

Wprowadzenie do Symfony
Poradnik na Podstawy Aplikacji Internetowych
wykonany 2023-10-15

Maciej Kaszkowiak, 151856

Spis treści

1	Czym jest Symfony?	2
1.1	Dlaczego wybrać Symfony?	2
2	Jak stworzyć projekt zaliczeniowy w Symfony?	2
2.1	Zacznijmy nasz projekt	2
2.2	Struktura projektu	3
2.3	Twig (szablony) i recipes	4
2.4	Nasza pierwsza podstrona	5
2.5	Poznajmy bundles	6
2.6	Wykorzystajmy bundles - użytkownicy	6
2.7	Migracje - czyli stwórzmy schemat bazy danych	7
2.8	Dependency Injection, Entity Manager - co to?	9
2.9	Konfigurowanie encji i walidacja	9
2.10	Debugging w Symfony - Profiler	12
2.11	Formularze - potęga Symfony	14
3	Odnosińniki	18
4	O dokumencie	18

1 Czym jest Symfony?

Symfony to popularny open-source'owy framework PHP, który jest wykorzystywany do tworzenia aplikacji internetowych i stron internetowych. Został stworzony przez Fabien Potenciera i jego zespół jako odpowiedź na potrzeby programistów PHP, którzy chcieli szybko tworzyć skalowalne i łatwe w utrzymaniu aplikacje internetowe.

Framework ten promuje dobre praktyki programistyczne, takie jak DRY (Don't Repeat Yourself) i konwencję nazewnictwa, co ułatwia zrozumienie i utrzymanie kodu przez programistów. Symfony jest również rozwijany przez dużą społeczność programistyczną, co oznacza, że istnieje wiele dostępnych zasobów, dokumentacji i rozszerzeń, które można wykorzystać w projektach opartych na tym frameworku.

Symfony dostępne jest na stronie <https://symfony.com/>.

1.1 Dlaczego wybrać Symfony?

Dobre praktyki programistyczne: Symfony narzuca dobre praktyki programistyczne, takie jak zastosowanie wzorca MVC (Model-View-Controller) do organizacji kodu, co prowadzi do czytelnego i łatwego do zrozumienia kodu. Ponadto, framework ten promuje DRY (Don't Repeat Yourself) i konwencję nazewnictwa, co ułatwia utrzymanie i rozwijanie aplikacji w dłuższym okresie czasu.

Szybki rozwój aplikacji: Dzięki gotowym komponentom, narzędziom do automatyzacji oraz wbudowanym funkcjom Symfony pozwala na szybkie tworzenie aplikacji. Programiści mogą skoncentrować się na implementacji logiki biznesowej, ponieważ wiele rutynowych zadań jest już rozwiązanych przez framework. To przyspiesza cały proces tworzenia aplikacji.

Bogata dokumentacja: Symfony oferuje obszerną i klarowną dokumentację, która ułatwia naukę frameworka. Dla początkujących programistów oraz doświadczonych developerów jest to bezcenne źródło wiedzy, które pomaga w rozwiązywaniu problemów i efektywnym wykorzystywaniu funkcji frameworka.

Aktywna społeczność: Symfony cieszy się dużą i aktywną społecznością programistyczną. Istnieje wiele dostępnych rozszerzeń, narzędzi oraz wsparcia online, które pomagają w rozwiązywaniu problemów i udoskonalaniu umiejętności programistycznych.

Stale utrzymanie i rozwijanie: Symfony jest frameworkiem, który jest ciągle aktualizowany i rozwijany. Deweloperzy Open Source oraz społeczność aktywnie pracują nad jego ulepszaniem, dodając nowe funkcje, poprawiając wydajność oraz zabezpieczenia, co sprawia, że Symfony jest gotowe do zastosowania w najnowszych projektach.

2 Jak stworzyć projekt zaliczeniowy w Symfony?

Posłużę się wymaganiami dla projektu zaliczeniowego dla studiów niestacjonarnych 2023/2024 v1.0 z przedmiotu Podstawy Aplikacji Internetowych w celu napisania prostej aplikacji.

Skupmy się na następujących punktach:

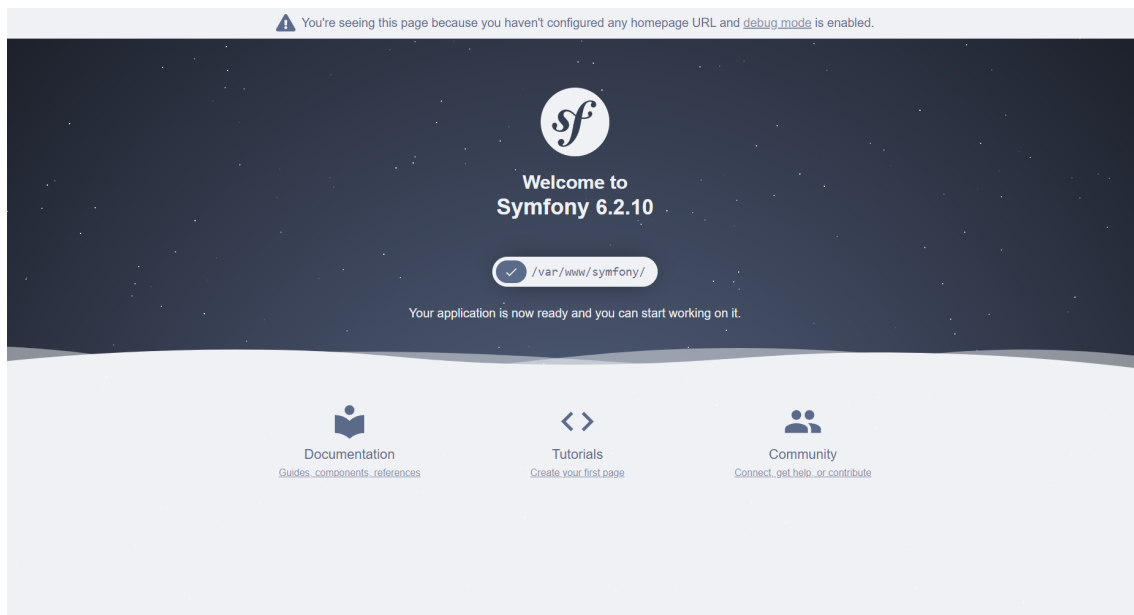
1. Zakładanie konta odbywa się przez formularz (wspólny dla trenera i sportowca – wybór roli w ramach formularza). Wymagane dane to login, hasło oraz wybór opcji: trenera lub sportowca. Zakładamy uproszczony schemat: dla sportowca zakładamy wybór pojedynczego trenera, który go prowadzi (spośród dostępnych trenerów).
2. Login może mieć tylko i wyłącznie postać poprawnego adresu mailowego (żadnych maili system jednak nie musi wysyłać).
3. Po zalogowaniu sportowiec może zapisać się do klubu (zostaje on automatycznie przypisany do trenera, którego wybrał przy zakładaniu konta).

