DNSite

Aplikacja webowa do zarządzania serwerem DNS

Dokumentacja techniczna

Inżynieria Oprogramowania
Wydział Fizyki i Informatyki Stosowanej
Informatyka Stosowana, 3 rok

Arkadiusz Kasprzak Jarosław Cierpich Jakub Kowalski
Konrad Pasik Krystian Molenda

Spis treści

1 Wstęp

DNSite to aplikacja webowa do zarządzania serwerem DNS. Dostarcza ona użytkownikowi możliwości łatwej i wygodnej edycji danych związanych z serwerem PowerDNS przechowywanych w bazie PostgreSQL.
Niniejsza dokumentacja techniczna została przygotowana dla pierwszego pełnego wydania aplikacji. Zawiera informacje przydatne przy dalszym rozwoju aplikacji.

2 Budowanie aplikacji

W celu uruchomienia zbudowania i uruchomienia aplikacji wymagane są:

- Java w wersji 8
- Maven
- Baza danych PostgreSQL 11
- Aplikacja pgAdmin 4

Należy zadbać o to, by przed uruchomieniem aplikacji baza danych była już stworzona - może natomiast nie zawierać tabel. W celu uruchomienia aplikacji należy pobrać ją z serwisu **Github** za pomocą polecenia:

```
git clone https://github.com/agh-ki-io/DNSite
```

W celach developmentu zaleca się budowanie i uruchamianie aplikacji za pomocą zintegrowanego środowiska programistycznego, np. **Intellij IDEA**. W tym przypadku należy otworzyć za pomocą tego środowiska projekt, a następnie wykonać na nim operacje *clean* oraz *install* w Mavenie. Po ich zakończeniu należy uruchomić projekt (klasa stanowiąca punkt początkowy to **DNSiteApplication**).
W wypadku budowania z poziomu konsoli należy natomiast przejść do katalogu *DNSite* i wykonać z jego poziomu polecenie:

```
mvn clean install
```

Proces może potrwać kilka minut. Należy zignorować pojawiające się komunikaty (również te oznaczone słowem *error*). Po zakończeniu procesu instalacji sukcesem, w celu uruchomienia aplikacji należy **z poziomu katalogu DNSite** wykonać polecenie:

```
java -jar target/dnsite-0.0.1-SNAPSHOT.jar
```

W przypadku pierwszego uruchomienia pojawi się okno konfiguracji. Proces konfiguracji został szczegółowo opisany w **Dokumentacji użytkownika** dostępnej dla projektu. Po zakończeniu konfiguracji aplikacja jest dostępna pod adresem: `http://localhost:8001/`

3 Stos technologiczny

Poniżej przedstawiono stos technologiczny użyty podczas tworzenia aplikacji:

- **Baza danych:** PostgreSQL 11
- **Backend:**
  - Java 8 (w momencie pisania dokumentacji użycie nowszej wersji nie pozwala na poprawne zbudowanie aplikacji)
  - Framework Spring
  - Spring Boot
  - Spring Security
  - Hibernate
  - JSP
  - log4j
  - snakeyaml
  - gson
- **Frontend:**
  - React (oraz: react-router, react-table, react-bootstrap)
  - JSP
  - CSS

Do **budowania aplikacji** użyty został *Maven*.

4 Warstwa frontend

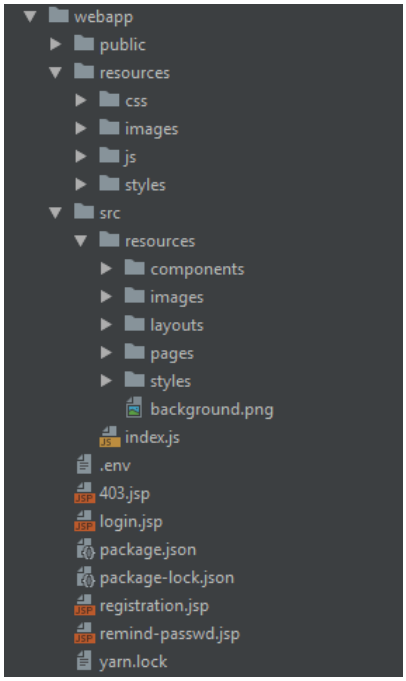
Ten rozdział opisuje sposób działania aplikacji po stronie frontendu. Omówione zostaną użyte technologie, organizacja kodu oraz najważniejsze komponenty, w tym komponent tabeli.

