PennBook: NETS 212 Final Report

Batchema Sombie BATCHEMA@SAS.UPENN.EDU
Divya Somayajula DIVYAS22@SEAS.UPENN.EDU
Ishaan Rao ISHAANR@SEAS.UPENN.EDU
Olivia O'Dwyer OODWYER@SEAS.UPENN.EDU

1. Overview

We built our PennBook project using React for the front-end, Node.js with Express for the back-end, Spark for the data analysis, and a DynamoDB database. All the required features are included in addition to some small extra credit features we implemented. The instructions for running our project are included in the README.

2. Project Components

2.1. Comments

We stored comments in a table with the associated unique post ID which was made up of the poster's username and the post timestamp, as well as the comment timestamp; this combination of three elements would provide a unique ID for each comment. We also stored the content, commenter username, and number of likes for an extra credit feature. We wrote routes on the back-end to add a comment to the database and get comments from a certain post. By querying the comments database for all comments with the same partition key with the post ID, we were easily able to return a collection of comments for a given post.

2.2. Likes

We implemented likes for comments, posts, and articles and also provided the ability to unlike any of these. For posts, we kept a variable in the posts table that tracked the number of likes so we could easily display the number. This number was increased or decreased each time a post was liked or unliked. We also stored likes in a post_likes table with the post ID and the liker's username. This allowed us to check whether a user had liked a post already. We wrote routes to like posts, unlike posts, and check whether a post had been liked by a user. Likes for comments were implemented similarly, with the number of likes stored in the comments table for each comment and likes by username stored in the comments_likes table. For articles, we also stored the number of likes in the articles table, but we kept track of likes in two other separate tables, article_likes_by_username and article_likes_by_article_id. This allowed us to easily query article likes both to check whether a user had liked an article for displaying on the front-end and also for tracking a user's news recommendation preferences (for adsorption).

2.3. Friends

When a friend request was sent, we stored two rows in the friends table. Our partition key was friendA, and our sort key was friendB. We wanted all friendships to be easily queryable by both friends, so we stored each friendship with two rows with both friends in the A and B positions. We also stored the timestamp, first when the request was sent, then when it was accepted, and the sender, as well as an accepted boolean field. If a request was rejected, we deleted both rows from the table. If it was accepted, we updated the timestamp and changed accepted to true. We got all pending friend requests by returning the rows in which accepted was equal to false. We also wrote a route to get new friendships for a particular user, meaning a list of the users who accepted this user's friend request in the past 3 days. This list is displayed on the home page and is a way for a user to know who recently accepted their friend request.

2.4. Users

We stored users in the accounts table with the necessary information required with an account with the username as the primary key. We also stored the hashed password, so no plaintext password was stored in the database. We used the bcrypt library to generate a random salt and store the hashed password and then compare and check the hashed password with user input to sign users in and change a user's password. We also implemented the functionality to allow users to change aspects of their profile by updating the accounts table with new information. We also implemented a lastActive route in order to track the last time a user performed an action online like liking or commenting or friend requesting someone, and this value is used to indicate whether the user is online or not.

2.5. Posts/Home Page/Wall

We stored posts and status updates in the posts table each having a unique primary and sort key combination: postee as the primary key and timestamp as the sort key. It was assumed that it would be impossible for two users to post on a user's wall at the same exact time. Other information stored in each post's row include the content of that post, the names of the poster and postee, as well as number of likes and comments. On a user A's home page, all posts/status updates where A is the postee and any of A's friends are the postee are displayed in reverse chronological order. The schema of our table made it very easy to retrieve these posts (simply had to query for all posts where the post_id was equal to the name of the postee - the logged in user or a friend). Similarly, on a user A's wall, all posts/status updates where A is the postee are displayed in reverse chronological order. These were retrieved by simply querying the posts table for where the postee = username of A. Normal posts from a user A to B are stored in the posts table as a row where A is the poster and B is the postee. Status updates by a user A are stored in the posts table as a row where A is both the poster and postee.

2.6. Visualizer

We used the library react-graph-vis to render the graph visualizer. This React component simply requires a graph object as a property, along with options (for styling) and events (for action handlers). The current structure of the graph (list of nodes and list of edges) was maintained in a React state. This means that anytime there was a node or edge added to the graph state, the visualizer would automatically be refreshed/re-rendered. This visualizer initially displays all friends of a user A, with edges outgoing from A to each friend B. These friends were retrieved by simply querying the friends table to find where friendA was user A. In the events object passed to this component, an onClick listener was defined. This onClick listener took in the id of the node that was clicked on (from which the username was easily retrievable from the graph object). We wrote a route that took in A's affiliation and the clicked-on user B's username and returned a list of B's friends that have the same affiliation as A. If these nodes weren't already on the graph, they were added to the graph and the edges accordingly, and the visualizer automatically re-rendered.

