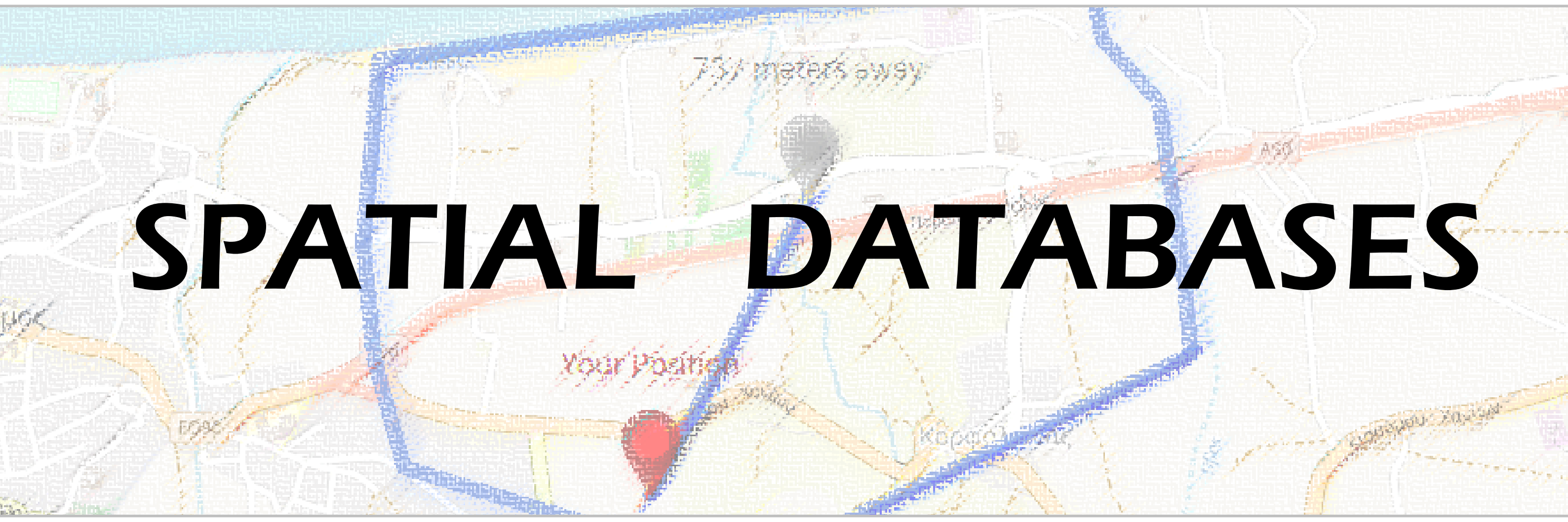
Exercise 13: Final Project

By Christina Zorenböhmer

05.03.2022



Full project documentation, code files, SQL dump, and data is

available on GitHub at

https://github.com/Christina1281995/spatial_database_festival_backend

To run this program

1. Create a database called 'festival' in postgres
2. Download the three python files in the src folder (if required, install any python modules)
3. Run main.py from your preferred IDE e.g. PyCharm

Running 'main.py' takes care of your connection to the database, the setup of all tables, PostGIS extension, and data.

Context

The initial motivation for this project was the final project of a 'Spatial Databses' course at the University of Salzburg. The aim is to create a spatial database that can serve as the backend to a festival, specifically it should:

- Serve as database backend to organise the booths, roller coasters, tents etc. (Positions, opening times, ...) as well as other facilities (garbage bins, toilets etc)
- User-specific, dynamic queries that return which events take place or facilities that are close to the current visitor’s position.
- User-specific and dynamic queries about the festival (digital maps, lists of events etc)

