Project Report

BeSocial - Event Management Platform

Ryan Ogrey
015562378
ryan.ogrey@sjsu.edu

Tri Nguyen
015767817
tri.n.nguyen01@sjsu.edu

Trung Tran
015690753
trung.tran01@sjsu.edu

Sai Manaswini Avadhanam
016445663
saimanaswini.avadhanam@sjsu.edu

Ashton Headley
014814410
Ashton.headley@sjsu.edu

10 May 2024
Professor Ramin Moazeni, PhD

Table of Contents

1. Application Goals and Description

1.1. Goals

In today's fast paced world, many individuals can find it hard to keep up with the events around them. Event organizers can struggle to advertise their events to their audience. With BeSocial, our goal is to connect those gaps, allowing event attendees and hosts to be connected under one platform. We aim to build a space where users can not only host and attend events but also share their experiences at different events. We want to encourage users to connect with other hosts and attendees to keep up with their peers who hosted and attended events, fostering a community under the platform BeSocial.

1.2. Description

The BeSocial Event Management Platform presents a new and improved take at event hosting and participation. It aims to streamline the process of hosting, promoting, and engaging with events, all from the comfort of your home. Following a simple account creation process, users immediately have the ability to find and host events that directly meet their needs. When hosting an event, users can elaborate on the capacity, location, time, category, and description of their event. They also have the power to delete their events in instances where the capacity fills or plans change. Other users can find local events by searching on the discover page. Alternatively, they can see if users who they follow are planning any events by checking out the event wall. If an event stands out to a user, but they aren't sure if they are ready to commit yet, they can use the bookmark feature to quickly navigate back to that event at a later date. If they choose to commit, a user can attend an event from the simple click of a button. If they happen to change their mind, they can easily unattend using the same process. Included in every event is an event details page where users can also comment and view the guest list for that particular event. Comments can be liked, giving the event host a better idea of potential suggestions or concerns that many agree upon. Each user also has a followers and following list where they can view users whom they have followed in the past or are currently following them now.

2. Application / Functional Requirements and Architecture

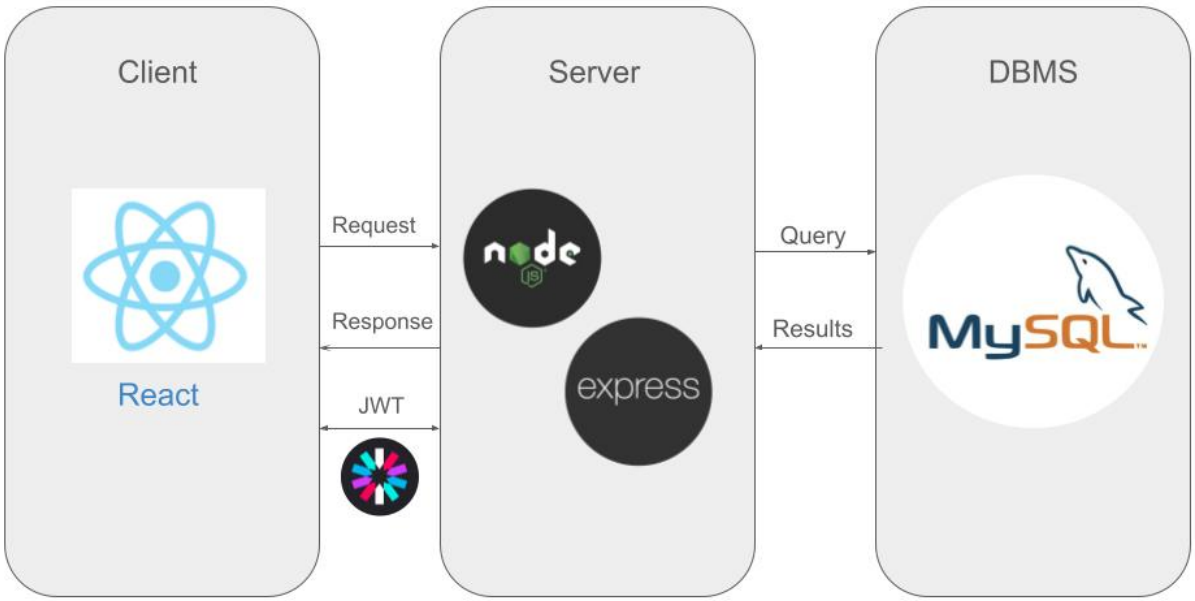
2.1. Application / Functional Requirements

- Let users create a new account. Account creation uses user authentication and gives the user access to all other application functions.
- Let users log in with preexisting account details that were set during account creation.