2.1 Zacznijmy nasz projekt

Symfony do działania wymaga PHP. Sam interpreter PHP musi komunikować się ze serwerem, np. z nginx lub Apache. Dodatkowo do większości aplikacji przyda nam się relacyjna baza danych. Aby ułatwić rozpoczęcie projektu, wykorzystajmy gotowe template do Symfony: <https://github.com/ger86/symfony-docker>

- Sklonujmy repozytorium: `git clone git@github.com:ger86/symfony-docker.git`
- Wykonajmy konfigurację początkową zgodnie z README.md - ustawmy zmienne środowiskowe w odpowiednich plikach .env
- Uruchom `docker compose up -d` w katalogu .docker

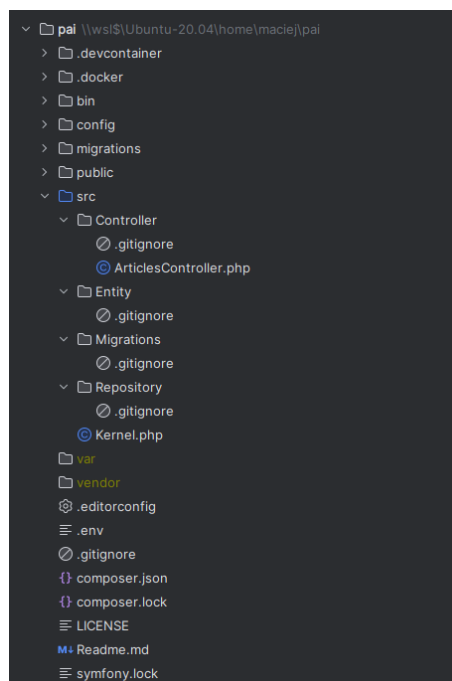
Po wykonaniu powyższych czynności, pod adresem *localhost* powinien być dostępny pusty projekt:



Rysunek 1: Pusty projekt

2.2 Struktura projektu

Wejdźmy w kod za pomocą dowolnego edytora. Osobiście będę korzystał z PHPStorm. W katalogu *src* znajduje się główna część kodu naszego projektu - obecnie jest niemalże pusta:



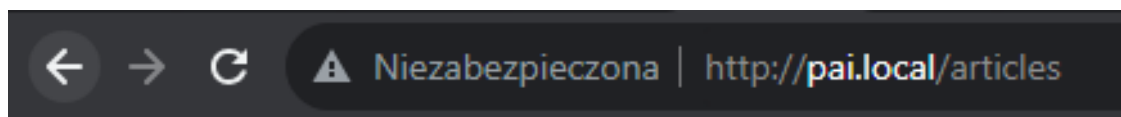
Rysunek 2: Struktura projektu

Wejdźmy w plik *Controller/ArticlesController.php* - to przykładowy kontroler dołączony w templatce:

```
1 <?php
2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8
9 class ArticlesController extends AbstractController
10 {
11     #[Route(path: '/articles', name: 'articles', methods: ['GET'])]
12     public function list(): Response
13     {
14         return new Response('Welcome to Latte and Code ');
15     }
16 }
```

Listing 1: Controller/ArticlesController.php

Możemy zauważyć, że nasz kontroler podłącza ścieżkę *articles* pod kontroler *ArticlesController->list()*, który zwraca odpowiedź tekstową. Sprawdźmy - wejdźmy w *localhost/articles*:



Rysunek 3: Działa!

Fajnie, ale pisanie całego HTMLa w kontrolerze byłoby *baaaardzo* uciążliwe. Wykorzystajmy więc Twiga!

2.3 Twig (szablony) i recipes

Twig to silnik szablonów dla języka programowania PHP. Jest to narzędzie często używane w połączeniu z frameworkiem Symfony, chociaż może być również stosowane samodzielnie. Twig zapewnia prosty i czytelny sposób definiowania szablonów, które są używane do generowania stron internetowych lub dowolnych innych treści tekstowych w PHP.

Wejdźmy w kontener PHP - wszystkie komendy związane z PHP uruchamiamy zawsze wewnątrz kontenera! - i wykonajmy polecenie *composer require template*.

Paczki zostaną automatycznie skonfigurowane:

```
Symfony operations: 2 recipes (561dabfd4bee7262baa401221aa62194)
- Configuring symfony/twig-bundle (>=5.4): From github.com/symfony/recipes:main
- Configuring twig/extra-bundle (>=v3.7.1): From auto-generated recipe
```

auto-generated recipe - co to? Już tłumaczę: Symfony Recipes to mechanizm, który automatyzuje proces konfigurowania paczek (bundles) oraz inne elementy w projekcie Symfony.

Configuring symfony/twig-bundle (>=5.4): From github.com/symfony/recipes:main:

Symfony/twig-bundle to oficjalna paczka Symfony, która zapewnia integrację z Twig, silnikiem szablonów używanym w Symfony. W tej operacji, Symfony pobiera odpowiednią konfigurację dla paczki symfony/twig-bundle z repozytorium Symfony Recipes na GitHubie (github.com/symfony/recipes:main). Jest to centralne miejsce, w którym społeczność Symfony utrzymuje gotowe przepisy konfiguracyjne (Recipes) dla różnych paczek. Symfony automatycznie konfiguruje symfony/twig-bundle na podstawie przepisu znajdującego się w repozytorium.

W skrócie mówiąc, oszczędza nam to ręcznej konfiguracji :)

Po wejściu w projekt możemy zauważyć, że pojawił się nowy katalog, a w nim jeden plik: *templates/base.html.twig*. Paczka go automatycznie tworzyła - to będzie nasz bazowy motyw wykorzystywany dla każdej szablonowanej podstrony:

```

1 <!DOCTYPE html>
2 <html>
3     <head>
4         <meta charset="UTF-8">
5         <title>{% block title %}Welcome!{% endblock %}</title>
6         <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org
7         /2000/svg%22 viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size
8         =%2296%22></text></svg>">
9         {# Run 'composer require symfony/webpack-encore-bundle' to start using
10        Symfony UX #}
11        {% block stylesheets %}
12            {{ encore_entry_link_tags('app') }}
13        {% endblock %}
14
15        {% block javascripts %}
16            {{ encore_entry_script_tags('app') }}
17        {% endblock %}
18    </head>
19    <body>
20        {% block body %}{% endblock %}
21    </body>
22</html>

```

Listing 2: templates/base.html.twig

2.4 Nasza pierwsza podstrona

Chcemy stworzyć podstronę startową - utwórzmy plik *templates/index.html.twig*:

```

1 {% extends 'base.html.twig' %}
2
3 {% block title %}Page Title{% endblock %}
4
5 {% block body %}
6     <h1>Welcome to my website!</h1>
7     <p>This is the content of the child template.</p>
8     <p>Hello {{ hello }}!</p>
9 {% endblock %}

```

Listing 3: templates/index.html.twig

I załadujmy go w kontrolerze. Utwórzmy nowy kontroler, nazwijmy go *IndexController.php*:

```

1 <?php
2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8
9 class IndexController extends AbstractController
10 {
11     #[Route(path: '/', name: 'index', methods: ['GET'])]
12     public function index(): Response
13     {
14         return $this->render('index.html.twig', [
15             'hello' => 'World!'
16         ]);
17     }
18 }

```

Listing 4: Controller/IndexController.php

Wejdźmy ponownie na *localhost*:



Rysunek 4: Działą - templatka została załadowana!

