Mini Project Report

Database Structure

I decided to use three tables for my database, titled: *users, sessions* and *traffic*. The structure and relationship between each table is as described below.

Users table

The purpose of this table is to store the username and password combinations of all users, linking these to a unique userid. This userid is then used to identify users in the *sessions table* (*users* is the parent table of *sessions*). Each row of this table corresponds to a different user. The specifics of each column are as follows:

userid: A unique integer used to efficiently identify users in the *sessions table* and in queries. This is the primary key of the *users table* and automatically increments by 1 for each new user. My reasons for making it the primary key are that it not only ensures unique IDs but also ensures each ID is not null.

username: The username chosen by a user. A free-form string.

password: This is the user's password, stored in hashed form using SHA1. String. When a user attempts a login, the password they enter is hashed in the same manner and the username/hash combination is validated by checking against the *users table*.

Sessions table

This table is where information about login sessions is kept, and is a child table of the *users table*, with the foreign key userid. It contains the unique magic tokens that are generated by the server whenever a user successfully logs in, linking these to a corresponding userid (identifying the user linked to each session). Each row of the table corresponds to a different successful login session, containing the userid (to identify the user in that session), the corresponding magic token unique to that session and the time of login and time of log-out (if applicable). Additionally, each entry contains an 'active' value, indicating whether the session is still active.

magic: A unique integer generated by the server during a successful login. This is the primary key and is used to validate current sessions and to uniquely identify each session.

userid: An integer that identifies the user corresponding to a given session, taken from the *users table* as a foreign key.

start: An SQL datetime of the form 'YYYY-MM-DD hh:mm:ss'. This automatically defaults to the current timestamp whenever a new session entry is added (and corresponds to the session login time).

end: An SQL datetime of the form 'YYYY-MM-DD hh:mm:ss'. This corresponds to the session logout time and is null while a session is still active.

active: An integer from the Boolean set (0,1). This automatically defaults to 1 whenever a new session entry is added, indicating that the session is currently active. When a user logs out this is updated to 0, indicating the session is no longer active and the session magic is invalid. This is managed through a column constraint called 'chk_active', allowing only one of (0,1) to be entered.

Traffic table

This table houses all traffic entries added via the web-interface. It is a child table of the *sessions table*, with the foreign key trafficid. The table includes the information relating to both 'add' and 'undo' entries, the latter of which are identified via a '0' in the 'include' column. This table is contains the magic token of the session that added each entry, the traffic data itself (including location, vehicle type and occupancy), the date and time each entry was added, the date and time each entry was undone (if applicable) and a Boolean value, 'include', indicating whether or not an entry has been undone.

trafficid: A unique integer that is automatically generated and incremented whenever a new traffic entry is added. This is the primary key of the table and is used to uniquely identify individual traffic entries.

location: The location of the traffic entry, input by the user. This is a free-form string.

type: The vehicle type of the traffic entry, input by the user. This can be one of eight pre-defined string values: ‘car’, ‘bus’, ‘bicycle’, ‘motorbike’, ‘van’, ‘truck’, ‘taxi’ or 'other'. This is managed through a column constraint called 'type_cats', allowing only one of the above values to be entered.

occupancy: The integer vehicle occupancy of the traffic entry, input by the user. This can be one of four pre-defined integer values: 1, 2, 3, 4 (managed via the column constraint 'occup_range', allowing only one of the above values to be entered).

magic: An integer that identifies the session corresponding to a given entry, taken from the sessions *table* as a foreign key.

added: An SQL datetime of the form 'YYYY-MM-DD hh:mm:ss'. This automatically defaults to the current timestamp whenever a new traffic entry is added (and corresponds to the date and time the entry was added).

removed: An SQL datetime of the form 'YYYY-MM-DD hh:mm:ss'. This is corresponds to the date and time an existing traffic entry was undone and is null unless the entry has been undone.

include: An integer from the Boolean set (0,1). This automatically defaults to 1 whenever a new traffic entry is added, indicating that the entry should be included in any totals. If a user chooses to undo the entry this is updated to 0, indicating that it should no longer be included in any totals. This is managed through a column constraint called 'chk_incl', allowing only one of (0,1) to be entered.

Add and Undo feature descriptions

Add function

The add function adds a new entry to the *traffic table*, generating a unique trafficid. It uses the unique session magic (a foreign key of *sessions*) to record the session that added each entry (and hence the userid corresponding to that session, via the *sessions table*).

In terms of the function’s mechanics, firstly, it checks that all required fields have been entered (location, type and occupancy). If not, it returns an error message informing the user that some required fields are missing, as well as an update of the traffic total. If all fields have been entered, the function then checks the user has a valid userid and magic combination. This is checked by passing them through the 'handle_validate' function, which will either return True - allowing the function to proceed - or False - where the user will be presented with an error message stating ‘User not logged in, entry not added’ alongside an update of the total. If the user's request has passed these checks, the new entry will be added to the *traffic table*. To do so, an 'INSERT INTO traffic' statement is used, with the new entry having the user selected location, type and occupancy, as well as the session's unique magic token - all of which are passed in parameterised form to prevent injection attacks. The session magic is included in the entry, as opposed to userid, as it allows traffic totals to be calculated on either a per session or per userid basis (each magic token links to a valid userid in the *sessions table*). As well as this, whenever a new traffic entry is added, the date and time of this addition is automatically stored in the 'added' column and a unique trafficid is also automatically generated (the primary key). The 'include' column also defaults to 1 to indicate that the entry has not yet been undone and should be counted in any totals.

