASED\_IMAGE NEW\_IMAGE\_NAME*

*Example: $docker commit 232c ubuntu\_php*

**List of docker images**

*$ docker image ls*

**Display all layers of image**

*$ docker history IMAGE*

*Example: $docker history ubuntu*

**Download image withou running( from Docker Hub)**

*$ docker pull IMAGE*

**Login and Logout to Docker Hub**

*$ docker login*

*$ docker logout*

**Pusing image to docker hub.**

Only offical images can have individual name. In otherwise your image name must be compatibile with scheme. Scheme:

USERNAME/IMAGE\_NAME

Docker is individual for every account.

Pushing image to docker hub:

*$ docker push IMAGE*

**Create tag name for image(rename)**

*$ docker tag OLD\_NAME NEW\_NAME*

**Copying files between PC and containers**

To use container use:

CONTAINER\_NAME:PATH\_TO\_FILE

*$ docker cp FROM TO*

*Example: $docker cp dreamy\_hellman:data.txt data.txt*

Above commnad copy data.txt from dreamy\_hellman container to our PC(where command was invoked)

**Run docker on port**

MACHINE\_PORT:DOCKER\_PORT

Pin docker port(inside container) to machine port. Then application is available at MACHINE\_PORT on ours machine.

The same port(-p 5000) unlock access to port only inside docker, not to outside. With two ports divided by colon we connect docker port with machine port and open access to appliaction with machine port. Or somethins simillar.

*$ docker run --publish 8080:8080 IMAGE*

**Set enviroment variables**

Use flag -e KEY=VALUES. Can be used more then one time.

**Inspect**

If you need more information abount something use **inspect** command.

*$ docker inspect IMAGE (default container)*

*$ docker network inspect*

**Theory**

Image == class

container == instance of class

During using container we can use first a few symbols from container id or use container name

Images are built with image layers

Layers contains changes

Layers can be shared between docker images

**Dockerfile**

Dockerfile is a collection of commands to build new container.

Every command create new container, new layer

Dockerfile shouls starts with uppercase and without any extension, like as in title.

Docker need context(not dockerfile itself) to build and work in this context. Files outside from context can't be used.

**Keyword: FROM <IMAGE>**

Based image to create new container.

**Keyword: COPY <LOCAL\_FILES/DIRS> <DOCKER\_LOCATION>**

Copy file/dir between PC and container.

Last argument is always DOCKER\_LOCATION.

**Keyword: ADD <LOCATION(LOCAL,WEB)|ARCHIVE .tar.gz> <DOCKER\_LOCATION>**

Add keyword work similiar as copy but add can download files from internet(location) and can extract tar.gz archive inside docker. (just put path to tar.gz)

**Keyword: RUN <COMMAND>**

Execute command inside container

**Keyword: CMD <COMMAND>**

Shell form

Execute commands after container startup

*CMD python app.py*

**Keyword: CMD [<ARGS>]**

Exec form

Build commnad, first should be program to run, others values are args for this program.

Can not connect commands, we build only one command(program)

**Keyword: ENTRYPOINT**

Entrypoint work simillar as CMD but we can combine args with run args.

*ENTRYPOINT ["ls"]...*

*$ docker run entry /test*

Aboce instructions combine commnad from entrypoint(id dockerfile) with args when we start container.

CMD override commands, ENTRYPOINT combine commands.

**Keyword: WORKDIR <PATH\_TO\_DIR>**

Change context location inside docker image/container.

**Build dockerfile**

*$ docker build PATH\_TO\_DIR\_WITH\_DOCKERFILE*

We can set custom name by **--tag** flag.

*Example: $ docker build . --tag test\_app*

To select dockerfile we can use **--file** tag.

**Docker Volumes**

**Create**

*$ docker volume create NAME*

**List**

*$ docker volume ls*

**Remove**

*$ docker volume rm NAME*

**Use volume**

*--volume VOLUME\_NAME:BIND\_LOCATION*

*$ docker run --volume my-volume:/data vol\_test*

Above command will run container with pined volume do 'data' directory on container. Everything saved files in this dir will be saved on volume. Volume lives independence from containers, so we can use this volume and data on this volume in an other container.