Po przeanalizowaniu kodu, możemy zauważyć że wykorzystaliśmy w Twigu bloki oraz zmienne, umożliwiając nam tworzenie rozszerzalnych templatek i przekazywanie danych z kontrolera do widoku.

2.5 Poznajmy bundles

Symfony posiada wiele paczek, które są nazywane *bundle*. Bundles to moduły, które zawierają gotowe do użycia funkcje i komponenty, które można łatwo integrować z aplikacją Symfony. Te paczki ułatwiają development, ponieważ programiści nie muszą pisać wszystkiego od zera, ale mogą korzystać z gotowych rozwiązań, oszczędzając czas i wysiłek.

Bundles mogą zawierać różnorodne elementy, takie jak kontrolery, szablony widoków, konfiguracje, style CSS, skrypty JavaScript, a także specjalne funkcje, które rozszerzają możliwości frameworka Symfony. Dzięki nim programiści mogą szybko dodać funkcje do swoich aplikacji, takie jak uwierzytelnianie użytkowników, obsługa formularzy, integracja z bazą danych, obsługa plików multimedialnych i wiele innych.

2.6 Wykorzystajmy bundles - użytkownicy

W tym miejscu odwołam się do oficjalnej dokumentacji Symfony pod tematem Security, która również w tym miejscu świetnie przedstawia jak zrobić rzecz X w Symfony. Zgodnie z zaleceniami, wykorzystajmy security bundle:

Po uruchomieniu *composer require symfony/security-bundle* możemy stworzyć użytkownika:

```
php bin/console make:user
The name of the security user class (e.g. User) [User]:
> User

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g. email, username,
> email

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be
Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

Skupmy się na tej części:

```
created: src/Entity/User.php
```

```
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

Komenda utworzyła nam encję, która zostanie powiązana z tabelą bazodanową z wykorzystaniem Doctrine, czyli biblioteki ORM wykorzystywanej przez Symfony. Doctrine pozwala na definiowanie encji, relacji między encjami oraz operacji na danych za pomocą prostych metod PHP, co ułatwia manipulację danymi w bazie danych.

Encja to klasa PHP, która reprezentuje strukturę danych w bazie danych. Każda encja jest zwykle powiązana z określoną tabelą w bazie danych. Klasa encji definiuje pola (np. właściwości użytkownika, takie jak imię, email, itp.) oraz metody, które umożliwiają dostęp i manipulację tymi danymi. Doctrine, jako ORM, używa encji do mapowania danych między bazą danych a obiektami PHP, co ułatwia pracę z danymi w aplikacji.

Repozytorium to klasa, która zawiera logikę dostępu do danych związanych z daną encją. Repozytoria są używane do wykonywania zapytań do bazy danych, takich jak pobieranie, zapisywanie, aktualizowanie i usuwanie danych. Doctrine automatycznie generuje podstawowe repozytoria dla każdej encji, ale można je także dostosować, aby obsłużyć bardziej zaawansowane operacje na danych.

W kontekście Symfony, konfiguracja odnosi się do ustawień i parametrów aplikacji, które są zdefiniowane w różnych plikach konfiguracyjnych. W przypadku Doctrine, konfiguracja obejmuje informacje dotyczące połączenia z bazą danych, takie jak adres hosta, nazwa użytkownika, hasło, nazwa bazy danych, ale także konfiguracje związane z encjami i repozytoriami.

2.7 Migracje - czyli stwórzmy schemat bazy danych

Encja została stworzona, ale nie istnieje jeszcze w bazie danych. W tym celu musimy stworzyć i uruchomić migrację.

Migracje bazodanowe to sposób zarządzania ewolucją schematu bazy danych w czasie, co oznacza, że możesz zmieniać strukturę bazy danych, dodając lub usuwając tabele, kolumny itp., w sposób kontrolowany i śledzony przez system kontroli wersji. Migracje pozwalają deweloperom na wprowadzanie zmian w schemacie bazy danych w sposób, który jest śledzony, dokumentowany i replikowalny na różnych środowiskach, takich jak lokalny komputer dewelopera, serwer testowy czy produkcji.

Wykonajmy następujące komendy:

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

```
# bin/console ma:mi
Success!
```

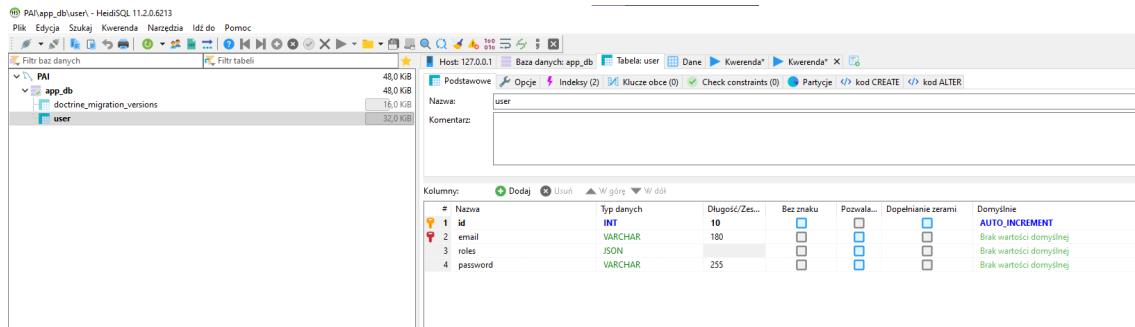
```
Next: Review the new migration "migrations/Version20231015112242.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
# bin/console do:mi:mi
```

```
WARNING! You are about to execute a migration in database "app_db" that could result in schema ch
>
```

```
[notice] Migrating up to DoctrineMigrations\Version20231015112242
[notice] finished in 136.9ms, used 12M memory, 1 migrations executed, 1 sql queries
```

```
[OK] Successfully migrated to version : DoctrineMigrations\Version20231015112242
```

Po połączeniu się z bazą danych przez zewnętrzny klient możemy zauważyć, że powstała nasza pierwsza tabela:



Rysunek 5: Działa!

Będziemy mogli zapisywać do bazy danych!

Stworzymy nowego użytkownika. Obecnie skupmy się na samej czynności zapisywania, nie implementując formularza. Sprawmy, aby po wejściu na *register* do bazy dodał się nowy użytkownik. Rozszerzymy nasz kontroler:

```

1 <?php
2
3 namespace App\Controller;
4
5 use App\Entity\User;
6 use Doctrine\ORM\EntityManagerInterface;
7 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
8 use Symfony\Component\HttpFoundation\Response;
9 use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
10 use Symfony\Component\Routing\Annotation\Route;
11
12 class IndexController extends AbstractController
13 {
14     #[Route(path: '/', name: 'index', methods: ['GET'])]
15     public function index(): Response
16     {
17         return $this->render('index.html.twig', [
18             'hello' => 'World!'
19         ]);
20     }
21
22     #[Route(path: '/register', name: 'register', methods: ['GET'])]
23     public function register(
24         UserPasswordHasherInterface $passwordHasher,
25         EntityManagerInterface $entityManager,
26     ): Response
27     {
28         $user = new User();
29         $plaintextPassword = 'haslo';
30
31         $user
32             ->setEmail('test@example.com');
33
34         $hashedPassword = $passwordHasher->hashPassword(
35             $user,
36             $plaintextPassword
37         );
38         $user->setPassword($hashedPassword);
39
40         $entityManager->persist($user);
41         $entityManager->flush();
42
43         return new Response("Registered!");
44     }
45 }
46 }

```

Listing 5: Controller/IndexController.php

2.8 Dependency Injection, Entity Manager - co to?

Kluczowym punktem jest sposób, w jaki przekazywane są argumenty do metody `register`. Wielokrotne przekazywanie obiektów jako argumentów metod kontrolera jest przykładem techniki znaną jako Dependency Injection (DI).