Psycopg2

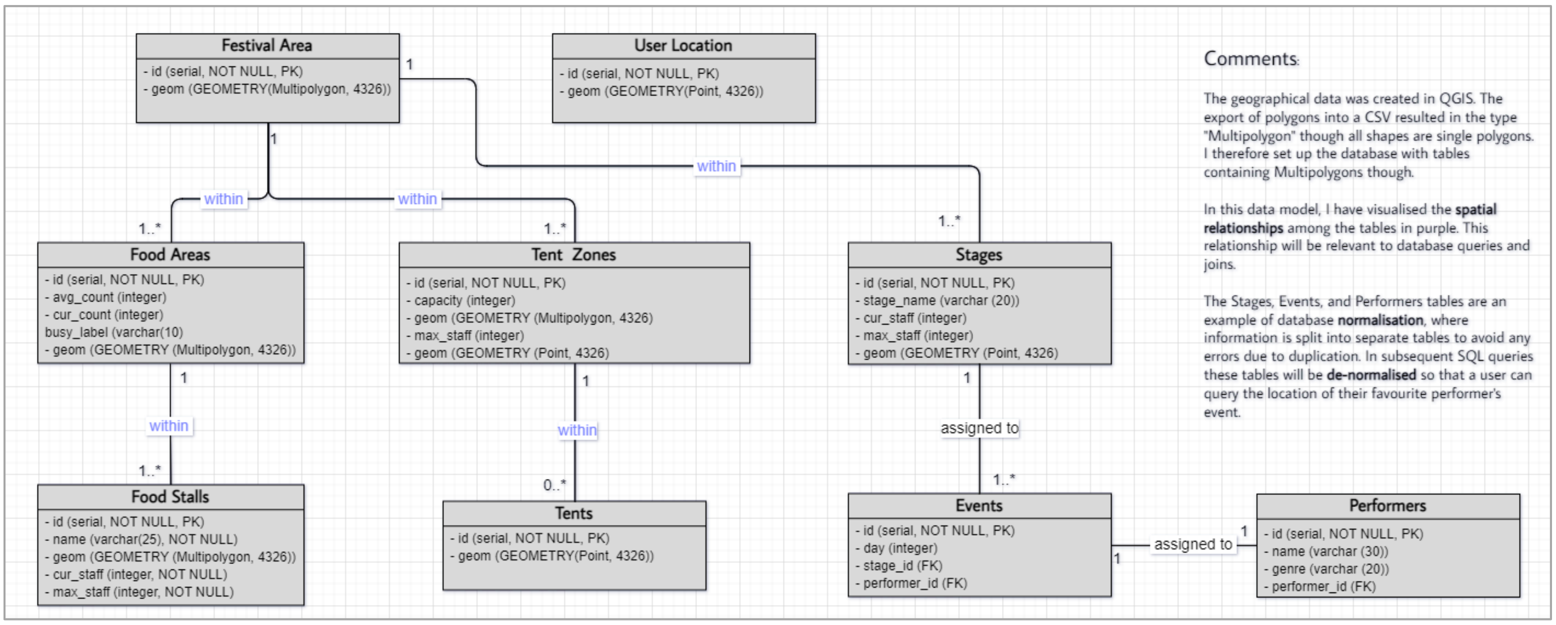
To create dynamic queries into a spatial database, there must be some form of user interaction. At the outset of this project I had no previous experience with any other postgres interaction points other than PG admin and QGIS. Both of those are direct communication channels with the database, where SQL statements are directly entered. In order to mediate, narrate, control and guide a user around the data that is in such a spatial database, there must be an additional step in between the database (or PG admin) and the user. The user should not need to worry about the queries and data flow to and from the database. Likewise, there should be minimal risk of the user damaging the data or inserting erroneous data. For all of those reasons, I searched the internet for an adequate "in-between" tool, and found psycopg2. It is the main database connector for the python language. By making use of it, I was able to work with both the user and the data. The finished programme consists of 3 python files:

1. The **"main.py"** file which only contains the execution logic for the programme. It calls the relevant functions.
2. The **"functions.py"** file contains all the functionalities needed to set up the database and execute the queries according to the user input. It also includes the function for a tkinter pop-up window with a map showing some relevant spatial information of the query.
3. The **"locations.py"** file containing some spatial information in the format that tkinter requires it for the map (which is a very different format than the geom datatypes available with postgres). If the geometry could have been converted from the

database tables' geom columns I would rather have done that, but I couldn't quite work out how to reformat it so much in an automated way. All three code files are saved here in the GitHub repository.

Data Model

A first concept for the database was created through simple brainstorming on the topic of "what would be a good use for an app during a festival?" and "what would I use an app for at a festival?". Based on those notes, a few key entities were identified: tents, tent areas, stages, performers, events, food stalls, food areas. With those entities in mind a set of concrete questions was developed which queries into the database should answer. With those entities and the questions / plain-text queries in mind, a data model was sketched (see image below). This data model is intentionally kept simple. It is only as complex as it needs to be for the intended queries. As such, it is **"fit for purpose"**. The decisions for the cardinalities are, of course, debatable. They were determined by my own decision: e.g. that a performer only belongs to one stage and one event.



Festival Area
- id (serial, NOT NULL, PK)
- geom (GEOMETRY(Multipolygon, 4326))
User Location
- id (serial, NOT NULL, PK)
- geom (GEOMETRY(Point, 4326))
1
1
within
within
within
1..*
1..*
1..*
Food Areas
- id (serial, NOT NULL, PK)
- avg_count (integer)
- cur_count (integer)
busy_label (varchar(10)
- geom (GEOMETRY (Multipolygon, 4326))
Tent Zones
- id (serial, NOT NULL, PK)
- capacity (integer)
- geom (GEOMETRY (Multipolygon, 4326)
- max_staff (integer)
- geom (GEOMETRY (Point, 4326)
Stages
- id (serial, NOT NULL, PK)
- stage_name (varchar (20))
- cur_staff (integer)
- max_staff (integer)
- geom (GEOMETRY (Point, 4326)
1
within
1..*
1
within
0..*
1
assigned to
1..*
Food Stalls
- id (serial, NOT NULL, PK)
- name (varchar(25), NOT NULL)
- geom (GEOMETRY (Multipolygon, 4326))
- cur_staff (integer, NOT NULL)
- max_staff (integer, NOT NULL)
Tents
- id (serial, NOT NULL, PK)
- geom (GEOMETRY(Point, 4326))
Events
- id (serial, NOT NULL, PK)
- day (integer)
- stage_id (FK)
- performer_id (FK)
1
assigned to
1
Performers
- id (serial, NOT NULL, PK)
- name (varchar (30))
- genre (varchar (20))
- performer_id (FK)

Comments:

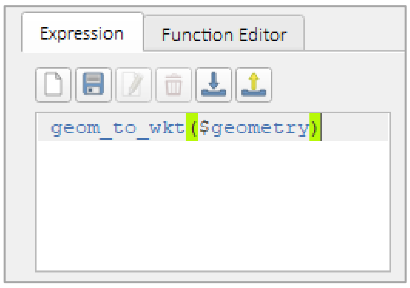
The geographical data was created in QGIS. The export of polygons into a CSV resulted in the type "Multipolygon" though all shapes are single polygons. I therefore set up the database with tables containing Multipolygons though.

