Software Documentation

<u>Our schema is consisted of 5 tables:</u>

1. <u>Locations table</u>- this is the main table. It contains an entry for each location that can be found in our app.

   The columns in this table are:

   - geonameid – id of the location
   - name – name of the location
   - latitude – latitude coordinate of the location
   - longitude – longitude coordinate of the location
   - fcode – 3-4 characters code indicating locations' type (park, hotel, etc.)
   - country – 2-3 characters code indicating in which country this location is
   - moddate – the date this location was added to the data base

   Primary key: geonameid

   Foreign key: Main table (foreign keys are defined in other tables)

   Indexes: Primary (geonameid), lat_lng_ind_locations (latitude, longitude), idx_locations_fcode (fcode), idx_locations_country (country)

2. <u>Alternatename table</u>- this table has entries for alternative names for locations.

   The columns in this table are:

   - alternatenameId – id of the alternate name (some places have more than 1 alternate name, thus we can't use the column geonameid as the primary key)
   - geonameid – id of the location
   - alternateName – the alternate name itself

   Primary key: alternatenameId

   Foreign key: geonameid_alternatename_fkey (geonameid)

   Indexes: Primary (alternatenameId), geonameid_alternate_idx (geonameid)

3. <u>Usersdescriptions table</u>- this table contains descriptions/reviews and ratings on the locations, given by the users. We initialized this table using a script we

wrote, that inserts descriptions to the table (so it won't be empty for the first users).

The columns in this table are:

- id – id of the description
- geonameid – id of the location which is described in this entry description
- description – the description/review itself
- rating – an int between 1 and 5
- date – the date of which the review was given

Primary key: id

Foreign key: geonameid_descs_fkey(geonameid)

Indexes: Primary (id), geonameid_descs_Ind (geonameid)

4. <u>Classandcodes table</u>: mapping the fcode (location type) of locations to fclass (category such as buildings, parks, water, etc.) and meaning of the fcode (for example the meaning of fcode "htl" is "hotel").

   The columns in this table are:

   - fcode – location type
   - fclass – locations' category as a one-character code
   - meaning – meaning of the fcode

   Primary key: fcode

   Foreign key: fcode_classandcodes_fkey (fcode)

   Indexes: Primary (fcode)

5. <u>Countries table</u>- mapping between name of country and its 2-3 characters codes. There was a need to save the 2 chars code and the 3 chars code since in the locations table each location has one code which is 2 chars or 3 chars.

   The columns of this table are:

   - ISO – 2 characters code of country
   - Country – the name of the country

Primary key: ISO

Foreign key: ISO_fkey (ISO)

Indexes: Primary (ISO)

<u>Procedures</u> – We wrote the queries as procedures in order to activate the queries in a more readable manner from the DB and thus from the code.
Each query has its own procedure.

<u>SQL queries and updates:</u>

Our DB is based on the DB geolocations taken from https://www.geonames.org. We had to make several changes in it so it would match our applications' needs.
We removed a lot of irrelevant entries from the DB that didn’t fit the application purpose, such as underwater locations, dangerous locations etc.
Furthermore, we removed irrelevant columns from ‘locations’ table and ‘alternatename’ table, such as height, timezone, language abbreviation etc.

<u>Queries we used in creating the DB:</u>

Creating the initial main table.	CREATE TABLE geoname (geonameid int PRIMARY KEY, name varchar(200), asciiname varchar(200), alternatenames varchar(10000), latitude decimal(10,7), longitude decimal(10,7), fclass char(1), fcode varchar(10), country varchar(2), cc2 varchar(60), admin1 varchar(20), admin2 varchar(80), admin3 varchar(20), admin4 varchar(20), population int, elevation varchar(10), gtopo30 int, timezone varchar(40), moddate date) CHARACTER SET utf8mb4;
Uploading the raw data to the table.	LOAD DATA INFILE 'D:\\MySQL\\Uploads\\allCountries.txt' INTO TABLE geoname (geonameid, name, asciiname, alternatenames, latitude, longitude, fclass, fcode, country, cc2, admin1, admin2, admin3, admin4, population, elevation, gtopo30, timezone, moddate)
Creating alternatename table.	CREATE TABLE alternatename (alternatenameId int PRIMARY KEY, geonameid int, alternateName varchar(55)) CHARACTER SET utf8mb4;

