
DOKUMENTACJA TECHNICZNA I OPIS PROTOKOŁÓW

Projekt z Sieci Komputerowych i Bezpieczeństwa

# Wyszukiwarka połączeń

*Autor:*
Krzysztof Barański

*Nadzorujący:*
dr Edward Szczypka

Styczeń 2015

## Spis treści

# 1 Ogólny opis projektu

## 1.1 Opis projektu

### 1.1.1 Cel

Celem projektu jest stworzenie oprogramowania do planowania podróży. Aplikacja w założeniu przeznaczona jest do planowania podróży na całym świecie używając wielu środków transportu takich jak: autobusy, pociągi, samoloty i inne.

### 1.1.2 Aktualna wersja

Obecnie aplikacja przeznaczona jest do wyszukiwania połączeń między kilkudziesięcioma największymi miastami w Polsce za pomocą dowolnego środka transportu. W obecnej wersji aplikacja pozwala wyznaczyć trasę z miasta A do miasta B, uwzględniając odległość i czas podróży, a także rozkłady jazdy dostarczane przez przewoźników.

### 1.1.3 Planowane rozszerzenia

W przyszłych wersjach, aplikacja miałaby uwzględniać także ceny biletów, oraz inne aspekty ważne przy planowaniu podróży. Oczywiście zostałaby rozszerzona baza miast. Zostałoby również dodane logowanie użytkowników, które umożliwiłoby przechowywanie powtarzających się danych takich jak np. adres czy numer telefonu.

## 1.2 Struktura projektu

### 1.2.1 Cel

Celem strukturalnym jest stworzenie aplikacji, która będzie zainstalowana bądź dostępna poprzez serwis WWW dla klientów i pobierająca od niego dane potrzebne do jak najlepszego zaplanowania jego podróży. Aplikacja kliencka będzie te dane wysyłać do jednego z wielu serwerów, które będą zajmowały się wyznaczaniem trasy. Każdy z serwerów będzie komunikował się z zarejestrowanymi przewoźnikami-serwerami posiadającymi bazę danych z ich rozkładem jazdy. Następnie wyznaczoną trasę serwer będzie wysyłał jako odpowiedź do klienta.

### 1.2.2 Aktualna wersja

W aktualnej wersji serwery nie komunikują się ze sobą, przez co gdy jeden jest przeciążony nie może zlecić zadania innemu serwerowi. Zapytania do serwerów są wysyłane z aplikacji zainstalowanych na komputerach klientów.

### 1.2.3 Planowane rozszerzenia

W przyszłych wersjach zostałby dodany serwis WWW obsługujący zapytania klientów, a także wraz z rozszerzaniem liczby miast zostałoby dodanych więcej serwerów obsługujących zapytania oraz komunikacji między nimi, w celu przyspieszenia czasu przesyłanych danych oraz możliwość obsługi większej liczby klientów.

# 2 Ogólna analiza wymagań stawianych aplikacji

## 2.1 Analiza problemów

### 2.1.1 Wymagania funkcjonalne

- Przesyłanie danych
- Weryfikacja danych
- Przechowywanie i zarządzanie danymi
- Prezentacja danych

### 2.1.2 Wymagania pozafunkcjonalne

Bezpieczeństwo przesyłu:

- poufność
- integralność
- dostępność

Dane odebrane przez serwer muszą zostać sprawdzone pod kątem potencjalnego ataku - SQL-injection oraz Cross Scripting

Zapytania do serwera powinny być wykonywane w osobnych wątkach, by skrócić czas wykonywania operacji i oczekiwania na odpowiedź.

## 2.2 Rozwiązania problemów

Poufność i integralność przesyłu danych jest zapewniona dzięki szyfrowaniu połączeń za pomocą klucza RSA. Zapobieganie atakom m. in. typu Cross Scripting, SQL-injection oraz DDoS jest zapewnione dzięki dokładnemu sprawdzaniu wprowadzonych oraz przesyłanych danych, pozwolenie używania tylko określonych znaków oraz wprowadzenie ograniczeń na czas i liczbę wykonywanych zapytań.
Dostępność jest zapewniona dzięki wielu serwerom, działającym niezależnie, więc w wyniku awarii jednego systemu, można udać się do innego.