2.7. News Recommendations

The news recommendation system uses the adsorption algorithm described in class. The infrastructure for the job is the same as that of HW3 (PageRank). The two main classes are in the SparkLivy package. They are AdsorptionJob.java and ComputeLivy.java. AdsorptionJob contains the code that runs the adsorption algorithm as described in class. The code pulls RDD data from AWS S3 and then proceeds as expected. SparkLivy.java is the main class of the recommendation system. When called, it makes requests to AWS DynamoDB and downloads the required information. It gets user category interests from the accounts table, the user to liked article information from articles_liked_by_username, the user-user relationships from the friends table, and the articles from the news_articles tables. This information is cleaned and saved to disk as .txt files and then uploaded to AWS S3. Once the data is uploaded to S3, the .txt files in local storage are deleted and a Livy Job is spawned. This uploads the jar file to Livy, and Spark workers perform adsorption. The Express server of the application, upon launch and every hour after, runs a shell command via the child_process library to launch the Livy jobs. When a user changes their interests, a similar function is used to call another job to perform the adsorption again. Once finished, the Livy job updates the table newsrecommendations with a list of 5 article recommendations for every user.

2.8. News Search

In order to make news search an efficient process, we preloaded a table articles_keyword with keyword as primary key and article_id as sort key, along with other attributes for each article. For each article A in the news_articles table, we tokenized the headline of A, and created a new Item for each token/keyword to insert into articles_keyword, using DynamoDB's BatchWrite. We also maintain a search_cache table which contains articles for previously searched queries, so future queries are sped up. For any search query Q, by a user, if Q is not in the cache table already, Q is tokenized, and for each token T, all entries in the articles_keyword where keyword equals T are retrieved. Once all news article results are compiled, they are first sorted by how many keywords they match, then next by reverse chronological order, and then displayed to the user. These results are put into the cache table. If Q is already in the cache table, then those articles are simply retrieved and displayed to the user (they will already be in the correct sorted order from the first time they were inserted into the cache).

2.9. Chat

We implemented a Facebook Messenger style chat interface, allowing users to create chats with their friends, as well as group chats with multiple members. Users also have the ability to add/remove friends from the chats. When a user receives a message, they will receive a pop up notification with a link to the chat that the message was sent from. They can join the chat and send real-time messages to the other users in that chat. Users do not have to be online to receive messages, however, as all chats and group chats are persistent. Upon logging back into the application, users can see a list of all of their chats, and any unread chats will be bolded for that user.

3. Design Decisions

3.1. Likes Table

We wanted to easily return whether a user had liked a post/article/comment or not as well as quickly getting the number of likes for displaying a post. We decided to both store the number of likes as a field in the posts table as well as creating a separate table, post_likes, that would store each post id and username that had liked the post. This allowed us to quickly query and update anytime we wanted to check if a user liked something, allowed them to unlike something, and adjust the number of likes on something.

3.2. Querying Friends

We were not sure what the best way to store friendships would be, since we wanted to be able to query quickly on both friend A and friend B in a friendship rather than having to scan through all the rows depending on which position a user was in. In order to make querying for all of someone's friends efficient, we decided to store a friendship as two rows, going both directions, so that for each friendship a person has, that person is in the primary key spot.

3.3. Handling User Session

Originally we planned to use express session to maintain the user's session, but we quickly ran into issues with session variables not persisting between routes, and credentials not being sent with cookies due to certain React settings. So we found that jsonwebtokens are a useful and secure way to maintain a user's session. Currently, we create a signed JWT when a user logs in that stores their username and fullname. Every time a user calls a route, this signed JWT is passed along as a Bearer Token, and its signature and origin is verified by the server. If the token expires or it has been modified, the user is immediately logged out of the application, and the token is discarded. We also discard tokens when a user manually logs out. The current expiration of the JWT session is 4 hours.

3.4. Socket.io

We decided to use socket.io to handle bidirectional communication between two users who are chatting with one another. Web sockets were unfamiliar to all of us for this project, so it was a challenge understanding how to use them effectively, as well as integrating it into a React/Express application. We utilized socket.io's room feature to ensure that for each chat, only users within that chat can receive socket messages for that chat. We also implemented notifications through socket.io, so that when a user is not in the chat session itself, but still active on the application, they will receive a popup notifying them of a message and attaching a link to the chat session from which the message originated.

4. Changes and Lessons Learned

4.1. Displaying a user's full name

At first, we were just displaying usernames along with posts and comments, but we decided we wanted to display full names and hadn't initially thought about storing them in the database, so we went back and added full name as a field in every database that relied upon displaying a full name (i.e. posts, comments, friends, chat) and updated our routes to account for this fact.

