# Projekt Zespołowy 2

## konsultacje dotyczące wirtualizacji, konteneryzacji oraz bezpieczeństwa

Temat projektu: Portal dla przyszłych inżynierów

Mateusz Krakowski

Tymoftyjewicz Maciej

Daniel Kobiałka

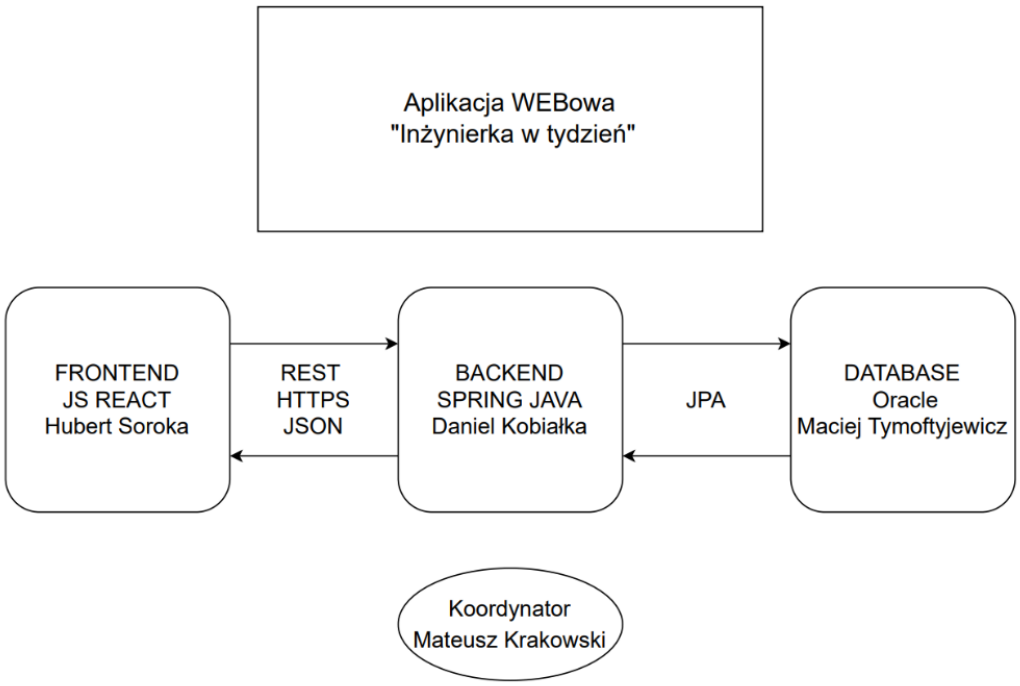
Soroka Hubert

8 stycznia 2023

# 1 Wirtualizacja i Konteneryzacja

Nasza aplikacja składa się z trzech niezależnych od siebie części:

- Frontend - Typesript/React
- Backend - Java/Spring
- Baza danych - Oracle




W kontekście konteneryzacji i wirtualizacji pomijamy bazę danych, ponieważ jest administrowana przez uczelnię. Oczywiście w przypadku, gdybyśmy utrzymywali swoją bazę, takie rozważania byłyby zasadne.

## 1.1 Konteneryzacja

Backend i frontent stanowią dwie osobne aplikacje, komunikujące się przez sieć. Każdą z nich można uruchomić i utrzymywać niezależnie od drugiej. Oznacza to możliwość użycia osobnych kontenerów lub maszyn wirtualnych dla każdej z nich. Użycie kontenerów nie byłoby skomplikowanym rozwiązaniem, które w zupełności wystarczyłoby naszym potrzebom. Backend i frontend uruchomione byłby na osobnych kontenerach dockera, połączone w jednej sieci w trybie bridge, który zapewniłby swobodną komunikację pomiędzy nimi, lecz były niewidoczne z zewnątrz, na czym nam zależy. Użycie kontenerów usprawniłoby też teoretyczne uruchomienie naszej aplikacji w złożonych środowiskach chmurowych (mało prawdopodobne, że dojdzie do tego w ramach PZSP2), problem organizacji aplikacji w kontenery byłby rozwiązany. Konteneryzacja nie zostanie zaimplementowana w ramach projektu z PZSP2, ponieważ nie jest istotna

dla działania aplikacji. Nie planujemy również przeprowadzać wdrożeń ani utrzymania, zatem potrzeba konteneryzacji nie występuje w naszym projekcie.