# 3 Opis używanych protokołów

## 3.1 Protokół client-server

Tekstowy protokół warstwy aplikacyjnej służący do przesyłania klientowi wyznaczonej przez serwer trasy podróż za pomocą bezpiecznego połączenia SSL.

### 3.1.1 Zapytanie

Zapytanie do serwera składa się z czterech linii:

- FROM_CITY
  - miasto początkowe. Dozwolone znaki to: litery alfabetu angielskiego, pauza oraz spacja.

- `TO_CITY`
  - miasto docelowe. Dozwolone znaki to: litery alfabetu angielskiego, pauza oraz spacja.
- `DATE`
  - data początku podróży. Format: YYYYMMDD. Dozwolone tylko cyfry.
- `TIME`
  - godzina początku podróży. Format: YYYYMMDD. Dozwolone tylko cyfry.

### 3.1.2 Odpowiedź

Odpowiedź serwera składa się z różnej liczby linii w zależności od wyniku zapytania.
Pierwszą linią jest zawsze kod wyniku zapytania. Możliwe kody to:

- `BADFORMAT` - zły format jakiejś linii w zapytaniu
- `BADQUERY` - niepoprawne zapytanie
- `IOERROR` - problem dostępem do bazy, pliku itp.
- `NOCONNECTION` - brak połączenia między podanymi miastami
- `NOTFOUND` - trasa nie została znaleziona
- `NOTEXISTS` - podane miasto nie istnieje w bazie
- `OK` - zapytanie poprawne i znaleziono trasę
- `TIMEOUT` - przekroczony czas zapytania
- `UNKNOWNERROR` - nieznany błąd

Jeżeli kodem wyniku jest `OK` to kolejna linia zawiera liczbę oznaczającą liczbę odcinków podróży. Następnie w kolejnych liniach znajduje się opis tych odcinków. Opis jednego odcinka składa się z następujących linii:

- `CARRIER_NAME`
  - nazwa przewoźnika dla danego odcinka trasy
- `FROM_CITY`
  - miasto początkowe odcinka
- `TO_CITY`
  - miasto docelowe odcinka
- `START_DATE`
  - data odjazdu w formacie YYYYMMDD
- `START_TIME`
  - godzina odjazdu w formacie HHMM
- `STOP_DATE`
  - data przyjazdu w formacie YYYYMMDD
- `STOP_TIME`
  - godzina przyjazdu w formacie HHMM

### 3.1.3 Implementacja w projekcie

Protokół jest zaimplementowany w podprojektach **client** i **server**. Po stronie klienta, klasą implementującą protokół jest klasa *QueryProtocol*, a po stronie serwera *QueryHandler*. Do komunikacji wykorzystano java'owy *SSLSocket*.

## 3.2 Protokół client-server

Tekstowy protokół warstwy aplikacyjnej służący do przesyłania serwerowi odpowiedzi na zapytania o rozkład jazdy za pomocą bezpiecznego połączenia SSL.

### 3.2.1 Zapytanie

Zapytanie do aplikacji przewoźnika składa się z czterech linii:

- `FROM_CITY`
  - miasto początkowe. Dozwolone znaki to: litery alfabetu angielskiego, pauza oraz spacja.
- `TO_CITY`
  - miasto docelowe. Dozwolone znaki to: litery alfabetu angielskiego, pauza oraz spacja.
- `DATE`
  - data początku podróży. Format: YYYYMMDD. Dozwolone tylko cyfry.
- `TIME`
  - godzina początku podróży. Format: YYYYMMDD. Dozwolone tylko cyfry.

### 3.2.2 Odpowiedź

Odpowiedź serwera składa się z następujących linii:

- `RESPONSE_CODE`
  - pierwszą linią jest zawsze kod wyniku zapytania. Możliwe kody to:
    - `BADFORMAT` - zły format jakiejś linii w zapytaniu
    - `BADQUERY` - niepoprawne zapytanie
    - `IOERROR` - problem dostępem do bazy, pliku itp.
    - `NOCONNECTION` - brak połączenia między podanymi miastami
    - `NOTFOUND` - trasa nie została znaleziona
    - `NOTEXISTS` - podane miasto nie istnieje w bazie
    - `OK` - zapytanie poprawne i znaleziono trasę
    - `TIMEOUT` - przekroczony czas zapytania
    - `UNKNOWNERROR` - nieznany błąd

