**TheFoodCourt**

**Software Documentation**

**Database Systems Course (2018-2019) – Final Project**

Ram Shimon (203537246)

Yuval Weiss (204330740)

Nitzan Yogev (312433279)

Michael Khaitov (308401082)

**DB Scheme structure** ………………………………………………………………………. **2**

**DB Optimizations Performed** ………………………………………………………. **4**

**Queries** …………………………………………………………………………………………………… **5**

**Code Structure** …………………………………………………………………………………. **12**

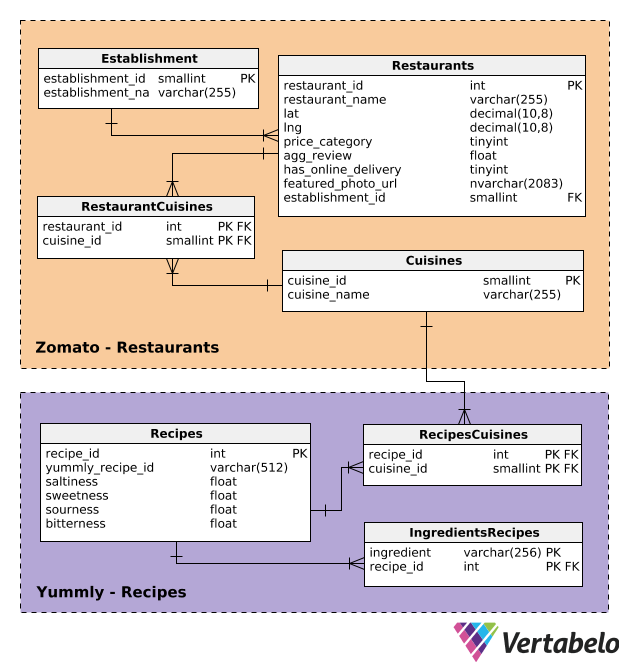
**API Used** ……………………………………………………………………………………………. **13**

**External Packages** …………………………………………………………………………… **13**

**General Flow of the App** ……………………………………………………………… **14**

**DB Scheme structure**

DB Scheme:



For more details, see `CREATE-DB-SCRIPTS.sql`.

Number of rows per table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Table Name** | Establishments | Restaurants | RestaurantsCuisines | Cuisines | Recipes | RecipesCuisines | IngredientsRecipes |
| **# rows** | 34 | 5,039 | 9,274 | 138 | 18,734 | 21,042 | 164,127 |

The design choice of our database was inspired by both the APIs we used (“Zomato”, “Yummly”) and our website’s design and purpose. As seen in the above image (of the DB scheme), we can split our tables into two parts, one for each API. The two parts are connected through the **Cuisines** table. To make sure that the table values (which are populated from the “Zomato” API) match, during our database population we have created a translation map to translate values of cuisine names between “Zomato” and “Yummly”. We will shortly describe each table design choices:

**Restaurants:** Each row in this table is a projection of values in the “Zomato” API restaurants tables. Only relevant columns for our website were stored in our database. We used “Zomato’s” restaurant\_id as our key. This enables us to find the restaurant easily (if needed in the future) in the “Zomato” API, without storing another column. The featured photo URL is bound by size to the maximum size of URLs in Microsoft’s Internet Explorer.

**Establishments:** The table is identical to the establishments table in “Zomato” API. Each restaurant has an establishment\_id associated with it.

**Cuisines:** The table is identical to the cuisines table in the “Zomato” API. Each restaurant can have multiple cuisine types, and therefore we use the following table to capture that relation.

**RestaurantsCuisines**: Used to connect between restaurants and their corresponding cuisines.

**Recipes:** Each row here is a projection of values in the “Yummly” API recipes tables. Since “Yummly” use an identifier which is a string (the recipe name), which might be long, we have added an AUTO INCREMENT id column as a primary key for each recipe. This is the primary key that is also linked in the following table. This allows us for faster search (small int value instead of long strings) and saves storage space.

**IngredientsRecipes:** From each “Yummly” recipe, we extract its ingredients and save them up in this table. Since ingredients can appear multiple times in the table, as well as each recipe, the primary key is a tuple of both.

**RecipesCuisines:** Used to connect between recipes and their cuisines. The cuisine data is stored in each recipe in the “Yummly” API, and we use the translation map mentioned above to match it to the relevant row in the **Cuisines** table.

