SPIS TREŚCI

2. ZROZUMIENIE SYMFONY

Zanim zaczniemy pracować z Symfony, musisz przede wszystkim zrozumieć, czym framework jest i dlaczego może stać się Twoim najlepszym narzędziem do tworzenia aplikacji. Symfony to nie tylko framework – to cała platforma, która dostarcza narzędzi, komponentów i rozwiązań, które ułatwią Ci życie jako programiście.

Symfony to framework PHP typu full-stack, co oznacza, że dostarcza kompletny zestaw narzędzi do budowy nowoczesnych aplikacji webowych. Z jego pomocą możesz tworzyć zarówno proste strony internetowe, jak i rozbudowane aplikacje, które obsługują miliony użytkowników. Co więcej, Symfony jest otwartoźródłowe, czyli całkowicie darmowe i rozwijane przez ogromną społeczność programistów z całego świata.

Jednak Symfony to coś więcej niż tylko „zestaw narzędzi". To framework oparty na zasadach najlepszych praktyk programistycznych, co oznacza, że kiedy z nim pracujesz, automatycznie uczysz się pisać lepszy kod. Dzięki niemu aplikacje są bardziej modularne, łatwe do rozbudowy i utrzymania. Wszystko to sprawia, że Symfony jest doskonałym wyborem, zarówno dla początkujących, jak i zaawansowanych programistów.

Zarówno do Symfony, jak i do Laravela, istnieje szereg paczek, które pomagają nam pisać coraz lepszy kod. Sam, odkąd zacząłem używać np. CS-Fixer czy Rector zauważyłem, że po jakimś czasie dokładnie zapamiętuję, co analizatory poprawiają i zaczynam sam tak pisać kod. Są to świetne narzędzia do utrzymywania kodu na najwyższym poziomie, jak i do nauki, aby taki kod pisać automatycznie.

Nie martw się, jeśli na początku coś będzie wydawało Ci się skomplikowane – wszystko wyjaśnię w sposób przystępny, a dzięki praktycznym przykładom szybko nabierzesz pewności w pracy z tym frameworkiem.

[...]

3. ŚRODOWISKO PRACY

W ebooku przedstawię Ci, jak pracuję z Symfony na środowisku Windows z użyciem XAMPP jako naszego serwera. XAMPP jest to program serwerowy, tworzy on serwer np. w systemie Windows, w którym mamy Apache, PHP oraz MySQL. Jest to taki serwer na naszym komputerze.

XAMPP jest o tyle prostym narzędziem, że jego instalacja odbywa się poprzez zwykły instalator. Pamiętać musimy, aby dodać odpowiednie wpisy w pliku hosts w Windows, oraz w plikach vHost dla Apache, żeby móc poruszać się lokalnie po różnych projektach, zmieniając tylko domenę w adresie.

Na serwerach testowych oraz produkcyjnych korzystam z Linuxa, a dokładnie z Ubuntu oraz Nginx. Nginx wydaje mi się być bardziej skalowalny i szybszy niż Apache, ale to może być tylko moja subiektywna opinia 😊

Nie będę tutaj przedstawiał konfiguracji Dockera. Jest to świetne rozwiązanie, aby wielu programistów pracujących przy projekcie miało dokładnie taki sam serwer lokalny, gdyż Docker jest narzędziem, które tworzy Maszyny Wirtualne wraz z ich konfiguracją i zainstalowanymi paczkami. Dockera można również używać na serwerach testowych oraz produkcyjnych – wtedy też wiemy, że, jeśli aplikacja ma problem lokalnie z serwerem, to będzie go miała na produkcji lub, jak potrzebujemy dodać nowy dodatek do PHP dla jakiegoś rozwiązania, wtedy znajdzie się on również na serwerach produkcyjnych. Natomiast w tym ebooku skupiam się na samym Symfony, dlatego też cały opis i konfiguracja będzie w „tym najprostszym" rozwiązaniu 😉

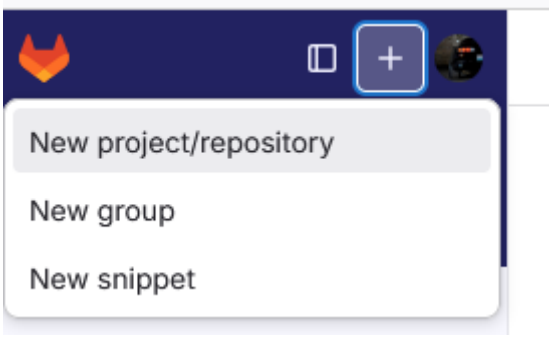
3.1. INSTALACJA I KONFIGURACJA SYMFONY

Instalacja i konfiguracja Symfony to kluczowe kroki w procesie tworzenia aplikacji opartych na tym frameworku. Dobrze skonfigurowane środowisko robocze zapewnia płynne doświadczenie podczas rozwijania projektu. Poniżej znajdziesz przewodnik dotyczący wymagań systemowych, konfiguracji lokalnego serwera, instalacji Composer oraz Symfony CLI.

