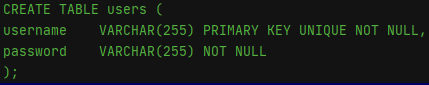
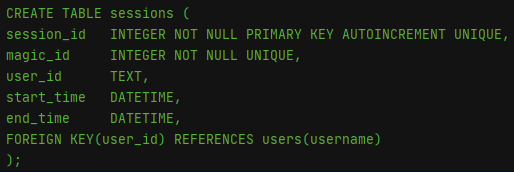
# Structure of Database

### Users Table

****

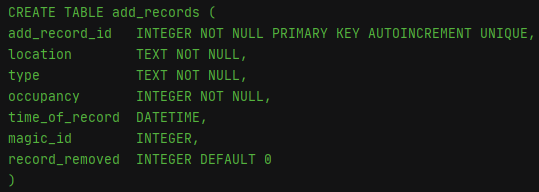
The users table seeks to store relevant information about the user. Since the only customer information that is required for this application to work is the username and password, this is the only information that is stored. The VARCHAR restriction within the database schema defaults the username and password fields to text. The number ‘255’ restricts the length of the text to 255 characters. The length was chosen due to the need to hash passwords within the database. The length of the actual password will usually be far shorter than 255 characters, but this is not necessarily the case for the hashed version. Username has a unique restriction. This is because it makes it harder for malicious users to imitate other users. Both username and password have a not null restriction for security reasons, again protecting against malicious users. Username is used as the primary key. This means that SQL can optimise retrieval of information by letting the database know that username will be referenced a lot. An alternative approach was to use an integer primary key associated with each username. I did not choose this approach as the information received from the interface does not have the user\_id, therefore I would have needed to include an additional SQL query. For simplicity, I used username as the primary key.

### Sessions Table

****

The sessions table was created to record when a user has logged in and out. Firstly, start\_time and end\_time have DATETIME typenames, resulting in numeric affinities. The DATETIME type makes manipulation of dates and times far easier than would otherwise be possible. This comes in particularly useful in tasks eight and nine. The primary key for this table is session\_id. It is worth noting that setting magic\_id would have been a more efficient way to access data relating to the sessions because the application uses magic\_id in the WHERE clauses in the SQL queries. This is because it pulls this data directly from the client, therefore magic\_id would make more sense in terms of efficiency. However, my implementation still works and was used to provide a simple implementation of tasks eight and nine. User\_id was set up as a foreign key. This means that an entry can only be inserted into the sessions table if the user exists within the user table. If the user does not exist in the users table, we would want to know why this is the case and delay them being able to input data until they have been added to the users table.

### Add Records Table

****

The add records table was created to keep track of the traffic which was added by users. Ideally, magic\_id would have a foreign key restriction such that a record could not be added unless there was a valid session recorded in the sessions table. However, due to my uncertainty on the specification of tasks eight and nine, I have avoided the foreign key restriction. Since this would just improve robustness rather than change functionality, this has been deemed a suitable decision. To keep track of whether a record has been un-done, I chose to have a column indicating whether to count the record or not. Therefore, only an additional WHERE clause was needed for summary statistics, as opposed to performing a join.

# Add and Undo Records

### Add Records

To add a record to the database, there are three main steps that must be completed:

1. Sessions validation
2. Check all fields are completed
3. Add data to database

Step one is straight forward to test given the handle\_validate function. If the user does not have a valid session open, an error is returned to the user stating that they are not logged in and need to return to the log in page. This is produced by building a response to refill the message area with the error message. The session total is also updated to return ‘—’ since the session is not valid, therefore, they will not have any data in the database under a session ID. If the user does have a valid session, we progress into step two.

Next, we check if all fields have been completed using a series of if, elif and else statements. Each if or elif statement checks whether a certain field has been completed. If the field has not been completed, an error message is returned to the user using the same method as producing the previous error – we concatenate the text response with a refill for the message area on the client side. If any of the three fields – location, occupancy or type – are not provided, the error message informs the user that they have not filled out all the required fields. Irrespective of whether all the fields have been provided, we update the total for that session by concatenating the text response. The value that is displayed is determined by the number of records that user has added during that session which have not been marked as removed, using an SQL select query. If all fields are filled out, we move onto step three.

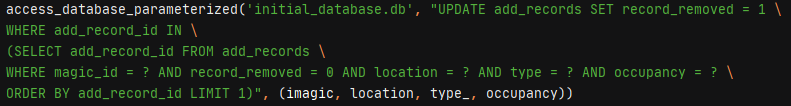
Step three involves sending off a query to the database to insert a new line into the add\_records table. The SQL query inserts location, occupancy, type and magic id. Note that the database schema dictates that the default value for the column record\_removed will be zero. Similar to before, the session total is updated but now this will reflect the increase in records added during the given session.

User and magic?

### Undo Records

1. Validation
2. Check all fields are completed
3. Remove data from database

Steps one and two act in the same manner as adding records. However, the final step functions slightly differently. The basis of the final step is based around and if – else set up. If there is a record matching input data provided by the user, the application will access the database and update that record such that record\_removed is equal to one. Note that the SQL query has a nested select query with a limit clause to ensure that not all records that match the input data change when a user wants to undo a single record. This is where the add\_record\_id comes in useful. This allows us to distinguish between individual records within a session that are exactly the same. It is worth noting that this implementation could have worked by using time\_of\_record where add\_record\_id was used. However, since add\_record\_id was set as the unique primary key, it made more sense to use this column for efficiency. Whether or not the input could be found in the database and marked as removed, the client-side total was updated. If a record could not be found, an error message was sent to the client informing the user that no matching record could be found.



# Errors and Malicious Input Prevention

### Parameterisation

Throughout the server.py file, I used parameterisation as the primary method of preventing harmful or malicious input entering the database. Parameterisation essentially lets SQL know what structure the full query should be in. For example, when comparing the username obtained on the client side, the following SQL query was used:

SELECT username FROM users WHERE username = ?

However, if we had used the following non-parameterised query, there would have been problems:

SELECT username FROM users WHERE username = ‘{}’ … .format(user)

Since SQL does not know the intended structure of the non-parameterised SQL query, a user could have potentially accessed any data they wanted. This is because they can mimic the rest of the SQL query in the input field on the client side. To give a concrete example of this, a user could have put *‘ OR 1=1 --*. The – would comment out the rest of the query and the query would return all usernames since 1=1 is always true. Other examples include the use of UNION, which append the results of additional queries onto the end of the initial query.

### Parameter Validation

Firstly, for logging in, adding records and removing records, all required fields must be completed. If this is not the case, an error message will be displayed informing the user there is an issue with that given field. This aspect of parameter validation is achieved using a series of if and elif statements. Secondly, the application also performs validation on specific fields. Namely, location must consist of only spaces and letters, otherwise the application will reject the input. This is to prevent users accidently typing in special characters. This string validation is performed on add and undo requests but not on user log ins since usernames and passwords may contain special characters. Other field specific validations include type and occupancy. The application checks the input from the client against a list of vehicle and occupancy definitions. If the input is not in the list of definitions, the input will be rejected.