Creating the main table based on the geoname table, without some irrelevant places we wanted to throw out. Complex query (not in).	CREATE TABLE Locations AS SELECT * FROM geoname WHERE fclass <> 'U' AND fcode NOT IN ('ADM1', 'ADM1H', 'ADM2', 'ADM2H', 'ADM3', 'ADM3H', 'ADM4', 'ADM4H', 'ADM5', 'ADM5H', 'ADMD', 'ADMDH', 'LTER', 'PCL', 'PCLD', 'PCLF', 'PCLH', 'PCLI', 'PCLIX', 'PCLS', 'PRSH', 'TERR', 'ZN', 'ZNB', 'ANCH', 'BNKX', 'CHNM', 'CHNN', 'CNLD', 'CNLSB', 'CRNT', 'DTCHD', 'DTCHI', 'DTCHM', 'FLTM', 'MOOR', 'MRSH', 'MRSHN', 'OVF', 'RSVT', 'SYSI', 'WHRL', 'AIRS', 'CAPG', 'CHN', 'CNLI', 'CNLN', 'CNLQ', 'CNLX', 'CUTF', 'DCKB', 'DTCH', 'FLTT', 'INLTQ', 'MFGN', 'MGV', 'PNDSF', 'RPDS', 'RSV', 'RSVI', 'STMQ', 'STMSB', 'SWMP', 'TNLC', 'WLL', 'WLLQ', 'WLLS', 'WTLD', 'WTLDI', 'WTRC', 'WTRH', 'BSNP', 'CNS', 'COLF', 'DEVH', 'FLDI', 'GASF', 'MILB', 'MNA', 'MVA', 'NVB', 'OILF', 'PEAT', 'QCKS', 'RES', 'RNGA', 'AGRC', 'AREA', 'BSND', 'CONT', 'FLD', 'GRAZ', 'GVL', 'INDS', 'LAND', 'LCTY', 'RESA', 'RESV', 'RGN', 'RGNE', 'RGNH', 'RGNL', 'SALT', 'SNOW', 'TRB', 'PPLF', 'PPLQ', 'PPLW', 'OILP', 'RTE', 'RYD', 'PTGE', 'RD', 'RDCUT', 'RDB', 'RDJCT', 'RJCT', 'RR', 'RRQ', 'STKR', 'TNL', 'TNLRD', 'TNLRR', 'TNLS', 'AIRB', 'AIRF', 'AIRH', 'ASYL', 'BDGQ', 'BLDA', 'BLDG', 'BLDO', 'BP', 'BRKS', 'BSTN', 'BTYD', 'BUR', 'CMPL', 'CMPLA', 'CMPMN', 'CMPO', 'CMPQ', 'CMPRF', 'CMTY', 'COMC', 'CRRL', 'CTHSE', 'CTRA', 'CTRF', 'CTRS', 'CVNT', 'DAMQ', 'DAMSB', 'DARY', 'DCKD', 'DCKY', 'DPOF', 'EST', 'ESTO', 'ESTR', 'ESTSG', 'ESTX', 'FNDY', 'FRM', 'FRMQ', 'FRMS', 'FRMT', 'GOSP', 'GOVL','GRVE', 'HMSD', 'HSE', 'HSEC', 'HSPL', 'HUT', 'HUTS', 'INSM', 'ITTR', 'JTY', 'LEPC', 'LNDF', 'LOCK', 'MFG', 'MFGB', 'MFGC', 'MFGCU', 'MFGLM', 'MFGM', 'MFGPH', 'MFGQ', 'MFGSG', 'ML', 'MLM', 'MLO', 'MLSG', 'MLSGQ', 'MLSW', 'MLWND', 'MLWTR', 'MN', 'MNAU', 'MNC', 'MNCR', 'MNCU', 'MNFE', 'MNN', 'MNQ', 'MNQR', 'MSSNQ', 'NOV', 'OBSR', 'OILJ', 'OILQ', 'OILR', 'OILT', 'OILW', 'PMPO', 'PMPW', 'PO', 'PPQ', 'PRN', 'PRNJ', 'PRNQ', 'PS', 'PSH', 'PSN', 'PSTP', 'QUAY', 'RDCR', 'RDIN', 'RKRY', 'RLGR', 'RNCH', 'RSD', 'RSGNL', 'RSTNQ', 'RSTPQ', 'RUIN', 'SCH', 'SCHA', 'SCHC', 'SCHL', 'SCHM', 'SCHN', 'SCHT', 'SECP', 'SHPF', 'SHSE', 'SLCE', 'SNTR', 'SPLY', 'STBL', 'STNB', 'STNC', 'STNE', 'STNF', 'STNM', 'STNR', 'STNS', 'STNW', 'SWT', 'TMB', 'TNKD', 'TRIG', 'TRMO', 'TWO', 'UNIP', 'UNIV', 'USGE', 'VETF', 'WALL', 'WEIR', 'WHRF', 'WRCK', 'WTRW', 'ADMF', 'AGRF', 'AIRQ', 'AQC', 'ARCHV', 'ART', 'ASTR', 'ATHF', 'ATM', 'BANK', 'BCN', 'BAR', 'BDLD', 'BLOW', 'BNCH', 'CFT', 'DLTA', 'DPR', 'DVD', 'FAN', 'FSR', 'GAP', 'HMCK', 'INTF', 'KRST', 'LAVA', 'LEV', 'NKM', 'NTK', 'NTKS', 'PAN', 'PANS', 'PLDR', 'RKFL', 'SAND', 'RK', 'RKS', 'SCRP', 'SDL', 'SINK', 'SLID', 'SLP', 'SPIT', 'SPUR', 'TAL', 'TRGD', 'TRR', 'CULT', 'GRVC', 'GRVO', 'GRVP', 'GRVPN', 'HTH', 'MDW', 'OCH', 'SCRB', 'VIN', 'VINS');
Deleting columns of irrelevant data. We dropped also the alternatenames column in order to put it in a separate table to prevent duplicate rows (of places that have multiple alternatenames).	ALTER TABLE geonames.locations drop column alternatenames, drop column admin1, drop column admin2, drop column admin3, drop column admin4, drop column population, drop column elevation;

