1. **Technologies Used**

Database: PostgreSQL installed with PostGIS extension for geospatial data. Docker image mdillon/postgis:9.5-alpine will be used which brings us PostgreSQL with version 9.5 and PostGis already installed.

Application: Application is developed with JAVA and Spring Boot.

Java: Java 8

Spring Boot: 2.2.5.RELEASE

Server to be deployed: By default from spring boot the embedded Tomcat server

Build Tool: Maven

Soap Api will be implemented from Spring with the following dependencies:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web-services</artifactId>  
</dependency>

<dependency>  
 <groupId>wsdl4j</groupId>  
 <artifactId>wsdl4j</artifactId>  
</dependency>

In order to communicate with the database, Spring JPA layers will be used with the following dependency:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

Because of our need to handle geospatial data that come from PostGis extension in PostgreSQL we need some more dependencies:

<dependency>  
 <groupId>com.vividsolutions</groupId>  
 <artifactId>jts</artifactId>  
 <version>${vividsolutions-jts.version}</version>  
</dependency>  
<dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-spatial</artifactId>  
 <version>${hibernate-spatial.version}</version>  
</dependency>

**For integration tests and test driven development Test Containers will be used. These Test Containers will startup a docker container with our database which we can test during development. In order to achieve that we need the following dependency:**

<dependency>  
 <groupId>org.testcontainers</groupId>  
 <artifactId>postgresql</artifactId>  
 <version>${testcontainers-postgresql.version}</version>  
 <scope>test</scope>  
</dependency>

In order to test our SOAP API endpoints the following dependency is needed:

<dependency>  
 <groupId>org.springframework.ws</groupId>  
 <artifactId>spring-ws-test</artifactId>  
 <version>${spring-ws-test.version}</version>  
 <scope>test</scope>  
</dependency>

In order to have less code in development and more simplified objects Lombock is used for automatic generation of getters, setters and constructors.

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>${lombock.version}</version>  
</dependency>

**If your IDE shows errors during project import make sure you have installed the plugin for Lombock!!!**

1. **Technical Conception**

Our Api must be fed with location data (latitude, longitude) and be able to return the closest point.

In order to achieve that we need to scan all rows in database. This will be time and resources expensive. Therefore a cache must be implemented.

Solution:

Our Database contain rows with many columns (id , coordinates, city, country, population, capital, request\_counter).

Our Application will keep in cache only the following information (id, coordinates, city) in order to save memory.

During application startup all database rows will be loaded in this smaller cache object. Cache will also be updated every day at 23:00 with the help of a Spring Scheduler. Here for simplicity 23:00 is static defined in code but could also be configured with an external parameter.

Our application will provide the following 3 SOAP Apis:

1. getNearestCityName(lat, lng) -> This method will return only the name of the city with the closest distance based on the provided coordinates. Application will scan only cache and return the city with the closest distance directly from cache. Before returning city name to the client, it will hit database once to increase the request\_counter for that row.

With this implementation the results are based on cache. So each change in database will not affect this api until a cache update occurs.

This function will be faster than the next one however it works based on a screenshot of our database during the last cache update.

1. getNearestPoint(lat, lng) -> This method will return all the information that we have in database for the row with the closest distance based on the provided coordinates. Application will scan only cache to find the closest object. After that it will hit database to retrieve all information for that object and increase the request\_counter.

With this implementation the results are based on cache and Database. Therefore we must consider the following situations:

- Row is deleted from database. In case the closest object is found in cache but not in database then the application will throw an illegalstateexception.

- Lat, Lng are changed for a row in Database. Here we run the risk of inconsistent data. The closest object found in cache and retrieved from database have different coordinates. The application will check for inconsistent coordinates and if found will throw an illegalStateException.

Both methods must increase the request\_counter in database by 1 with each request.

Each update must first read the value of request\_counter and then increase it by one. When thousands of requests try to increase the same counter at the same time will produce inconsistent data. Here a mechanism is implemented so that each update occurs inside a Repeatable Read Transaction. So only 1 thread can commit an update to request\_counter each time. According to PostgreSQL documentation this transaction level will lead to failures if inconsistent data are to be produced.

*\*\*  
 \* Official postgres documentation https://www.postgresql.org/docs/9.5/transaction-iso.html  
 \* Repeatable Read Isolation Level  
 \* A repeatable read transaction cannot modify or lock rows changed by other transactions after the repeatable  
 \* read transaction began.  
 \* Applications using this level must be prepared to retry transactions due to serialization failures.  
 \* if the first updater commits (and actually updated or deleted the row, not just locked it) then the  
 \* repeatable read transaction will be rolled back with the message  
 \* ERROR: could not serialize access due to concurrent update  
 \*  
 \* Spring JPA will throw org.springframework.dao.CannotAcquireLockException according to  
 \* https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/dao/CannotAcquireLockException.html*

When an update fails we must try again later. How many times we can afford to try to update the request\_counter is configurable in application.yaml with the property **maxCounterUpdateIncreaseAttempts**

1. getPointsWithGreaterCounter(counter) -> This method will fetch all data directly from database according to the counter that each row has. request\_counter could be considered an index in that search but because of the frequency of change and frequency of duplicate values would do more harm than good. More analysis here is needed of how to optimize this function.

