

# [SDPB] Wyznaczanie trasy

## Dokumentacja końcowa projektu

Michał Aniserowicz, Jakub Turek

### Temat projektu

Napisać aplikację wyznaczającą i porównującą trasę przejazdu na terenie Warszawy środkami komunikacji miejskiej i samochodem z opcją ustawienia godziny wyjścia z domu żeby zdążyć na czas. Aplikacja może być aplikacją mobilną lub przeznaczoną na komputer klasy PC.

### Założenia

Temat został uszczegółowiony poniższymi założeniami:

- Aplikacja zaprojektowana w architekturze klient-serwer.
- Serwer posiada dwie odpowiedzialności:
  - Przechowuje dane.
  - Dostarcza logikę związaną z przeliczaniem tras.
- Klient jest aplikacją dostępową umożliwiającą wprowadzenie następujących danych:
  - Lokalizacja początkowa i docelowa (punkty na mapie).
  - Docelowy czas przyjazdu.
  - Przejazd komunikacją miejską lub samochodem.
- Klient na wyjściu prezentuje najszybszą trasę dla podanych danych wejściowych oraz godzinę wyjścia/wyjazdu, która pozwala zdążyć na czas.
- Dane nie są dostarczane z systemów zewnętrznych (np. *Google Maps*, *Jak dojadę*), ale składowane w bazie danych opracowanej w ramach projektu.
- Uproszczenie algorytmu wyznaczania trasy:
  - Założenie, że czas przejazdu tego samego odcinka drogi komunikacją miejską i samochodem jest identyczny.
  - Brak uwzględnienia informacji o godzinach przyjazdu środków komunikacji miejskiej na przystanki. Zakłada się, że czas wymagany na przesiadkę jest stały i jest parametrem algorytmu.

- Wyznaczana jest zawsze pojedyncza najszybsza trasa dla wybranego środka komunikacji.
- Jako punkt początkowy i końcowy wybierane są te lokalizacje spośród danych znajdujących się w bazie, które są najbliższe lokalizacjom wskazanym przez użytkownika.
- W przypadku wskazania komunikacji miejskiej jako środka transportu punktem początkowym i końcowym podróży są zawsze przystanki najbliższe wskazanym lokalizacjom. Nie jest uwzględniana możliwość dojścia do przystanku.

## Model danych

W aplikacji został wykorzystany sieciowy model danych:

- Miasto jest opisane przy pomocy węzłów oraz łuków:
  - Współrzędne węzła (punkty) są opisane długością oraz szerokością geograficzną.
  - Węzły opisują różne lokalizacje miejskie: przystanki autobusowe, skrzyżowania, etc.
  - Łuki reprezentują połączenia drogowe pomiędzy węzłami. Są skierowane, więc można na nich opisać ruch jednokierunkowy.
  - Informacja o relacjach pomiędzy dwuwymiarowymi strukturami nie jest przechowywana.
- Nie ma ograniczeń odnośnie planarności sieci:
  - Testowy zbiór danych jest siecią planarną.
  - Aplikacja bez modyfikacji zadziała dla sieci nieplanarnych (tunele, przejazdy wielokondygnacyjne).

## Reprezentacja logiczna modelu danych

Relacyjny model baz danych nie jest dobrze przystosowany do opisu modelu sieciowego. Informacje o strukturze grafowej można przechowywać na dwa sposoby:

- W postaci znormalizowanych tabel - osobne tabele dla wierzchołków i krawędzi.
- W postaci zdenormalizowanych tabel - rekord tabeli zawiera pełną informację o węźle i wszystkich połączonych z nim węzłach (wejściowych i wyjściowych). Węzły wyjściowe i wejściowe przechowywane są w postaci dwóch osobnych list.

W przypadku drugiego podejścia problemem jest przechowywanie informacji o koszcie danej krawędzi. Wymaga ona opakowania informacji w słownikową strukturę danych, co z kolei uniemożliwia wykonywanie zapytań języka SQL. Alternatywą może być przechowywanie tylko informacji o ścieżkach. Przy tym podejściu każdy rekord zawiera pełne

informacje o węźle początkowym i końcowym. Jest to jednak duży narzut pamięciowy, gdyż informacja o każdym węźle jest przechowywana  $n$  razy, gdzie  $n$  to stopień danego węzła.