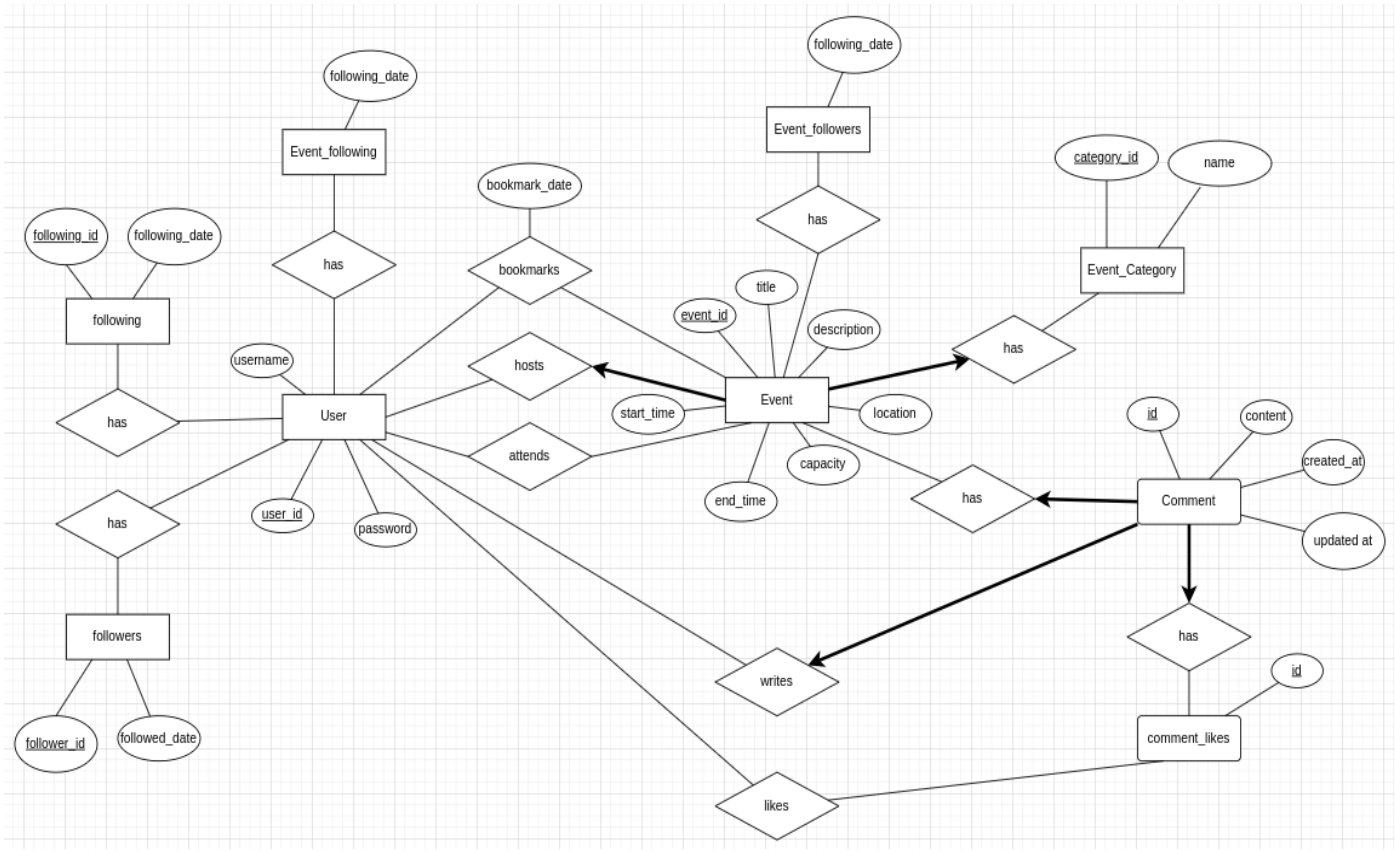
- Let users log out, preventing them from accessing all elements of the application other than the home page.
- Let users host (create) an event.
- Let users set event details (location, start time, end time, description, capacity, and category) when creating an event.
- Let users delist (delete) their events.
- Let users write comments on all events, including theirs.
- Let users like comments, including ones they created.
- Let users check in (follow) an existing event.
- Let users check out (unfollow) of an event that they no longer wish to attend.
- Let users follow other users.
- Let users unfollow users that they currently follow.
- Let users view everyone who they follow with a following list.
- Let users view everyone who follows them with a followers list.
- Let users view events hosted by users they follow through an event wall.
- Let users view all events hosted by other users on a discovery wall.
- Let users bookmark/unbookmark an existing event.
- Let users view an event's details, including guest list, event specifics, and comments.