  Jeżeli kod wyniku jest różny od `OK` to odpowiedź nie zawiera kolejnych linii.
- `CARRIER_NAME`
  - nazwa przewoźnika, który wysyła odpowiedź

- `FROM_CITY`
  - miasto początkowe
- `TO_CITY`
  - miasto docelowe
- `START_DATE`
  - data odjazdu w formacie YYYYMMDD
- `START_TIME`
  - godzina odjazdu w formacie HHMM
- `STOP_DATE`
  - data przyjazdu w formacie YYYYMMDD
- `STOP_TIME`
  - godzina przyjazdu w formacie HHMM

#### 3.2.3 Implementacja w projekcie

Protokół jest zaimplementowany w podprojektach **server** i **carrier**. Po stronie serwera, klasą implementującą protokół jest klasa *QueryProtocol*, a po stronie przewoźnika *QueryHandler*. Do komunikacji wykorzystano java'owy *SSLSocket*.

# 4 Komunikacja Klient-Serwer

## 4.1 Nawiązywanie i wznawianie połączenia

### 4.1.1 Założenia

Mamy wiele serwerów, który realizują zapytania wielu klientów. Każdy klient ma przydzielony jeden serwer, co oznacza, że każdy serwer ma przydzieloną pewną pulę klientów, których obsługuje. Klient zawsze powinien mieć możliwość wykonywania podstawowych operacji oferowanych przez system. Wymogiem stawianym klientowi przed pierwszym skorzystaniem z naszej aplikacji jest konieczność konfiguracji adresu serwera i portu na którym nasłuchuje.

### 4.1.2 Zagrożenia

Aktualnie, dane wprowadzane przez klienta nie są na tyle poufne, żeby potrzebna była autoryzacja za pomocą loginu i hasła, przez co nie ma zagrożeń związanych z nieautoryzowanym dostępem do danych. Dodatkowo, połączenie między klientem a serwerem jest szyfrowane, więc próba wydobycia danych podczas ich przesyłania jest nie dość, że bardzo utrudniona to jeszcze nie współmierna do zysków.

### 4.1.3 Analiza

Klient jest w stanie korzystać z aplikacji dopóki funkcjonuje jakikolwiek ze znanych mu serwerów - prawdopodobieństwo, że wszystkie jednocześnie przestaną funkcjonować, jest niewielkie.

# 5 Komunikacja Serwer - Baza danych oraz Przewoźnik - Baza danych

## 5.1 Komunikacja

### 5.1.1 Założenia

Baza danych powinna być połączona z serwerem bezpiecznym połączeniem.

### 5.1.2 Zagrożenia

Atak `Man in the middle`, próba nawiązania zewnętrznego połączenia. Istotnym zagrożeniem może się także okazać atak na serwer realizowany za pomocą jednoczesnego wysyłania zapytań od wielu klientów.

### 5.1.3 Rozwiązanie

Baza jest połączona z serwerem połączeniem bezpośrednim, lokalnym. Porty powiązane z bazą są zablokowane dla zewnętrzynych połączeń. Wszystkie próby nawiązania połączenia z zewnątrz oraz utraty połączenia z serwerem są logowane i analizowane. W momencie, kiedy zostaje przekroczony limit aktywnych połączeń serwer-klient, serwer przestaje przyjmować nowe.

### 5.1.4 Analiza

Baza jest zabezpieczona od ataków zewnętrznych. Pozostaje kwestia ataków wewnętrznych.

## 5.2 Zapytania SQL

### 5.2.1 Założenia

Baza danych zagrożona jest też od wewnątrz - potrzebujemy zabezpieczyć ją przed atakami wewnętrzymi.

### 5.2.2 Zagrożenia

Atak `SQLInjection` - zapytania SQL tworzone są na podstawie danych przesłanych przez klientów - mogą być spreparowane tak, by zaatakować bazę od środka.

### 5.2.3 Rozwiązanie

Dane od klienta są szczegółowo analizowane pod kątem niedozwolonych znaków.

### 5.2.4 Analiza

Atak `SQLInjection` jest dobrze znany i bazy danych implementują bardzo skuteczne metody do zapobiegania im. Wystarczy odpowiednio korzystać z tych dostarczonych i przetestowanych funkcjonalności.