In addition to the above queries we made some more queries to delete more irrelevant data and to normalize the DB.

Each entry in the locations table had an attribute for fclass and an attribute for fcode, but there is consistency between fcode and fclass. Therefor we created the table classandcodes to map between fcodes and fclasses and deleted the column fclass from the locations table in order to normalize the dataset and prevent duplicate data.

<u>Queries used during the application run (as defined procedures):</u>

In order to organize the code, we used procedures and called them from the code with suitable parameters.

Add a description to a location in the DB.	CREATE DEFINER=\`root\`@\`localhost\` PROCEDURE \`add_description\`(IN geonameid INT, IN descrip varchar(55), IN rating INT)
	BEGIN
	INSERT into geonames.usersdescriptions
	values(default, geonameid, descrip, rating, curdate());
	END

Get all the countries in the countries table. Complex query (order by).	CREATE DEFINER=\`root\`@\`localhost\` PROCEDURE \`get_all_countries\`() BEGIN SELECT country FROM countries ORDER BY country ASC; END
Get descriptions of a location of a given id (geonameid), order them by descending modifying date. Complex query (order by).	CREATE DEFINER=\`root\`@\`localhost\` PROCEDURE \`get_description\`(IN geoid INT) BEGIN SELECT geonameid, description, rating, date FROM geonames.usersdescriptions WHERE geonameid = geoid ORDER BY date desc; END

<table>
<tr>
<td>Get details of up to 500 locations which meet the next requirements:
- Belong to the given country
- Located in specific coordinations range (calculated by radial search)
- Are in distance of less than the given radius
- Their fcode belongs to the given userfcl (fclass)
Complex query (avg, inner join, left join, like, group by, order by).</td>
<td>