In this data model, I have visualised the **spatial relationships** among the tables in purple. This relationship will be relevant to database queries and joins.

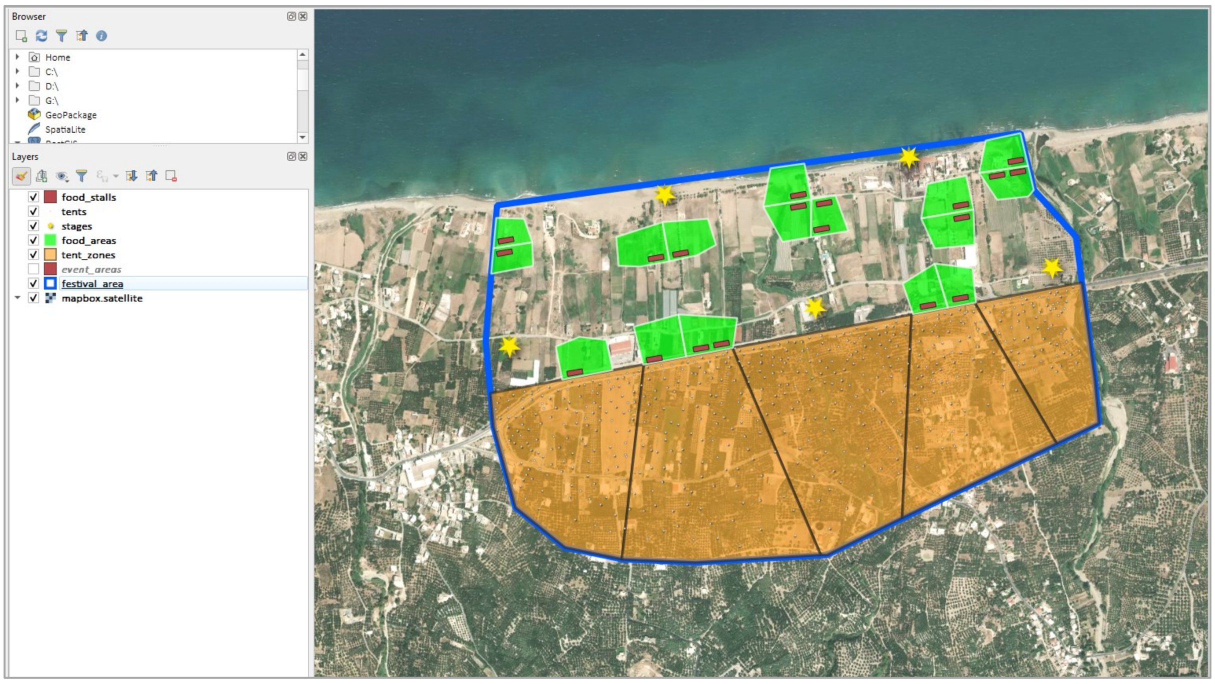
The Stages, Events, and Performers tables are an example of database **normalisation**, where information is split into separate tables to avoid any errors due to duplication. In subsequent SQL queries these tables will be **de-normalised** so that a user can query the location of their favourite performer's event.

Setting up the Database, the Tables, and the Data

QGIS was used to create fictional data for the database. The tables were set up according to the data model and the geom columns were calculated using QGIS's field calculator and the expression shown to the right. A screenshot of the QGIS project is shown below. Once the data was satisfactorily created, it was exported as a CSV file and uploaded to GitHub where it could later be accessed through the python programme.