4.2. Tracking Online Users

We initially kept track of online users by creating a boolean field online and storing it with each user's account and changing this when a user logged in or out. However, this meant when a tab was closed or the user hadn't visited the page in a while but was still logged in, it would still show the user as active. Instead, whenever a route is called from an online user, we update the last_active timestamp in the accounts table. Then, when displaying online friends or users, we take users whose last_active timestamp is in the last ten minutes.

4.3. News Search Cache and Pagination

At first, we retrieved and displayed all the corresponding articles from the articles_keyword table for every search query made by the user. However, this resulted in quite a bit of delay in our application due to retrieving 1000s of articles, rendering them, and fetching all of their likes and comments information. In order to speed up the process, especially for popular queries with many results, we decided to paginate our results. For the first time a particular query is made, we fetch all the corresponding articles, sort them, and put them into our cache table (mapping from search query to list of article ids). Then, we simply take the first 10 results and display that as the first page. If the user wishes to see the next 10 results, we don't have to recompute the results again because it is already stored in our cache, and can index into and return the next 10. This resolved our problem of significant delays in the application, because we only have to process and render 10 articles at a time. In general it also helps for popular queries made by many users over time, because we don't have to keep retrieving and sorting the same articles over and over again.

4.4. Integration with Java

Java gave us a few issues even though it is platform independent and despite the fact that we used Maven as a package manager. We dealt with many AWS errors due to internal usages of S3 http file streaming inside Hadoop. We had to work around not throwing certain errors and bypassing some internal library calls that we had no control over. Furthermore, we decided to trigger Livy with a Maven command. However, it turns out AWS Linux yum package manager did not have the latest versions of Java and Maven. We therefore had to manually install the binaries and get comfortable with manipulating bash shell variables. All in all, we became more versed in some DevOps topics beyond just writing code.

5. Extra Credit

5.1. Replicating Facebook-style Likes for Posts

We implemented Facebook-style likes, where the number of likes is displayed on each post and a user can only like a post once; they can like and unlike posts. The implementation of this is discussed above in Section 2.2.

5.2. Liking Comments and Articles

We implemented Facebook-style likes for comments and articles as well.

5.3. Pending/Accepting Friend Requests

We implemented friend requests so that one user sends a request and the other can accept or deny the request.

6. Work Distribution

6.1. Batchema (25%)

News Recommendation System (Spark and Livy jobs, server cron job, on demand recommendations update trigger, news recommendations routes and database management). Code deployment environment set up (Maven and Java installations, process.env set up for AWS sdk and AWS credentials management config, shell environments configurations)

6.2. Divya (25%)

Visualizer, News Search, routes/DB functions for account changes, routes/DB functions for home page and wall, a few routes/DB functions for friendships

6.3. Ishaan (25%)

All of the React frontend (Home, News Feed, Chat, Visualizer, Profile, Search, Login/Signup etc.), routes/DB functions for chat, socket.io for chat, session management with JWTs

6.4. Olivia (25%)

Account creation and password hashing, routes/DB functions for friends, routes/DB functions for comments, routes/DB functions for likes

7. Acknowledgments

We would like to thank Professors Andreas Haeberlen and Zachary Ives for their teachings this semester, as well as Bharath Jaladi for being our project shepherd.

8. Screenshots

Below are screenshots of some of our core features:

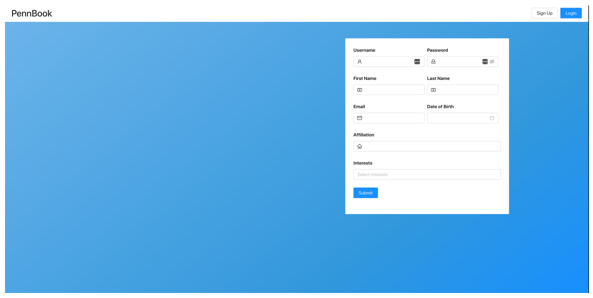
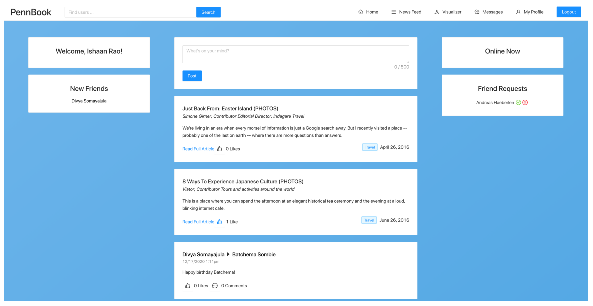


Figure 1. Signup



Welcome, Ishaan Rao!