4.1 Użyte technologie

Pod względem użytych technologii warstwa frontend aplikacji DNSite stanowi hybrydę: główna część aplikacji napisana została z wykorzystaniem biblioteki **React**, natomiast strony służące m.in. do logowania, rejestracji i przypominania hasła napisane zostały w technologii **JSP (JavaServer Pages)**.
Hybrydowość warstwy frontend wynika z błędnego początkowego założenia, że cała aplikacja oparta będzie na technologii JSP. Kiedy okazało się, że znacznie wygodniejsze będzie użycie React'a, gotowe były już strony do logowania i rejestracji. Podjęta została decyzja o niewprowadzaniu zmian - wynikło to z obecności innych, ważniejszych dla działania aplikacji, zadań do wykonania.
W nowszej części aplikacji użyte zostały: *React*, *react-router*, *react-bootstrap* oraz *react-table*. Zdecydowaną zaletą React'a jest projektowanie oparte o komponenty. Jest to rozwiązanie, które charakteryzuje prostota i elegancja. Kolejnym ułatwieniem są jasne praktyki dotyczące przepływu danych w aplikacji. Zastosowany wzorzec to jednokierunkowy przepływ danych. Ponadto React jest bardzo wszechstronną oraz popularną biblioteką, posiada zaangażowaną społeczność oraz rozwijany przez nią ekosystem.

4.2 Organizacja kodu

Struktura kodu składającego się na warstwę frontend wygląda następująco:



Część aplikacji wykonana w technologii JSP widoczna jest na dole obrazka - są to pliki:

- **403.jsp** - obsługa kodu 403
- **login.jsp** - obsługa logowania
- **registration.jsp** - obsługa rejestracji nowych użytkowników
- **remind-password.jsp** - obsługa panelu przypominania hasła

Pliki odpowiadające za *stylowanie* tych stron znajdują się w katalogu `webapp\resources`

Część aplikacji zbudowana za pomocą React'a znajduje się natomiast w katalogu `webapp\src`. Najważniejsze pliki to:

- **App.js** - ogólny komponent, który zbiera wszystkie komponenty
- **Footer.js** - komponent odpowiadający za renderowanie stopki aplikacji
- **Header.js** - komponent odpowiadający za renderowanie nagłówka aplikacji tj. Tytułu aplikacji

- **Navigation.js** - komponent odpowiadający za renderowanie nawigacji aplikacji,
- **UserBlock.js** - komponent odpowiadający za renderowanie bloku zawierającego nawigacje do zmiany hasła oraz wylogowania się z aplikacji
- **MainPage.js** - komponent ten wykorzystuje komponenty biblioteki react-router. Dzięki komponentowi Switch w komponencie MainPage zostanie wyrenderowany maksymalnie jeden komponent naraz tj. ten który będzie odpowiadał adresowi URL, natomiast dzięki komponentowi Route możliwe jest określenie jaki komponent ma zostać wyrenderowany w przypadku odwiedzenia odpowiedniego adresu URL, jeżeli podany adres URL nie istnieje, aplikacja komponent Error404, który informuje o błędzie
- **ChangePassword.js** - komponent odpowiadający za renderowanie inputów umożliwiających zmianę hasła użytkownika
- **Error404.js** - komponent renderuje informacje o błędzie 404
- **Home.js** - komponent renderuje informacje o aplikacji oraz o autorach

Pliki *.css* w katalogu *styles* wykorzystywane są do stylowania odpowiadających im komponentów.
W katalogu *images* przechowywane są 2 pliki:

- **icon.png** - obraz wykorzystywany jest jako ikona aplikacji w karcie przeglądarki
- **background.png** - obraz wykorzystywany jest jako tło fragmentów aplikacji

4.3 Routing

Do obsługi routingu po stronie przeglądarki wykorzystana została biblioteka **react-router**. React-router jest bardzo popularną biblioteką, wykorzystywaną często w projektach *SPA (Single-Page Application)*. Dzięki tej bibliotece zapytanie do serwera wykonywane jest wyłącznie raz, na początku działania aplikacji. Po zmianie URL strona nie jest odświeżana, aplikacja przechwytuję zmianę URL i w oparciu o konfigurację Routera wyświetla odpowiedni widok dla danego URL. React-router wykonuje 3 podstawowe funkcje: modyfikuje URL, po wykonaniu modyfikacji ponownie renderuje aplikację, rozpoznaje URL i określa jakie komponenty mają być wyświetlane dla danej lokalizacji. Dzięki bibliotece react-router możemy też wykorzystać interfejs przeglądarki np. cofanie do poprzedniej strony bez odświeżania.