**Anonymous volume**

To create anonymous volume just don't gives a name volume in create or pinning to container.

*$ docker volume create*

*$ docker run --volumes LOCATION IMAGE*

**Bind machine directory to docker directory**

Use ABSOLUTE path instead volume name. Path to machine directory must by absolute.

*$ docker run --volume //c/data/docker/app:/data vol\_test*

**Docker compose**

Read documentation about docker compose: <https://docs.docker.com/compose/>

**Up docker compose**

Run docker-compose.yml in current dir.

*$ docker-compose up*

**Docker network**

**List**

*$ docker network ls*

**Create**

*$ docker network create NAME*

**Remove**

*$ docker network rm NAME*

**Connect container to network**

*$ docker network connect NETWORK CONTAINER*

**Disconnect container from network**

*$ docker network disconnect NETWORK CONTAINER*

# Phase IV

FROM niemczukpiotr/ubuntu\_symfony

COPY . /usr/src/app

WORKDIR /usr/src/app

EXPOSE 8000

RUN apt-get install php-sqlite3 --yes

ENTRYPOINT ["symfony", "serve"]

# Phase V

Phase V

# Azure

* What is location and eastus in group resource ?
* --dns-name-label, what is that ?
* QUERY
* --sku, what is that ?

**CLI**

**Finding information**

*$ az find SOMETHING*

**Create resource group**

*$ az group create --name NAME --location LOCATION(eastus)*

**Create container instance**

To login you need provide data to container registry(if it's in use) you can find it on data protal in access key.

*$ az container create --resource-group GROUP\_NAME --name NAME --image DOCKER\_IMAGE --dns-name-label DNS\_NAME\_LABEL --ports PORTS*

**Status container instance**

*$ az container show --resource-group RESOURCE\_GROUP --name NAME --query QUERY --out TYPE*

--out -> type of output

**Pull logs from container(show logs)**

*$ az container logs --resource-group RESOURCE\_GROUP --name NAME*

**Attach to output streams**

*$ az container attach --resource-group RESOURCE\_GROUP --name NAME*

**Delete container**

*$ az container delete --resource-group RESOURCE\_GROUP --name NAME*

**Delete group**

*$ az container delete --name RESOURCE\_GROUP\_NAME*

**Create Azure container registry**

*$ az acr create --resource-group RESOURCE\_GROUP --name UNIQUE\_NAME --sku Basic --location LOCATION --admin-enable*

**Push image to container registry**

*$ az acr build --registry REGISTRY\_NAME --image IMAGE\_NAME .*

**Create App Service Plan**

*$ az appservice plan create --name PLAN\_NAME --resource-group RESOURCE\_GROUP --location LOCATION --sku TIER(FREE) --is-linux*

**Create App Service(Web App) with ACR**

*$ az webapp create --resource-group RESOURCE\_GROUP --plan PLAN --name APP\_NAME -i LINK\_TO\_IMAGE\_IN\_ACR*

**Log in to container registry**

*$ az acr login NAME*

# Phase VI

Phase VI

# Phase VII

Phase VII

# Azure DevOps

Azure DevOps

# Phase VIII

Phase VIII

# Napotkane problemy

Napotkane problemy

# Podsumowanie

Podusmowanie

# Bibliografia

[1] NgFor Documentation; <https://v2.angular.io/docs/ts/latest/api/common/index/NgFor-directive.html>; [24.04.2021]

[2] Jasmine Framework: <https://jasmine.github.io/>; [26.04.2021]

[3] How to install and Use Composer on Ubuntu 20.04; Erika Heidi; <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-composer-on-ubuntu-20-04> [05.05.20201]

[4] Twig Documentation; [Home - Twig - The flexible, fast, and secure PHP template engine (symfony.com)](https://twig.symfony.com/) ; [08.05.2021]

[5] Doctrine relataions; <https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/reference/association-mapping.html>; [12.05.2021]

Table of Figures

[Figure 1: Architektura Angulara; source: https://v2.angular.io/docs/ts/latest/guide/architecture.html 5](#Figure!0|sequence)