**Bezpieczeństwo Systemów Komputerowych**

**Sprawozdanie z zajęć projektowych**

Autorzy:

Dorota Tomczak, 165145

Karolina Makuch 165276

Wersja dokumentu: 2.0

**Historia zmian**

|  |  |  |
| --- | --- | --- |
| Wersja | Data | Opis zmian |
| 1.0 | 23.04.19 | Utworzenie dokumentu, dodanie źródeł |
| 1.1 | 12.05.19 | Dodanie interfejsów użytkownika, wyników testów (projekt 1) |
| 1.2 | 13.05.19 | Dodanie charakterystyki zadania i wniosków (projekt 1) |
| 1.3 | 19.05.19 | Dodanie opisu implementacji (projekt 1) |
| 2.0 | 10.06.19 | Dodanie opisu interfejsu użytkownika(projekt 2) |
| 2.1 | 11.06.19 | Uzupełnienie opisu zadania, bazy danych, testów oraz wniosków (projekt 2) |

1. **Projekt 1 – Szyfrowanie plików wraz z przekazaniem klucza sesyjnego**
   1. ***Charakterystyka zadania projektowego***

Zadanie projektowe obejmowało utworzenie dwóch aplikacji: serwerowej do szyfrowania plików oraz klienckiej do ich deszyfracji. Aplikacja serwerowa oczekuje na połączenia od klientów, a po jego nawiązaniu generuje klucz sesyjny, szyfruje plik a następnie przesyła go do klienta. Wyboru pliku można dokonać w każdym momencie działania aplikacji.

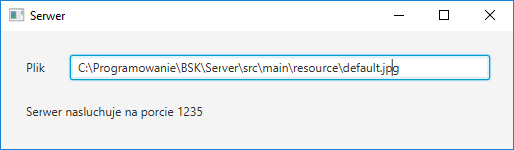
Aplikacja kliencka jest uruchamiana na maszynie wirtualnej. Umożliwia logowanie i rejestrację w celu obsługi wielu użytkowników. Po rejestracji tworzone są klucze RSA – publiczny oraz prywatny – które zostają następnie zapisane na dysku w oddzielnych folderach. Aby móc odbierać zaszyfrowane i deszyfrować pliki należy się najpierw zalogować. Następnie klient może wysłać do serwera żądanie otrzymania zaszyfrowanego pliku (uprzednio wypełniając wszystkie niezbędne parametry potrzebne do szyfrowania). Po nawiązaniu połączenia wysyłany jest klucz publiczny klienta i inne dane potrzebne do szyfrowania. Serwer zwraca klientowi zaszyfrowany klucz sesyjny wraz z innymi danymi potrzebnymi do deszyfracji. Następnie wysyła plik, który jest zapisywany na dysku klienta. Deszyfracji dokonuje się wybierając plik z dysku. Po wybraniu niezaszyfrowanego pliku lub nieautoryzowanej próby deszyfracji aplikacja o tym nie informuje.

Wybranym algorytmem blokowym do szyfrowania plików jest algorytm Blowfish.

* 1. ***Projekt interfejsów użytkownika***

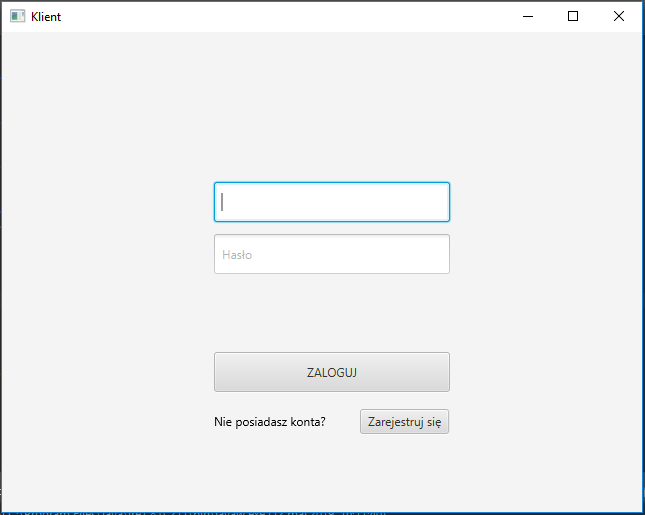
Projekt interfejsów użytkownika wykonano z wykorzystaniem JavaFX oraz narzędzia wspomagającego SceneBuilder. Screenshoty zostały wykonane na urządzeniu z systemem Windows.