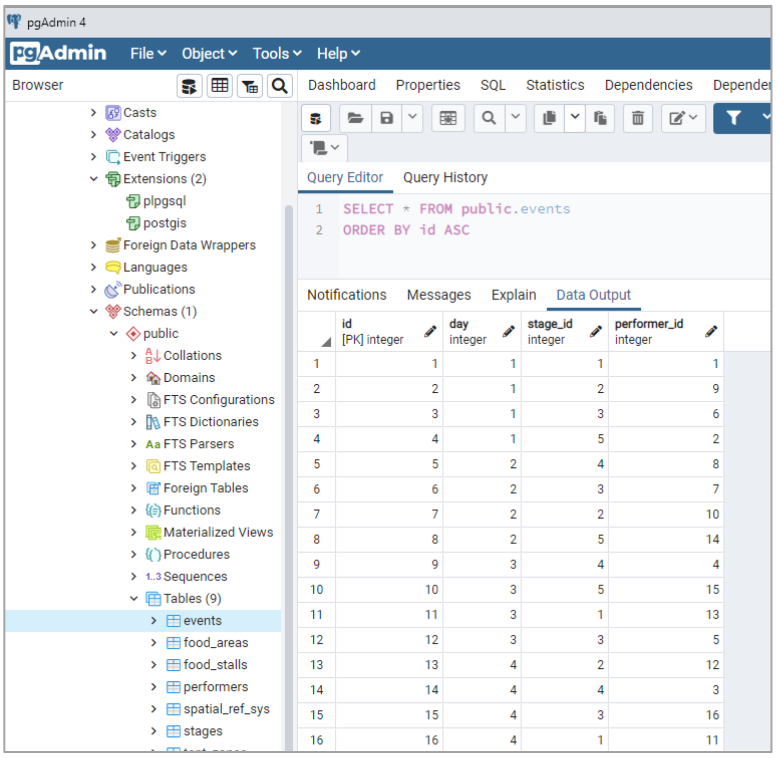


```
geom_to_wkt($geometry)
```



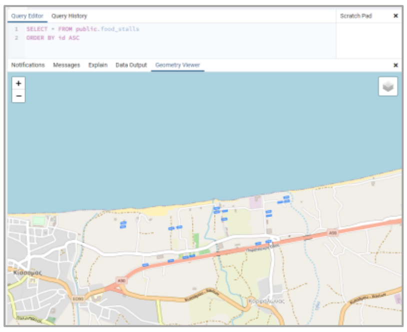
The only step conducted in PG admin was to create a database with the name "festival". The remainder of the setup is implemented with psychopg2. A set of functions is executed to connect with the database, check for the existence of a PostGIS extension and the database tables. Then, both are added and the tables are filled with data taken from CSV files stored online in this GitHub repository. Since there are quite a few functions, a summarized list for reference is placed at the bottom of this documentation. Please also refer to the functions.py file itself.

While creating the programme, PG admin was used to check that all inserts and queries work. After creating the function to download the CSV data from GitHub, to format it into a pandas DataFrame and then to insert it into the correct table, PG Admin was used to validate that the data had indeed arrived in the table in the correct format. The image to the right shows the events table, which contains the events' IDs, the day of the events, and as foreign keys it also refers to the respective stage and performers.



```
SELECT * FROM public.events
ORDER BY id ASC
```

	id [PK] integer	day integer	stage_id integer	performer_id integer
1	1	1	1	1
2	2	1	2	9
3	3	1	3	6
4	4	1	5	2
5	5	2	4	8
6	6	2	3	7
7	7	2	2	10
8	8	2	5	14
9	9	3	4	4
10	10	3	5	15
11	11	3	1	13
12	12	3	3	5
13	13	4	2	12
14	14	4	4	3
15	15	4	3	16
16	16	4	1	11



```
SELECT * FROM public.food_stalls
ORDER BY id ASC
```

Thanks to the option of viewing the geom column directly in PG Admin, I was also easily able to validate that the geometry was created correctly.

User Interaction

Although not required, a central element for this project is the user interaction. In the following section I detail the logic of the available queries. The tables below show the different interactions available to the user. Please note that the user's input is represented in the queries through the "%s" symbol. In later stages of this project, the SQL commands were re-written into preprared statements as plans.

Decision 1 - "Who is using this app?"

Option	Value
1	Festival Staff
2	Festival Visitor

Decision 2 - "What would you like to do right now?"

For Staff: Topic Food		
1	*<u>Find out if more members of staff are needed at any food stalls.</u>* Calculate the difference between current staff and maximum staff for all of the food stalls. Then check if any food stall that don't have a full team are very busy (i.e. need more help). Return name of food stall and how many more staff members can go there.	SELECT name, max_staff - cur_staff FROM food_stalls WHERE id IN (SELECT id FROM food_stalls WHERE cur_staff < max_staff) AND id IN (SELECT s.id FROM food_stalls AS s, (SELECT geom FROM food_areas WHERE busy_label = 'average') AS a WHERE ST_Intersects(s.geom, a.geom));
2	*<u>Update the number of staff members at a food stall.</u>* First give user a list of name and id of all food stalls, then show user the current staff number for the food stall they have selected. Then update the current staff number according to their input.	SELECT id, name FROM food_stalls; SELECT cur_staff FROM food_stalls WHERE id = %s;" *% (id)* UPDATE food_stalls SET cur_staff = %s WHERE id = %s;" *% (new_count, id)*

3	*Update the number of visitors in a food area* First give user a list of the food areas. Then update the visitor number according to the user's input. Then update the label of the food area. Then inform the user.	SELECT id, cur_count, avg_count, busy_label FROM food_areas ORDER BY id ASC; UPDATE food_areas SET cur_count = %s WHERE id = %s;" *% (new_count, id)* UPDATE food_areas SET busy_label = CASE WHEN cur_count - avg_count > 5 THEN 'high' WHEN avg_count - cur_count > 5 THEN 'low' ELSE 'average' END WHERE id = %s;" *% id* SELECT busy_label FROM food_areas WHERE id = %s;" *% id*

For Staff: Topic Events		
4	*Find out if more staff is needed at a stage for an event today* Join performers and stages tables to the events table so that the performer and the stage name can be queried via the day (which is stored in the events table). This is an example of de-normalising the tables.	SELECT events.id, performers.name, stages.stage_name, stages.max_staff - stages.cur_staff, stages.max_staff FROM events INNER JOIN performers ON performers.id = events.performer_id INNER JOIN stages ON stages.id = events.stage_id WHERE day = %s AND stages.max_staff - stages.cur_staff > 0; *% day*
5	*Update the number of staff members at a stage* List current staff and maximum staff for the user, and ask the user for an ID. Get the maximum allowed staff and update the staff at the selected stage if entry was valid. Lastly, select the stage name for display in the map.	SELECT id, stage_name, cur_staff, max_staff FROM stages ORDER BY id ASC; SELECT max_staff FROM stages WHERE id = %s; % id UPDATE stages SET cur_staff = %s WHERE id = %s; *% (staff, id)* SELECT stage_name FROM stages WHERE id = %s; *% id*