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `radial_search`(IN currLat DECIMAL(10,7), IN currLng DECIMAL(10,7), IN minLat DECIMAL(10,7), IN maxLat DECIMAL(10,7), IN minLng DECIMAL(10,7), IN maxLng DECIMAL(10,7),IN userfcl varchar(20), IN radius INT, IN country varchar(50))
BEGIN
SELECT locations.geonameid, name, latitude, longitude, alternateName, AVG(rating) as avgRate, fclass, meaning,
(6371 * acos(cos(radians(currLat)) * cos(radians(latitude)) * cos(radians(longitude)-radians(currLng)) + sin(radians(currLat)) * sin(radians(latitude))))
AS distance
FROM locations INNER JOIN classandcodes ON (locations.fcode = classandcodes.fcode)
INNER JOIN countries ON (countries.country = country)
LEFT JOIN usersdescriptions ON (locations.geonameid=usersdescriptions.geonameid)
LEFT JOIN alternatename ON (locations.geonameid=alternatename.geonameid)
WHERE (latitude BETWEEN minLat AND maxLat)
AND (longitude BETWEEN minLng AND maxLng)
AND userfcl LIKE CONCAT('%',fclass,'%')
AND countries.iso = locations.country
GROUP BY geonameid
HAVING distance < radius
ORDER BY distance ASC
LIMIT 500;
END
```

</td>
</tr>
</table>

Get details of up to 500 locations which meet the next requirements: - Belong to the given country - Located in specific coordinations range (calculated by radial search) - Are in distance of less than the given radius - Their fcode belongs to the given userfcl (fclass) - Their average rate is bigger/equal to the given score Complex query (avg, inner join, left join, like, in, nested query, group by, order by)	CREATE DEFINER=\`root\`@\`localhost\` PROCEDURE \`radial_search_rating\`(IN currLat DECIMAL(10,7), IN currLng DECIMAL(10,7), IN minLat DECIMAL(10,7), IN maxLat DECIMAL(10,7), IN minLng DECIMAL(10,7), IN maxLng DECIMAL(10,7), IN userfcl varchar(20), IN radius INT, IN score INT, IN country varchar(50)) BEGIN SELECT locations.geonameid, name, latitude, longitude, alternateName, AVG(rating) AS avgRate, fclass, meaning, (6371 * acos(cos(radians(currLat)) * cos(radians(latitude)) * cos(radians(longitude)-radians(currLng)) + sin(radians(currLat)) * sin(radians(latitude)))) AS distance FROM locations INNER JOIN classandcodes ON (locations.fcode = classandcodes.fcode) INNER JOIN countries ON (locations.country = countries.iso) LEFT JOIN usersdescriptions ON (locations.geonameid=usersdescriptions.geonameid) LEFT JOIN alternatename ON (locations.geonameid=alternatename.geonameid) WHERE (latitude BETWEEN minLat AND maxLat) AND (longitude BETWEEN minLng AND maxLng) AND userfcl LIKE CONCAT('%',fclass,'%') AND locations.country IN (select ISO from countries where Country = country) GROUP BY geonameid HAVING distance <= radius AND avgRate >= score ORDER BY distance ASC LIMIT 500; END

Code Structure:

SQLHandler.js – A JavaScript (Node.js) class that responsible on managing the communication with the DB server. It uses 'Bluebird' package in order to send queries and receive data asynchronously.
The class has a constructor which can be either hard-coded or given external parameters.
It creates a pool of connections to the DB server which send queries and receive data, while managing the amount of connections needed for those tasks.
The SQLHandler uses the procedures defined in the DB server in order to preform the needed queries.

server.js – A TypeScript (Node.js) that holds an SQLHandler object, and manages the communication with the front of the app. The server sends the initial information it gets from its' SQLHandler object to the client, and gives back the user input and requests to the SQLHandler. It also parses some of the user information before giving it to the SQLHandler and vice versa.

script.js – A TypeScript which operates the client. It receives the user input and presents the data given from the DB server to the user in a graphical manner. It includes 2 parts:
One is the user's "dashboard" which helps the user ask for the locations it wants and observe some textual data about them (distance from destination, rating, favorite places, etc.).
The second is an interactive map which presents all the locations and their information according to the user request.

insertingDescs.js – A TypeScript that inserts randomly chosen descriptions and ratings to different locations. It was made and used in order to have initial descriptions and ratings to some of the places.

Data Sources:

DB – We used data sets from GeoNames Data Base (https://www.geonames.org) as our data set.

Open Layers OSM map – In our web app we display an OSM map loaded from https://cdn.jsdelivr.net/gh/openlayers/openlayers.github.io@master/en/v6.4.3/css/ol.css.

Nominatim service – In order to convert the destination the user inputs into latitude and longitude coordination, we used the nominatim service by sending an HTTP request to https://nominatim.openstreetmap.org with the input from the user, and got the coordination as the result.

External Packages:

Mysql – A JavaScript package that enables connecting and managing the connections with a DB.
With this package we get an API to communicate, send queries, get information from the DB and much more.

Bluebird – A JavaScript package that enables async programing.
Useful in case of querying a DB, where async approach is needed in order to get the answers for the queries and process them.

Application Flow:

The user enters country, destination and radius, and chooses categories and ratings filter. Then, the input is sent to the server which parses it and makes some calculations and passes the data to the SQL handler.

The SQL handler calls the suitable method which in turn calls the suitable procedure (query) at the SQL server. The result of this query is all the locations that meet the users' requirements according to his input. The result is received by the SQL handler, which in turn passes the data to the server, the server creates an object containing this data and sends it to the client.

The results are displayed in a table and on a map. The user can click on the map or on the table to see a popup with details about the selected location, and can also see reviews, add his own review and add this location to a list of the locations he wants to visit.

At any point, the user can perform a new search.