W aplikacji serwerowej zastosowano jeden ekran, na którym wyświetlane są dwie podstawowe informacje: ścieżka do pliku aktualnie wybranego do szyfrowania oraz informacja w jakim obecnie stanie znajduje się serwer (oczekiwanie na połączenie, szyfrowanie pliku, wysyłanie). Po kliknięciu na pole z nazwą pliku otwiera się menadżer plików, przez który można dokonać zmiany pliku przeznaczonego do szyfrowania.

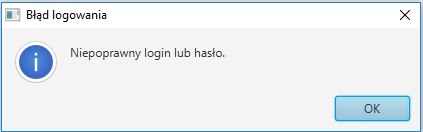


*Okno aplikacji serwerowej*

W aplikacji klienckiej obecne są 4 ekrany: logowania, rejestracji, szyfrowania i deszyfrowania. Ekranem startowym jest ekran przeznaczony do logowania. Po wpisaniu poprawnego loginu i hasła aplikacja przekierowuje do ekranu szyfrowania. W przeciwnym wypadku wyświetla się komunikat o nieprawidłowych danych. Po kliknięciu w przycisk „Zarejestruj się” aplikacja przekierowuje do ekranu rejestracji.

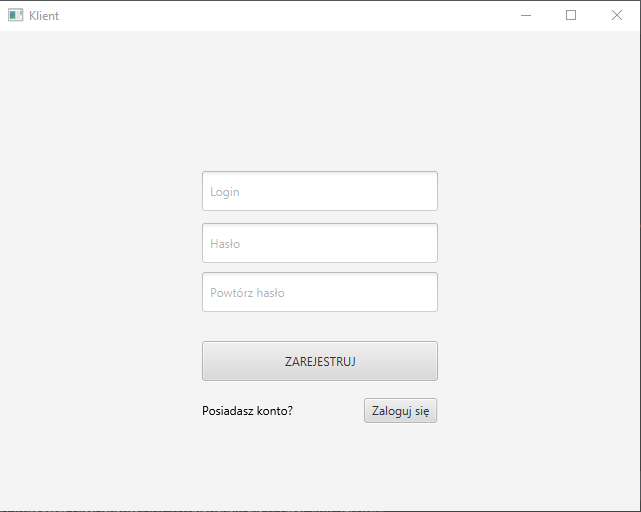


*Ekran logowania aplikacji klienckiej*



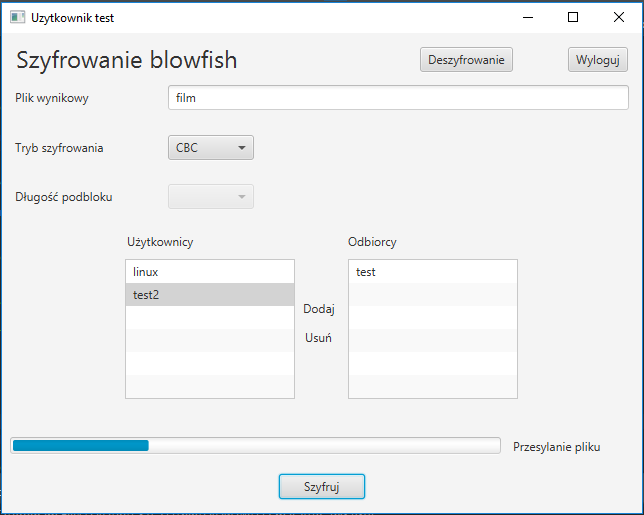
*Komunikat o błędzie logowania*

Na ekranie rejestracji widoczne są 3 pola, które trzeba uzupełnić, aby móc dokonać rejestracji. Pierwsze musi zawierać unikalną nazwę użytkownika, a drugie i trzecie takie same łańcuchy znaków. Po udanej rejestracji aplikacja przekierowuje do ekranu logowania (tak samo po wciśnięciu przycisku „Zaloguj się”). W przypadku nieudanej rejestracji wyświetla się stosowny komunikat.



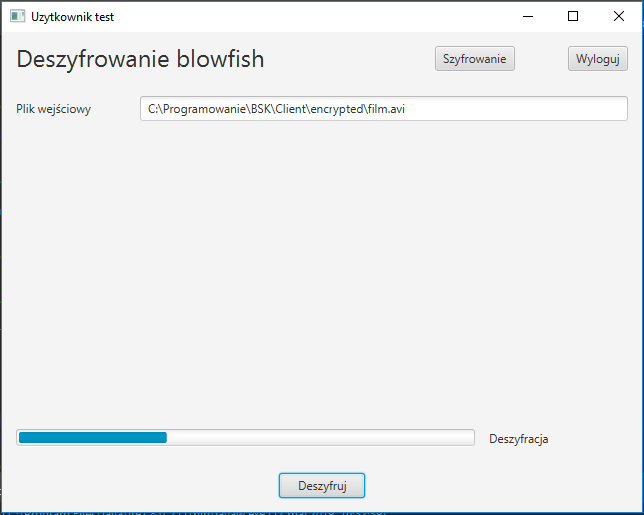
*Ekran rejestracji aplikacji klienckiej*