2.2. Architecture

Three-tier architecture: Client - Server - DBMS



3. ER Data Model



4. Database Design

4.1. Tables & Relationships

Our 10 relations are listed below:
Users (user_id: Integer, username: String, password: String)
Events (event_id: Integer, title: String, description: String, location: String, capacity: Integer, start_time: Date, end_time: Date, created_at: Timestamp, updated_at: Timestamp, category_id: Integer)
Event_following (user_id: Integer, event_id: Integer, following_date: Date)
Event_followers (event_id: Integer, user_id: Integer, follower_date: Date)
Categories (category_id: Integer, name: String)

Bookmark (user_id: Integer, event_id: Integer, bookmark_date: Timestamp)
Comments (comment_id: Integer, content: String, created_at: Timestamp, updated_at: Timestamp)
Comment_likes (like_id: Integer)
Followers (following_id: Integer, following_date: Date)
Following (following_id: Integer, following_date: Date)

4.2. Relational Schemas

User (user_id: Integer, username: String, password: String)

Hosts(user_id, event_id)
Foreign Key: (user_id) References Users, (event_id) References Events

Attends(user_id, event_id)
Foreign Key: (user_id) References Users, (event_id) References Events

Bookmark (user_id: Integer, event_id: Integer, bookmark_date: Timestamp)
Foreign Key: (user_id) References Users, (event_id) References Events

Events (event_id: Integer, category_id: Integer, host_user_id: Integer, title: String, description: String, location: String, capacity: Integer, start_time: Date, end_time: Date, created_at: Timestamp, updated_at: Timestamp)

Has(event_id, category_id)
*Foreign Key: (*event_id*) References Events, (category_id) References Categories*

Categories (category_id: Integer, name: String)

Event_following (user_id: Integer, event_id: Integer, following_date: Date)
Foreign Key: (user_id) References users(user_id)
Foreign Key: (event_id) References event(event_id)

Event_followers (event_id: Integer, user_id: Integer, follower_date: Date)
Foreign Key: (event_id) References event(event_id)
Foreign Key: (user_id) References users(user_id)

Comments (comment_id: Integer, content: String, created_at: Timestamp, updated_at: Timestamp)
Foreign Key: (user_id) References users(user_id)

Has(comment_id, like_id)
Foreign Key: (comoment_id) References comments, (like_id) References comment_likes

Comment_likes (like_id: Integer)

Has(user_id, id)
Foreign Key: (user_id) References Users, (id) References Followers

Followers (id: Integer, follower_id: Integer, followed_date: Date)
Foreign Key: (id) References users(user_id)
*Foreign Key: (*follower_id*) References users(user_id)*

Has(user_id, id)
Foreign Key: (user_id) References Users, (id) References Following

Following (id: Integer, following_id: Integer, following_date: Date)
Foreign Key: (id) References users(user_id)
*Foreign Key: (*following_id*) References users(user_id)*