**Distance between 2 Points**

private static double distFrom(double lat1, double lng1, double lat2, double lng2) {  
 double earthRadius = 6371000; //meters  
 double dLat = Math.*toRadians*(lat2-lat1);  
 double dLng = Math.*toRadians*(lng2-lng1);  
 double a = Math.*sin*(dLat/2) \* Math.*sin*(dLat/2) +  
 Math.*cos*(Math.*toRadians*(lat1)) \* Math.*cos*(Math.*toRadians*(lat2)) \*  
 Math.*sin*(dLng/2) \* Math.*sin*(dLng/2);  
 double c = 2 \* Math.*atan2*(Math.*sqrt*(a), Math.*sqrt*(1-a));  
 return (earthRadius \* c);  
}

With this function we can calculate the distance between 2 Points (lat1, lng1) and (lat2, lng2) considering the curvature of earth also.

**Algorithm to search for object with minimum distance**

private PointOfInterestCache getNearestPoint(double lat, double lng){  
 PointOfInterestCache nearestPoint = null;  
 Double minDistance = null;  
 for (PointOfInterestCache pointOfInterest : pointOfInterestsCache){  
 Double distance = *distFrom*(lat, lng,

pointOfInterest.getMapPoint().getX(),

pointOfInterest.getMapPoint().getY());  
 if (minDistance == null || distance < minDistance){  
 minDistance = distance;  
 nearestPoint = pointOfInterest;  
 }  
 }  
 return nearestPoint;  
}

Here we make a sequential scan which is O(n). In this one loop we will keep in the end in memory only the object with the minimum distance.

Parallel streams were also considered but according to one study

<https://dzone.com/articles/should-i-parallalise-streams> will affect only big data sets from 1.000.000 rows or more. In our Application we are going to handle about 20.000 rows, so they were excluded. If we need to handle much bigger data sets another analysis is needed here.

1. **Live Example and time analysis**

Our database contains about 15.500 rows. Because of localized environment with both database and application on the same machine with very limited resources each request will take longer than production based environments.

1. **Testing of getNearestPoint -> finds closest from cache, retrieves full info from DB, updates counter in DB.**

**2020-09-20 16:36:27.832** INFO 19072 --- [nio-9595-exec-1] c.b.g.endpoints.PointOfInterestWs : **Request came with following coordinates x:55.0 y99.0**

**2020-09-20 16:36:27.957** TRACE 19072 --- [nio-9595-exec-1] o.s.ws.server.MessageTracing.sent : **Sent response** [<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Header/><SOAP-ENV:Body><ns2:getNearestPointResponse xmlns:ns2="map-points"><ns2:pointOfInterest><ns2:city>Nizhneudinsk</ns2:city><ns2:capital xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/><ns2:country>Russia</ns2:country><ns2:population>43326</ns2:population><ns2:requestCounter>2</ns2:requestCounter><ns2:coordinates><ns2:lat>54.8977</ns2:lat><ns2:lng>99.0277</ns2:lng></ns2:coordinates></ns2:pointOfInterest></ns2:getNearestPointResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>] for request [<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:map="map-points">



So method **getNearestPoint needed only 0.125 of a second**

1. **Testing of getNearestCityName-> finds closest from cahce, updates counter in DB.**

**2020-09-20 16:44:25.300** INFO 19072 --- [nio-9595-exec-3] c.b.g.endpoints.PointOfInterestWs : **Request came with following coordinates x:12.0 y24.0**

**2020-09-20 16:44:25.318** TRACE 19072 --- [nio-9595-exec-3] o.s.ws.server.MessageTracing.sent : **Sent response** [<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Header/><SOAP-ENV:Body><ns2:getNearestCityNameResponse xmlns:ns2="map-points"><ns2:cityName>Nyala</ns2:cityName></ns2:getNearestCityNameResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>] for request [<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:map="map-points">



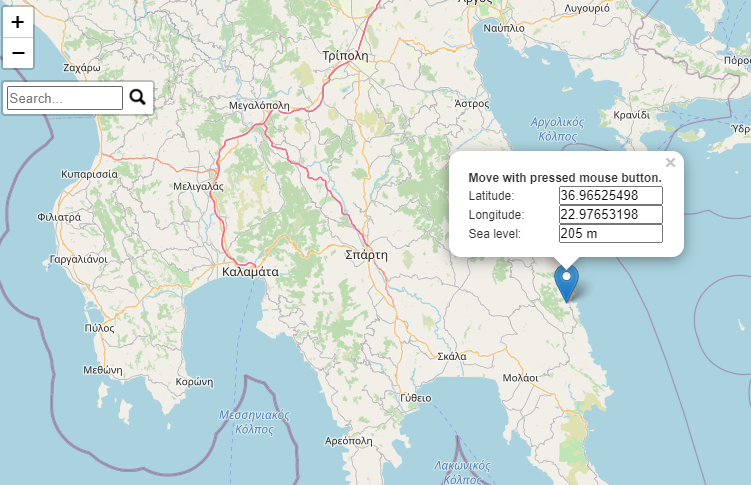
So method **getNearestCityName needed only 0.018 of a second**