New Friends
Divya Somayajula

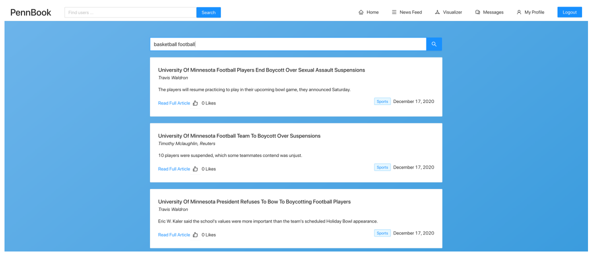
Andreas Haeberlen

Just Back From: Easter Island (PHOTOS)
Simone Girner, Contributor Editorial Director, Indagare Travel
We're living in an era when every morsel of information is just a Google search away. But I recently visited a place -- probably one of the last on earth -- where there are more questions than answers.
Read Full Article 0 Likes Travel April 26, 2016

8 Ways To Experience Japanese Culture (PHOTOS)
Viator, Contributor Tours and activities around the world
This is a place where you can spend the afternoon at an elegant historical tea ceremony and the evening at a loud, blinking internet cafe.
Read Full Article 1 Like Travel June 26, 2016

Divya Somayajula ▸ Batchema Sombie
12/17/2020 1:11pm
Happy birthday Batchema!
0 Likes 0 Comments

Figure 2. Home



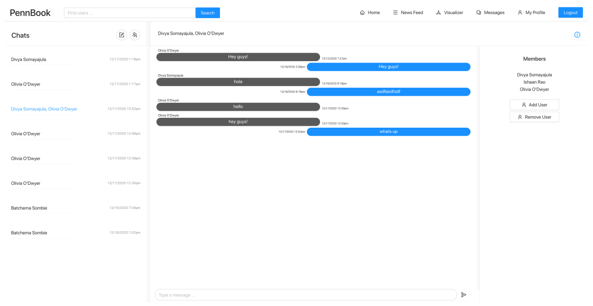
basketball football

University Of Minnesota Football Players End Boycott Over Sexual Assault Suspensions
Travis Waldron
The players will resume practicing to play in their upcoming bowl game, they announced Saturday.
Read Full Article 0 Likes Sports December 17, 2020

University Of Minnesota Football Team To Boycott Over Suspensions
Timothy Mclaughlin, Reuters
10 players were suspended, which some teammates contend was unjust.
Read Full Article 0 Likes Sports December 17, 2020

University Of Minnesota President Refuses To Bow To Boycotting Football Players
Travis Waldron
Eric W. Kaler said the school's values were more important than the team's scheduled Holiday Bowl appearance.
Read Full Article 0 Likes Sports December 17, 2020

Figure 3. News Search



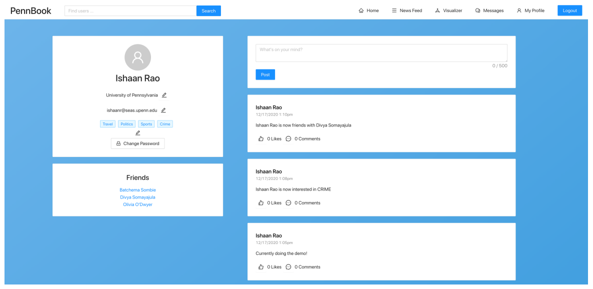
Divya Somayajula	12/17/2020 1:18pm
Olivia O'Dwyer	12/17/2020 1:17pm
Divya Somayajula, Olivia O'Dwyer	12/17/2020 12:52pm
Olivia O'Dwyer	12/17/2020 12:48pm
Olivia O'Dwyer	12/17/2020 12:38pm
Olivia O'Dwyer	12/17/2020 12:35pm
Batchema Sombie	12/16/2020 7:04pm
Batchema Sombie	12/16/2020 7:02pm

Divya Somayajula, Olivia O'Dwyer

Olivia O'Dwyer
Hey guys! 12/12/2020 7:27pm
12/16/2020 2:56pm Hey guys!
Divya Somayajula
hola 12/16/2020 9:18pm
12/16/2020 9:19pm asdfasdfsdf
Olivia O'Dwyer
hello 12/17/2020 12:30pm
Olivia O'Dwyer
hey guys! 12/17/2020 12:43pm
12/17/2020 12:52pm whats up

Divya Somayajula
Ishaan Rao
Olivia O'Dwyer

Figure 4. Chat



Ishaan Rao
University of Pennsylvania
ishaanr@seas.upenn.edu
Travel Politics Sports Crime

Batchema Sombie
Divya Somayajula
Olivia O'Dwyer

Ishaan Rao
12/17/2020 1:10pm
Ishaan Rao is now friends with Divya Somayajula
0 Likes 0 Comments

Ishaan Rao
12/17/2020 1:08pm
Ishaan Rao is now interested in CRIME
0 Likes 0 Comments

Ishaan Rao
12/17/2020 1:05pm
Currently doing the demo!
0 Likes 0 Comments

Figure 5. Profile