Algorytm wyszukiwania połączeń w sieci wymaga dokonania wielu złączeń. Przykładowo dla postaci znormalizowanej, po wejściu do każdego węzła musimy najpierw pobrać informację o łukach, które ten węzeł tworzy, a następnie pobrać informacje o połączonych z nim węzłach. Jeżeli na etapie oszacowania kosztu potrzebna jest informacja o węzłach poprzedzających, wtedy wykonywane będą kolejne złączenia. Jest to istotny narzut obliczeniowy. Zakładając, że w ramach modelu danych będzie ujęte wiele typów relacji problem potęguje się.

Opisany w powyższym akapicie narzut nie wyklucza wykorzystania relacyjnej bazy danych do rozwiązania zadanego problemu. Autorzy projektu postanowili jednak skorzystać z innego podejścia i wykorzystać bazy danych o reprezentacji logicznej lepiej nadającej się do modelowania dziedziny problemu.

Interesującą alternatywę zapewnia ruch **Not Only SQL**, a konkretnie jego podzbiór - grafowe bazy danych. Ich logiczny model danych jest spójny z sieciowym modelem struktury miasta, dlatego bardzo dobrze nadają się do rozwiązania postawionego problemu. Jednym z najbardziej dojrzałych projektów grafowych baz danych jest **Neo4j** (<http://www.neo4j.org>). Zaletą tej bazy w kontekście tematyki przestrzennych baz danych jest posiadanie warstwy abstrakcji do obliczeń przestrzennych - **Neo4j Spatial**. Ze względu na wymienione zalety i odpowiedniość struktury do dziedziny problemu, do implementacji projektu autorzy skorzystali właśnie z tej bazy danych.

## Schemat modelu danych miasta

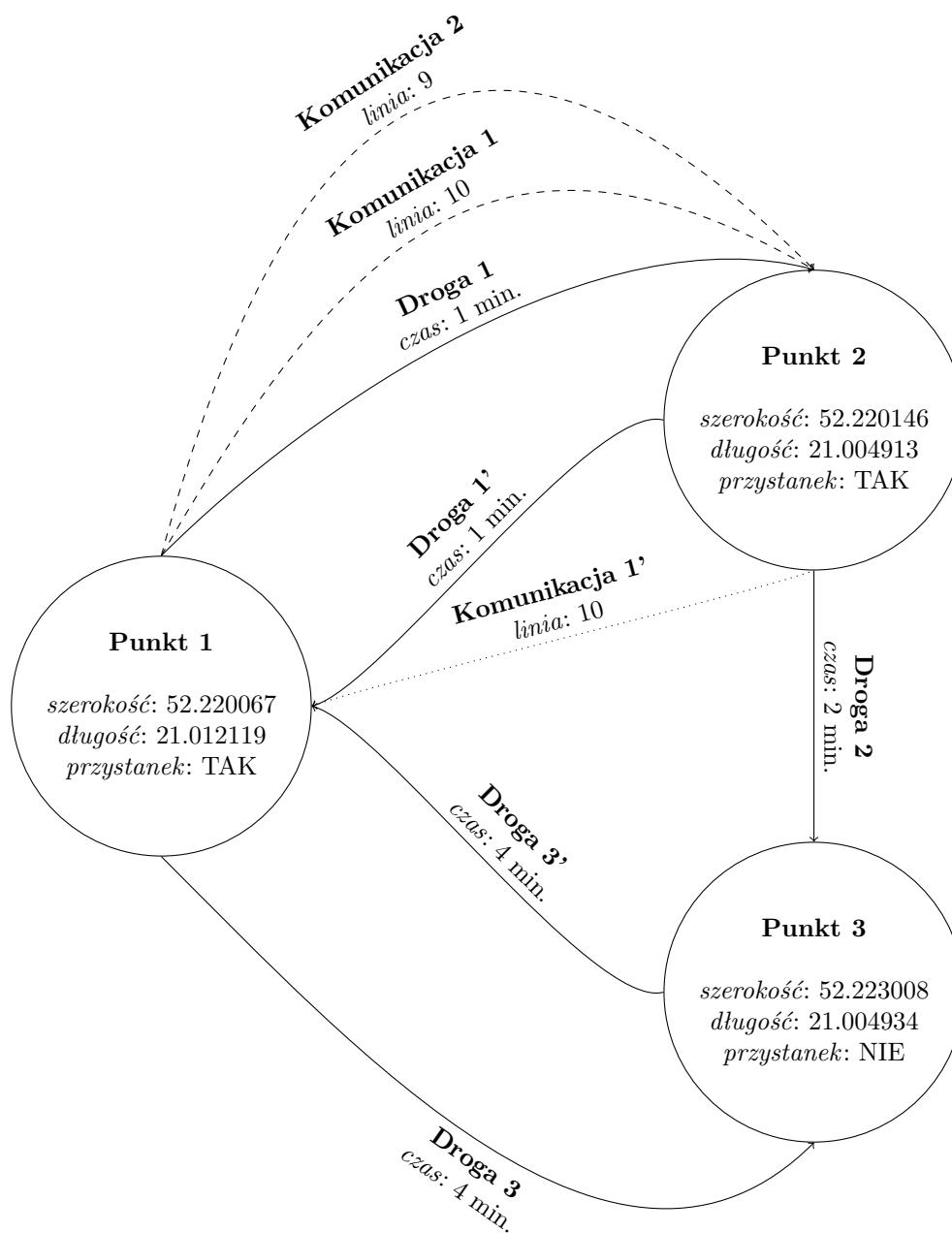
Wykorzystany model danych został schematycznie przedstawiony na rysunku 1.