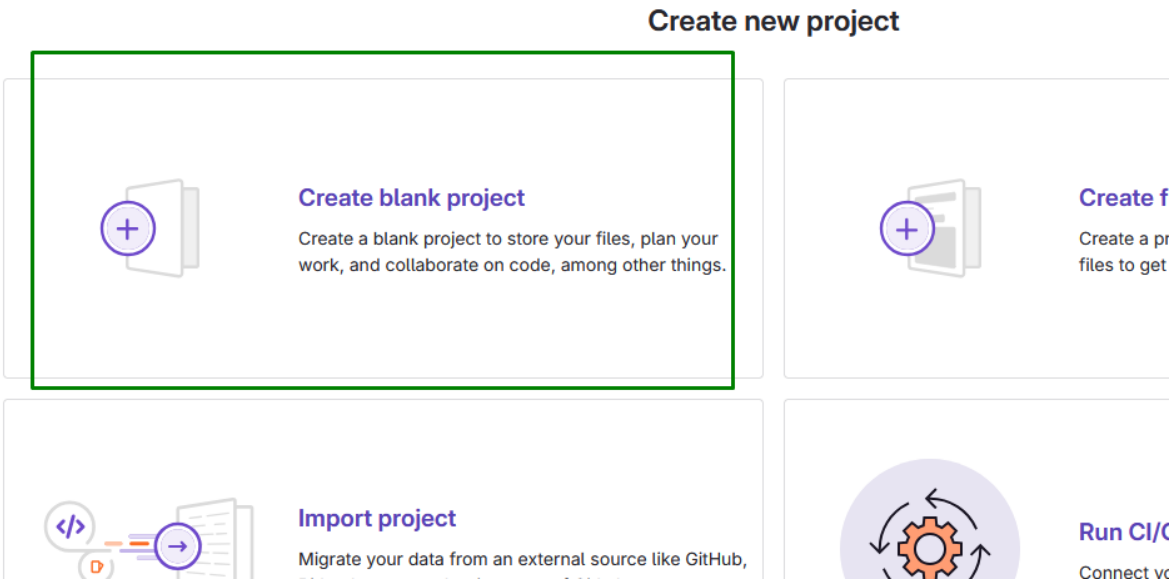
3.1.1.TWORZENIE REPOZYTORIUM GITLAB

Zacznijmy od tego, że musisz mieć założone konto na GitLab. Jest to darmowa platforma, więc nie ma się czego obawiać. Możesz na niej zakładać repozytoria prywatne – które widzisz tylko Ty lub osoba, której je udostępnisz lub publiczne, do których każdy będzie mógł zajrzeć. Stwórzmy sobie zatem nasz repozytorium.

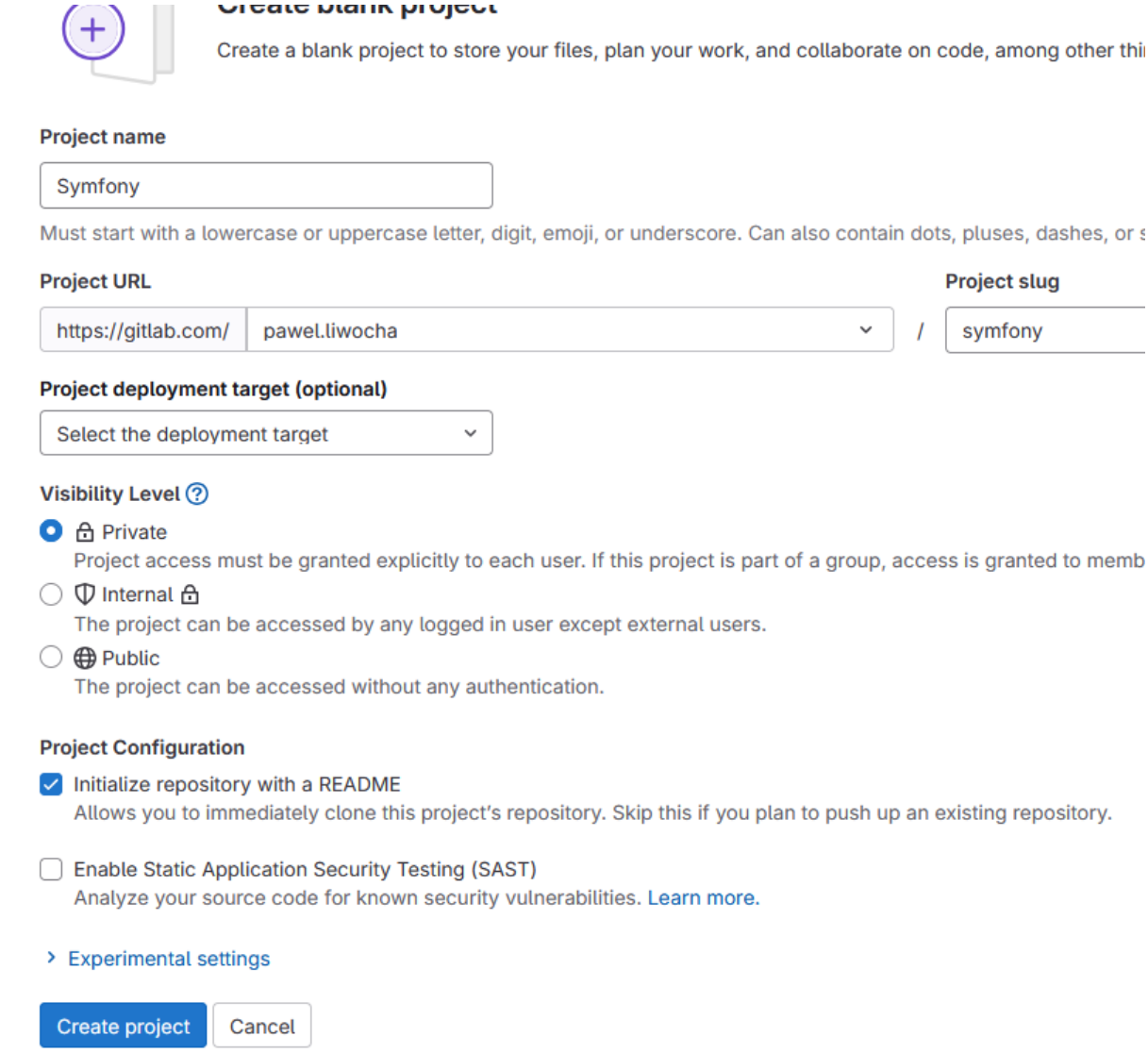
Klikamy plusika przy swoim awatarze i wybieramy *New project/repository*:



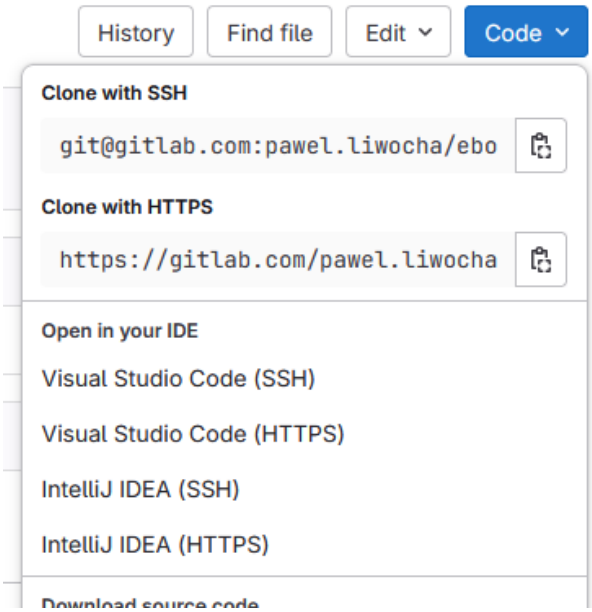
Następnie wybieramy *Create blank project*:



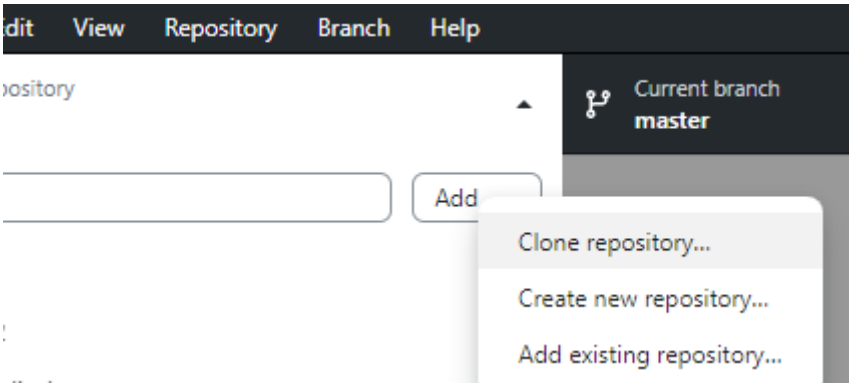
Tutaj dodajemy nazwę swojego projektu oraz slug, czyli adres url, jaki będzie miało. Możemy wybrać, czy ma być to projekt prywatny, czy publiczny oraz zainicjować go plikiem Readme:



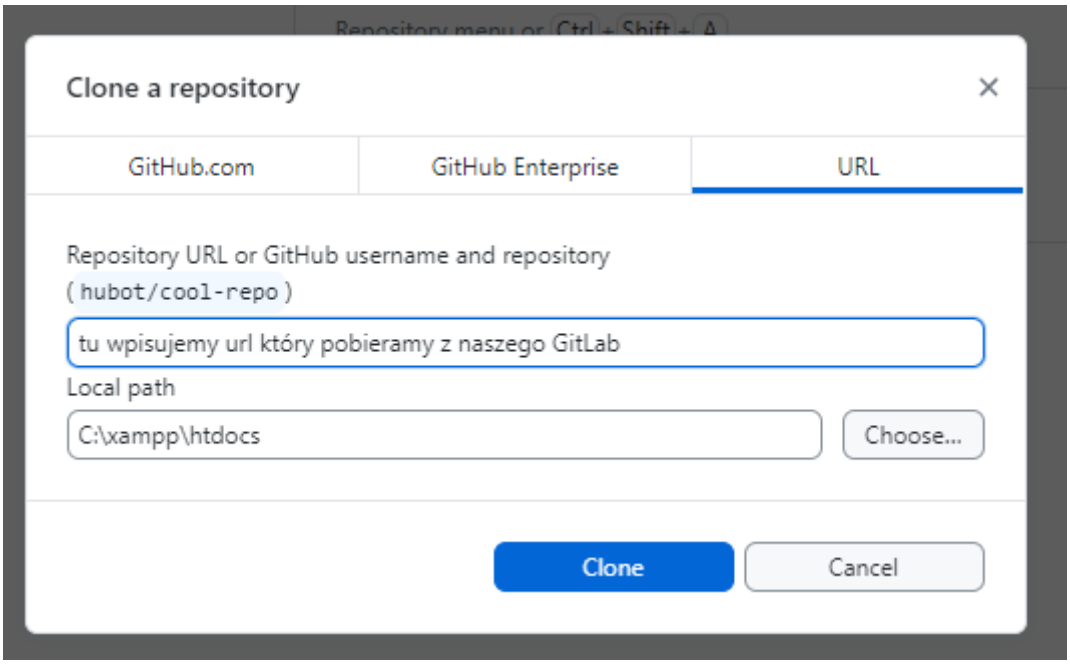
Ok udało się 😊 Pobierzmy URL do naszego repo, za pomocą którego pobierzemy je:



Teraz pobierzmy je do siebie na dysk i zacznijmy pracę. Używając GitHub Desktop możemy to zrobić w ten sposób:



Następnie wpisujemy url który pobieramy z naszego GitLaba i wybieramy folder, gdzie ma się ono skopiować:



I to wszystko. Mamy nasze repozytorium gotowe do pracy.

[...]

4. BUDOWA APLIKACJI KROK PO KROKU

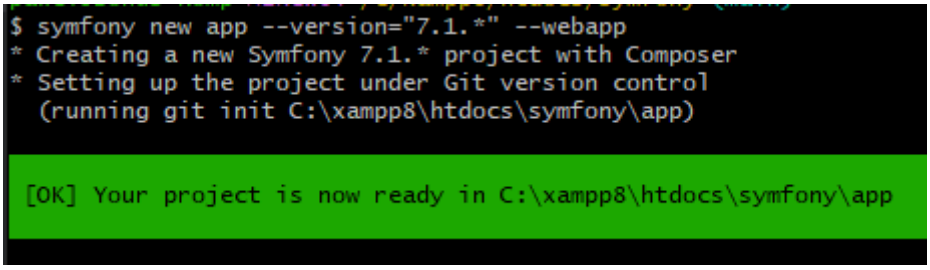
Zanim zaczniemy w ogóle budować aplikację, najpierw pobierzmy i zainstalujmy Symfony. Pobierzmy obecnie najnowszą wersję 7.1:

```
symfony new . --version="7.1.*" --webapp
```

I niestety, ale Symfony nie pozwoli nam się zainstalować do folderu, który nie jest pusty, a użycie flagi *–force* nie zadziała. Więc zrobimy to trochę „na piechotę”, zainstalujemy Symfony do folderu, a potem przekopiujemy wszystkie pliki do naszego głównego 😊

```
symfony new app --version="7.1.*" --webapp
```

Mamy Symfony w folderze app, teraz wszystko przekopiujmy do folderu wyżej (bez folderu .git).



Ja nie będę używał dockera, więc pliki dockerowe od razu usunę aby nie trzymać zbędnych rzeczy w gitcie.

Spójrzmy na plik *composer.json*, w którym mamy zapisane, jakich paczek używamy. Jest ich naprawdę sporo, ale wszystkie się przydadzą do naszej aplikacji. Nawet dołożymy więcej z czasem, ale krok po kroku 😉

Ja do pliku composer.json dodaję jeszcze dwie linijki na początku, aby był on zgodny ze standardami oraz przyszłe analizatory nie wytknęły nam błędu:

```
"name": "pawel.liwocha/symfony",
"description": "Symfony app for ebook",
```

Sercem naszego projektu są foldery **config**, **templates** oraz **src**. W folderze **vendor** znajduje się kod wszystkich paczek, które mamy zadeklarowane w pliku composera. Folder **var** przechowuje cache oraz logi naszej aplikacji.

Oczywiście, wszystkie foldery, które widzimy, są mega ważne. Folder **public** jest tak naprawdę otwarciem naszej aplikacji, to do niego kieruje konfiguracja Apache, czy Nginx. Folder **migrations** zawiera wszystkie migracje (zapytania tworzące) naszej bazy

danych, czy folder **bin**, w którym są pliki umożliwiające wykonywanie różnych poleceń bashowych naszego Symfony.