4.4 Komponent ReusableTable

TO SIE POJAWI JUTRO

5 Warstwa backend

Ten rozdział opisuje sposób działania aplikacji po stronie backendu. Omówione zostaną rozwiązana stanowiące o przepływie danych w aplikacji, jej bezpieczeństwie, tworzeniu kopii zapasowych danych przechowywanych w bazie, czy tworzących historię operacji na niej.

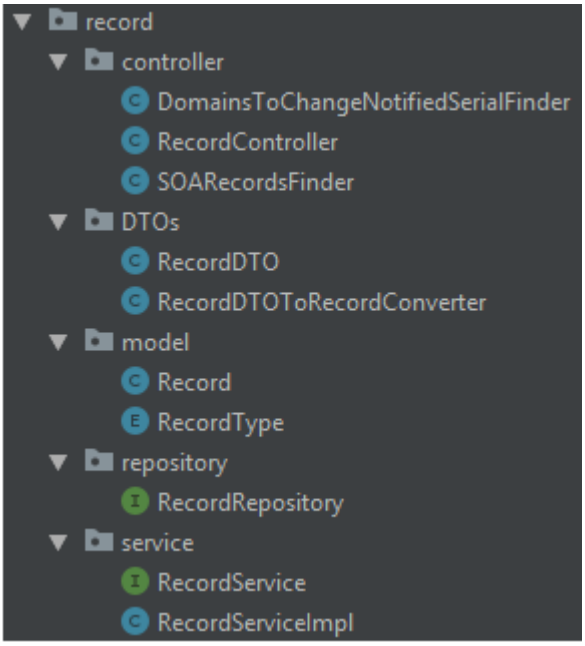
5.1 Struktura warstwy backend, rdzeń aplikacji

Struktura warstwy backend została stworzona w taki sposób, aby być zgodną z ustalonym standardem dla frameworku Spring, czyli dla każdej tabeli w bazie danych, której odwzorowanie tworzymy po stronie backendu tworzona jest następująca struktura pakietów:

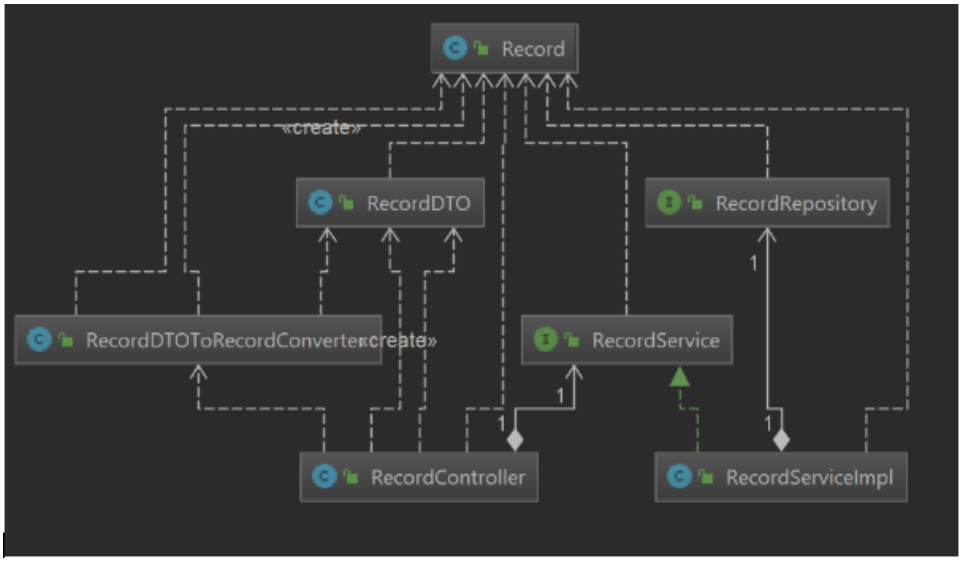
- **model** - znajduje się tutaj klasa, której pola mapowane są na pola tabeli w bazie danych
- **controller** - klasy odpowiadające za przekazywanie requestów między bazą danych oraz frontendem. Wykonują również potrzebne przetworzenia danych. Odpowiadają za dodatkową logikę w związku z DNS (np.: notifiedSerial)[1]
- **service** - odpowiada za logikę dostępu do danych
- **repository** - wykorzystuje JPA do dostępu do bazy danych, wykorzystywany przez service
- **DTO - Data Transfer Object** - model pozwalający na *obcięcie* danych wysyłanych z rzeczywistego modelu do frontendu. Dodatkowo klasa konwertująca pozwalająca na swobodne przejścia DTO <–> model