**DB Optimizations Performed**

**Indexes**:

When designing the table and its indexes, we used one main assumption as a guidline:

*All online queries (queries performed by clients) are `SELECT` queries, and not `INSERT` or `UPDATE` queries.*

This key assumption was crucial to our index choice. Since each added index increases insert and update time (due to modification needed after the query to update the index), the assumption above enabled us to add indexes without impacting online performance in terms of query time. We did keep in mind that still, indexes needed to be loaded into memory, thus we still needed to pick our indexes intelligently.

The indexes mentioned below are additional indexes to those created automatically for primary keys. Those are not mentioned here.

* **flavor\_index:** the index improves query performance for the `restaurant\_by\_taste` query (see next chapter). The index is composed of four columns – the exact ones used in the query. Since the query searches up ranges of flavors (e.g. `sweetness BETWEEN X AND Y) we used a BTREE index.
* **restaurant\_location\_index:** the index improves query performance for all restaurant queries using location filtering. We used a normal BTREE index (and not SPATIAL), since our location filtering for restaurant is simple – a square search around a given location, only needing range querying for the lat,lng columns, and not complex geometric queries.
* **restaurant\_review\_index:** the simple index is used to optimize restaurant filtering by user’s review averages (the agg\_review column). Since the query is using comparisons (e.g. `greater than` we used a BTREE for this index as well.
* **restaurant\_price\_index:** similar to the above, but for the price\_category column

We have also considered a hash index over the **ingredient** column in the **IngredientsRecipes** table, in order to optimize queries that look up for a specific ingredient value. However, considering the existing BTREE index over this column, the size of the table and the time the queries take without adding the index, we have decided that the index does not justify the added memory usage of it.

**Queries**

We list below the main query our application uses. Some queries are not mentioned here, but can be viewed in the **sql\_queries.py** python script. These queries are small queries with mainly functionality which is transparent to the users. For each query below we show the SQL query code, we describe the query and show sample results.

|  |
| --- |
| **1: Restaurant Query Wrapper** |
| SELECT restaurant\_name,         lat,         lng,         price\_category,         agg\_review,         has\_online\_delivery,         featured\_photo\_url,         establishments.establishment\_id,         establishment\_name  FROM   establishments,         (%s) AS source  WHERE  establishments.establishment\_id = source.establishment\_id  AND    %f <= source.lat  AND    source.lat <= %f  AND    %f <= source.lng  AND    source.lng <= %f  AND    source.price\_category = %s  AND    source.agg\_review >= %s  AND    source.has\_online\_delivery = %s  AND    source.establishment\_id = %s |
| This query is a modular query used to wrap queries that return restaurant values. The input query is added as the **(%s)** with alias **source**. Each **AND** condition is added only if the filtering condition is added. The query adds the establishment name to the restaurant from the **Establishments** table. The modularity is added through the **restaurant\_query\_builder** function in the **db.py** python file. |
| Example result:  **"Dunkin Donuts" "40.71504167" "-74.00769167" "2" "3" "0" "https://b.zmtcdn.com/data/res\_imagery/16758439\_CHAIN\_212347c0633d3b3c9b006a875bd28882\_c.png" "281" "Fast Food"** |