For Visitors: Topic Food		
6	*Find out which food areas are not busy* Select the name of all food stalls where the surrounding food area is marked as "low" in terms of how busy it is.	SELECT name FROM food_stalls WHERE id IN (SELECT s.id FROM food_stalls AS s, (SELECT geom FROM food_areas WHERE busy_label = 'low') AS a WHERE ST_Intersects(s.geom, a.geom));
7	*Find the closest food stall* First get user input about location. Delete any previous record for the location and add the new one. The query the closest food stall. The geom columns are cast as geography to get units in meters (rather than degrees). The distance result is then cast as integer to get a rounded neat number.	DELETE FROM user_location; INSERT INTO user_location(id, geom) VALUES(1, ST_GeomFromText('Point(%s %s)', 4326)); *% (userX, userY)* SELECT name, (ST_Distance(s.geom::geography, u.geom::geography)::integer) AS distance FROM food_stalls AS s, user_location AS u ORDER BY distance ASC LIMIT 1;

8	*Find the closest not busy food stall*
	
	First get user input about location. Delete any previous record for the location and add the new one. The following query selects the name of a food stall which is the closest to the user and which is in a food area that is labelled as 'low' for busyness.
		DELETE FROM user_location;
		
		INSERT INTO user_location(id, geom) VALUES(1, ST_GeomFromText('Point(%s %s)', 4326)); *% (userX, userY)*
		
		SELECT s.name, (ST_Distance(s.geom::geography, u.geom::geography)::integer) AS distance
		FROM food_stalls AS s,
		user_location as u
		WHERE s.id IN (
		    SELECT a.id FROM food_stalls AS a,
		    food_areas AS b
		    WHERE ST_Intersects(a.geom, b.geom)
		    AND
		    b.busy_label = 'low')
		ORDER BY distance ASC
		LIMIT 1;

For Visitors: Topic Events		

9	*Find out which events are on today*
	
	Query the event, performers and stages in a joined table according to a given day.
		SELECT events.id, performers.name, stages.stage_name
		FROM events
		INNER JOIN performers ON performers.id = events.performer_id
		INNER JOIN stages ON stages.id = events.stage_id
		
		WHERE day = %s; *% day*

10	*Find out when and where my favourite artist is playing*
	
	First give user a list of the id, names, and genres of the artists and the ask them for the id. Then query the day and stage of the performance.
		SELECT id, name, genre FROM performers ORDER BY id ASC;
		
		SELECT events.day, stages.stage_name, performers.name FROM events
		INNER JOIN performers ON performers.id = events.performer_id
		INNER JOIN stages ON stages.id = events.stage_id
		
		WHERE performers.id = %s; *% id*

11	*Find out what events are happening near me today*
	
	Select the id, performer's name, stage name and distance to the user of the relevant events for a given day and rank them according to distance.
		SELECT e.id, e.name, e.stage_name, (ST_Distance(e.geom::geography, u.geom::geography)::integer) AS distance
		FROM
		  (SELECT events.id, performers.name, stages.stage_name, stages.geom
		  FROM events
		  INNER JOIN performers ON performers.id = events.performer_id
		  INNER JOIN stages ON stages.id = events.stage_id
		  WHERE day = %s) AS e,
		user_location AS u
		ORDER BY distance ASC; *% day*

12	*Find the closest stage*
	
	First get user input about location. Delete any previous record for the location and add the new one. The geom columns are cast as geography to get units in meters (rather than degrees).
		DELETE FROM user_location;
		
		INSERT INTO user_location(id, geom) VALUES(1, ST_GeomFromText('Point(%s %s)', 4326)); *% (userX, userY)*
		
		SELECT s.stage_name, (ST_Distance(s.geom::geography, u.geom::geography)::integer) AS distance

	The distance result is then cast as integer to get a rounded neat number.	FROM stages AS s,
		user_location AS u
		ORDER BY distance ASC
		LIMIT 1;

For Visitors: Topic Tents		

13	*In which zone can I put my tent? I.e. where is still space?*
	
	Select the tent area id, and how many more tents still fit in a tent area based on the difference of current tents and maximum tents for the tent areas.
		SELECT n.id, n.more_tents FROM
		(SELECT t.id, t.capacity - t.tent_count AS more_tents
		FROM
		(SELECT tent_zones.id, tent_zones.capacity, COUNT(tents.geom) AS tent_count
		FROM tent_zones
		LEFT JOIN tents ON ST_Contains(tent_zones.geom, tents.geom)
		GROUP BY tent_zones.id) AS t)
		AS n
		WHERE n.more_tents > 0;

14	*How far am I from my tent?*
	
	First, give user overview of how many tents there are and ask for a valid ID (=tent number). Then calculate the distance from user to that tent in rounded meters
		SELECT COUNT(tents.geom) FROM tents;
		
		SELECT (ST_Distance(t.geom::geography, u.geom::geography)::integer) AS distance
		FROM (SELECT geom, id FROM tents WHERE ID = %s) AS t,
		user_location AS u
		ORDER BY distance ASC
		LIMIT 1; '% id'

As already mentioned, these queries are embedded in python script. The python code is designed to perform the following:

- **Execute the queries** as prepared statement plans to prevent SQL injection attacks
- Performs various **validity checks** on user input before sending the queries (e.g. for updates to the database the user input is checked for the correct format and whether the input is a within a valid range)
- **Guides the user** through the query process through descriptions.