Poniżej przykład realizacji opisanej struktury dla obsługi *rekordów DNS*:

[1] Duża część *logiki biznesowej* została zaimplementowana po stronie controller, co powinno zostać przeniesione do service. Wynika to z niewystarczającego doświadczenia developerów na początku projektu.



oraz przykładowy diagram zależności dla tabeli rekordów:



Nie wszystkie tabele wykorzystywane przez PowerDNS zostały zaimplementowane z pełną obsługą. Można je rozpoznać po tym, że wewnątrz ich pakietu znajduje się jedynie pakiet *model*. Są to: tsig_keys, domain_meta_data, cryptokey.

Oprócz tego w części backendowej aplikacji znajduje się moduł **utils** w którym znajdują się klasy odpowiedzialne za:

- Dodatkowe funkcjonalności, np.: BackupPostgresql
- Adnotacje pozwalające na łatwą walidację danych przekazywanych do modeli, np.: IpAddress oraz IpAddressValidator
- DTO dla naruszeń walidacji, klasy do zaawansowanego przetwarzania DTO (SOARecordsCreator, DomainIdExtractor)
- Skonfigurowanie połączenia z bazą danych przy pierwszym uruchomieniu (pakiet hibernate)
- Wspólną logikę dla notifiedSerial (rekordy SOA w content posiadają również notifiedSerial)

5.2 Logika notifiedSerial

NotifiedSerial to wartość ustawiana dla *domeny DNS*, pozwalająca na powiadomienie pozostałych serwerów o aktualizacji zawartości domeny. W przypadku, gdy utworzona zostaje nowa domena, do jej pola *notifiedSerial* przypisywana jest następująca wartość typu integer: *YYYYMMDD00* - jest to liczba dziesięciocyfrowa, której pierwsze 8 cyfr przedstawia datę dodania.
Wartość notifiedSerial jest zwiększana w przypadku, gdy któryś rekord typu **NIE** SOA przypisany do domeny zostanie zmieniony. W przypadku, gdy notfiedSerial jest ustawiony to jest on zwiększany o 1. Jeśli jego wartość nie jest ustawiona, to tworzony jest on na nowo zgodnie z wcześniejszym algorytmem.

5.3 Bezpieczeństwo i użytkownicy

W katalogu **security** znajdują się wszystkie klasy odpowiedzialne za bezpieczeństwo aplikacji. Zostały one napisane głównie przy użyciu *frameworka Spring Security*.

Użytkownicy aplikacji mają przydzielone odpowiednie role:

- ADMIN - ma dostęp do całej aplikacji, może wykonywać każdą dostępną aplikację, w tym przyznawać innym użytkownikom dostęp do strony. Pierwszy użytkownik aplikacji automatycznie otrzymuje status administratora.
- USER - jego prawa są ograniczone, nie ma dostępu do głównej części aplikacji. Użytkownik o tym statusie oczekuje na przyznanie dostępu do aplikacji.

Przyznanie użytkownikowi z rolą *USER* dostępu do strony przez administratora zmienia jego rolę na *ADMIN*.

W katalogu **config** znaleźć można konfigurację odpowiedzialną między innymi za uniemożliwienie użytkownikom korzystania z elementów aplikacji, do których nie mają dostępu.
Linia:

```
.antMatchers("/resources/**",
    "/registration","/remind-passwd").permitAll()
```

definiuje miejsca w aplikacji, do których dostęp ma każdy bez względu na swoje uprawnienia.
Kod:

```
.loginPage("/login")
```

definiuje domyślną stronę logowania.
Linia:

```
.exceptionHandling().accessDeniedPage("/403");
```

sprawia, że gdy użytkownik próbuje wejść na stronę, do której nie ma dostępu, to otrzymuje on w odpowiedzi stronę z kodem 403.
Hasła są **hashowane** przy użyciu **bCryptPasswordEncoder()**.

