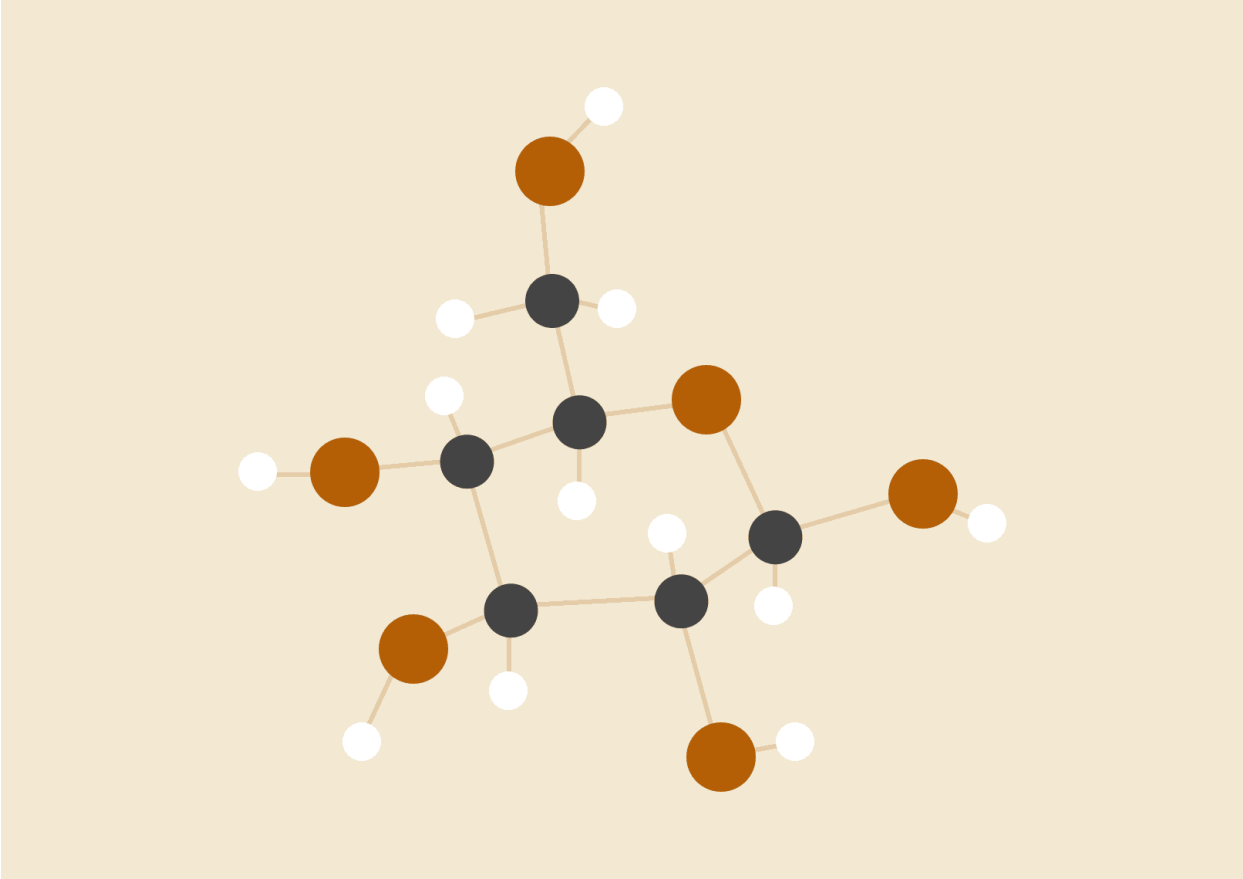
Database Design Project Report



Mark Matis, Joshua Wood, Owen Moore, Sam Hollenbeck

Github Link: https://github.com/jswood23/CSCE310

05/01/2023

CSCE 310

Changes to Final Project Design

Our database design has not had any changes to the final ERD that we established before beginning the coding portion of this project. We have been following closely the functionality outline and specifications that we agreed upon beforehand.