4.3. Normalization

Events(event_id, title, description, location, capacity, category_id, category_name, start_time, end_time, host_user_id, username, password, created_at, updated_at, bookmark_date)
FD 1: event_id -> title, description, location, capacity, category, start_time, end_time
FD 2: host_user_id -> user_name, password
FD 3: host_user_id -> event_id, bookmark_date
FD 4: category_id -> category_name
Decompose FD2 because of transitive dependency
R1: Events(id, title, description, location, capacity, category_id, category_name, host_user_id, start_time, end_time, created_at, updated_at, bookmark_date)
R2: Users(host_user_id, username, password) => R2 is in BCNF because no FD is lost and there is no transitive dependency
Decompose FD3 because of transitive dependency
R1 Events(id, title, description, location, capacity, category_id, category_name, host_user_id, start_time, end_time, created_at, updated_at)
R3: Bookmark(host_user_id, event_id, bookmark_date) => R3 is in BCNF because no FD is lost and Every functional dependency, X → Y in R, X contains a candidate key host_user_id
Decompose FD4 because of transitive dependency:
R1: Events(id, title, description, location, capacity, category_id, host_user_id, start_time, end_time, created_at, updated_at)

R4: Categories Table(category_id, name) => R4 is in BCNF because no FD is lost and for very functional dependency, X → Y in R, X contains a candidate key id

Follow(id, follower_id, following_id, following_date, followed_date)
FD: id -> follower_id, followed_date, following_id, followed_date
We made a lossless join decomposition for better manage the data:
R1:Follower(id, follower_id, followed_date)
R2: Following(id, following_id, followinng_date)

- Every non-candidate key attribute is fully functionally dependent on a candidate key id
- No transitive dependencies
- Every functional dependency, X → Y in R, X contains a candidate key id => BCNF

Event_follow(user_id, event_id, following_date, follower_date)
FD1: user_id, event_id -> following_date
FD2: user_id, event_id -> follower_date
Decompose based on the FD:
R1: Event_following (user_id, event_id, following_date)
R2: Event_Followers Table(event_id, user_id, follower_date)

- Every non-candidate key attribute is fully functionally dependent on the candidate keys user_id and event_id
- No transitive dependencies observed.
- Every functional dependency, X → Y in R, X contains a candidate key event_id => BCNF

Comments(comments_id, event_id, user_id, content text, created_at, updated_at)

- No partial dependencies observed.
- No transitive dependencies observed.
- Every functional dependency, X → Y in R, X contains a candidate key comments_id => BCNF

Comments_like(like_id, comments_id, user_id)

- Every non-candidate key attribute is fully functionally dependent on the candidate keys like_id
- No transitive dependencies observed.
- Every functional dependency, X → Y in R, X contains a candidate key like_id => BCNF

4.4. Indexing: Hash-based method

4.4.1 Functions:

createHashIndex() function (called every time a new user/ event is created)

- Create new table if not exist with the attribute of binary type hashed_value and string type original_value:
- Store the id in original_value then hash the input id and store it in the hashed_value.

deleteHashIndexRow() function (called every time an event is deleted):

- Delete the hash_value from the hash table based from the input id

searchHashIndex() function (called every times a search query is made)

- Select the original value from the hash table where the hashed_value is equal to input value

4.4.2 Tables:

User_id_hash_index table is implemented store the hash of the id of the users table to help in querying(searching) the id of the users more effective

Events_id_hash_index table is implemented to store the hash of the id of the events table to help in querying(searching) the id of the events more effective

5. Major Design Decisions

When it came to selecting the technology stack and architecture, we wanted to choose a bundle of libraries and frameworks that go well with each other. We know that we wanted to develop in JavaScript, and the technologies that we decided to pick were React, Node.js, Express.js, MySQL, and JWT. The popular MERN JavaScript stack (MongoDB, Express.js, React, Node.js) was an inspiration for our decision since it is very well structured and documented. In our architecture, the client-side is powered by React, which provides a responsive and interactive user experience. When a user interacts with the frontend, these actions are handled by React components, which will later communicate with our server-side component. On the server side, we make Express.js run on Node.js to develop robust API endpoints. These endpoints are important in order to execute SQL queries with our database management system, MySQL. Furthermore, to keep our application secure and manage sessions efficiently, we have integrated JWT (JSON Web Tokens) as our authentication middleware. This setup not only supports secure user authentication but also maintains session integrity across the application.