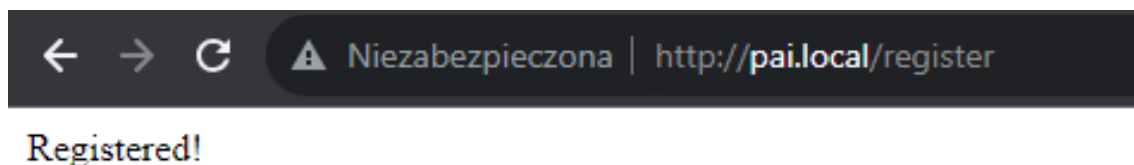
Dependency Injection to wzorec projektowy, który polega na przekazywaniu obiektów (zależności) do innych obiektów, zamiast pozwalania obiektom na samodzielne tworzenie ich zależności. W tym przypadku, metoda `register` przyjmuje dwie zależności: `UserPasswordHasherInterface` i `EntityManagerInterface`.

Dzięki temu podejściu, nie musisz ręcznie tworzyć obiektów tych zależności wewnątrz metody. Symfony automatycznie rozpoznaje typy zależności przekazane jako argumenty i dostarcza odpowiednie instancje tych typów podczas wywoływania metody.

Jak możemy zauważyć, w kodzie odwołujemy się również do `EntityManager` w celu zapisania encji `User`. `EntityManager` to jedna z kluczowych części Doctrine, które jest narzędziem do mapowania obiektowo-relacyjnego (ORM), co oznacza, że pozwala programistom pracować z danymi w bazie danych, używając obiektów PHP, zamiast klasycznych zapytań SQL.

Doctrine wykorzystuje model Data Mapper zamiast Active Record, przez co zmiany nie niosą się automatycznie, jak w Laravelu, tylko musimy wywołać `persist()` w celu oznaczenia obiektu jako nowego do zapisania, a następnie `flush()`, aby skomunikować się z bazą danych.

Wejdźmy na `/register`:



Rysunek 6: Zarejestrowano - czy na pewno?

Komunikat pokazał, że zarejestrowano użytkownika. Czy na pewno? Podejrzymy w bazie danych:

A screenshot of a database query result. The title bar says "app_db.user: 1 razem wierszy (około)". The table has four columns: "id", "email", "roles", and "password". There is one row of data with the following values: "1", "test@example.com", "[]", and "\$2y\$13\$M3PGunG.NNKwE86a2NUjD.t2oyyc/LPZTG6...".

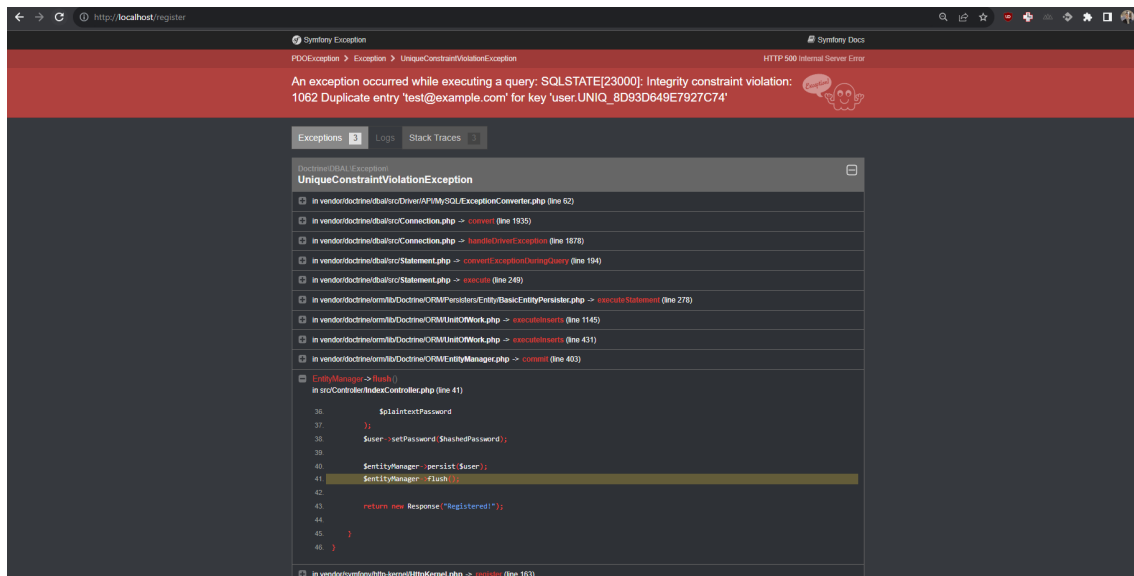
id	email	roles	password
1	test@example.com	[]	\$2y\$13\$M3PGunG.NNKwE86a2NUjD.t2oyyc/LPZTG6...

Rysunek 7: Zarejestrowano - na pewno :)

Udało się, zapisaliśmy pierwszego użytkownika w bazie danych.

2.9 Konfigurowanie encji i walidacja

Spróbujmy wejść na podstronę ponownie:



Rysunek 8: Błąd?

Pojawił się błąd:

An exception occurred while executing a query: SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry 'test@example.com' for key 'user.UNIQ_8D93D649E7927C74'

Ten błąd oznacza, że próbujesz wstawić nowy rekord do tabeli w bazie danych, który narusza ograniczenie unikalności (unique constraint) na kolumnie o nazwie 'user.UNIQ_8D93D649E7927C74'. W konkretnym przypadku, unikalność jest naruszona dla wartości 'test@example.com' w tej kolumnie. To oznacza, że w danej tabeli już istnieje rekord z wartością 'test@example.com', a próbujesz dodać kolejny rekord z tą samą wartością, co jest niedozwolone ze względu na ograniczenie unikalności.

Jak Symfony to wykryło? Wejźmy do wcześniej utworzonej klasy `src/Entity/User.php`:

```
1 class User implements UserInterface, PasswordAuthenticatedUserInterface
2 {
3     /* ... */
4
5     #[ORM\Column(length: 180, unique: true)]
6     private ?string $email = null;
7
8     /* ... */
```

Listing 6: src/Entity/User.php

Atrybut *ORM*

Column określa nam, że kolumna powinna mieć maksymalnie 180 znaków, powinna być unikalna, zaś typ jest odczytywany z typu zmiennej. Na tej podstawie generowany jest kod SQL, który tworzy nam schema w bazie danych!