# 6 Komunikacja Serwer - Przewoźnik

## 6.1 Analiza

Dane, które przewoźnik przesyła do serwera nie są aktualnie poufne (bowiem rozkład jazdy zazwyczaj jest znany każdemu np. poprzez stronę internetową). System musi być integralny, fałszywy przewoźnik nie powinien móc "podpiąć się" do sieci, nikt nie powinien móc rozsyłać fałszywych danych. W takim razie, każdy serwer zna klucze publiczne wszystkich swoich przewoźników i na odwrót. Każdy wysyłany komunikat jest podpisywany przez nadawcę, a każdy odbierany jest weryfikowany.

# 7 Klient - Aplikacja

## 7.1 Implementacja

Aplikacja jest napisana w Javie 1.8 i budowana oraz konfigurowana przez skrypty napisane w Bashu. Aplikacja kliencka jest aplikacją z interfejsem tekstowym. Do swojego działania wymaga podprojektu *commons*, w którym znajdują się klasy wspólne dla całego projektu.

## 7.2 Klasy

Aplikacja kliencka składa się z następujących klas (wraz z krótkim opisem funkcjonalności):

- `Application` - klasa główna aplikacji, pobieranie danych od użytkownika
- `Config` - klasa opakowująca plik konfiguracyjny
- `QueryProtocol` - klasa służąca do komunikacji z serwerem, implementacja protokołu *client-server*
- `RouteUtils` - klasa będąca mostem pomiędzy, aplikacją a protokołem
- `Utils` - funkcje wspólne dla całej aplikacji

## 7.3 Działanie

Opis działania jest bardzo prosty. Aplikacja najpierw pobiera dane od użytkownika, sprawdza czy mają poprawny format i ewentualnie ponawia zapytanie o dany fragment wyświetlając stosowny komunikat. Następnie przekazuje zapytanie do implementacji protokołu *client-server*, w którym jest wysyłane do serwera. Po otrzymaniu odpowiedzi wyświetla czy trasa została wyznaczona i jeżeli tak, to wyświetla ją czytelnej dla użytkownika formie.

## 7.4 Dodawanie własnego klienta

Jest możliwość dodania własnej aplikacji klienta. Jedynym wymogiem jest zaimplementowanie protokołu *client-server* opisanego w niniejszej dokumentacji.

# 8 Serwer - Aplikacja

## 8.1 Implementacja

Aplikacja jest napisana w Javie 1.8 i budowana oraz konfigurowana przez skrypty napisane w Bashu. Do swojego działania wymaga podobnie jak aplikacja kliencka podprojektu *commons*, w którym znajdują się klasy wspólne dla całego projektu.

## 8.2 Klasy

Aplikacja serwerowa składa się z następujących klas (wraz z krótkim opisem funkcjonalności):

- `QueryProtocol` - klasa służąca do komunikacji z przewoźnikiem, implementacja protokołu *server-carrier*
- `Config` - klasa opakowująca plik konfiguracyjny
- `ApplicationServer` - kod głównego wątku serwera
- `QueryHandler` - klasa służąca do komunikacji z klientem, obsługa zapytania, implementacja protokołu *client-server*
- `CarriersAccessor` - połączenie aplikacji i "bazy danych" zawierającej spis przewoźników
- `DistancesAccessor` - połączenie aplikacji i "bazy danych" zawierającej odległości między miastami
- `PathCalculator` - wyliczanie najkrótszej ścieżki między dwoma miastami, algorytm Dijkstry
- `RouteSearch` - wyszukiwanie najszybszego połączenia na danej trasie
- `Application` - klasa główna aplikacji, start serwera