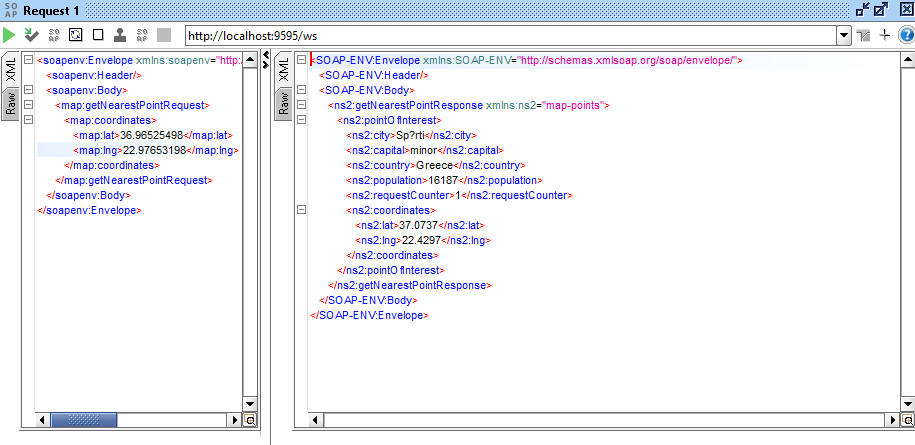
1. **getPointsWithGreaterCounter -> finds directly from DB all rows with greater request\_counter than the one provided**



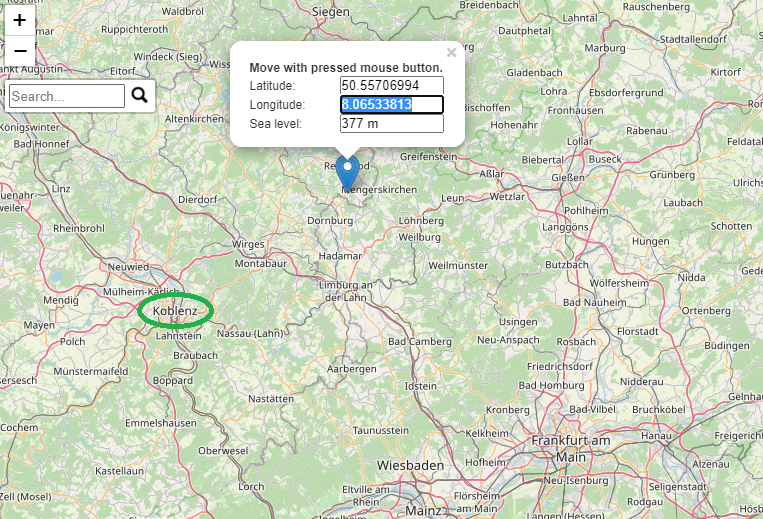
This took only 0.84 of a second however there are not many rows with greater request\_counter in DB to be retrieved. Here maybe more analysis is needed.

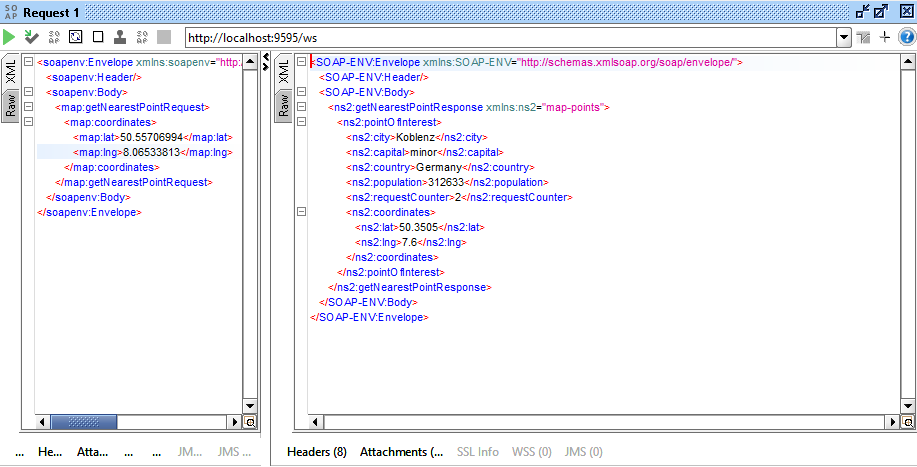
1. **Testing functionality**





Sparti is correctly retrieved as the closest big city to that point. Data encoding faces some problems which could not be solved because of the small time frame that was available.





Koblenz in Germany is correctly retrieved as the closest big city to that point.