Założenie jest poprawne. Zmodyfikujmy więc nasz kontroler, aby dodać drugiego użytkownika:

```
1 $user
2     ->setEmail('to nie jest email!!!!11');
```

Listing 7: Controller/IndexController.php

I wejźmy na `/register` ponownie:

id	email	roles	password
1	test@example.com	[]	\$2y\$13\$M3PGunG.NNKwE86a2NUjD.t2oyyc/LPZTG6...
3	to nie jest email!!!11	[]	\$2y\$13\$JEF.Wol7kExA/TOV0QdmmuYObFObw33ZA...

Rysunek 9: Halo, to nielegalne!

Chwila - przecież powinniśmy móc dodawać tylko email! Dodajmy walidację. Zainstalujmy komponent odpowiadający za walidację: *composer require symfony/validator*

Po instalacji możemy dodać nasz pierwszy constraint:

```

1 use Symfony\Component\Validator\Constraints as Assert;
2
3 class User implements UserInterface, PasswordAuthenticatedUserInterface
4 {
5     #[ORM\Column(length: 180, unique: true)]
6     #[Assert\Email(
7         message: 'Dobrodzieju, podana przez Wacpana wartosc {{ value }} nie jest
8         poprawnym adresem poczty elektronicznej!',
9     )]
10    private ?string $email = null;

```

Listing 8: Entity/User.php

Spróbujmy ponownie dodać użytkownika z niepoprawnym adresem email. Po uruchomieniu /register... nic się nie stanie! Dlaczego? Ponieważ musimy jeszcze wywołać walidację. Przy formularzach walidacja uruchamiana jest automatycznie, co oszczędza nam roboty, ale tutaj pokazujemy mechanizm od podstaw :)

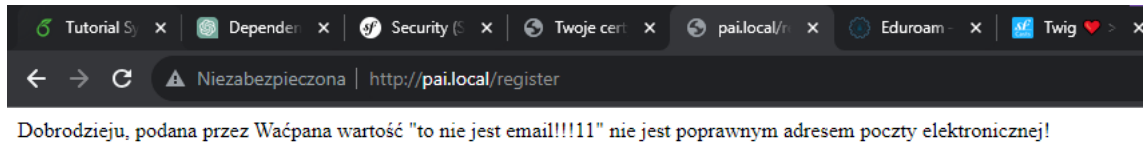
```

1     #[Route(path: '/register', name: 'register', methods: ['GET'])]
2     public function register(
3         UserPasswordHasherInterface $passwordHasher,
4         EntityManagerInterface $entityManager,
5         ValidatorInterface $validator,
6     ): Response
7     {
8         $user = new User();
9         $plaintextPassword = 'haslo';
10
11         $user
12             ->setEmail('to nie jest email!!!11');
13
14         $hashedPassword = $passwordHasher->hashPassword(
15             $user,
16             $plaintextPassword
17         );
18         $user->setPassword($hashedPassword);
19
20         $errors = $validator->validate($user);
21
22         if (count($errors) > 0) {
23             $errorMessages = [];
24             foreach ($errors as $error) {
25                 $errorMessages[$error->getPropertyPath()] = $error->getMessage();
26             }
27
28             return new Response(implode("<br>", $errorMessages), Response::
29             HTTP_BAD_REQUEST);
30         }
31
32         $entityManager->persist($user);
33         $entityManager->flush();
34
35         return new Response("Registered!");
36     }

```

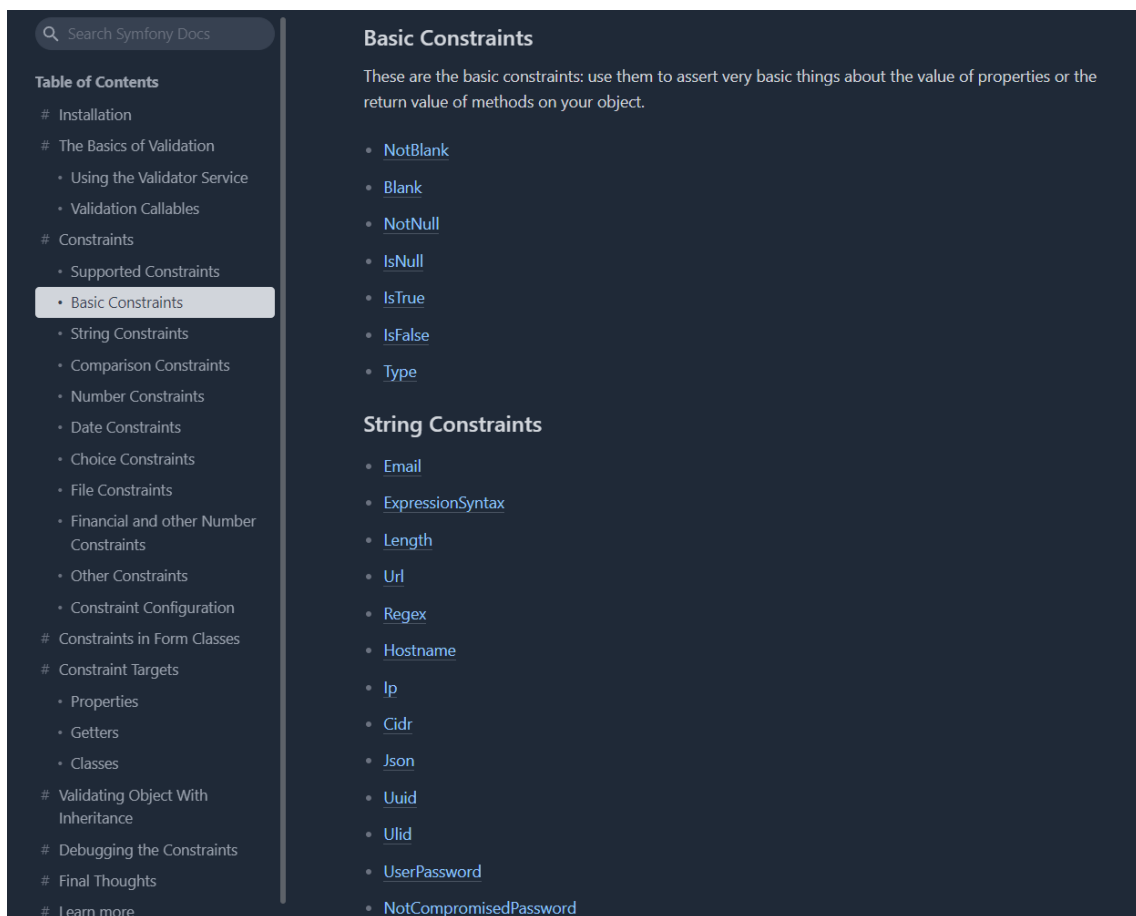
Listing 9: Controller/IndexController.php

W przypadku wystąpienia błędów z walidatora, pojawi się komunikat na ekranie, a użytkownik nie zostanie zapisany:



Rysunek 10: Waćpanie, tak nie można!