Na schemacie pokazane są trzy punkty:

1. O współrzędnych (52.220067, 21.012119). Przystanek komunikacji miejskiej.
2. O współrzędnych (52.220146, 21.004913). Przystanek komunikacji miejskiej.
3. O współrzędnych (52.223008, 21.004934).

Na schemacie pokazane są dwa typy relacji. Relacja narysowana pełną linią o nazwie **Droga** oznacza istnienie drogi pomiędzy punktami. Relacja narysowana linią przerywaną o nazwie **Komunikacja** oznacza natomiast istnienie ścieżki przejazdu komunikacji miejskiej jednej linii pomiędzy tymi punktami. Z relacji przedstawionych na schemacie można więc odczytać:

- Z punktu 1 do punktu 2 prowadzi droga 1, której przebycie zajmuje jedną minutę. Istnieje droga powrotna 1', która prowadzi z punktu 2 do punktu 1.
- Z punktu 1 do punktu 2 można dojechać dwoma liniami komunikacji miejskiej: 9 i 10. Z punktu 2 do punktu 1 wraca tylko linia numer 10.
- Punkty 2 i 3 połączone są drogą jednokierunkową, której przebycie zajmuje 2 minuty.



Rysunek 1: Schematyczne przedstawienie modelu danych miasta.

- Punkty 1 i 3 są połączone drogą dwukierunkową, której przejechanie zajmuje 4 minuty.

## Indeksy

W modelu danych założone są dwa indeksy:

- Indeks bitmapowy na polu *przystanek*.
- Indeks przestrzenny na polach *szerokość* i *długość* geograficzna<sup>1</sup>.

## Metoda wyznaczania trasy

Algorytm wyznaczania najszybszej trasy jest dwukrokowy:

1. W bazie danych odnajdywane są dwa punkty: najbliższe początkowi i końcowi trasy spośród wszystkich punktów zdefiniowanych w bazie danych. W przypadku wyznaczania trasy komunikacji miejskiej wyznaczane są takie najbliższe punkty, że są one przystankami komunikacji.
2. Za pomocą algorytmu Dijkstry wyznaczana jest najszybsza droga pomiędzy znalezionymi punktami:
  - Dla trasy samochodowej/pieszej pod uwagę brane są tylko i wyłącznie łuki, które reprezentują relację **Droga**. Ewaluacja kosztu polega na dodawaniu czasu podróży dla każdego segmentu drogi.
  - Dla podróży komunikacją miejską pod uwagę brane są tylko i wyłącznie łuki, które reprezentują relację **Komunikacja**. Dla każdego segmentu wykonywana jest ewaluacja kosztu:
    - Koszt jest zwiększany o czas podróży dla danego segmentu.
    - Wykonywane jest sprawdzenie czy do węzła wejściowego analizowanego łuku prowadzi łuk, który obsługuje ta sama linia komunikacyjna. Jeżeli tak to przesiadka jest zbędna i ewaluacja kosztu kończy się. W przeciwnym razie koszt całkowity ścieżki jest powiększany o zadany czas oczekiwania na przesiadkę.