When creating our tables in MySQL, we decided to not squeeze in redundant relations into one table so that our tables are optimally organized and follows normalization principles. An example of where we decide to split a table into two tables was the following and followers table. Originally, we planned to have the following user ID and followers ID in one table. However, this would create an unorganized and redundant table, which would slow down query efficiency and lead to a slower and harder data fetch. Therefore, we decided to split the table into two, making

querying a lot more efficient and easier. Another key decision that we made was to use multiple endpoints for different event pages. Our application has four pages that would require endpoints to fetch event details: Discovery page, event wall page, my hosted event page, and bookmarked events page. These pages all fetch event details, however, the events need to follow specific constraints. For example, the event wall page should only fetch events posted by hosts that the user is following. To implement these constraints, one method was to use one endpoint for all four pages, but adjust the constraints through the frontend so that it doesn't show the events that needs to be filtered out. Another method would be to have multiple endpoints and implement constraints through the backend. We decided to do the latter approach because we concluded that by implementing constraints from the backend, it would give us more control over the data output towards the frontend. This makes the code easy to scale and control, ensuring that data is organized and then sent to the frontend, rather than organizing messy data after it is sent to the frontend.

6. Implementation Details

Setup:
First we setup Express.js for backend routing; CORS to allow the frontend and backend to communicate without restricted access; bcrypt for password hashing; crypto for hash indexing, and initialize MySQL database connection. We utilized AWS's RDS service to host a cloud MySQL database, which can be connected by all team members with its credentials. We also set up an authentication middleware in the frontend to protect routes from unauthorized users without a token created during the registration or login process.

Frontend components and the corresponding endpoints that are called:
Register:

- Post register (input user credentials into database)

Login:

- Post login (authenticate user credentials for login)

UserWall:

- Get user_events (get events details)
- Post event_walls (to create event)
- Delete events (to delete event)

EventDetails:

- Get events/id (get specific event details)
- Get events/id/comments (get comments of that specific event)
- Get get_event_follower/id (get list of attending users for that event)

- Get users/id (get host details)

Discover:

- Get events (get events details)
- Post post_event_following (when the attend button is clicked)
- Post bookmark (when bookmark button is clicked)
- Post unbookmark (when unbookmark button is clicked)

EventWall:

- Get event_wall (get events details)
- Post post_event_following (when attend button is clicked)
- Post bookmark (when bookmark button is clicked)
- Post unbookmark (when unbookmark button is clicked)

Event Attending:

- Get get_event_following (get events details)
- Post post_event_unfollowing (when unattend button is clicked)
- Post bookmark (when bookmark button is clicked)
- Post unbookmark (when unbookmark button is clicked)

UserProfile:

- Get profile (get user's profile details like followings and followers list)

OtherProfile:

- Get profile/id (get other user's profile including followings and followers list)
- Get following (get current following status)

7. Demonstration of System Run

7.1. Demonstration Video (Including Setup)

https://youtu.be/qF69XQkszG4
Demo starts: 8:16

7.2. Github

https://github.com/trungtran1234/BeSocial

8. Conclusions

8.1. General Conclusions

Throughout this project, we were able to utilize a relational database management system, in this case, MySQL, alongside other client-side and server-side technologies to create a meaningful application for event management. We were able to use the material taught in this course to effectively manage a database connected to a real application. We successfully designed the database with tables and relationships to define entities and relations required for the application. We were able to create a database schema and ER diagram to clearly visualize the conceptual details of the database to reflect the application. Additionally, we also implemented multiple queries for fetching and inserting data into the application's database. At the end, we were able to demonstrate efficient usage of the database, relations, and queries to build a strong foundation for our application.