Bardzo ważnym plikiem jest plik **.env** oraz plik, którego standardowo nie ma, ale trzeba go utworzyć, czyli plik **.env.local**. Dlaczego ? Otóż plik .env jest czymś w rodzaju słownika zmiennych systemowych, to tutaj znajdują się odwołania do połączenia z bazą danych, ustawienie naszej aplikacji, w jakim trybie działa – dewelopersko, czy produkcyjnie oraz np. dane do serwera mailowego, z którego będą wysyłane wiadomości.

Pamiętajmy, aby w pliku .env **nie wpisywać** wartości „realnych", to jest plik, który siedzi w repozytorium, tutaj wpisujemy nazwy tych zmiennych, natomiast wartości realne (czyli np. użytkownik i hasło do bazy danych), trzymajmy w pliku .env.local, który nie jest wrzucany do repozytorium, a jego wartości nadpisują te w .env. Tak samo na środowisku produkcyjnym, czy testowym, zawsze musimy utworzyć plik .env.local i tam wpisać wartości testowe, czy produkcyjne.

Mamy również plik *.gitignore* - zawiera on pliki i katalogi, które nasz Git ma ignorować i nie wrzucać do naszego repozytorium. Są tu wpisy właśnie o *.env.local* czy folderach *var* oraz *vendor*. Ja korzystając z PHPStorma, jako IDE do programowania, dorzucam tutaj też katalog *.idea*, czyli miejsce, gdzie PHPStorm trzyma swoją konfigurację dla danego projektu. Taki plik wygląda tak:

```
###> symfony/framework-bundle ###
/.idea/
/.env.local
/.env.local.php
/.env.*.local
/config/secrets/prod/prod.decrypt.private.php
/public/bundles/
/var/
/vendor/
###< symfony/framework-bundle ###

###> phpunit/phpunit ###
/phpunit.xml
.phpunit.result.cache
###< phpunit/phpunit ###

###> symfony/asset-mapper ###
/public/assets/
/assets/vendor/
###< symfony/asset-mapper ###
```

Jeśli pracujemy w XAMPP, który ma w sobie Apache, bądź nasz serwer produkcyjny, czy testowy będzie miał Apache, powinniśmy dodać plik *.htaccess* do folderu *public*, do którego będzie się odwoływał. Plik ten pozwala na użycie „zwykłego tekstu" jako URL, bez rozszerzenia o pliki.

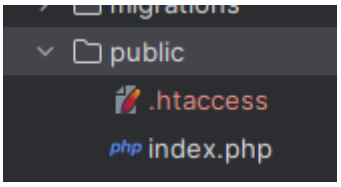
Więc dodajmy taki plik o takiej przykładowej treści:

```
RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

Dokładnie w tym miejscu:



Dzięki temu na serwerach z Apache, wszystkie nasze adresy będą działały poprawnie.

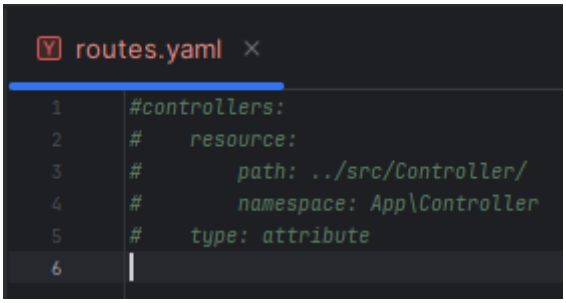
[...]

4.1.1. ROUTING W SYMFONY

Zatem zacznijmy od routingu. Cały routing aplikacji, czyli miejsce, w którym są definiowane adresy url naszej aplikacji, znajduje się w katalogu *config*. Dokładnie to tutaj zdefiniujemy główne drzewo naszych adresów, skierujemy podstawowe url'e na konkretne foldery. Reszta gałęzi naszego drzewa urli będzie już sięgała do folderów Kontrollerów, gdzie będą przypisywane ostateczne „końcowe" adresy do funkcji.

Jak wspomniałem na początku tego ebooka, przypomnę jeszcze raz, gdy już dochodzimy do kluczowego momentu, czyli kodu. Wszystko, co tutaj przedstawiam, stanowi moją subiektywną ocenę oraz pokazuję, w jaki sposób ja pracuję. Nie jest to jedyny słuszny model pracy, ale moje doświadczenie i wiele aplikacji napisanych w Symfony, pokazują, że sprawdza się znakomicie, jest czytelne dla innych oraz łatwe i otwarte na zmiany.

Nasz główny routing będzie znajdował się w folderze *config/routes*. Dlatego edytuję plik *config/routes.yaml* i komentuję wszystko, co tam jest:



Oraz tworzę plik *attributes.yaml* w folderze podanym wyżej. W tym pliku zdefiniuję kilka routów oraz wyjaśnię, o co chodzi. To bardzo proste, ale niesamowicie przydatne przy większych aplikacjach, aby jasno podzielić sobie kontrolery w projekcie. Tak wygląda teraz mój plik:

```
kernel:
    resource: ../../src/Kernel.php
    type: attribute

controllers_web:
    resource: ../../src/Controller/Web/
    type: attribute
    prefix: /
    name_prefix: web_

controllers_api:
    resource: ../../src/Controller/Api/
    type: attribute
    prefix: /api/
    name_prefix: api_
```

Pierwszy wpis to deklarujący, gdzie znajduje się Kernel symfony, jest to standardowy wpis.

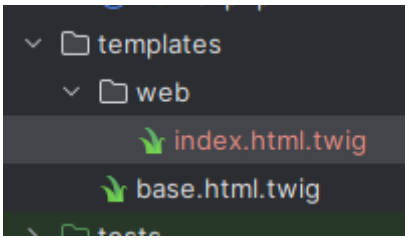
[...]

4.1.2. SYSTEM SZABLONÓW TWIG

Twig to potężny i wszechstronny system szablonów, stworzony z myślą o upraszczaniu pracy z warstwą widoku w aplikacjach. Jest domyślnym silnikiem szablonów w Symfony, który pozwala na łatwe oddzielenie logiki aplikacji od jej prezentacji. Tworzenie interfejsów użytkownika za pomocą Twig sprawia, że praca staje się prostsza i bardziej efektywna – zarówno dla backendowców, jak i dla frontendowców.

Pliki widoków są zapisane z rozszerzeniem *.twig*, jest to połączenie standardowego HTMLa z możliwościami dynamicznego generowania treści w oparciu o zmienne przekazywane z kontrolerów Symfony. Nasza aplikacja wygenerowała nam standardowy plik *base.html.twig,* o który będziemy rozszerzali nasze widoki. W tym pliku mamy podstawowe deklaracje HTMLa, na potrzeby tego ebooka nie będziemy go modyfikowali, a jedynie rozszerzymy inne pliki właśnie o niego.

Dla porządku w naszej aplikacji, pliki widoków są w folderze *templates,* możemy tu tworzyć kolejne foldery i pliki, co za chwilę pokażę. Stwórzmy więc nowy widok Twiga, a następnie dodamy do niego jakąś treść z kontrolera.



Stworzyliśmy nowy plik, teraz rozszerzmy go o plik bazowy:

W pierwszej linii dodaliśmy kod

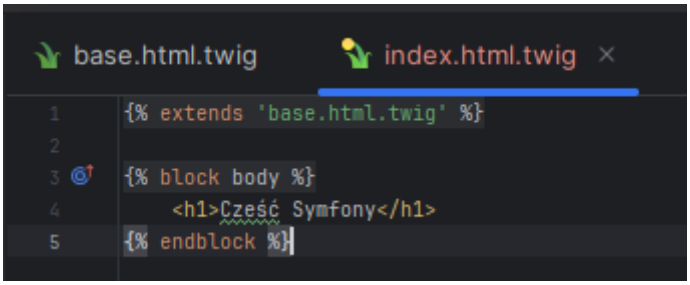
```
1. {% extends 'base.html.twig' %}
```

Dzięki czemu nasz nowy twig jest rozszerzony o *base.html.twig* i możemy „do niego” wrzucać treści.

Dodamy teraz treść która wyświetli się w bloku *body* pliku base:

```
3. {% block body %}
4.     <h1>Cześć Symfony</h1>
5. {% endblock %}
```

Plik teraz wygląda tak:



Żeby skorzystać z tego widoku, w naszym kontrolerze musimy zmienić odpowiedź.

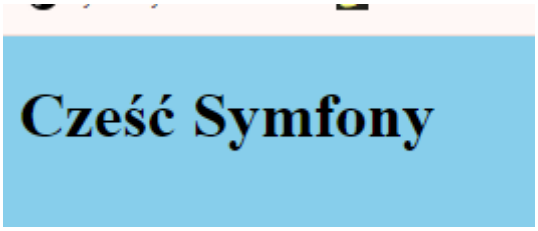
Jak pamiętasz obecnie zwracamy mu treść html przygotowaną w kontrolerze:

```
16. return new Response(
17.     '<html><body>Dzień dobry</body></html>'
18. );
```

Zamienimy to teraz na użycie twiga:

```
16. return $this->render('web/index.html.twig');
```

Używamy funkcji *render*, która renderuje nam podany w ścieżce widok utworzony chwilę wcześniej. Proszę, o to efekt:



[...]

4.1.3. ZABEZPIECZENIE APLIKACJI PRZED ATAKAMI

Teraz pora na nieco bardziej zaawansowany temat: zabezpieczenie aplikacji. O ile autentykacja i autoryzacja pomagają określić, kim jest użytkownik i co może robić, to nie zabezpieczą aplikacji przed atakami, które mogą pochodzić z zewnątrz. Na szczęście, Symfony dostarcza nam narzędzia, dzięki którym możemy skutecznie bronić się przed najczęściej występującymi zagrożeniami. W tym podrozdziale przyjrzymy się dwóm najpopularniejszym typom ataków – CSRF (Cross-Site Request Forgery) i XSS (Cross-Site Scripting), a także SQL Injection. Dowiesz się, jak je rozpoznać, jak się przed nimi bronić i co oferuje Symfony, by ułatwić nam to zadanie.

CSRF – Cross-Site Request Forgery

Wyobraź sobie sytuację: Użytkownik zalogował się do Twojej aplikacji, a w międzyczasie otworzył nieco podejrzaną stronę w innym oknie przeglądarki. Strona ta, nie prosząc o pozwolenie, wysyła żądanie do Twojej aplikacji w imieniu użytkownika, powodując niechciane akcje – np. zmianę hasła lub złożenie zamówienia. To właśnie jest atak CSRF. W skrócie: CSRF to atak, w którym złośliwa strona nakłania użytkownika do wykonania działań, których ten użytkownik wcale nie chciał wykonać.

No dobra, ale jak się przed tym bronić? Na szczęście Symfony ma na to gotowe rozwiązanie w postaci tokenów CSRF.

Jak działa mechanizm CSRF?

Za każdym razem, gdy tworzysz formularz, Symfony generuje unikalny token CSRF, który jest dołączany do formularza jako ukryte pole. Kiedy użytkownik wysyła formularz, Symfony sprawdza, czy token jest poprawny. Jeśli token nie pasuje lub jest nieobecny, Symfony odrzuca żądanie.

Ustawienie takie możemy włączyć w naszym configu, w pliku *framework.yaml*, nazywa się *csrf_protection: true*.

Kiedy tworzyliśmy nasze logowanie, właśnie takie zabezpieczenie się dodało, w pliku *login.html.twig*, mamy taką linijkę:

```
23. <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate')
}}" >
```

Jest to ukryty input, który zawiera właśnie token CSRF, a dokładnie jest to token „o nazwie" *authenticate*. Następnie w momencie odebrania danych możemy porównać te tokeny. Dajmy przykład, przykład opisuje, gdy tworzymy cały formularz ręcznie:

W jakimś twigu dodajemy sobie taki input:

```
<input type="text" name="token-admin-parking-edit" value="{{ csrf_token('token-
admin-user-edit') }}" required/>
```

[...]

6. ANALIZA JAKOŚCI KODU

Oprócz samego działania aplikacji, którą napisaliśmy, powinniśmy też poświęcić czas na utrzymanie naszego kodu. Pracując przy jednej aplikacji, czasami możemy przed rok nie wrócić do jakiegoś miejsca, a dobrze byłoby w momencie zajrzenia w ten zakamarek, aby kod był dla nas – ale nie tylko - jasny i zrozumiały. Jakość kodu wpływa bezpośrednio na wydajność zespołu, zrozumiałość dla przyszłych programistów oraz oczywiście na samą aplikację. Dlatego warto poświęcić chwilę na analizę, ocenę i poprawę jakości. Na szczęście, w PHP mamy dostęp do całego arsenału narzędzi, które pomogą nam w tej trudnej, ale bardzo satysfakcjonującej pracy. Przyjrzyjmy się kilku takim paczką i zobaczmy, co potrafią oraz, co mogą nam zaoferować.

6.1 LINT

To absolutna podstawa, jeśli chcesz uniknąć najbardziej oczywistych błędów, takich jak brakujący średnik, złe nawiasy, czy błędy składniowe. Linty to narzędzia, które w czasie

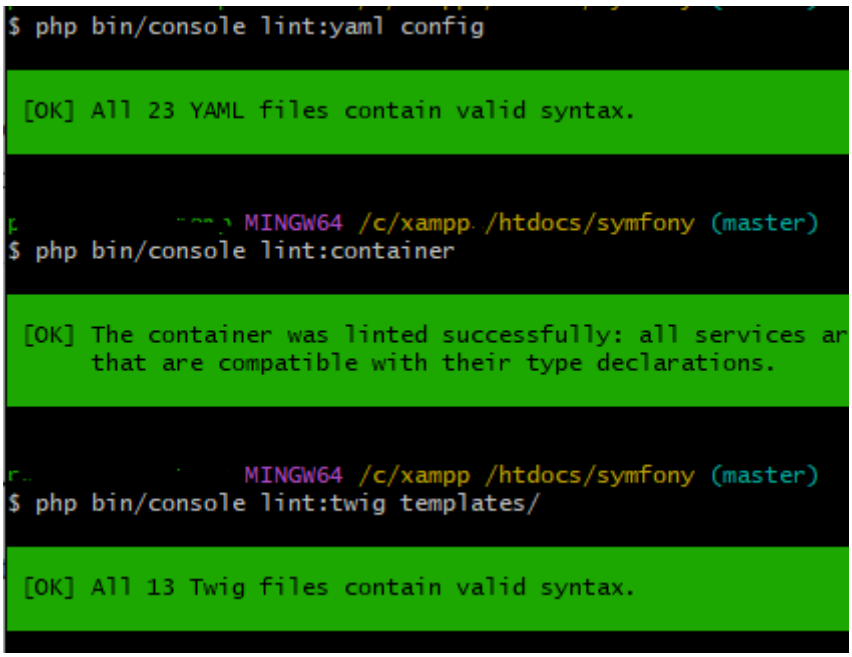
rzeczywistym wskazują Ci takie problemy, zanim Twój kod trafi do repozytorium czy na produkcję.

Lintowanie, w połączeniu z analizą statyczną (PHPStan, PHPMD), tworzy niesamowicie skuteczny tandem. Możesz lintować kod już w trakcie jego pisania i mieć pewność, że Twoja aplikacja będzie nie tylko działała poprawnie, ale też będzie łatwa do utrzymania.

W Symfony mamy kilka komend do sprawdzenia:

```
lint:container
lint:twig
lint:xliff
lint:yaml
```

Tak więc sprawdźmy naszą aplikację:



Wygląda, ze jest wszystko Ok, czego i Tobie życzę przy każdym projekcie !

8. INTEGRACJA Z GITLAB I CI/CD

W tym rozdziale zajmiemy się tym, jak zintegrować swoją aplikację Symfony z GitLabem oraz, jak wykorzystać potęgę CI/CD, aby uczynić proces rozwoju i wdrażania Twojego projektu bardziej efektywnym.

8.1 CO TO JEST CI/CD? DLACZEGO WARTO TO STOSOWAĆ?

CI/CD to skrót od Continuous Integration (ciągła integracja) i Continuous Deployment (ciągłe wdrażanie). Możesz pomyśleć o CI/CD, jak o tajnej broni każdego dobrego programisty, która pozwala zaoszczędzić mnóstwo czasu i zredukować liczbę błędów w produkcji. Co dokładnie oznaczają te terminy?

- **Ciągła integracja** to praktyka, która polega na regularnym łączeniu zmian w kodzie z główną gałęzią repozytorium. Dzięki temu, za każdym razem, gdy programista wprowadza zmiany, zautomatyzowane testy są uruchamiane, aby upewnić się, że wszystko działa poprawnie. Można to porównać do codziennego sprzątania biurka – lepiej to zrobić na bieżąco, niż czekać na koniec miesiąca.
- **Ciągłe wdrażanie** to proces automatycznego wdrażania aplikacji na środowisko produkcyjne po przejściu pomyślnie wszystkich testów. Oznacza to, że nie musisz martwić się o ręczne wprowadzanie zmian, które mogą prowadzić do błędów ludzkich.

Czemu warto to stosować? Oto kilka kluczowych korzyści:

Szybsze wykrywanie błędów: Dzięki regularnej integracji kodu, błędy są wychwytywane niemal natychmiast po ich powstaniu. Wyobraź sobie, że robisz ciasto i dodajesz składniki jeden po drugim. Jeśli coś jest nie tak, zauważysz to szybko, zanim przejdziesz do następnego kroku. To samo dotyczy kodu – szybkie wykrywanie problemów oznacza mniej pracy później.

Poprawa jakości kodu: CI/CD sprzyja lepszemu pisaniu testów. Kiedy masz zautomatyzowane testy uruchamiane przy każdym pushu do repozytorium prowadzi to do wyższej jakości kodu i większego zaufania do aplikacji. Kto by nie chciał, aby ich oprogramowanie było mniej podatne na błędy?

Skrócenie czasu wdrażania: Proces wdrażania staje się szybki i bezproblemowy. Dzięki automatyzacji możesz wdrażać nowe funkcje i poprawki w rekordowym tempie, co daje Ci przewagę nad konkurencją. Wyobraź sobie, że możesz dodać nową funkcjonalność do swojego oprogramowania i w ciągu kilku minut mieć ją na żywo!

Zwiększenie wydajności zespołu: Automatyzacja testów i wdrożeń pozwala zespołom skupić się na pisaniu nowego kodu zamiast na ręcznym wdrażaniu i testowaniu. To jak dodanie turbo do silnika – nagle Twoje tempo pracy wzrasta, a Ty możesz skoncentrować się na tym, co naprawdę ważne.

Większa satysfakcja użytkowników: Regularne aktualizacje i poprawki sprawiają, że użytkownicy są bardziej zadowoleni z Twojego oprogramowania. Nic nie irytuje bardziej niż stara, nieaktualna aplikacja. Dzięki CI/CD możesz dostarczać nowe funkcjonalności i poprawki na bieżąco, co przekłada się na lepsze doświadczenie użytkowników.

Lepsze zarządzanie ryzykiem: Automatyzacja procesów związanych z testowaniem i wdrażaniem zmniejsza ryzyko wprowadzenia błędów do środowiska produkcyjnego. Dzięki temu, że aplikacja jest regularnie testowana, masz większą pewność, że każda nowa wersja jest stabilna i działa zgodnie z oczekiwaniami.

[...]