|  |
| --- |
| **2: Discover new cuisines from cuisine** |
| SELECT   cuisinetocuisine.cuisine\_id,           cuisinetocuisine.cuisine\_name,           ( cuisinefreq / cuisine\_receipe\_count.receipe\_weight ) AS match\_value  FROM     (                     SELECT    cuisines.cuisine\_id,                               cuisines.cuisine\_name,                               *Count*(cuisines.cuisine\_id) AS cuisinefreq                     FROM      (                                         SELECT    ingredient,                                                   *Count*(ingredient) AS maxingredients                                         FROM      cuisines                                         LEFT JOIN recipescuisines                                         ON        cuisines.cuisine\_id = recipescuisines.cuisine\_id                                         LEFT JOIN ingredientsrecipes                                         ON        recipescuisines.recipe\_id = ingredientsrecipes.recipe\_id                                         WHERE     cuisines.cuisine\_id = %s                                         GROUP BY  ingredientsrecipes.ingredient) AS commoningredients                     LEFT JOIN ingredientsrecipes                     ON        commoningredients.ingredient = ingredientsrecipes.ingredient                     LEFT JOIN recipescuisines                     ON        ingredientsrecipes.recipe\_id = recipescuisines.recipe\_id                     LEFT JOIN cuisines                     ON        recipescuisines.cuisine\_id = cuisines.cuisine\_id                     WHERE     cuisines.cuisine\_id <> %s                     GROUP BY  cuisines.cuisine\_id) AS cuisinetocuisine,           (                    SELECT   cuisine\_id,                             count(cuisine\_id) AS receipe\_weight                    FROM     recipescuisines                    GROUP BY cuisine\_id) AS cuisine\_receipe\_count  WHERE    cuisine\_receipe\_count.cuisine\_id = cuisinetocuisine.cuisine\_id  ORDER BY match\_value DESC limit 3 |
| This query, given a cuisine (**cuisine\_id**), finds new cuisines (i.e. cuisines which are not the input cuisine) which share the greatest number of ingredients for which the cuisines’ recipes have. Because some cuisines might have more recipes than other cuisines, we normalize the results with the number of recipes each cuisine has, and calculate it as a **match\_value**. We return the top three matching cuisine with the highest **match\_value**. The results of the query are cached in a cache mechanism we implemented in the backend (see “Code Structure” section for more details). |
| Example result, given input cuisine “Greek” (**cuisine\_id = 45**):   |  |  |  | | --- | --- | --- | | **cuisine\_id** | **cuisine\_name** | **match\_value** | | **70** | **Mediterranean** | **9.4146** | | **147** | **Moroccan** | **8.8618** | | **491** | **Cajun** | **8.1741** |   . |

|  |
| --- |
| **3: Query restaurants by ingredients** |
| SELECT restaurants.\*  FROM   restaurants,         restaurantscuisines,         (SELECT recipescuisines.cuisine\_id          FROM   ingredientsrecipes,                 recipescuisines,                 cuisines,                 (SELECT cuisine\_id,                         **Count**(recipe\_id) cuisine\_recipe\_cnt                  FROM   recipescuisines                  GROUP  BY cuisine\_id) AS CuisineRecipesCount          WHERE  recipescuisines.recipe\_id = ingredientsrecipes.recipe\_id                 AND ingredientsrecipes.ingredient = '%s'                 AND CuisineRecipesCount.cuisine\_id = recipescuisines.cuisine\_id          GROUP  BY recipescuisines.cuisine\_id          ORDER  BY **Count**(recipescuisines.cuisine\_id) / cuisine\_recipe\_cnt DESC          LIMIT  3) AS CuisinesByIngredient  WHERE  restaurantscuisines.cuisine\_id = CuisinesByIngredient.cuisine\_id         AND restaurants.restaurant\_id = restaurantscuisines.restaurant\_id |
| Given an ingredient (queried before from the **IngredientsRecipes** table), this query returns restaurants which cuisine types match the top 3 cuisines that are linked to the given input ingredient, adjusted by weight (as in the previous query) of the number of recipes for each cuisine. This query is then wrapped with the restaurant query wrapper (query 1). |
| Example result, given input ingredient “shimeji mushrooms”:  **"16766735" "Ginza Japanese Restaurant" "40.59290833" "-73.95017222" "3" "3.7" "0" "" "21"** |