Ekran szyfrowania umożliwia: wpisanie nazwy, którą będzie miał po odszyfrowaniu odebrany plik; wybranie trybu szyfrowania (ECB, CBC, CFB, OFB) z rozwijanej listy; wybranie długości podbloku dla trybów CFB i OFB z rozwijanej listy (8, 16, 24, 32, 64); wybranie użytkowników, który będą mogli odszyfrować odebrany plik. Pasek postępu umożliwia obserwację postępu przesyłania pliku z aplikacji serwerowej. Szyfrowanie następuje po naciśnięciu przycisku „Szyfruj”. Dwa przyciski w prawym górnym rogu – „Deszyfrowanie” i „Wyloguj” – odpowiadają odpowiednio za: przejście do ekranu deszyfrowania i wylogowania z aplikacji (przejście do ekranu logowania). Nazwa okna aplikacja przedstawia login obecnie zalogowanego użytkownika (tak samo na ekranie deszyfrowania).

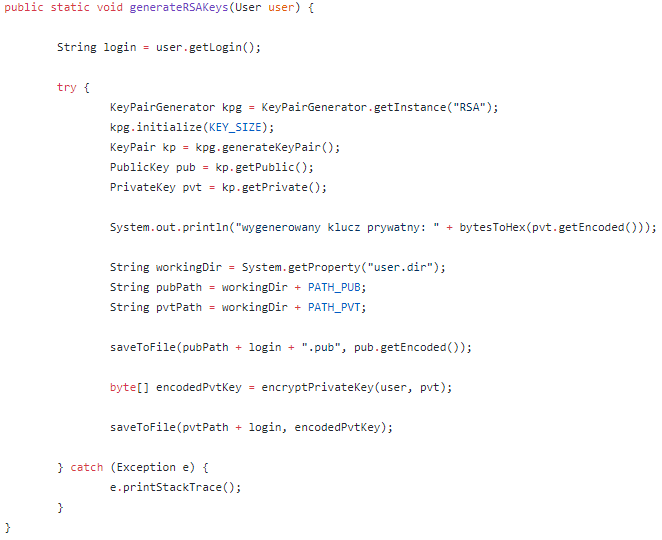
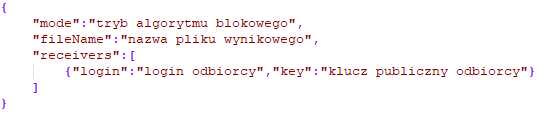
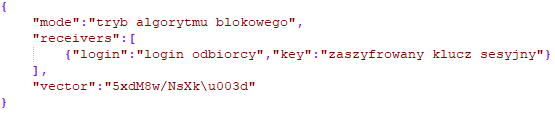
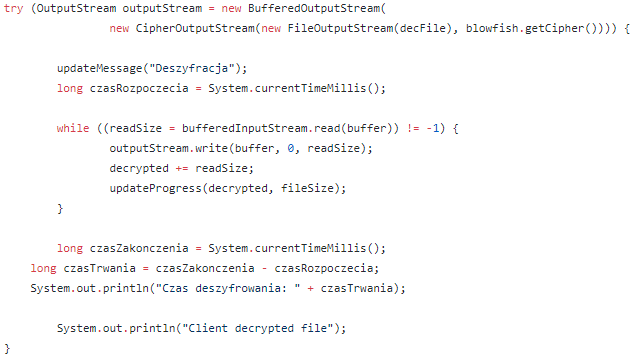
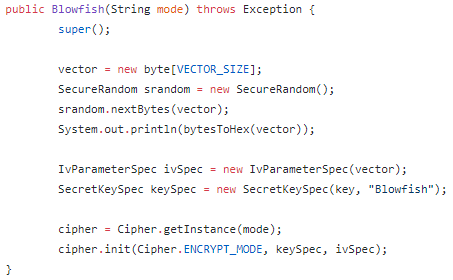
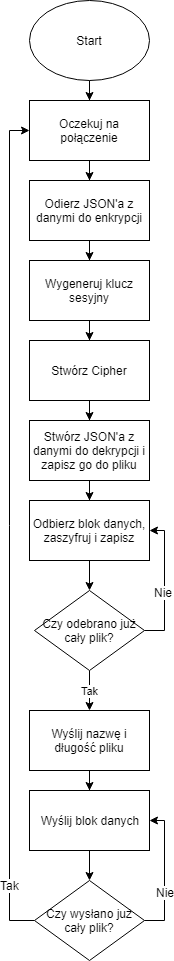
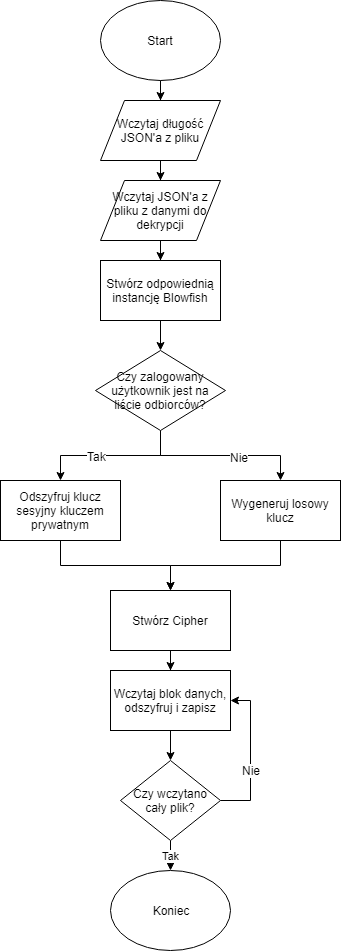