Warstwa bezpieczeństwa aplikacji zbudowana jest ponadto z następujących klas:

- **UserServiceImpl** - pozwala ona na mapowanie obiektów na rekordy w bazie. W funkcji *save* do zapisania hasła używany jest wspomniany wcześniej *enkoder* - hasło nie jest więc przechowywane w postaci czystego tekstu. Klasa ta pozwala również na aktualizację danych w bazie (hasło) czy też przeszukiwanie bazy.

- **UserController** - za pomocą tej klasy wystawiane jest API służące późniejszej obsłudze użytkowników po stronie frontendu. Większość funkcji w tej klasie nie wymaga osobnego komentarza - wyjątkiem jest funkcja *registration*, służąca do rejestracji. W tej funkcji początkowo dokonujemy walidacji danych formularza otrzymanych z warstwy frontend. Następnie wykonywane jest sprawdzenie, czy użytkownik jest pierwszym rejestrowanym w aplikacji. Jeśli tak jest, to zostaje on automatycznie zalogowany i przyznana zostaje mu rola ADMIN. W przeciwnym wypadku do wszystkich administratorów wysyłana jest wiadomość e-mail z informacją, że nowy użytkownik oczekuje na przyznanie dostępu do aplikacji. Administratorzy mogą zadecydować o przyjęciu nowego użytkownika.

- encja **User** - reprezentuje użytkownika aplikacji w bazie danych. Zawiera następujące atrybuty: *id*, *username*, *password*, *role*, *firstName*, *lastName*, *registrationDate*, *lastLoginDate*, *email*, *isUserAccepted*.

- **SecurityServiceImpl** - najważniejszą funkcją w tej klasie jest *autologin*. Jeśli aplikacja wykrywa, że użytkownik jest zalogowany do sesji, to nie jest wymagane ponowne jego logowanie - jest automatycznie przenoszony do aplikacji.

- **AdministrationController** - w klasie tej wystawiane jest API służące potwierdzaniu lub odrzucaniu nowych użytkowników.

- **EmailServiceImpl** - klasa odpowiadająca za implementację usługi mailowej aplikacji. Korzysta z *MimeMessage*. Istnieją dwa scenariusze użycia tej usługi:

  - Po rejestracji nowego użytkownika do wszystkich administratorów wysyłana jest wiadomość e-mail z informacją o tym zdarzeniu oraz linkiem aktywacyjnym.
  - Po wprowadzeniu loginu użytkownika w panelu przypominania hasła, pod adres e-mail powiązany z tym loginem zostaje wysyłane nowe, wygenerowane losowo hasło.

Warstwa bezpieczeństwa aplikacji korzysta również z modułu **utils**. Znajdują się w nim narzędzia pomocnicze:

- Klasa **PasswordUtils** - zawiera narzędzia takie jak walidacja wprowadzonego hasła zgodnie z założonymi wymaganiami czy też sprawdzenie pakietu danych pobranych podczas zmiany hasła.

- Klasa **PasswordGenerator** - odpowiada za generowanie tymczasowego hasła. Pozwala ona na generowanie hasła dostosowanego do wymagań bezpieczeństwa (wielkie oraz małe litery, cyfry). Przykład użycia klasy PasswordGenerator znajduje się w *PasswordUtils.generateTemporaryPassword()*. Wygenerowane hasło jest wysyłane w wiadomości e-mail do użytkownika, którego login został podany w panelu przypominania hasła.

5.4 Tworzenie kopii zapasowej danych

Aplikacja posiada system tworzenia kopii zapasowych danych przechowywanych w bazie (**backup**). Implementacja tego mechanizmu znajduje się w module *utils*. Backup jest tworzony przy pomocy klas *ScheduledTasks* oraz *BackupPostgresql*.