|  |
| --- |
| **4: Query restaurants by taste** |
| SELECT restaurants.\*  FROM   restaurants,         restaurantscuisines,         (                SELECT cuisine\_id                FROM   (                                SELECT   recipescuisines.cuisine\_id,                                         (*Count*(recipescuisines.cuisine\_id)/cnt) AS weight                                FROM     recipes,                                         recipescuisines,                                         (                                                  SELECT   recipescuisines.cuisine\_id,                                                           *Count*(recipescuisines.cuisine\_id) AS cnt                                                  FROM     recipes,                                                           recipescuisines                                                  WHERE    recipes.recipe\_id = recipescuisines.recipe\_id                                                  GROUP BY recipescuisines.cuisine\_id ) AS numrecipespercuisine                                WHERE    saltiness BETWEEN %s  AND      sweetness BETWEEN %s  AND      sourness BETWEEN %s  AND      bitterness BETWEEN %s  AND      recipescuisines.recipe\_id = recipes.recipe\_id                                AND      recipescuisines.cuisine\_id = numrecipespercuisine.cuisine\_id                                GROUP BY recipescuisines.cuisine\_id                                ORDER BY weight DESC) AS matchingtastes                WHERE  NOT EXISTS                       (                              SELECT \*                              FROM   (                                              SELECT   recipescuisines.cuisine\_id,                                                       (count(recipescuisines.cuisine\_id)/cnt) AS weight                                              FROM     recipes,                                                       recipescuisines,                                                       (                                                                SELECT   recipescuisines.cuisine\_id,                                                                         count(recipescuisines.cuisine\_id) AS cnt                                                                FROM     recipes,                                                                         recipescuisines                                                                WHERE    recipes.recipe\_id = recipescuisines.recipe\_id                                                                GROUP BY recipescuisines.cuisine\_id ) AS numrecipespercuisine                                              WHERE    saltiness BETWEEN %s  AND      sweetness BETWEEN %s  AND      sourness BETWEEN %s  AND      bitterness BETWEEN %s  AND      recipescuisines.recipe\_id = recipes.recipe\_id                                              AND      recipescuisines.cuisine\_id = numrecipespercuisine.cuisine\_id                                              GROUP BY recipescuisines.cuisine\_id                                              ORDER BY weight DESC limit 5) AS notmatchingtastes                              WHERE  matchingtastes.cuisine\_id = notmatchingtastes.cuisine\_id ) limit 3) AS cuisinesbytaste  WHERE  restaurants.restaurant\_id = restaurantscuisines.restaurant\_id  AND    restaurantscuisines.cuisine\_id = cuisinesbytaste.cuisine\_id |
| Similar to the third query, this query returns restaurants as well. However, here the input are the user’s flavors preferences (i.e. whether he likes/dislikes salt/sweet/sour/bitter food). This query searches for the cuisines that matches the user’s tastes, but also don’t match the tastes opposite of the user’s. This is done by filtering out such cuisines. All of the values are weighted as well (similar to the above queries) to account for cuisines with high/low number of recipes. Finally, this query can be then wrapped by the restaurant query wrapper. |
| Example result, given input (dislike salty food, likes sweet food, dislikes sour food, dislikes bitter food):  **"16761060" "Bouchon Bakery & Cafe" "40.76841410" "-73.98270370" "4" "4.3" "0" "https://b.zmtcdn.com/data/res\_imagery/16761060\_RESTAURANT\_ce0cc676161eedd94d28f01a5007a958\_c.jpg" "1"** |

|  |
| --- |
| **5: Find unique ingredients of cuisine** |
| SELECT \*  FROM   (                  SELECT   ingredient,                           *Count*(ingredient)                  FROM     ingredientsrecipes,                           recipescuisines                  WHERE    ingredientsrecipes.recipe\_id = recipescuisines.recipe\_id                  AND      recipescuisines.cuisine\_id = %d                  GROUP BY ingredient                  ORDER BY count(ingredient) DESC) AS ingredientsofcuisine  WHERE  NOT EXISTS         (                SELECT \*                FROM   (                                SELECT   ingredient                                FROM     ingredientsrecipes,                                         recipescuisines                                WHERE    ingredientsrecipes.recipe\_id = recipescuisines.recipe\_id                                AND      recipescuisines.cuisine\_id <> %d                                GROUP BY ingredient                                ORDER BY count(ingredient) DESC limit %d) AS ingredientsofothercuisines                WHERE  ingredientsofothercuisines.ingredient = ingredientsofcuisine.ingredient ) limit 10 |
| Given an input cuisine, this query finds the top ingredients that are both common for that cuisine, but are not common in general (thus, are associated with the cuisine). The query, after ordering the ingredients by how they are common for that cuisine, filters out the most common ingredients of cuisines which are not the input cuisine. If the query with high value of filtering doesn’t return a value, the backend server automatically performs a second query with a smaller number of filtering. This query results, similar to query number 2, are cached in the backend server (see “Code Structure” section). |
| Example result, given input “Greek cuisine” (**cuisine\_id = 156**):   |  |  | | --- | --- | | **ingredient** | **Count(ingredient)** | | **feta** | **26** | | **dill** | **22** | | **pitted kalamata olives** | **20** | | **greek seasoning** | **14** | | **ground lamb** | **12** | | **hummus** | **11** | | **phyllo dough** | **9** | | **crumbled feta** | **9** | | **chopped fresh mint** | **8** | | **roasted red peppers** | **8** |   . |