File List

1. Accounts/
    a. change-permissions.php - Joshua Wood
    b. login.php - Joshua Wood
    c. logout.php - Joshua Wood
    d. manage-accounts.php - Joshua Wood
    e. register.php - Joshua Wood
    f. reset-password.php - Joshua Wood
    g. welcome.php - All Team Members
2. Meetings/
    a. Create-meeting.php - Mark Matis
    b. Edit-meetings.php - Mark Matis
    c. Previous-meetings.php - Mark Matis
    d. Single-meeting-edit.php - Mark Matis
3. Items/
    a. Create-items.php - Owen Moore
    b. Edit-items.php - Owen Moore
    c. Get-items.php - Owen Moore
    d. Delete-items.php - Owen Moore
4. Reviews/
    a. Create-review.php - Sam Hollenbeck
    b. Select-review.php - Sam Hollenbeck
    c. Single-review-edit.php - Sam Hollenbeck
    d. View-reviews.php - Sam Hollenbeck
    e. Welcome.php - Sam Hollenbeck
5. Config.php - Joshua Wood
6. Header.php - All Team Members
7. Index.php - Joshua Wood
8. Create_tables.txt - All Team Members

Functionality Set 1 - Joshua Wood

I was responsible for the accounts system, including the creation and editing of accounts along with their permissions. Each page that is not intended for non-users checks the session to see if the user is logged in and whether their permissions level is high enough to view the page. The accounts database table was created with the following command:

```
CREATE TABLE accounts (
    account_key INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name_first VARCHAR(50) NOT NULL UNIQUE,
    name_last VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    permission INT DEFAULT 0,
    bio VARCHAR(400),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

The Login and Create Account pages use sql commands to search or update the accounts table as well as create a session for the user once they are logged in. Additionally, the Manage Accounts page lists the accounts and gives options to change the permissions on each account. Naturally, this requires admin permissions to access so that the feature is closed off from the majority of the users. When a user makes an account for the first time, the system checks if there are any already existing users. If not, then the system makes this first user an admin. Otherwise, the user is given a permission level of 0, which prevents them from using any of the major features in the website. Once an admin sees a new account in the list and recognizes them as a member, they may give the account a permission level of 1 (member permissions) or 2 (admin permissions). Finally, they can choose to delete the account if they don't recognize the new user.

SQL commands used:

- The Register page uses a **SELECT** to see if any accounts exist with the same email, an **INSERT** to add a new row for an account, and a **SELECT** to check if any other accounts exist (for automatic admin privileges).
- The Login page uses a **SELECT** to see if an account exists that matches the username and hashed password.
- The Reset Password page uses an **UPDATE** to change the hashed password stored for the user.
- The Manage Accounts page (and change-permissions helper) uses a **SELECT** to get information on each account, a **DELETE** to allow admins to delete accounts, and an **UPDATE** to allow admins to change permissions on accounts.

Functionality Set 2 - Mark Matis

I was responsible for the creation and management of book club meetings, which counted as Functionality Set 2 - Scheduling. I coded the ability to add meetings, create meetings, edit meetings, and delete meetings.

To do this, I created two separate tables - a table titled Meeting, and a table titled Bridge. Bridge was, as the name suggests, a bridge table between users and meetings - because a user can be in many meetings, and a meeting can have many users, but many to many relationships aren't valid! So the bridge made two one-to-many relationships, which made the SQL a little more complex but it was worth it. And the Meeting table was just a lot of information about the meeting, such as location and organizer. The bridge table was made with:

```
CREATE TABLE bridges (
    account_key INT NOT NULL  REFERENCES accounts(account_key),
    meeting_key INT NOT NULL  REFERENCES meetings(meeting_key),
    PRIMARY KEY (account_key, meeting_key)
);
```

And the meetings table was made with:

```
CREATE TABLE meetings (
    meeting_key INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    start_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    end_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    address VARCHAR(400),
    item_key INT NOT NULL  REFERENCES items(item_key),
    meet_desc VARCHAR(400),
    organizer INT NOT NULL  REFERENCES accounts(account_key)
);
```

My feature had different functionality for two different user types. The **first** type of user, a regular member, could: see the meetings they are a part of, edit or delete meetings they are the organizer of, and can remove themselves from the roster of a meeting. The **second** type of user, an admin, can do everything that the user can, but also can *create* new meetings!

To implement the “view meetings you are a part of” feature, I created a **SELECT** SQL statement which used two **INNER_JOINS** on the user table, the bridge table, and the meetings table. This page also contains the “remove myself from meeting” feature, where I used the **DELETE** statement on the bridge table using the user’s id and the meeting id that was selected.

For the “create meeting” feature, I used an **INSERT** statement of the meeting table, then one on the bridge table, and then iteratively added the attendees to the bridge table using more **INSERT** statements, and the options for the meeting were fetched and shown to the user using **SELECT** statements.

For the “edit meeting” feature, I used the **UPDATE** statement on the meetings table for the given meeting_key, to update it as requested in the form on the page. I also used the **SELECT** to get all meetings that the user organizes, **DELETE** to remove users from the bridge table that were no longer invited to the meeting, and **INSERT** to add users to the bridge table that were now invited.

For the “delete meeting” feature, I used **SELECT** to get all meetings that the user organizes, and I used the **DELETE** statement on the meetings table and the bridge table to completely delete it.

To see my functionality set in action, watch the attached demo video of our project!

Functionality Set 3 - Owen Moore

I was responsible for the creation and management of book club items, which counted as Functionality Set 3 - Items. I coded the ability to add items, view all items, edit existing items, and delete items.

To implement the items table I created a single table in the database containing all of the information about the item, such as Title, author, isbn, and date it was added. The item table was made with:

```
CREATE TABLE items (
    item_key INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    item_title VARCHAR(100) NOT NULL,
    author VARCHAR(50) NOT NULL,
    isbn INT NOT NULL,
    date_added DATETIME DEFAULT CURRENT_TIMESTAMP,
    summary VARCHAR(250) NOT NULL
);
```

My feature had different functionality for two different user types. The **first** type of user, a regular member, is only allowed to view all of the items in the database. We did this because it did not make sense to allow regular user to add whatever they wanted to the database. To control this we allowed the **second** user type, an admin, the ability to do everything, add a new item, delete items, edit items, and view all the items.

For the “Add new item” function, I used an **INSERT** statement to insert all the given values from the user into the database and used the date time function to record the current time the user adds the item.

For the “Delete item” function, I used a **DELETE** statement to delete the item from the database using the item_key for that item.

For the “Update item” function, I first used a **SELECT** statement to select all the values with the given primary key the user wishes to edit. The select statement is used to get and store all of the item’s attributes to save for the update if the user chooses not to make any changes to those attributes. I then used an **UPDATE** statement to update the item with the given key’s values in the database.

For the “See all items” function, I used a **SELECT** statement to select all the items from the table along with all their attributes to display to the user.

Functionality Set 4 - Sam Hollenbeck

I was responsible for the creation and management of book club reviews, which counted as Functionality Set 4 - Reviewing. I coded the ability to post reviews, edit reviews, view reviews, and delete reviews. A **member** can post, edit and delete their own reviews and can view all reviews. An **admin** can do all this but moderate others' reviews by being able to edit and delete all reviews. Any **user** who is not a member or admin does not get access to this functionality.

To accompany this functionality, I created the Reviews table. The reviews table was made with:

```
CREATE TABLE reviews (
    review_key INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    account_key INT NOT NULL REFERENCES accounts(account_key),
    item_key INT NOT NULL REFERENCES items(item_key),
    header VARCHAR(255) NOT NULL,
    body VARCHAR(400) NOT NULL,
    stars INT DEFAULT 0,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

For the “Create review” feature, I used an **INSERT** statement to the review table. I needed to use a simple **SELECT** statement to get all the current item keys from the items database to list them for the user.

For the “Edit review” feature and the “Delete review”, I first used a **SELECT** to get all the reviews. There was a **WHERE** clause that only retrieved reviews that the user was allowed to edit or delete – all for the admin or reviews where the account_key of the review equaled the account_key of the current user. There also were two **JOIN**s to provide helpful information from two other tables: the name of the user who made the review from the accounts table (JOIN accounts ON reviews.account_key = accounts.account_key) and the title of the book from the review table (JOIN items ON reviews.item_key = items.item_key. If a user chooses to delete a review, I used a **DELETE** statement with a WHERE clause to drop only reviews that had a matching review_key. If the user chooses to edit a review, I used an **UPDATE** statement with a **WHERE** clause to update only reviews that had a matching review_key.

For the “View Meeting” feature, I used a **SELECT** to get all the reviews. There were two **JOIN**s to provide helpful information from two other tables: the name of the user who made the review from the accounts table (JOIN accounts ON reviews.account_key = accounts.account_key) and the title of the book from the review table (JOIN items ON reviews.item_key = items.item_key.

Thank you for a great semester!