## 8.3 Działanie

Na początku uruchamiany jest nowy wątek z kodem serwera. Tworzony jest SSL Server Socket i ustawiony w tryb nasłuchiwania. W momencie zaakceptowania klienta, do ExecutorService wysyłane jest zadanie obsłużenia klienta. Obsługa znajduje się w klasie `QueryHandler`. Tam jest pobierane zapytanie od klienta i walidowane pod względem znaków, formatu, poprawności itp. Następnie wyznaczana jest najkrótsza ścieżka między miastami: początkowym i docelowym, a potem na znalezionej ścieżce jest wykonywane wyszukiwanie najlepszego pod względem czasu dotarcia połączenia. Algorytm sprawdza czy jest możliwe bezpośrednie połączenie między miastem początkowym i kolejno: ostatnim, przedostatnim itd. Jeżeli takie znajdzie resztę trasy wyznacza rekurencyjnie. W trakcie wykonywania algorytmu następują odwołania do "baz danych" (w cudzysłowie, bo aktualnie to są zwykłe pliki) oraz zapytania do przewoźników o połączenia między miastami wraz z datą i godziną. Jeżeli w trakcie obsługi zapytania klienta pójdzie nie tak, wysyłany jest stosowny kod odpowiedzi i obsługa klienta kończy działanie. Jeżeli natomiast połączenie zostanie pomyślnie znalezione, zostaje ono wysłane do klienta zgodnie z protokołem *client-server*.

### 8.4 Dodawanie własnego serwera

Jest możliwość dodania własnej aplikacji serwera. Jedynym wymogiem jest zaimplementowanie protokołu *client-server* opisanego w niniejszej dokumentacji. Jeżeli dodatkowo serwer chce się łączyć z przewoźnikami, musi on implementować także protokół *server-carrier* również opisany w dokumentacji.

## 9 Przewoźnik - Aplikacja

### 9.1 Implementacja

Aplikacja jest napisana w Javie 1.8 i budowana oraz konfigurowana przez skrypty napisane w Bashu. Do swojego działania potrzebuje zainstalowaną aplikację SQLite3 oraz podobnie jak poprzednie aplikacje, wymaga podprojektu *commons*, w którym znajdują się klasy wspólne dla całego projektu.

### 9.2 Klasy

Aplikacja przewoźnika składa się z następujących klas (wraz z krótkim opisem funkcjonalności):

- `Config` - klasa opakowująca plik konfiguracyjny
- `QueryExecutor` - wykonuje zapytanie do bazy danych zawierającej rozkład jazdy
- `TimetableAccessor` - połączenie aplikacji i "bazy danych" zawierającej rozkład jazdy przewoźnika
- `QueryHandler` - klasa służąca do komunikacji z serwerem, obsługa zapytań serwera, serwerowa implementacja protokołu *server-carrier*
- `ApplicationServer` - kod głównego wątku serwera
- `Application` - klasa główna aplikacji, start serwera

### 9.3 Działanie

Opis działania tej aplikacji jest bardzo prosty. Jest to typowy przykład serwera wykonującego zapytania. Na początku uruchamiany jest nowy wątek z kodem serwera. Tworzony jest SSL Server Socket i ustawiony w tryb nasłuchiwania. W momencie zaakceptowania klienta (czyli de facto naszego serwera z projektu), do ExecutorService wysyłane jest zadanie obsłużenia go. Obsługa znajduje się w klasie `QueryHandler`. Tam jest pobierane zapytanie od klienta (naszego serwera) i walidowane pod względem znaków, formatu, poprawności itp. Następnie wykonywane jest zapytanie do bazy danych. Jeżeli baza danych odpowie niepustym rekordem, jest on wysyłany do klienta w formacie opisanym w protokole *server-carrier*. W przeciwnym wypadku wysyłany jest stosowny `ResoinseCode`.

### 9.4 Dodawanie własnego przewoźnika

Jest możliwość dodania własnej aplikacji przewoźnika. Jedynym wymogiem jest zaimplementowanie protokołu *server-carrier* opisanego w niniejszej dokumentacji.

# 10 Commons

## 10.1 Implementacja

W podprojekcie *commons* zawarte są klasy wspólne dla wszystkich podprojektów.

## 10.2 Klasy

Podprojekt *commons* składa się z następujących klas (wraz z krótkim opisem funkcjonalności):

- `Carrier` - klasa opisująca przewoźnika
- `Query` - klasa opisująca zapytanie klienta, wzorzec builder
- `Route` - klasa opisująca całą trasę
- `Segment` - klasa opisująca pojedynczy odcinek trasy, wzorzec builder
- `DatabaseAccessor` - interface dostępu do baz danych
- `DateTimeUtils` - formatery daty i czasu
- `Pair` - klasa do przechowywania par, immutable
- `ResponseCodes` - enum z kodami odpowiedzi serwerów
- `ServerUtils` - funkcje wspólne dla serwerów
- `Validator` - klasa z różnymi walidatorami