|  |
| --- |
| **6: Optimal franchise to open query** |
| SELECT restaurant\_name  FROM   (SELECT restaurants.\*,                 restaurantscuisines.cuisine\_id          FROM   restaurants,                 restaurantscuisines          WHERE  EXISTS (SELECT \*                         FROM   (SELECT \*                                 FROM   (SELECT restaurant\_name                                         FROM   restaurants                                         GROUP  BY restaurant\_name                                         HAVING **Count**(restaurant\_name) > 10) AS                                        Franchises                                 WHERE  NOT EXISTS (SELECT \*                                                    FROM   (SELECT restaurant\_name                                                            FROM   restaurants                                                            WHERE                                                   lat BETWEEN %f AND %f                                                   AND lng BETWEEN %f AND                                                       %f) AS                                                           LocationRestaurants                                                    WHERE                                            Franchises.restaurant\_name =         LocationRestaurants.restaurant\_name))         AS         FranchisesNotInLocation         WHERE  restaurants.restaurant\_name =         FranchisesNotInLocation.restaurant\_name)         AND restaurants.restaurant\_id = restaurantscuisines.restaurant\_id) AS         OptionalFranchises  WHERE  NOT EXISTS (SELECT \*                     FROM   (SELECT cuisine\_id                             FROM   restaurants,                                    restaurantscuisines                             WHERE  lat BETWEEN %f AND %f                                    AND lng BETWEEN %f AND %f                                    AND restaurants.restaurant\_id =                                        restaurantscuisines.restaurant\_id                             GROUP  BY cuisine\_id                             ORDER  BY **Count**(cuisine\_id) DESC                             LIMIT  15) AS CuisinesInLocation                     WHERE  CuisinesInLocation.cuisine\_id =                           OptionalFranchises.cuisine\_id)  GROUP  BY restaurant\_name |
| This unique query, given a location (**lat,lng**) gives suggestion on which franchises should be opened in that location in order to maximize success. First, franchises are defined by restaurants for which there are more than 10 branches of. The query searches for franchises that are not already existing with a certain L1 distance from the given location (to avoid opening two McDonald’s in the same location for example), and then filters out franchises that cuisine type is one of the top 15 cuisine types in that location (so the new franchise will give a new value). |
| Example result, given input “Times Square Location” (**lat: 40.758899, lng: -73.9873197**):   |  | | --- | | **Restaurant\_name** | | **Kennedy Fried Chicken** | | **Golden Krust** | | **Taco Bell** |   . |
| **7: Find common ingredients with a given ingredient** |
| SELECT   ingredient,           *Count*(ingredient)  FROM     recipes,           ingredientsrecipes  WHERE    recipes.recipe\_id = ingredientsrecipes.recipe\_id  AND      EXISTS           (                  SELECT re.recipe\_id                  FROM   recipes            AS re,                         ingredientsrecipes AS ir                  WHERE  ir.ingredient = %s                  AND    re.recipe\_id = ir.recipe\_id                  AND    recipes.recipe\_id = re.recipe\_id)  AND      ingredientsrecipes.ingredient <> %s  GROUP BY ingredient  ORDER BY count(ingredient) DESC limit 10 |
| Given an input ingredient, this query finds the ingredients that appear the most in recipes where the input ingredient is used. The query returns the 10 most common ingredients. This allows users who want to make recipes with their favorite ingredient to know which ingredient usually appears with it, so they can stock them to maximize recipe options! |
| Example result, given input “tequila”:   |  |  | | --- | --- | | **ingredient** | **Count(ingredient)** | | **lime juice** | **32** | | **salt** | **31** | | **lime** | **25** | | **triple sec** | **21** | | **olive oil** | **17** | | **fresh lime juice** | **13** | | **jalapeno** | **12** | | **agave nectar** | **11** | | **limes** | **11** | | **ice** | **11** |   . |

**Code Structure**

Our code is split in to three parts: DB Population scripts, Frontend and Backend. We will describe in short the structure of each part:

**DB Population Scripts [Python, MySQL]:**

DB Population scripts are simple standalone python scripts. There are two main scripts – one for the “Zomato” API and one for the “Yummly” API. Each script was created to match the services given by the relevant API. The connection to the API is done using GET requests through the `requests` python library. We use json and sometimes decoding to process the values we get from the API and then MySQLdb to store the values in our databases.

**Frontend [HTML, CSS, JS]:**

The frontend is comprised from two parts. The first one is the JavaScript controller file, and the second one is the HTML web page.

Our site contains a single HTML page: ‘TheFoodCourt.html’, which contains the site’s core layout and the DOMs (Document Object Models) that are bound to the site’s controller. The visual components on the site were built by using both HTML’s basic tags and CSS.