Pierwsza z wymienionych klas odpowiada za wykonywanie czynności zaraz po uruchomieniu aplikacji (initialDelay = 0) oraz cyklicznie co godzinę pracy (fixedRate=1000*60*60).
Implementacja tworzenia backupu znajduje się w klasie BackupPostgresql. Korzysta ona z pliku konfiguracyjnego aplikacji *dbconfig.yaml* w którym:

- *backupLocalization* to ścieżka do katalogu w którym mają być tworzone pliki z kopią zapasową bazy danych.
- *pg_dumpLocalization* to ścieżka do narzędzia **pg_dump.exe** przy pomocy którego tworzony jest backup bazy danych.

Oba pola muszą być poprawnie podane aby tworzenie kopii zapasowej nie działało. W przypadku nie podania, lub błędnego podania którejś ze ścieżek kopia zapasowa nie będzie tworzona. Do odczytania pliku *dbconfig.yaml* wykorzystywana jest klasa **DbConfigService**.
Tworzenie kopii zapasowej bazy danych polega na uruchomieniu zewnętrznego procesu z poziomu aplikacji - `PostgreSQL\11\bin\pg_dump.exe` wraz z odpowiednimi argumentami wywołania. Jest to oficjalne narzędzie do tworzenia backupu bazy danych w PostgreSQL i powinno instalować się automatycznie podczas instalacji PostgreSQL. Więcej o pg_dump.exe można znaleźć pod adresem: `https://www.postgresql.org/docs/devel/app-pgdump.html`
Wszystkie niezbędne informacje do uruchomienia procesu tworzącego kopię zapasową bazy danych znajdują się w pliku *dbconfig.yaml*, który jest odczytywany podczas tworzenia obiektu klasy BackupPostgresql. Pliki z kopią zapasową zapisywane są we wskazanym w *backupLocalization* katalogu w plikach o nazwach tworzonych według formatu *backupddMMyyyy_HHmmss* - odpowiednio data i godzina wykonania kopii zapasowej - w formacie .sql. Wczytywanie kopii zapasowej z poziomu aplikacji nie jest wspierane. Więcej informacji na temat wczytywania kopii zapasowej można znaleźć pod adresem: `http://www.postgresqltutorial.com/postgresql-restore-database/`

5.5 Tworzenie historii operacji na bazie danych

Aplikacja zawiera moduł odpowiadający za zbieranie historii operacji na bazie danych. Za każdym razem, gdy wykonywana jest jedna z określonych operacji bazodanowych, do specjalnej tabeli w bazie danych zapisywane są najważniejsze informacje o danej operacji.

6 Testy

W katalogu `src\test` znajdują się napisane dla aplikacji **testy jednostkowe**. Testują one komponenty aplikacji napisane w języku Java. Przygotowane zostały za pomocą narzędzie **JUnit** oraz frameworka **Mockito**. Testy pokrywają funkcjonalności takie, jak: generowanie haseł czy napisane na potrzeby projektu adnotacje służące do walidacji danych.

7 Lista możliwych rozszerzeń i poprawek

Poniżej lista proponowanych przez nas rozszerzeń i poprawek w aplikacji, które ze względu na ograniczony czas trwania projektu nie zostały przez nas wprowadzone:

- zmiana sposobu działania modułu odpowiedzialnego za historię operacji na bazie danych - powinien on być zbudowany na bazie **AOP (programowanie aspektowe)**
- dodanie mechanizmu przywracania bazy danych za pomocą pliku kopii zapasowej
- przeniesienie logiki biznesowej z klas typu controller do service - dotyczy to m.in. domen i rekordów
- dodanie brakującej logiki dla niektórych tabel, np. *tsigkeys*
- zwiększenie restrykcyjności walidacji (np dokładniejsza analiza nazw domen)
- przeprowadzenie migracji części frontendu wykonanej w technologii JSP do Reacta
- lepszy podział komponentu *ReusableTable*
- eliminacja niewykrytych do tej pory błędów w logice tabeli
- dodanie obsługi mechanizmu *Smart Copy* dla stopki i filtrów w tabeli
- dodanie możliwości filtrowania zarówno po starej, jak i nowej wartości w wierszu (aktualnie działa tylko dla starej).
- wdrożenie obsługi *DNSSEC* do projektu
- wdrożenie systemu replikacji za pomocą *Slony*

- wdrożenie frameworka do testów przeprowadzanych po stronie React'a
- zwiększenie pokrycia kodu testami