An example of the code for task 5 ("Update number of staff at a stage")

First the SQL queries are prepared as part of the setup() function:

```
# Task 5
cur.execute("prepare plan5_1 as "
            "SELECT max_staff FROM stages WHERE id = $1;")
cur.execute("prepare plan5_2 as "
            "UPDATE stages SET cur_staff = $1 WHERE id = $2;")
cur.execute("prepare plan5_3 as "
            "SELECT stage_name FROM stages WHERE id = $1;")
```

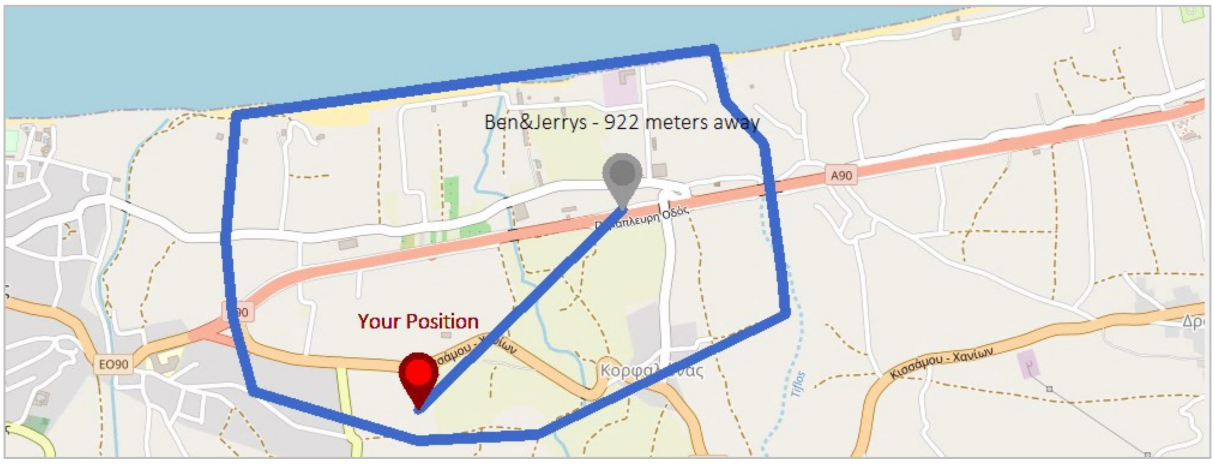
Then, the perform_tasks() function interacts with the user and performs several checks are performed. In this code snippet a few additional comments have been added to make the process more understandable.

```python
if task == "5" or task == "5.":
    # Task 5 -> Update the number of staff members at a stage.

    # 5.1.
    # Get information to show user
    cur.execute("SELECT id, stage_name, cur_staff, max_staff FROM stages ORDER
         BY id ASC;")
    stages = cur.fetchall()
    print("\nID | Stage Name | Current Staff | Maximum Staff")
    for row in stages:
        print(f"{str(row[0])} | {str(row[1])} | {str(row[2])} |
           {str(row[3])}")

    # Get user input on which stage they want to update
    stage = input("\nFor which stage do you want to update the staff? Please
                   enter the id.")

    # Check that the stage ID is valid
    if int(stage) in range(1, 6):

        # Execute plan to get the allowed maximum staff for the selected stage
        cur.execute("execute plan5_1 (%s)" % stage)
        result = cur.fetchone()
        maximum_staff = int(result[0]) + 1

        # Get user input for new staff count
        staff = input("What is the new staff count? ")

        # Check validity of input against the maximum allowed
        if staff.isdigit():
            if int(staff) < maximum_staff:

                # 5.2.
                # Execute plan to update item in database
                cur.execute("execute plan5_2 (%s, %s)" % (staff, stage))
                con.commit()

                # 5.3.
                # Execute plan to get stage name and staff for display in map
                cur.execute("execute plan5_3 (%s);" % stage)
                name = cur.fetchone()

                # Set arguments for display in map
                args[str(name[0])] = f"{str(name[0])} staff: {str(staff)}"
                show_a_map = True

            else:
                print("The value you entered is larger than the maximum.
                    Please try again next time!")

        else:
            print("The value you entered wasn't recognised as a digit. Please
                try again next time!")
```