*Ekran szyfrowania*

Ekran deszyfrowania umożliwia wybranie pliku przeznaczonego do deszyfracji po naciśnięciu na pole obok etykiety „Plik wejściowy”. Przycisk „Deszyfruj” rozpoczyna deszyfrację. Pasek postępu wskazuje na postęp deszyfracji. Dwa przyciski w prawym górnym rogu – „Szyfrowanie” i „Wyloguj” – odpowiadają odpowiednio za: przejście do ekranu szyfrowania i wylogowania z aplikacji (przejście do ekranu logowania).



*Ekran deszyfrowania*

* 1. ***Opis implementacji zadanej funkcjonalności***
     1. ***Ogólne***Obie aplikacje zostały napisane w języku Java (SE 1.8) w środowisku Eclipse. Oprócz standardowych bibliotek wykorzystano również bibliotekę gson w wersji 2.8.5 oraz mssql-jdbc w wersji 7.2.0.
     2. ***Hashowanie hasła***Skrót hasła zostaje utworzony podczas rejestracji, a następnie zapisany w bazie (poprzedzony wygenerowaną solą o długości 44 znaków). Przy próbach logowania skrót hasła i sól są pobierane z bazy, skrót wpisanego hasła zostaje wygenerowany z tą samą solą, a na końcu oba skróty są ze sobą porównywane w celu uwierzytelnienia. Wszystkie operacje związane z funkcją skrótu zawierają się w klasie PasswordUtils aplikacji klienckiej.   
        Sól to po prostu ciąg losowo wygenerowanych bajtów metodą nextBytes() klasy SecureRandom. Skróty haseł tworzone są poprzez algorytm PBKDF2WithHmacSHA512.
     3. ***Tworzenie kluczy RSA***Klucze RSA tworzone są po udanej rejestracji nowego użytkownika. Dzieje się to w metodach klasy RSAKeysUtils aplikacji klienckiej. Obiekt klasy KeyPairGenerator generuje oba klucze. Następnie są one zapisywane na dysku w dwóch oddzielnych folderach w plikach o nazwie użytkownika, dla którego zostały stworzone. Klucz prywatny jest przed zapisem szyfrowany algorytmem Blowfish w trybie CBC. Kluczem użytym do szyfrowania jest skrót hasła użytkownika, które podał przy rejestracji.  
          
        
     4. ***Protokół komunikacyjny***Do komunikacji między maszynami zastosowano protokół sieciowy TCP.   
        1. Serwer oczekuje na połączenie na porcie 1235. Gdy klient nawiąże z nim połączenie wysyła dane potrzebne do szyfrowania w formacie JSON.  
          
        2. Serwer odbiera dane i na ich podstawie dokonuje szyfrowania pliku. Na początek pliku zapisuje dane potrzebne do dekrypcji. Następnie wysyła do klienta nazwę pod którą ma zostać zapisany zaszyfrowany plik.  
          