## Architektura aplikacji

Aplikacja została zaprojektowana w architekturze klient-serwer. Szczegółowy schemat architektury przedstawia rysunek 2.

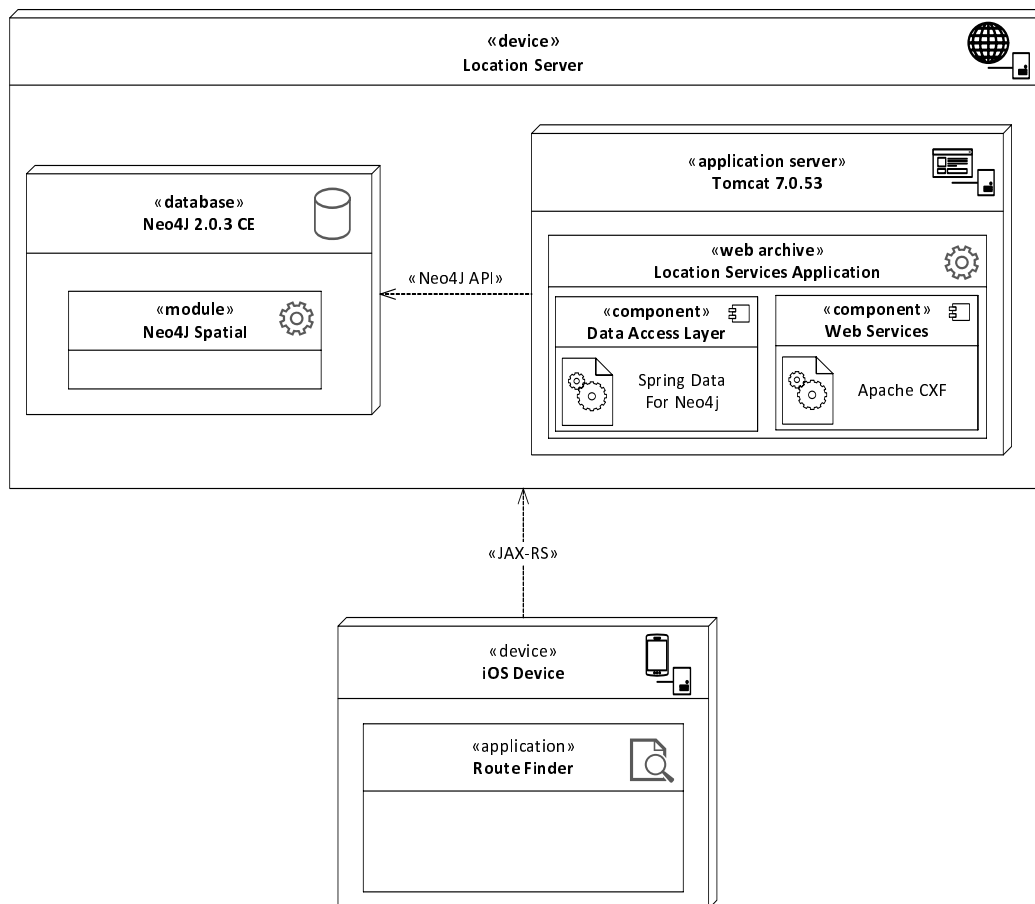
### Serwer

Serwer posiada następujące odpowiedzialności:

- Przechowuje informacje o sieci miasta.

---

<sup>1</sup>W bazie danych Neo4j punkt jest reprezentowany jako łańcuch znaków WKT (Well-Known Text).  
Przykład: POINT( 52,223008 21,004934 ).



Rysunek 2: Schemat architektury aplikacji.