Działa! Symfony posiada wiele wbudowanych walidatorów:

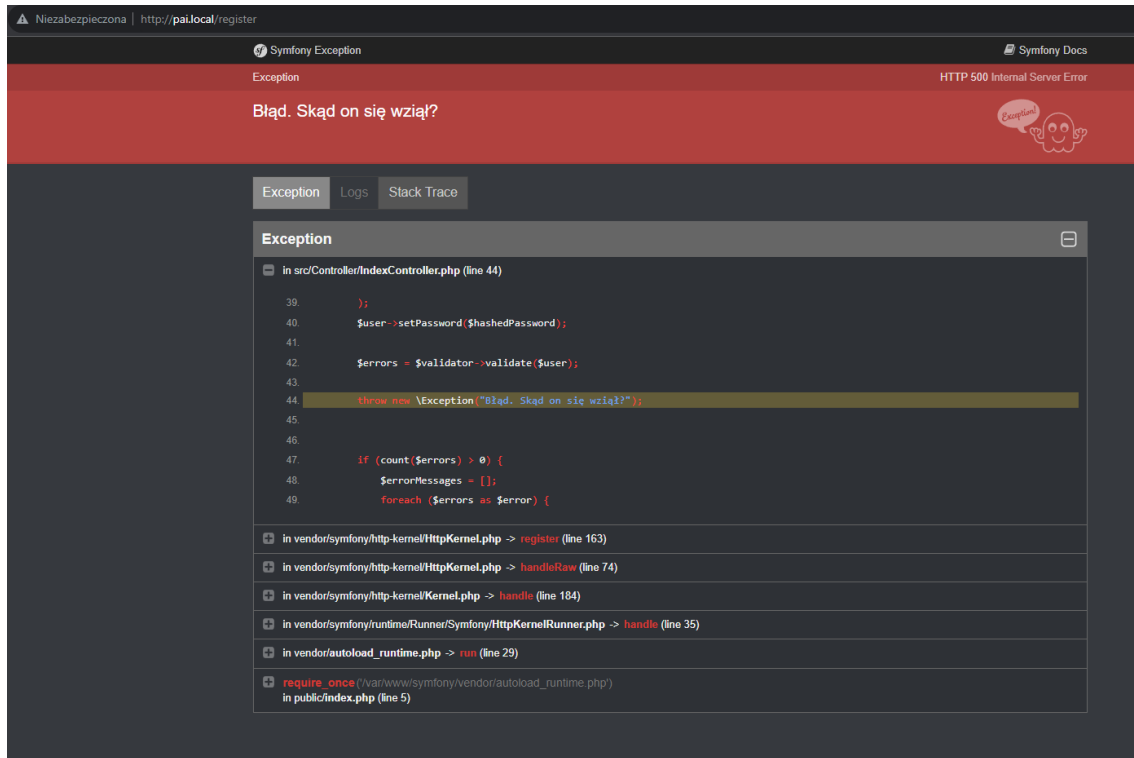


Rysunek 11: Walidacja - zrzut ekranu oficjalnej dokumentacji

Jak widać, jest ich naprawdę multum.

2.10 Debugging w Symfony - Profiler

Symfony jest świetne pod kątem debugingu. Jak widać na poprzednim zrzucie ekranu, przy błędzie pojawia się duży, czerwony stacktrace. Dodajmy gdzieś exception: `throw new Exception("Bład. Skąd on się wziął?");`



Rysunek 12: Błąd

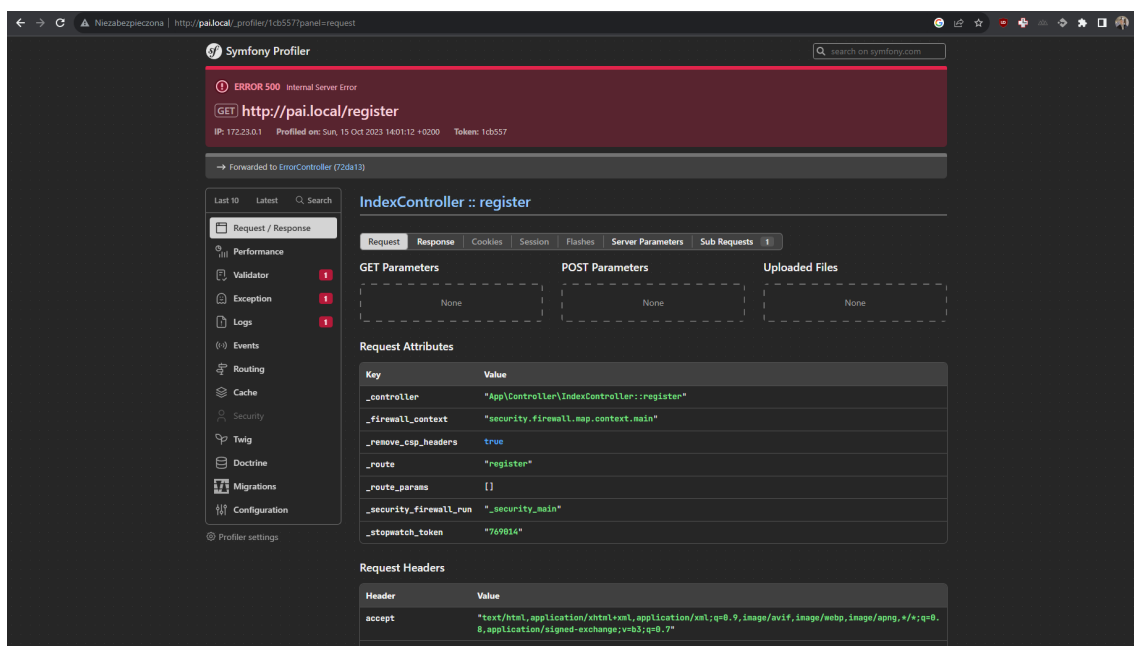
Dlaczego Symfony jest świetne? Zainstalujmy paczkę aby się dowiedzieć: `composer require --dev symfony/profiler-pack`.

Po zainstalowaniu możemy zauważyć, że pojawił się na dole czarny pasek:



Rysunek 13: Czarny pasek na dole ekranu - Profiler

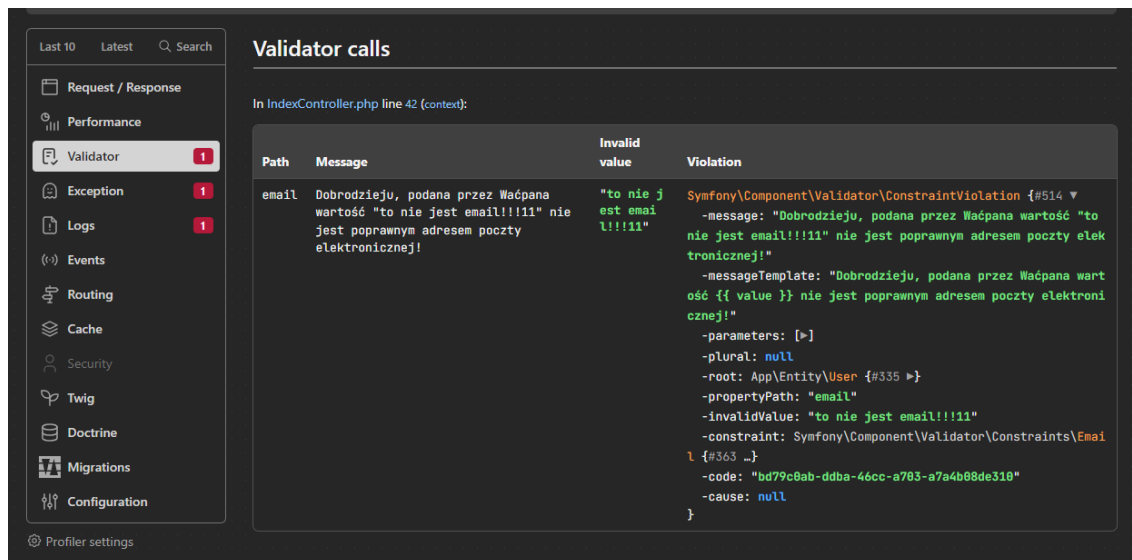
Kliknijmy w lewy dolny róg, a pojawi się pełen ekran Profilera:



Rysunek 14: Profiler

Profiler pozwala na śledzenie żądań HTTP, analizę wydajności, monitorowanie zapytań SQL, śledzenie błędów i wyjątków, przeglądanie logów aplikacji oraz sprawdzanie, które fragmenty kodu zajmują najwięcej czasu wykonania. Dzięki Profilerowi deweloperzy mogą precyzyjnie zlokalizować i zrozumieć problemy w aplikacji, ułatwiając szybkie debugowanie, optymalizację kodu oraz eliminowanie błędów.

Wejźmy przykładowo w zakładkę *Validator*:



Rysunek 15: Sekcja Validator w Profilerze

Informacje w Profilerze pojawiają się zawsze. Sam Profiler można wyłączyć na środowisku uat/prod za pomocą zmiennych środowiskowych.

2.11 Formularze - potęga Symfony

Zaletą silnego systemu formularzy w Symfony jest jego intuicyjny i rozbudowany mechanizm, który pozwala programistom łatwo tworzyć, walidować i przetwarzać formularze. Framework automatyzuje wiele kroków, takich jak generowanie kodu HTML, obsługa walidacji danych, zarządzanie błędami i przetwarzanie danych wejściowych od użytkownika. Dzięki temu, programiści mogą efektywnie budować interaktywne interfejsy użytkownika, skupiając się na logice aplikacji, a nie na szczegółach obsługi formularzy, co przyspiesza rozwój aplikacji i poprawia jakość kodu.

Stwórzmy nasz pierwszy formularz. Zainstalujmy paczkę: *composer require form*

W tym miejscu stwórzmy dodatkową encję - nazwijmy ją Zawody (SportsEvent). Nie będziemy w tutorialu przedstawiali operowania na encji User, ponieważ nie pokaże nam to zalet operowania na Symfony.

Do tworzenia encji wykorzystajmy Maker Bundle. Maker Bundle to już zainstalowana paczka, która ułatwia pisanie kodu:

make

make:auth

make:command

make:controller

make:crud

make:docker:database

make:entity

make:fixtures

make:form

make:message

make:messenger-middleware

make:migration

make:registration-form

make:reset-password

Creates a Guard authenticator of different flavors

Creates a new console command class

Creates a new controller class

Creates CRUD for Doctrine entity class

Adds a database container to your docker-compose.yaml

Creates or updates a Doctrine entity class, and optional

Creates a new class to load Doctrine fixtures

Creates a new form class

Creates a new message and handler

Creates a new messenger middleware

Creates a new migration based on database changes

Creates a new registration form system

Create controller, entity, and repositories for use w

<code>make:serializer:encoder</code>	Creates a new serializer encoder class
<code>make:serializer:normalizer</code>	Creates a new serializer normalizer class
<code>make:stimulus-controller</code>	Creates a new Stimulus controller
<code>make:subscriber</code>	Creates a new event subscriber class
<code>make:test</code>	<code>[make:unit-test make:functional-test]</code> Creates a new t
<code>make:twig-component</code>	Creates a twig (or live) component
<code>make:twig-extension</code>	Creates a new Twig extension with its runtime class
<code>make:user</code>	Creates a new security user class
<code>make:validator</code>	Creates a new validator and constraint class
<code>make:voter</code>	Creates a new security voter class

My wykorzystamy komendę `bin/console make:entity`:

```
# bin/console make:entity
```

Class name of the entity to create or update (e.g. AgreeablePuppy):

```
> SportsEvent
```

```
created: src/Entity/SportsEvent.php
```

```
created: src/Repository/SportsEventRepository.php
```

Entity generated! Now let's add some fields!

You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):

```
> name
```

Field type (enter ? to see all types) [string]:

```
>
```

Field length [255]:

```
>
```

Can this field be null in the database (nullable) (yes/no) [no]:

```
>
```

```
updated: src/Entity/SportsEvent.php
```

Add another property? Enter the property name (or press <return> to stop adding fields):

```
> date
```

Field type (enter ? to see all types) [string]:

```
> datetime
```

Can this field be null in the database (nullable) (yes/no) [no]:

```
>
```

```
updated: src/Entity/SportsEvent.php
```

Add another property? Enter the property name (or press <return> to stop adding fields):

```
>
```

Success!

Next: When you're ready, create a migration with `php bin/console make:migration`

```
#
```

Powinna utworzyć nam się encja *SportsEvent*. Stwórzmy i wykonajmy migrację:

```
# bin/console ma:mi
```

Success!

Next: Review the new migration "migrations/Version20231015121816.php"

Then: Run the migration with `php bin/console doctrine:migrations:migrate`

See <https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html>

```
# bin/console do:mi:mi
```

WARNING! You are about to execute a migration in database "app_db" that could result in schema ch

>

```
[notice] Migrating up to DoctrineMigrations\Version20231015121816
```

```
[notice] finished in 143.5ms, used 16M memory, 1 migrations executed, 1 sql queries
```

```
[OK] Successfully migrated to version : DoctrineMigrations\Version20231015121816
```

#

Baza została zaktualizowana, utworzyliśmy encję SportsEvent z nazwą i datą. Utwórzmy dla niej formularz:

bin/console make:form

The name of the form class (e.g. GentlePizzaType):

> SportsEventType

The name of Entity or fully qualified model class name that the new form will be bound to (empty)

> SportsEvent

created: src/Form/SportsEventType.php

Success!

Next: Add fields to your form and start using it.