### 1.2 Wirtualizacja

W ramach naszego projektu nie przewidujemy używania maszyn wirtualnych w kontekście uruchamiania na nich elementów naszej aplikacji, co najwyżej w ramach testów. W naszej opinii, kontenery stanowią prostsze i wystarczające rozwiązanie.

## 2 Zapewnienie podstawowego poziomu bezpieczeństwa

Aby zapewnić bezpieczne działanie i ułatwić administrowanie zawartością, wprowadziliśmy następujące mechanizmy.

### 2.1 Poziomy uprawnień

Można rozróżnić 3 poziomy uprawnień:

- Użytkownik niezalogowany

  Nie ma żadnych uprawnień do aplikacji, po próbie wejścia na dowolną stronę aplikacji, zostaje przekierowany na stronę do logowania. Razem ze stroną do rejestracji, są to jedyne strony, z których mogą korzystać niezalogowani użytkownicy.

- Użytkownik zalogowany

  Może korzystać ze wszystkich głównych funkcji aplikacji, ma pełną kontrolę nad treścią (fiszkami i grupami), którą sam stworzył. Nie może edytować ani oglądać treści tworzonej przez innych użytkowników ani tworzyć treść w nieswoim imieniu. Każde konto po rejestracji ma ten poziom uprawnień.

- Administrator

  Może to samo co użytkownik, lecz bez ograniczeń co do zarządzania nieswoją treścią. Nie ma możliwości tworzyć takiego konta inaczej, niż przez wykonanie poleceń na bazie danych.

Na potrzeby naszej aplikacji taki podział wystarcza, ponieważ nie planujemy uprzywilejowanych użytkowników, poza już istniejącymi uprawnieniami administratora, które są przydatne podczas testowania manualnego, ale rozszerzenie go (np. poprzez dodanie roli moderator) nie byłoby problematyczne.

## 2.2 Uwierzytelnienie z tokenem JWT

Aby móc weryfikować uprawnienia i osobowość zalogowanego użytkownika, zaimplementowaliśmy system uwierzytelnienia wykorzystujący token JWT. Jego działanie najłatwiej wytłumaczyć na przykładzie listy kroków:

1. Użytkownik wysyła żądzanie POST z danymi do logowania pod /account/login.
2. Serwer sprawdza, czy dane są poprawne, jeśli tak, odsyła token JWT, z dołączonym skrótem wykonanym z użyciem HMACSHA-256, z kluczem umieszczonym na serwerze, zawierający:
   - nazwę użytkownika,
   - uprawnienia (czy jest administratorem),
   - skrót z hasła,
   - datę wystawienia,
   - podmiot wystawiający, w naszym wypadku "MHMD".
3. Frontend zapamiętuje otrzymany token, przy użyciu react-secure-storage,
4. Użytkownik wysyła dowolne żądanie, oprócz /account/login i /account/register, załączając przechowany token w nagłówku Authorization.
5. Zanim serwer zacznie przetwarzać żądanie właściwe, następuje walidacja i dekodowanie tokenu. Ponownie wyliczany jest skrót HMACSHA-256 z tym samym sekretem. Jeśli nie będzie zgodny ze skrótem w tokenie, token nie występuje lub nie jest w właściwym formacie, lub dane z tokenu nie zgadzają się ze stanem w bazie danych, walidacja nie przechodzi i zwracana jest odpowiedź Unauthorized. Jeśli walidacja przejdzie, zapytanie jest wykonywane zgodnie z uzyzkanymi uprawnieniami.
6. Uzytkownik otrzymuje odpowiedź z serwera z danymi.