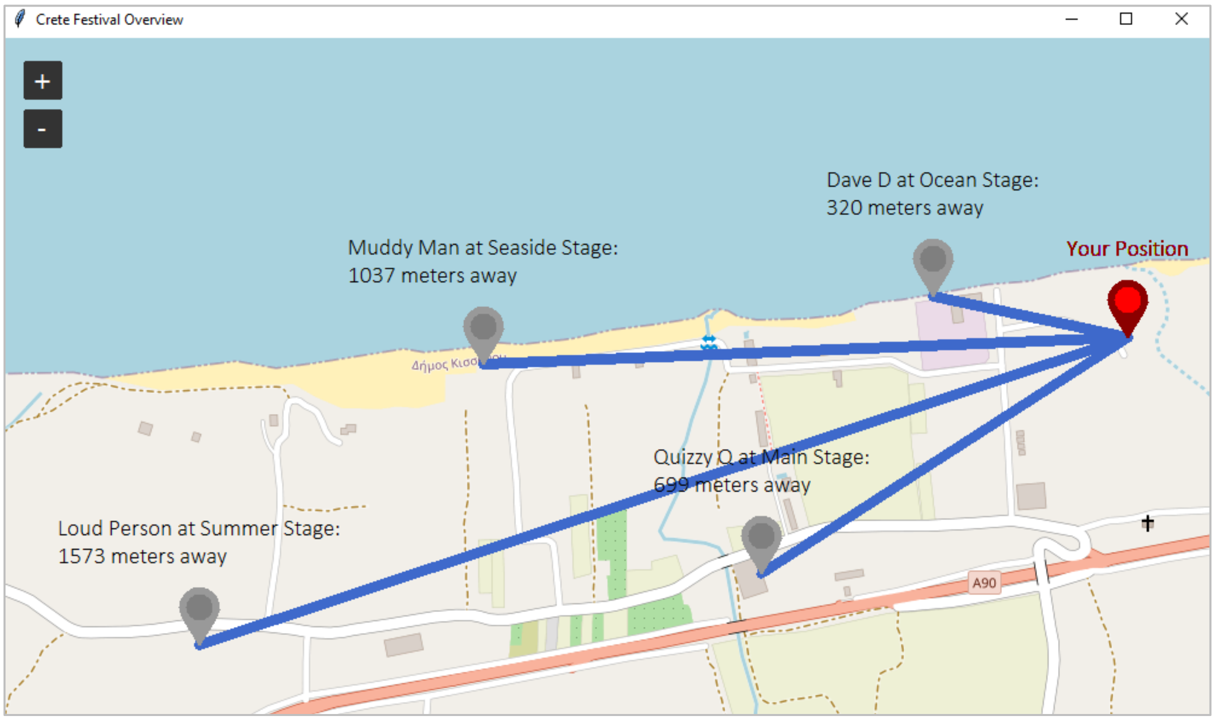
A Map as an Additional Output

The tkinter map pop-up window is a very simple spatial visualisation of the query. Since this project isn't programmed in a Jupyter Notebook or any equivalent there are limited options for displaying a map. Here are some examples:

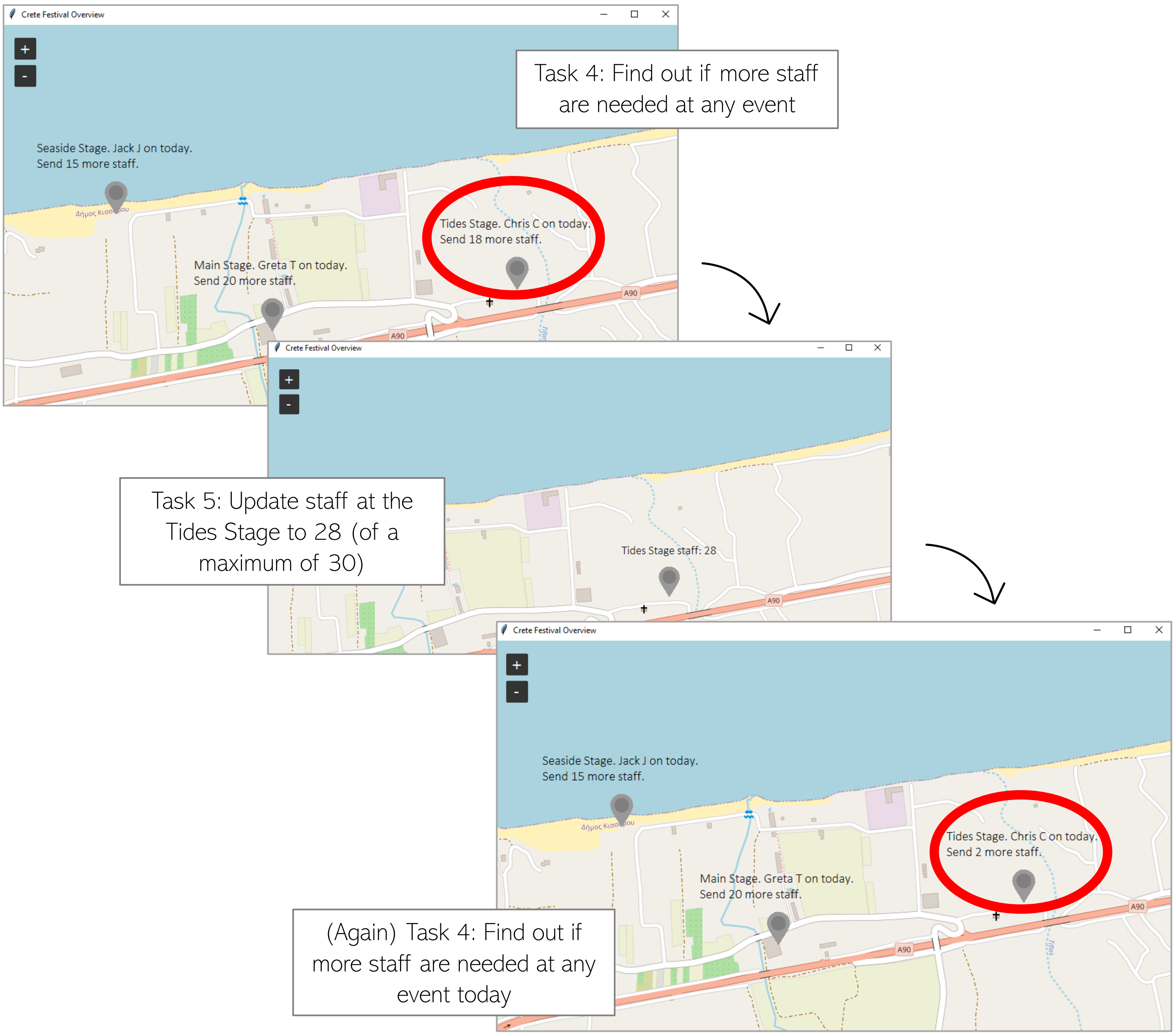
An output of task 8 ("Find the closest not busy food stall"):