        3. Serwer wysyła do klienta plik w paczkach o wielkości 4096 bajtów.  
        4. Klient odbiera plik i zapisuje go na dysku.
     5. ***Klucz sesyjny***Klucz sesyjny tworzony jest po nawiązaniu połączenia z serwerem przez klienta. Użyto do tego obiektu klasy SecureRandom, który uzyskano wywołując metodę getInstanceStrong(), która zwraca możliwie najbezpieczniejszą dostępną instancję tej klasy. Jako wartość początkowa generatora podano aktualny czas systemowy, a następnie wykonano na nim metodę nextBytes(). Uzyskane w ten sposób klucze mają stałą długość 16 bajtów. Są szyfrowane kluczami publicznymi RSA użytkowników, którzy nawiązali połączenie z serwerem.
     6. ***Dekrypcja***Najpierw ze wskazanego pliku pobierany jest int, który określa liczbę bajtów zapisanego dalej JSON-a. Następnie wczytywany jest JSON, który zawiera dane potrzebne do deszyfracji. Znajdują się tam informacje o trybie szyfrowania, odbiorcach oraz ew. wektorze.  
        Zostaje utworzona odpowiednia instancja klasa Blowfish dzięki statycznej metodzie getBlowfish() klasy BlowfishBase. Dla trybu ECB zostanie zwrócony obiekt klasy BlowfishECB, a w przeciwnym wypadku po prostu Blowfish.  
        W konstruktorze BlowfishBase następuje sprawdzenie, czy zalogowany użytkownik jest na liście odbiorców. Jeśli tak zaszyfrowany klucz sesyjny jest odszyfrowywany jego kluczem prywatnym. W przeciwnym wypadku generowany jest losowy klucz.  
        Dla trybów poza ECB wykorzystywany jest wektor z otrzymanego JSONA. Następnie tworzony jest obiekt klasy Cipher i zostaje zainicjowany w trybie DECRYPT\_MODE, aby posłużył do deszyfrowania.  
        Plik jest wczytywany w paczkach po 4096 bajtów, deszyfrowany i zapisywany do pliku wynikowego w folderze decrypted.  
        
     7. ***Enkrypcja***Analogicznie do dekrypcji z tą różnicą, że wektor jest tworzony, a nie odczytywano z JSON-a oraz klucz sesyjny również. Cipher jest inicjowany w trybie ENCRYPT\_MODE.  
          
        
     8. ***Algorytmiczny schemat działania serwera (ServerTask)***
     9. ***Algorytmiczny schemat działania klienta (DecryptTask)***  
        ******
  2. ***Wyniki przeprowadzonych testów***

Pomiarów dokonano korzystając z metody System.currentTimeMillis() odejmując czas startu od czasu zakończenia.

Środowisko testowe

|  |  |
| --- | --- |
| Host | Maszyna wirtualna |
| Procesor: Intel Core i3-311M CPU 2x 2.40GHz  RAM: 8GB  System: 64-bit Windows 10  Dysk SSD: Adata SU650 | RAM: 4 GB  System: 64-bit Linux Mint Mate 19 |

Operacje szyfrowania były wykonywane na hoście, deszyfrowanie na maszynie wirtualnej. Pliki były przesyłane z hosta do maszyny.

Plik 737KB

|  |  |  |  |
| --- | --- | --- | --- |
| Tryb | Czas szyfrowania [ms] | Czas przesyłania [ms] | Czas deszyfrowania [ms] |
| ECB | 49 | 838 | 1739 |
| CBC | 46 | 927 | 1616 |
| CFB8 | 130 | 580 | 2792 |
| CFB64 | 34 | 873 | 1917 |
| OFB8 | 159 | 571 | 2379 |
| OFB64 | 45 | 1009 | 2281 |

Plik 14,2 MB

|  |  |  |  |
| --- | --- | --- | --- |
| Tryb | Czas szyfrowania [ms] | Czas przesyłania [ms] | Czas deszyfrowania [ms] |
| ECB | 365 | 20220 | 16670 |
| CBC | 406 | 20234 | 17464 |
| CFB8 | 2055 | 15226 | 25754 |
| CFB64 | 390 | 15020 | 16258 |
| OFB8 | 2143 | 13987 | 27236 |
| OFB64 | 402 | 16469 | 16869 |

Plik 128 MB

|  |  |  |  |
| --- | --- | --- | --- |
| Tryb | Czas szyfrowania [ms] | Czas przesyłania [ms] | Czas deszyfrowania [ms] |
| ECB | 2878 | 128185 | 120490 |
| CBC | 3020 | 127570 | 121343 |
| CFB8 | 18217 | 128101 | 202887 |
| CFB64 | 3378 | 126879 | 122989 |
| OFB8 | 19186 | 123472 | 238570 |
| OFB64 | 3626 | 124191 | 123965 |

* 1. ***Wnioski***

Pierwsza obserwacja: im większy rozmiaru pliku tym dłuższe czasy szyfrowania, przesyłania i deszyfracji. Powód jest oczywisty: im większa ilość danych do przetworzenia, tym więcej operacji trzeba wykonać.

Czasy szyfrowania są kilkukrotnie krótsze od czasów deszyfracji, co wynika z faktu przeprowadzenia deszyfracji na maszynie wirtualnej. Opóźnienia są przez to nawet kilkukrotne.