- Zwraca informacje o najbliższych punktach z bazy danych dla danej szerokości i długości geograficznej<sup>2</sup>.
- Zwraca informacje o najbliższych przystankach komunikacji z bazy danych dla danej szerokości i długości geograficznej<sup>2</sup>.
- Zwraca informacje o najszybszej ścieżce dotarcia z punktu o identyfikatorze A do punktu o identyfikatorze B. W zależności od wywołania metody zwracana jest ścieżka dla samochodu lub komunikacji miejskiej.

## Klient

Klient posiada następujące odpowiedzialności:

- Pozwala zdefiniować punkt początkowy i końcowy podróży poprzez zaznaczenie na mapie.
- Umożliwia wprowadzenie godziny przyjazdu.

<sup>2</sup>Przeszukiwanie domyślnie ograniczone jest do promienia 5 kilometrów. Ograniczenie to można zmienić w pliku `config.properties`.

- Rysuje najszybszą ścieżkę na mapie na podstawie informacji pobranych z serwera.
- Oblicza czas wyjścia z domu na podstawie obecnego czasu i długości podróży odebranej z serwera.
- Pozwala na określenie czasu przesiadki.

## Zasilanie danymi

Aby ułatwić inicjalne zasilanie bazy danych serwera utworzony został moduł o nazwie **data-populator**:

- Moduł ten wypełnia bazę danych wartościami pobranymi ze zdefiniowanego repozytorium.
- Domyślnym repozytorium są trzy pliki płaskie, które odpowiednio zawierają informacje o węzłach mapy, łukach dróg oraz łukach komunikacji miejskiej.

Rysunek 3 przedstawia przykład pliku definiującego punkty wejściowe dla węzłów mapy. Linie rozpoczynające się od znaku **#** są traktowane jako komentarz. Kolejne kolumny oznaczają: identyfikator punktu w obrębie repozytorium, szerokość geograficzną, długość geograficzną oraz flagę oznaczającą czy dany punkt jest przystankiem komunikacji miejskiej. Ostatnia kolumna, w której znajduje się opis punktu, nie jest wprowadzana do bazy danych.

#	id	latitude	longitude	pt_stop	description
	EN_0001	52.220067	21.012119	true	Plac Politechniki
	EN_0002	52.220146	21.004913	true	Nowowiejska/Al. Niepodległości
	EN_0003	52.223008	21.004934	true	Koszykowa/Chałubińskiego
	EN_0004	52.227885	21.001865	true	Al. Jerozolimskie/Jana Pawła
	EN_0005	52.230014	21.011886	true	Marszałkowska/Al. Jerozolimskie
	EN_0006	52.219893	21.018152	true	Plac Zbawiciela
	EN_0007	52.223232	21.015984	true	Plac Konstytucji
	EN_0008	52.226229	21.014161	true	Marszałkowskiej/Hoża
	EN_0009	52.228353	21.010203	false	Nowogrodzka/Poznańska

Rysunek 3: Plik repozytorium zawierający dane wejściowe dla węzłów mapy.

Rysunek 4 przedstawia przykład pliku definiującego łuki reprezentujące drogi. Kolumny oznaczają kolejno: identyfikator drogi w obrębie repozytorium, identyfikator węzła początkowego drogi, identyfikator węzła końcowego drogi oraz czas podróży podany w sekundach.

Rysunek 5 przedstawia przykład pliku definiującego łuki reprezentujące transport publiczny. Kolumny oznaczają kolejno: identyfikator wpisu w obrębie repozytorium, numer linii komunikacji oraz identyfikator drogi, po której porusza się komunikacja.

#	id	from	to	travel_time_in_sec
	R0_0001	EN_0001	EN_0002	180
	R0_0002	EN_0002	EN_0003	120
	R0_0003	EN_0003	EN_0004	240
	R0_0004	EN_0004	EN_0005	300
	R0_0005	EN_0001	EN_0006	120
	R0_0006	EN_0006	EN_0007	120
	R0_0007	EN_0007	EN_0008	120
	R0_0008	EN_0008	EN_0005	240

Rysunek 4: Plik repozytorium zawierający dane wejściowe dla łuków reprezentujących drogi.

#	id	line	route
	PTRO_0001	10	R0_0001
	PTRO_0002	10	R0_0002
	PTRO_0003	10	R0_0003
	PTRO_0004	9	R0_0004
	PTRO_0005	15	R0_0005
	PTRO_0006	15	R0_0006
	PTRO_0007	15	R0_0007
	PTRO_0008	15	R0_0008

Rysunek 5: Plik repozytorium zawierający dane wejściowe dla łuków reprezentujących komunikację publiczną.