An output of task 11 ("Find out which events are happening near me today"):



The outputs of task 4 and 5 to show how the query results change if the user updates values:



Task 4: Find out if more staff are needed at any event

Seaside Stage. Jack J on today.
Send 15 more staff.

Tides Stage. Chris C on today.
Send 18 more staff.

Main Stage. Greta T on today.
Send 20 more staff.

Task 5: Update staff at the Tides Stage to 28 (of a maximum of 30)

Tides Stage staff: 28

(Again) Task 4: Find out if more staff are needed at any event today

Seaside Stage. Jack J on today.
Send 15 more staff.

Tides Stage. Chris C on today.
Send 2 more staff.

Main Stage. Greta T on today.
Send 20 more staff.

A Summarized list of the functions

Function	Purpose	Returns
def **connection_params**()	Welcomes the user and helps set up the connection parameters for the database.	**host**, **database**, **user**, **password** - all the parameters needed to connect with psycopg2
def **connect**(host, database, user, password)	Establishes the connection to the database via psycopg2	**con** - a psycopg2 connection
def **check_exists**(con, table)	Checks for exisitng tables in the database.	**exists** - the database's boolean return value from the query (TRUE or FALSE)
def **sql_in**(con, sql_statement)	Inserts a statement into a given table.	/
def **sql_return**(con, sql_statement)	Queries a return value from the database.	**result** - the database's return values
def **df_inserts**(con, df, table)	Used to insert the existing data into the database tables in the beginning. It iteratively inserts data from a github CSV into the a specified table.	/
def **setup**(con)	Connects to database, checks for PostGIS extension (otherwise creates one), uses the above function "check_exists" to see if tables already exist and if not it creates them with the "df_inserts".	/
def **get_dataframe**(link)	A simple function to get a pandas dataframe from an online CSV.	**result** - a pandas dataframe
def **preprare_plans**(con)	A function that is called once before the user chooses which task to perform to set up the execution plans in the database.	/
def **perform_task**(con, task)	A pretty lengthy function that handles each task number entered by the user. It calls the different execution plans in the database and performs simple interaction with the user, while checking for valid inputs.	**map** - a boolean value to indicate whether or not a map should be loaded at the end, **args** - arguments for the map, e.g. which item to show
def **update_position**(userY, userX, con)	Takes the user's input for position and updates the loction in the database. For tasks that don't require the user location, the default is set to 0,0.	/
def **map**(userY, userX, show)	Creates a pop up map via tkinter and loads markers or paths into the map according to the arguments (="shows") and the user's position. This function is only called if the boolean variable "map" is set to True.	/
def **decide()**	User interaction logic to find out what task the user wants to do. This function also asks for the user's location if the chosen task requires it. If none is required the user's position is set to 0,0.	**userX** - user's x position, **userY** - user's y position, **task** - the chosen task number

Reflection and Conclusion

In sum, this programme is not as dynamic as I would have like it to be. The fact that the tkinter pop-up map requires a geometry input which is quite different from that given in postgres made it very difficult to dynamically display items in the tkinter map dynamically. The psycopg2 return values from an SQL query were formatted in an object that I could not edit in the same way as editing strings is possible (e.g. with the .split() or .replace() functions).

Postgres geom as text	"POINT(23.66940043 35.49462787)"
Tkinter geom type	map_widget.set_marker(35.49462787，23.6694004）

Many lines of code also dedicated to the print() function, simply to make the output in the console more user-friendly. In a professional environment, much more work would have to flow into the dynamism of the programme and the user interaction (which should be in the form of an actual app).

These shortcomings are outweighed by the newly gained skills in handling a spatial database, creating dynamic SQL queries, working with psycopg2 and rendering a simple tkinter map. In particular, I highly enjoyed bringing together the complete process of conceptualising an idea for a database/ app and then implementing it in a functioning programme. I especially like to focus on making processes easy for the user, which is why I spent a lot of time on making the installation as straightforward as possible and setup as effortless as possible for the user (e.g. by creating a function that automatically loads all data into the database). This project has undoubtedly taught me valuable skills for any future projects involving both spatial or non-spatial databases.