Once the entry has been added, an updated traffic total is calculated using a 'SELECT COUNT(*)' statement where 'include' is 1 (only counting entries that have not been undone) and magic is equal to the current session's magic (ensuring only entries added during the current session are counted). This is then displayed on the webpage via the 'build_response_refill' function, along with a message stating that the entry was successfully added.

Undo function

The undo function works by disabling the entry in question, keeping it in the *traffic table* but disqualifying it from any subsequent totals.

Before updating the entry, the function first performs the same checks as the add function does - namely checking all required fields have been filled (returning an error message and the current total if not) and that the user's magic and userid are valid (again returning an error message and the current total, but without carrying out the undo). If both checks are passed, the function will proceed to perform one final check: that the location, type and occupancy of the vehicle specified do indeed correspond to an entry that was previously added to the *traffic table* during the current session. This is achieved using a 'SELECT' command where the location, type and occupancy are equal to those entered by the user; the magic token matches that of the current session and include is 1 (i.e. they have not yet been undone). The number of results is counted and if there are no matching entries the user is presented with the error message 'No such entry exists' and the current total is again returned. If, however, there are matching entries, the most recent match is disabled. To do so, an 'UPDATE' statement is used, setting the 'include' column of the matching entry to 0 (disabling it) and the 'removed' column to the current timestamp (to record the date and time of the correction). The unique trafficid of the most recent match is also used with a 'WHERE' clause to identify the correct entry.

Once the entry has been disabled, the new traffic total is counted and returned, along with a message stating that the entry was successfully undone.

Preventing errors and malicious behaviour

Error Prevention

To prevent the server encountering unhandled exceptions, I have pre-empted the following scenarios, all of which could have led to unhandled errors and caused the server to deviate from its expected behaviour. Instead, these errors will be properly handled, and suitable error messages will be displayed to the user.

The first few error scenarios involve the user logging in via the 'login' page. The 'handle_login_request' function starts by checking that both the username and password fields have been filled in by the user, and if one or more fields have been omitted it will return to the user the error message 'Error: Missing username or password.'. This ensures that neither the username nor password field is empty, which could have led to an error when attempting to hash an empty password or validate empty username/passwords. Secondly, 'handle_login_request' also checks that the username/password combination supplied is valid (via 'handle validate') and if not returns an error message stating that either the username or password is incorrect. This is of course an incredibly important feature as it ensures the integrity of the login system, allowing access to only a defined set of username/password combinations. Finally, the login function checks the userid corresponding to an entered username against those included in the *sessions table*. If an active session with that userid already exists, the user will be denied access to the app, with the error message 'User already logged in, please log out first'. This prevents a user opening multiple active sessions at once without logging out.

The next error relates to nearly all actions taken by the user away from the 'login' page. It involves the user attempting to access pages or add/undo entries when they are not authorised to do so (i.e. not currently logged in). To prevent this, each relevant function in the server performs a validation check with the user's cookies before any action is taken. This is done by passing the magic and userid to the 'handle validate' function, which will check these against active sessions in the *sessions table*, and if no match is found will return 'False' - indicating the user has invalid session cookies and their request should not be granted. They will then receive the error message: 'User not logged in, action not performed'.

The final set of error scenarios relate to the add/undo entry features. Both the 'handle_add_request' and 'handle_undo_request' functions are designed to begin by checking that all required fields have been entered by the user (including location, vehicle type and occupancy). This prevents the creation of unhandled errors when one or more of these fields have been missed by the user (leading to the server attempting to add/update traffic entries with empty values, where one from a defined set is expected). As well as this, my 'handle_undo_request' function anticipates the case where a user attempts to undo an entry that was never previously added. This could lead to an internal error where the server tries to update a traffic entry that does not exist. To prevent this, the function checks that at least one traffic entry (that has not been undone) matches the given description and the session magic. If no such entry can be found, an error message stating 'No such entry exists' is returned and the server does not attempt to update the non-existent entry.

Malicious behaviour prevention

Parameterised queries

To prevent SQL injection attacks, all queries that depend on a user input are parameterised. This is done via the sqlite3 'Cursor' object's 'execute(query, parameters)' method, where query contains a '?' in place of user inputs and parameters is a tuple of the user inputs. By doing so, the execute method will correctly escape the inputs (and any quotation marks inside them) and hence prevent injection attack attempts.

Session validation

Whenever a user attempts to open a page or add/undo traffic entries, their session cookie is validated by the server. This checks whether the combination of magic and userid are present and 'active' in the *sessions table* - ensuring that the user has previously provided a correct username/password combination, and that they had not yet logged out of that session. This is a crucial precaution against malicious behaviour, preventing a user without the correct credentials accessing any page or adding/undoing any entries they are not authorised to. Additionally, instead of relying on usernames to validate and record sessions, it is the userid's that are transferred and validated (alongside the magic). This prevents the username, which could contain sensitive information, being viewed through transferred cookies.

Hashed passwords

To prevent attackers viewing sensitive password information, all passwords are stored in hashed form using the SHA1 convention (via the 'hashlib' module). When a user logs in, the password entered is also hashed and checked - alongside their username - against the *users table*.

Input validation

As well as using parameterisation to prevent injection attacks, the server also checks that inputs are within their expected domain. For example, the vehicle type must be within a predefined set of valid types, and the literal input is cross-checked against this before proceeding to any database queries.