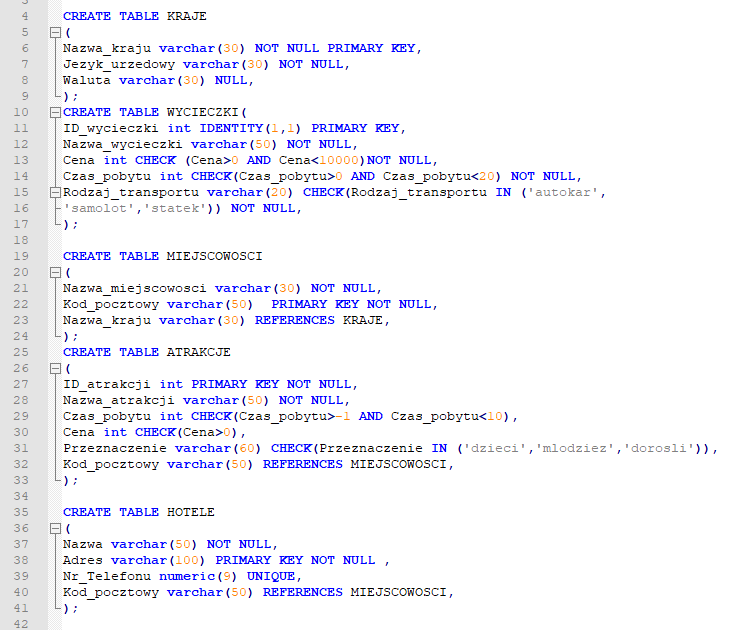
Im mniejsza długość podbloku w trybach CDB i OFB tym dłuższe czasy szyfrowania i deszyfrowania. Jest tak ponieważ dane są szyfrowane w jednostkach mniejszych niż rozmiar bloku (przetestowano dla 8 i 64), więc ponownie wiąże się to z większą liczbą operacji do wykonania. Dla podbloków o długości 64 bitów osiągane czasy dorównują trybom CBC i ECB lub są od nich lepsze.

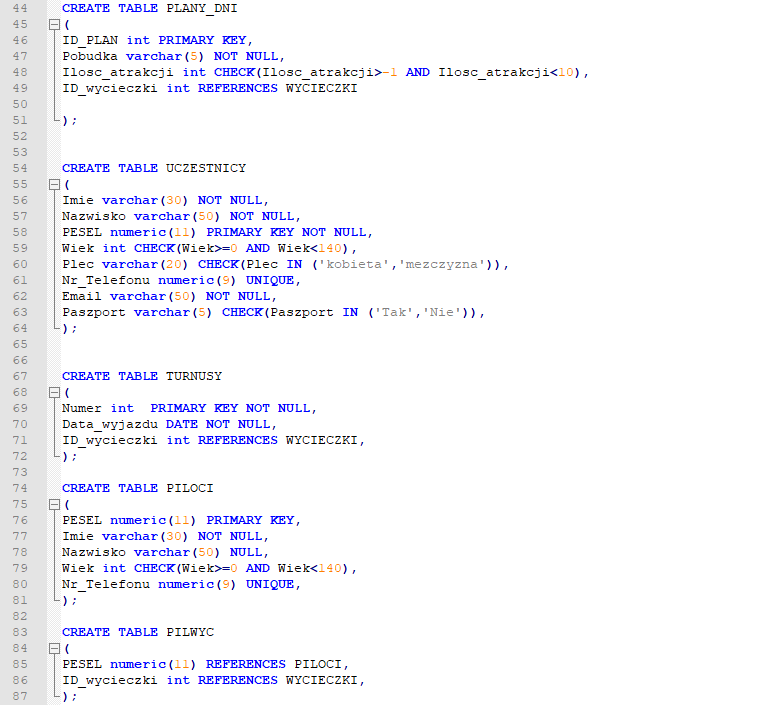
1. **Projekt 2 – Implementacja mechanizmów kontroli dostępu do baz danych**
   1. ***Charakterystyka zadania projektowego***

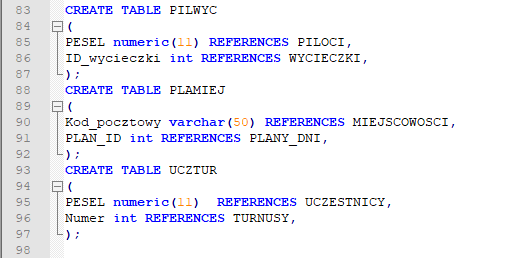
Głównym celem zadania projektowego było zaprojektowanie i implementacja wybranego na początku zajęć modelu autoryzacji i kontroli dostępu w wytworzonej bazie danych. Należało również zaimplementować podstawowe operacje SQL na obiektach oraz umożliwić dostęp do bazy danych z poziomu przeglądarki internetowej wraz z tunelem SSL.