Tokeny posiadają też datę ważności, po upłynięciu której, należy się zalogować jeszcze raz.

## 2.3 Szyfrowanie haseł

Po otrzymaniu od użytkownika hasła w postaci tekstowej, na backendzie następuje szyfrowanie z użyciem soli. Zabezpieczone hasło i sól są składowane w bazie danych, dzięki czemu nawet po włamaniu się do bazy danych, atakujący nie może bezpośrednio przeczytać hasła (można to obejść zamieniając hasło i sól na wartości pobrane ze znanego konta, ale to wymaga uprawnień do edycji danych w bazie). Zaszyfrowane hasło (a właściwie 16 bajtowy sekretny klucz, generowany na jego podstawie) jest tworzony z użyciem PBE (Password Based Encryption), konkretnie PBKDF2 z wykorzystaniem HMACSHA-1 oraz 16 bajtowej, losowej soli. Liczba iteracji jest ustawiona na 16384, lecz można ją zwiększyć, jeśli nie będzie to prowadzić do problemów z wydajnością.

### 2.4 Hasła aplikacji

Wszystkie hasła i klucze, przeznaczone do użytku przez i tylko przez aplikację (np. hasło do bazy danych, klucze szyfrowania) nie są dostępne w repozytorium kodu, muszą być ręcznie dopisane na docelowej maszynie. Samo repozytorium też docelowo nie powinno być publiczne, ale aby ułatwić dostęp prowadzącym i studentom, jest dostępne dla użytkowników wydziałowego GitLaba.

## 3 Zidentyfikowanie najistotniejszych zagrożeń bezpieczeństwa i przeciwdziałanie

### 3.1 Ataki Injection (SQL lub JS)

Zagrożenie zneutralizowane. Wszystkie pola, z których przesyłane są dane od użytkownika, są obarczone ograniczeniem długości i typu inputu. Dodatkowo, w przypadku danych logowania, można użyć tylko określonej listy znaków specjalnych (nie można używać np. "lub %). Jpa korzysta z wbudowanego mechanizmu "prepared statments", który pozwala na oddzielenie inputu użytkownika od query wysyłanego do bazy danych.

### 3.2 XSS

Zagrożenie zneutralizowane. Dzięki stosowaniu wcześniej wymienionej walidacji danych przesyłanych od użytkownika, użytkownik nie może wstrzyknąć złośliwego kodu na serwer.

Do przechowywania danych po strone klienta wykorzystujemy react-secure-storage. Przy każdym żądaniu jest dołączany token JWT, pobierany stamtąd. Nawet gdyby atakującemu udało się umieścić tam złośliwy kod, token jest każdorazowo walidowany, a sam token jest odsyłany tylko, kiedy jest generowany. Atakujący nie ma możliwości umieścić złośliwego kodu na serwerze, który byłby odsływany wielu użytkownikom (persistent XSS).

### 3.3 Ataki DoS

W obecnym stanie projektu ataki DoS są zagrożeniem, które ciężko w pełni zneutralizować. Aby im przeciwdziałać, skorzystamy z firewalla, który może zablokować niepożądany ruch w sieci. Innym rozwiązaniem byłoby skorzystanie z środowiska chmurowego.

W aplikacji istnieje tylko jeden endpoint, pozwalający zdmodyfikować bazę danych bez logowania się do aplikacji - /account/register. Aby utrudnić atak DDoS polegający na masowym tworzeniu kont przez ogromną liczbę użytkowników, należałoby umieścić captche oraz rejestrację z walidacją poprzez potwierdzenie mailowe. Zagrożenie jest raczej niezbyt prawdopodobne, ze względu na tematykę aplikacji (platforma edukacyjna), a testy zmasowanej rejestracji i wysyłanie maili są raczej kosztowne czasowo i nie wnoszą wartości względem istoty działania aplikacji. Z tych względów nie zaimplementowaliśmy tych środków przeciw masowej rejestracji.