## Komunikacja

Komunikacja odbywa się przy pomocy serwisów w specyfikacji JAX-RS<sup>3</sup>. Zostały zdefiniowane dwa serwisy.

**Najbliższy punkt na mapie** Metoda jest dostępna przez odwołanie do relatywnej ścieżki `/nearest/{lat}/{lon}/publicTransport/{stop}`. Parametr `lat` należy zastąpić szerokością geograficzną w formacie zmiennoprzecinkowym z kropką. Parametr `lon` należy zastąpić długością geograficzną w tym samym formacie. Parametr `stop` należy natomiast wypełnić wartością `true` lub `false` jeżeli odpowiednio ma zostać zwrócony przystanek komunikacji miejskiej lub dowolny punkt. Serwis zwraca odpowiedź w formacie przedstawionym na rysunku 7.

**Najszybsza droga** Metoda może być wywołana poprzez odwołanie do relatywnej ścieżki `/shortestPath/{sId}/{fId}/publicTransport/{pT}/changeDuration/{cD}`. Parametr `sId` należy zastąpić poprzez identyfikator węzła początkowego. Parametr `fId` należy zastąpić poprzez identyfikator węzła końcowego. Oba identyfikatory mogą pochodzić z wywołania serwisu dla najbliższego punktu na mapie. Parametr `pT` należy zastąpić wartością `true` lub `false` w zależności od tego czy wyszukana ma być droga dla komunikacji miejskiej lub samochodu. Ostatni parametr `cD` należy zastąpić liczbą całkowitą,

---

<sup>3</sup>Java API for RESTful Web Services.



```
{
  "id": 15,
  "latitude": 52.219893,
  "longitude": 21.018152
}
```

Rysunek 6: Przykładowa odpowiedź serwisu odnajdującego najbliższy punkt w bazie danych.

która oznacza ilość sekund doliczanych na oczekiwanie przy przesiadce. Odwołanie do `/changeDuration/{cD}` jest opcjonalne dla najszybszej ścieżki samochodowej i można je pominąć. Serwis zwraca odpowiedź w formacie przedstawionym na rysunku 7.

## Budowanie i uruchamianie aplikacji

Moduł serwerowy oraz kliencki aplikacji budowane są osobno. Część serwerowa jest budowana przy użyciu aplikacji Maven (<http://maven.apache.org>). Uruchomienie polecenia **mvn clean install** w ścieżce projektu spowoduje ściągnięcie wszystkich zależności, uruchomienie testów jednostkowych oraz integracyjnych, a także wygenerowanie pliku web-archive (**.war**), który następnie trzeba *zdeployować* na serwer aplikacji Java.

Budowanie aplikacji klienckiej jest realizowane przy pomocy menadżera zależności CocoaPods (<http://cocoapods.org>). Uruchomienie polecenia **pod install** w podfolderze z klientem spowoduje ściągnięcie wszystkich zależności oraz wygenerowanie pliku projektu programu Xcode, za pomocą którego można skompilować i wyeksportować aplikację.

## Implementacja

### Moduły

Kod źródłowy projektu został podzielony na cztery moduły:

**data-populator** służy do wypełniania bazy danych inicjalnymi wartościami. Odczytuje wartości z plików płaskich.

**data-access** interfejs abstrahujący dostęp do danych.

**web-services** moduł udostępniający dostęp do bazy danych z użyciem serwisów REST.

**ios-client** aplikacja kliencka.

### Środowisko uruchomieniowe

Projekt był uruchamiany i testowany w następującym środowisku:

- System operacyjny: OS X Mavericks 10.9.3 (dla aplikacji serwerowej), iOS 7.1.1 (dla aplikacji klienckiej).

```
[
  {
    "id": 10,
    "routeFrom": {
      "id": 15,
      "latitude": 52.219893,
      "longitude": 21.018152
    },
    "routeTo": {
      "id": 329,
      "latitude": 52.220067,
      "longitude": 21.012119
    },
    "duration": 300,
    "line": 9
  },
  {
    "id": 73,
    "routeFrom": {
      "id": 329,
      "latitude": 52.220067,
      "longitude": 21.012119
    },
    "routeTo": {
      "id": 412,
      "latitude": 52.228353,
      "longitude": 21.010203
    },
    "duration": 240,
    "line": 15
  }
]
```

Rysunek 7: Przykładowa odpowiedź serwisu odnajdującego najszybszą ścieżkę pomiędzy punktami.