* 1. ***Projekt struktury bazy danych***

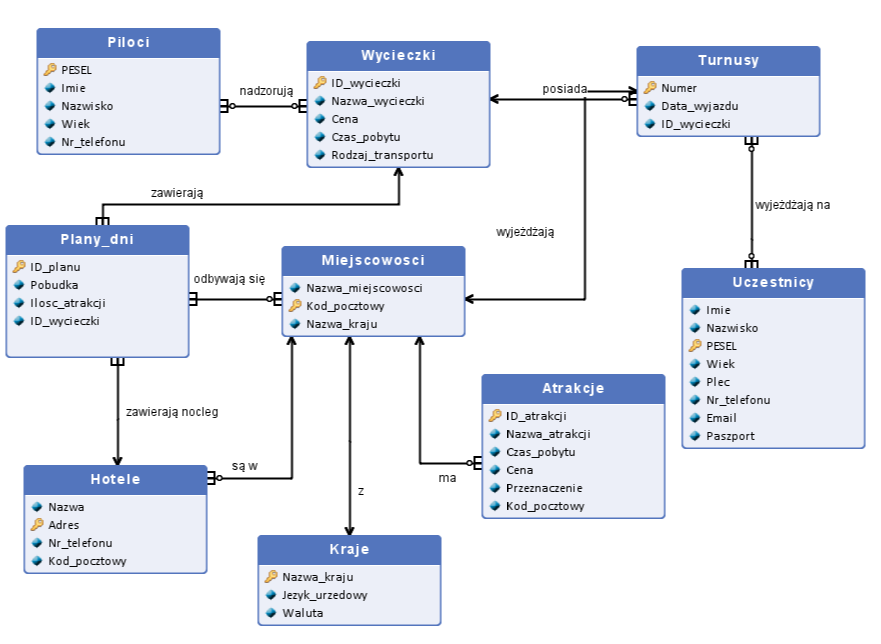
Relacyjna baza danych została zaprojektowana dla biura podróży. Zawiera ona dziewięć encji. Została utworzona na SQL Serverze za pośrednictwem programu SQL Management Studio.



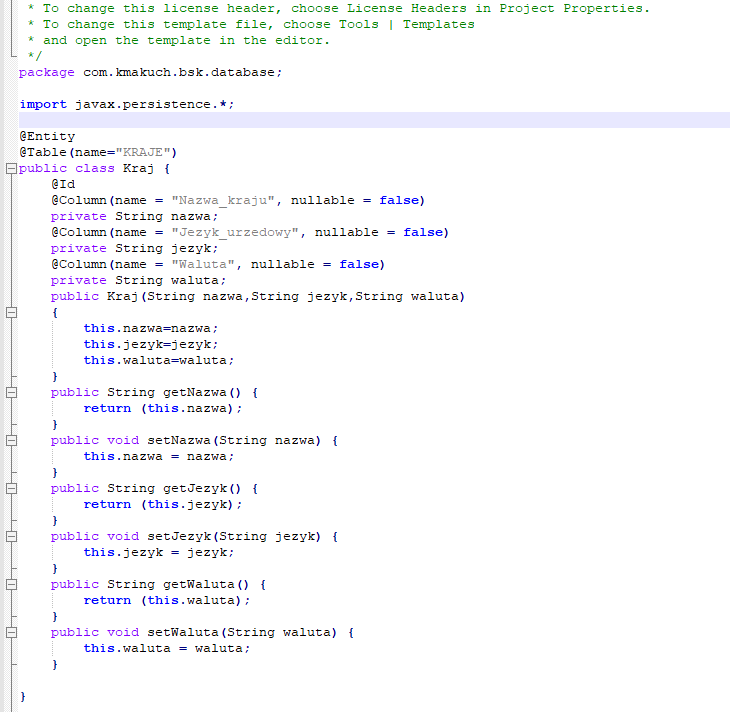




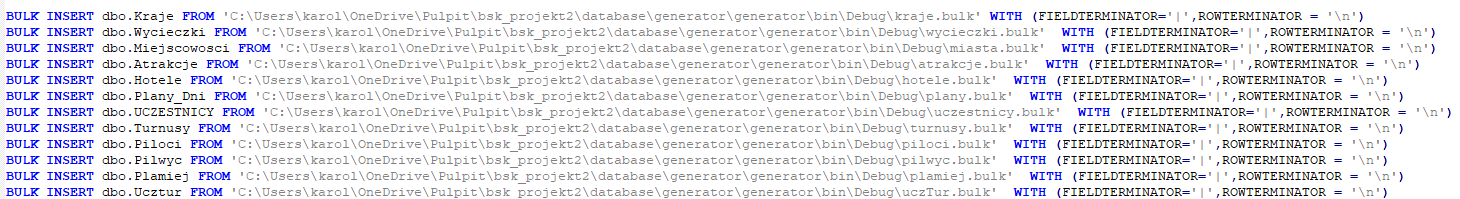
Został również przygotowany diagram ERD:



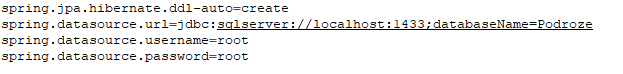
Każda encja posiada swój odpowiednik w postaci klasy. Poszczególne adnotacje zostały zimportowane z *javax.persistence.\*3*



Inicjalnie baza została wypełniona danymi wytworzonymi wcześniej przez napisany przez nas generator.



Niestety próba połączenia aplikacji z bazą danych nie powiodła się. W celu uzyskania połączenia zostało wykorzystane dependency spingboota4 (odpowienie liniki w pom.xml):



Przygotowanie dostępu użytkowników do bazy danych zostało wykonane na podstawie modelu RBAC. Zostały utworzone dwie role: administratora oraz pilota wycieczki.

Administrator otrzymał przywileje zapisywania oraz usuwania danych z poszczególnych tabel, zaś pilot wycieczki otrzymał pierwszy z nich. Przywileje zostały zapisane w formie kolekcji o typie Privilege.



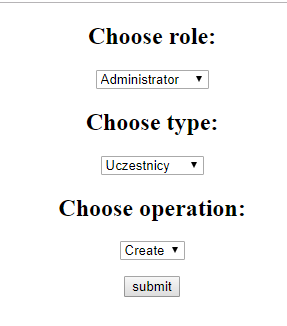
Użytkownik oprócz przypisanej roli posiada id oraz name.



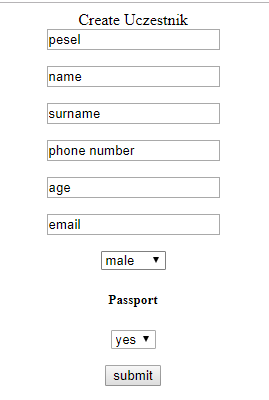
* 1. ***Projekt interfejsu użytkownika***

Interfejs użytkownika został napisany w języku HTML. Zrzuty ekranu zostały wykonane za pomocą programu Lightshot w przeglądarce internetowej Chrome na systemie operacyjnym Windows.

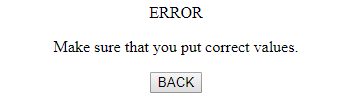
Początkowa strona zawiera trzy selecty. Pierwszy z nich odpowiada roli użytkownika ( poszczególna rola ogranicza możliwość operacji na bazie danych). Kolejny z nich odpowiada encjom. Umożliwia użytkownikowi dodanie danych do wybranej tabeli. Ostatni odpowiada za wybór operacji, która ma zostać wykonana.



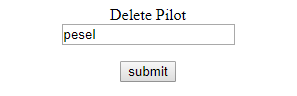
Po wybraniu opcji create oraz naciśnięciu przycisku submit, użytkownik zostaje przekierowany do kreatora. Typy enum zostały przedstawione w formie selectów.



W przypadku wpisania stringa przez użytkownika w inputa, który powinien zawierać integera po naciśnięciu przycisku submit pojawi się komunikat o błędzie.



Po wybraniu przez użytkownika operacji delete pojawia się input umożliwiający wpisanie klucza głównego danej, która ma zostać usunięta.



Gdy użytkownik nie posiada przywileju do usuwania danych z tabeli, zostaje automatycznie przekserowany do strony początkowej.

* 1. ***Wyniki testów***

Z racji problemów z połączeniem z bazą danych testy nie zostały przeprowadzone.

* 1. ***Wnioski***

Problemy z połączeniem z docelową bazą danych uniemożliwiły wykonanie kluczowych obserwacji. W końcowej fazie projektu pojawiał się błąd wymagający @Bean o nazwie entityManager. Niestety posiadana przez nas wiedza, poszukiwanie rozwiązania w internecie nie pomogło rozwiązać tego problemu.

1. **Literatura**

[1] <https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html>, (02.03.19).

[2] <https://docs.oracle.com/cd/B14099_19/idmanage.1012/b15565/oracle/security/crypto/core/Blowfish.html>, (02.03.19).

[3] <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html> (06.06.19)

[4] <https://spring.io/projects/spring-boot> (06.06.19)