## 3.4 Man in the middle

Aby uniemożliwić ataki typu man in the middle, staraliśmy się jak najbardziej ograniczyć przesyłanie i wrażliwych danych w prostej postaci. Jedyne miejsce, gdzie w aplikacji są przesyłane i używane hasła w niezabezpieczonej postaci, to logowanie i rejestracja, a konkretnie żądania POST /account/login i /account/register. Aby uniemożliwić przechwycenie haseł przesyłanych plaintextem, należy zaszyfrować kanał komunikacyjny, którym są przesyłane. W tym wypadku pomiędzy frontendem a backendem. Nalezy wykorzystać protokół HTTPS zamiast domyślnego HTTP. Aby móc to zrobić, potrzebny jest podpisany przez zaufaną trzecią stronę certyfikat. Przeprowadziliśmy próbę z samopodpisanym certyfikatem, lecz przeglądarki blokowały takie zapytania, ponieważ nie mogły zweryfikować poprawności certyfikatu. Próba umożliwienia działania aplikacji z niezweryfikowanym certyfikatem za wszelkę cenę mija się z celem, ponieważ w jego miejscu powinien się znajdować zdobyty legalnie certfikat z zaufanego źródła. Jest to już zagadnienie bardziej administracyjne niż programistyczne, dlatego na ten moment pracujemy bez certyfikatu, korzystając z HTTP, świadomi zagrożenia tego typu atakami.

Podobnie ma się sprawa z połączeniem pomiędzy bazą danych a backendem. Istnieje możliwość połączenia się z wykorzystaniem SSL, lecz ponownie potrzeba podpisanego certyfikatu. W przypadku bazy danych jest to mniej istotne, ponieważ nie ma możliwości podejrzeć samego hasła, ponieważ jest wysyłane i składowane w postaci zaszyfrowanej.

## 3.5 Naruszenie ochrony danych osobowych (data breach)

Aby przeciwdziałać naruszeniu danych osobowych stosujemy szyfrowanie haseł oraz zasadę najmniejszego uprzywilejowania. Niestety, może dojść do sytuacji w której złamane/ukradzione zostanie hasło administratora, co otworzy dostęp do danych osobie nieupoważnionej. W przypadku naszej aplikacji, nie wymagamy, a nawet nie zalecamy podawać danych osobowych, więc nawet w przypadku przejęcia danych z bazy przez osoby trzecie nie powinno dojść do wycieku wrażliwych danych.

## 3.6 XSRF

Spring zapewnia domyślną ochronę przed Cross-Site Request Forgery, z której korzystamy. Opiera się na dodatkowym tokenie, dołączanym do żądań.

Nie korzystamy w naszej aplikacji z cookies (a gdybyśmy zaczęli, upewnilibyśmy się, że są niedostępne z poziomu typescripta oraz że są tylko ustawiane i pobierane przez serwer), lecz z localStorage (a właściwie react-secure-storage, który zapewnia dodatkowe bezpieczeństwo zapisanych danych poprzez ich zaszyfrowanie), który nie jest automatycznie dołączany do żądań, jak niezabezpieczone cookies. Dane w nim składowane nie są dostępne poza stroną naszej aplikacji. Dodatkowo, aby utrudnić przeprowadzenie XSRF, nie ma zapytań typu GET, które zmieniają stan aplikacji. Tokeny JWT, umożliwiające przeprowadzenie żądań modyfikujących bazę danych, mają czas wygaśnięcia, co zmniejsza okno umożliwiające taki atak. W logice aplikacji sprawdzane jest czy użytkownik ma uprawnienia do modyfikacji zasobu.