‘fcController.js’ is the controller attached to our site. It binds our application data to the attributes of the HTML DOM elements. Its functions are invoked by the site’s input forms and button click events. Then, it gets the required data from the backend: the data retrieval is done by using Fetch API (an API which provides a generic Request and Response objects). Finally, the retrieved data is displayed to the user by updating the HTML page asynchronously (using angular).

**Backend [Python, Flask, MySQL]:**

The backend is comprised from two parts. The first one is our database – a MySQL database on the university servers. The second is comprised from the backend server. Our backend server is made up of a few python scripts. The main one being the `server.py` script which has all the “server” functionality (all the Flask functionality). Each query against the database is done through the “db.py” file, which is managed by a class we have created so there is only one cursor working with the server at any given time. All the queries that include user input are protected against invalid inputs, especially SQL injection – either by the cursor exposed functionality of the MySQLdb library, or by checks that we apply when the integration of the user input is done before the query. The SQL queries themselves are stored in another python script – ‘sql\_queries.py’. We have introduced two special functionalities to the server as well:

1. **Caching** – some queries are cached. This means that when the query first executes, we store the result in memory, and make it valid for 24 hours. Therefore, if at any time in these 24 hours the same query is requested, we get the value immediately from the memory instead of running it through the database. This allows for much faster execution time, and reduces load on our database. The queries which are cached are chosen carefully, and only queries which have high probability to reoccur are cached. This is done to prevent saving a lot of query results in memory, since this won’t be efficient. We have decided ono a timed cache since user experience won’t be affected if the result of these queries will be “outdated” by at most a day. Even if there was an update to the relevant tables in that 24 hour period, most of the time the query results won’t change, and if they do – they won’t change significantly. Therefore, limiting the cache persistence to 24 hours was a good balance between not using a cache and using a cache that is updated every time the source tables are updated.
2. **Logging** – we use a logging system to log every request by users and errors. Each exception that occurs during the server run is stored as an error with the exception value and the location where it happened. This is true for faulty SQL queries as well. As stated, apart from the errors, we also store every user generated request to the server. This allows later to analyze usage and create statistics of popular (and unpopular) functionalities and more, to improve the website in terms of usability, reach and performance. The error logs can help detect bugs and unexpected or unpredicted user interactions, which will help reduce bugs and improve performance and usability.

**API Used**

We use two main APIs for our website:

1. “Zomato” <https://developers.zomato.com/> - for restaurant data.
2. “Yummly” <https://developer.yummly.com/> - for recipe data.

Apart from these APIs, we also use:

1. “MapBox” <https://www.mapbox.com/> - to display maps in the website.

**External Packages/Libraries**

We use the following external packages:

**DB Populations:**

1. Python:
   1. ‘requests’ – to connect to the web APIs.
   2. ‘MySQLdb’ – to connect to the MySQL database.

**Frontend:**

1. HTML
2. CSS
   1. angucomplete CSS stylesheet
   2. datatables.bootstrap4 CSS stylesheet
   3. Bootstrap CSS stylesheet
   4. bootstrap-toggle CSS stylesheet
   5. MapBox CSS stylesheet
3. Javascript:
   1. JQuery Javascript library - basic library for JS, required for other libraries.
   2. Angular Javascript library – web application framework.
   3. JQuery dataTables – basic library used by the other data tables’ libraries.
   4. angucomplete – used for displaying the auto complete results for ingredients.
   5. angular-datatables – used to implement the data tables in the site.
   6. datatables.bootstrap4 – used to implement the data tables in the site.
   7. Bootstrap – used to create the site’s visual components.
   8. bootstrap-toggle – used to create toggle buttons.
   9. MapBox Javascript library

**Backend:**

1. Python:
   1. ‘MySQLdb’ – to connect to the MySQL database.
   2. ‘simplejson’ – to process jsons.
   3. ‘logging’ – to implement the logging feature mentioned above.
   4. ‘flask’ – to create the backend server.
   5. ‘datetime’ – to manage the timed cache mentioned above.
2. MySQL

**General Flow of the App**

When opening the frontend application, the site’s basic layout is loaded. When the user invokes a controller function (by inputting text into an input form, or by selecting different filters and clicking the ‘Search’ button). Then, the controller generates the proper request path for the required query, and requests the query’s result from the backend. The controller’s data retrieval handling is asynchronous: while the backend processes the request the controller keeps operating normally. Finally, when the backend’s response is received, the controller uses the DOMs bound to it in the HTML page and displays the data to the user.

**(AND THEN WE WILL CONTINUE WITH THE BACKEND)**