Find the documentation at <https://symfony.com/doc/current/forms.html>

#

Pojawił się nowy plik - *Form/SportsEventType.php*:

```
1 <?php
2
3 namespace App\Form;
4
5 use App\Entity\SportsEvent;
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9
10 class SportsEventType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options): void
13     {
14         $builder
15             ->add('name')
16             ->add('date')
17         ;
18     }
19     public function configureOptions(OptionsResolver $resolver): void
20     {
21         $resolver->setDefaults([
22             'data_class' => SportsEvent::class,
23         ]);
24     }
25 }
```

Listing 10: Form/SportsEventType.php

Wyświetlmy nasz formularz, tworząc nowy kontroler:

```
1 #[Route(path: '/form', name: 'form', methods: ['GET', 'POST'])]
2 public function form(Request $request, EntityManagerInterface $entityManager):
3     Response
4     {
5         $sportsEvent = new SportsEvent();
6
7         $form = $this->createForm(SportsEventType::class, $sportsEvent);
8
9         $form->handleRequest($request);
```

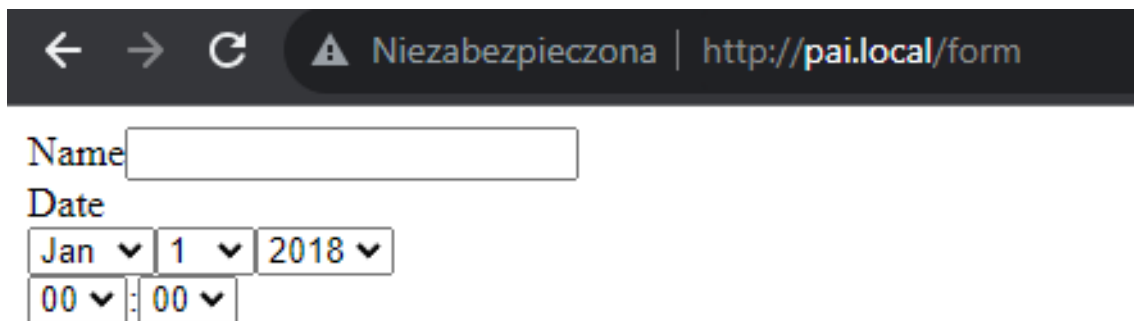


```
10     if ($form->isSubmitted() && $form->isValid()) {
11         $entityManager->persist($sportsEvent);
12         $entityManager->flush();
13         return $this->redirectToRoute('index');
14     }
15
16     return $this->render('form.html.twig', [
17         'form' => $form->createView(),
18     ]);
19 }
```

Utwórzmy też plik *form.html.twig*:

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Formularz{% endblock %}
4
5 {% block body %}
6     {{ form(form) }}
7 {% endblock %}
```

I wejdźmy na */form*:

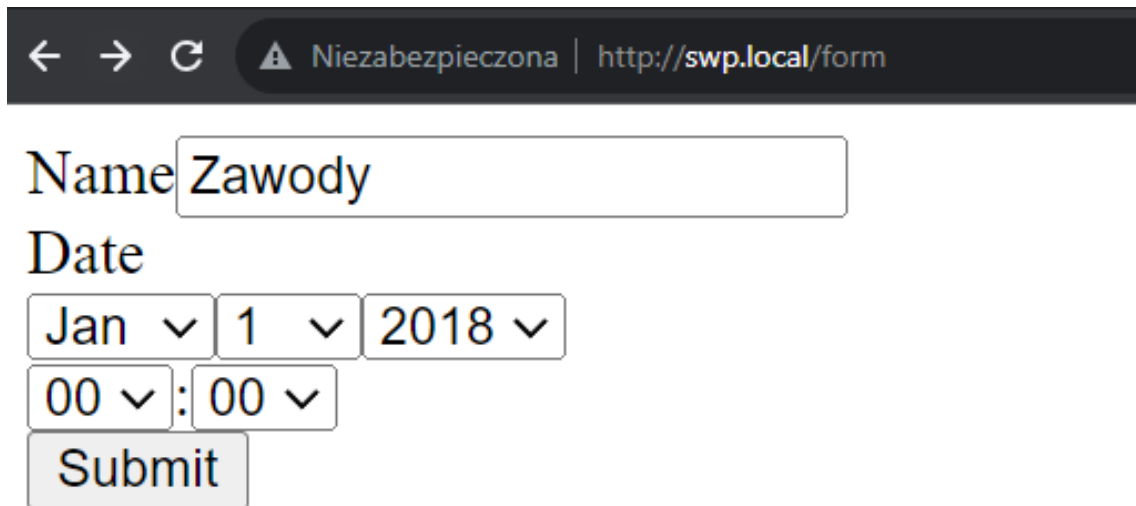


Rysunek 16: Formularz jest widoczny!

Pojawił się formularz! Kod w kontrolerze wspiera wysyłkę, jednak nie możemy go wysłać po stronie przeglądarki. Dodajmy do formularza jeszcze pole typu *SubmitType*, które umożliwi wysyłkę.

```
1 use Symfony\Component\Form\Extension\Core\Type\SubmitType;
2
3 class SportsEventType extends AbstractType
4 {
5     public function buildForm(FormBuilderInterface $builder, array $options): void
6     {
7         $builder
8             ->add('name')
9             ->add('date')
10            ->add('submit', SubmitType::class, [
11                'label' => 'Submit',
12            ])
13        ;
14    }
```

Odświeżmy stronę:



← → ↻ ⚠ Niezabezpieczona | http://swp.local/form

Name

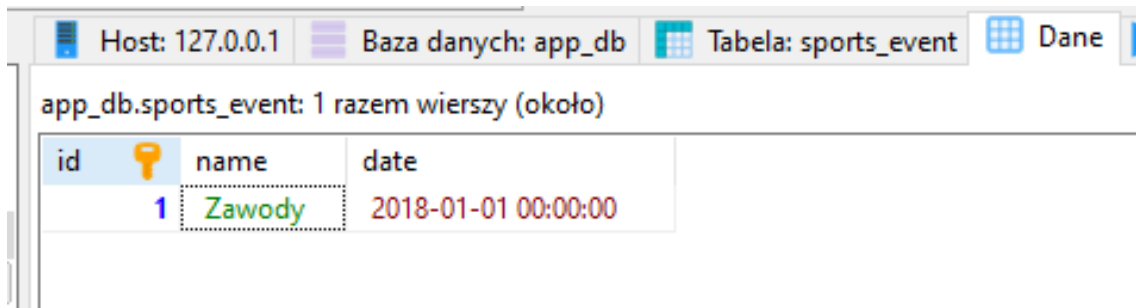
Date

Jan ▾ 1 ▾ 2018 ▾

00 ▾ : 00 ▾

Rysunek 17: Pojawił się przycisk Submit

Super - spróbujmy wypełnić formularz. Po naciśnięciu Submit zaobserwujemy przekierowanie na stronę główną, a po wejściu do bazy danych:



Host: 127.0.0.1 Baza danych: app_db Tabela: sports_event Dane

app_db.sports_event: 1 razem wierszy (około)

id	name	date
1	Zawody	2018-01-01 00:00:00

Rysunek 18: Działą - zawody zostały zapisane w bazie!

Wszystko działa - za pomocą niewielkiej ilości kodu udało nam się stworzyć formularz zapisujący do bazy!

3 Odnosińniki

Kod dla powyższego poradnika razem z chronologiczną historią commitów <https://github.com/mkaszkowiak/cs-put/podstawy-aplikacji-internetowych>

Oficjalna dokumentacja Symfony, obszerna, zrozumiała oraz aktualna <https://symfony.com/>

SymfonyCasts, nieoficjalna dokumentacja, świetnie pokazuje przykłady <https://symfonycasts.com/screencasts/recipe>

4 O dokumencie

Niniejszy dokument może być redystrybuowany i modyfikowany bez zgody autora.
Drobny disclaimer - autor pisał na szybko i dokument może zawierać błędy stylistyczne ;)