- Wirtualna maszyna Java: JRE 1.8.0.
- Serwer aplikacyjny: Apache Tomcat 7.0.53.
- Baza danych: Neo4j 2.0.2 CE.

Kod źródłowy projektu (części serwerowej) jest zgodny z najnowszą specyfikacją języka Java w wersji 8. Wykorzystywane elementy języka to między innymi wskaźniki na funkcje, wyrażenia lambda oraz nowy interfejs programistyczny dla dat. Z tego względu aplikacja nie uruchomi się w starszej wersji maszyny wirtualnej.

## Środowisko developerskie

Projekt był implementowany przy użyciu:

- IntelliJ IDEA Ultimate 13.1.2. Zaawansowane IDE do programowania w języku Java.
- Xcode 5.1.1. Domyślne IDE dla użytkowników systemu OS X umożliwiające tworzenie aplikacji na systemy OS X / iOS.

## Testy jednostkowe

Projekt posiada zestaw testów jednostkowych, które pokrywają wszystkie moduły zaimplementowane na potrzeby tematu:

- Testy jednostkowe operacji bazodanowych z wykorzystaniem biblioteki testującej dla Spring Data.
- Testy jednostkowe modułu wczytującego i wypełniającego bazę danych inicjalnymi wartościami.
- Testy integracyjne serwisów, które automatycznie uruchamiają serwer aplikacyjny, uruchamiają na nim serwisy i przeprowadzają testy klienckie.
- Testy jednostkowe komunikacji z serwisami po stronie aplikacji klienckiej.

## Wykorzystane oprogramowanie stron trzecich

Projekt wykorzystuje liczne elementy oprogramowania stron trzecich:

- JUnit 4.1.1. Biblioteka umożliwiająca testowanie jednostkowe aplikacji w języku Java.
- Apache Maven 3.2.1. Narzędzie służy do budowania części serwerowej aplikacji. Pozwala na zarządzanie zależnościami i przenośne budowanie projektu bez potrzeby dostarczania skompilowanych zależności wraz ze źródłami projektu.
- Spring Framework 4.0.3.RELEASE. Zestaw bibliotek programistycznych o szerokim zakresie funkcjonalności do tworzenia aplikacji enterprise opartych o język Java. W projekcie wykorzystywany jest głównie jako kontener DI<sup>4</sup>.
- Neo4j API 2.0.2. Interfejs programistyczny dla języka Java umożliwiający korzystanie z bazy danych Neo4j.
- Neo4j Spatial 0.12. Moduł dla bazy danych Neo4j do realizacji operacji i indeksów przestrzennych.

---

<sup>4</sup>Dependency Injection - wzorec architektoniczny polegający na usuwaniu bezpośrednich zależności między komponentami.

- Spring Data Neo4j 3.0.1.RELEASE. Wtyczka dla Spring Framework, która realizuje mapowanie obiektowe komponentów grafowej bazy danych Neo4j.
- Apache CXF 2.7.11. Biblioteka, która służy do tworzenia web-service'ów. Dostarcza implementację dla interfejsów JAX-WS oraz JAX-RS.
- Google Gson 2.2.4. Biblioteka, która służy do automatycznej serializacji obiektów POJO do formatu JSON.
- CocoaPods 0.22.0. Menedżer zależności dla języka Objective-C.
- RestKit 0.20.0. Biblioteka umożliwiające prostą komunikację z serwisami RESTful dla Objective-C.
- XCTAsyncTestCase 0.1.0. Biblioteka umożliwiające testy jednostkowe kodu wykonywanego asynchronicznie w Objective-C.
- MBProgressHUD 0.8. Kontrolka dla systemu iOS umożliwiająca prezentację paska postępu ładowania.
- PXAlertView 0.1.0. Alternatywna kontrolka służąca do prezentacji komunikatów systemowych na platformie iOS.