8.2. Lessons Learned

From completing this project, we not only learned many new technologies but also new techniques and lessons from mistakes we made. One important thing that we learned was that team communication is very important when multiple people are working on the code at the same time. Poor communication can lead to a lot of merge conflicts which take up a lot of time to resolve. Another lesson that we learned was to clearly plan out the requirements and interaction needed between the frontend, backend, and database for a certain component to be polished. Before starting work on a component, we should analyze the required inputs, outputs, and constraints of the component so that we can have a bigger picture of how we can create the endpoints and queries necessary to execute the needed operation. What taught us this lesson was when we were creating the event details component, after creating the query to fetch all the data needed, we finally realized that the host username was not part of the table we were fetching from. This forced us to create a more complex query to fetch the host username. This happened multiple times throughout the development process. We also learned that global state management can be challenging when implementation across different components are inconsistent. With effective global state management, it can save us additional queries, so we will definitely have that in consideration for the future. Overall, both communication and technical lessons were learned from this project, which will help us a lot in the future.

8.3. Possible Improvements

There is a lot of room for improvement when it comes to the full functionality of our application. Firstly, we can add a function for users to add their own profile pictures. Another feature that would greatly improve our application is a search function, allowing users to search for specific events. A sort feature could also be implemented to help user sort events by categories, time

created, location, capacity, and more. Enabling direct messaging for users can also help users to keep up with one another regarding events. We can also use a RESTful API that implements real locations into the event creation form, letting users select legitimate locations for the event. An additional technology that we could use is Redux, which would help us with state management across components so that our application is not only more efficient but the code will be cleaner as well. The event creation form should also be able to let the host upload an image associated with the event. With all these possible improvements, the most important improvement would be to ensure that the database is optimized so that it is scalable and efficient even with complex features.

8.4. Individual Contributions

Ryan Ogrey:

- Implemented the frontend HTML/CSS styling for all aspects of the application.
- Made all text input boxes and page elements dynamic to large inputs and many database entries.
- Created the signup and login pages.
- Sorted the events on all pages by date, moving those with potential errors (designated by NULL) to the back.
- Wrote the README.
- Helped brainstorm details for presentation slides.
- Completed Description, Application / Functional Requirements, Tables & Relationships, and helped Tri with the Relational Schemas section of the project report.

Tri Nguyen:

- Helped brainstorm details for presentation slides.
- Draw the ER diagram
- Record the demo
- Implemented the user log out
- Implemented the hash indexing method for events and users table
- Created and implemented the endpoint for events to fetch guest list (list of attending users)
- Created the UserWall.js (Event Hosted page)
- Created the EventFollowing.js (Event Attended page)
- Created and implemented the endpoint for the Event Hosted Page to fetch user's hosting events

- Created and implemented the endpoint to enable users to delete hosted events in the Event Hosted page
- Created and implemented the endpoints to enable users to attend events and unattend events
- Created and implemented the endpoint to enable the Event Attending page to fetch events users are attending
- Add to the eventItem component with attend, delete, and guestList feature
- Complete Relational Schemas with Ryan
- Wrote the 4.4. Indexing: Hash-based method and 4.3 Normalization

Trung Tran:

- Initialized the environment
- Set up cloud MySQL database with AWS
- Created and implemented the endpoint for user registration and user login
- Implement the authentication system for user login
- Created event creation form, event object, and event SQL table
- Created event wall for following users' events
- Created user profile with following feature along with following and follower list
- Created Bookmark page as well as endpoints for user to bookmark events
- Created event details page
- Created event comment section and like feature
- Worked with Ashton to finish event_category
- Helped brainstorm details for presentation slides.
- Completed Architecture, Implementation, Major Design Decisions, and Conclusions for the report

Sai Manaswini Avadhanam:

- Designed the Logo for the application
- Sourced images to be used for various buttons and backgrounds on the frontend
- Designed and wrote all presentation slides.
- Presented the slide portion of the presentation.
- Spell and grammar checked the final report

Ashton Headley:

- Worked on event_category: created the table on MySQL and helped write backend code to link it to the event_form.
- Set up presentation slides.
- Helped brainstorm details for presentation slides.