Vin Liu

Qi Zhang

Professor Torsten Suel

CS-GY 6083

18 December 2019

**Project Report**

In this document, we derive a relational schema for a web application that supports location-based user communities. The web application allows a user to register an account to access the services and build up social networks within two levels of locality, hoods, and blocks. A user can add friends and neighbors in his/her social networks and send direct messages with them. A user can also post and reply messages within a community of a block or a hood to make his or her messages visible to all the community members. A user can filter and display incoming messages with various scopes. We will explain how we achieve those core functional requirements as well as many user scenarios in this document.

**1. Logical Design**

**ER model**

The first step is the logical design. We started with the ER model. we modeled six entities as well as the relationships associate among them. Then used the ER diagram to express the overall logical structure of our database. In part 2 of the project, we decided to add another relation, Approval, to describe the action when a user who is an existing block member approves a pending member of that block. With the aid of Approval schema, we can ensure one user can only approve a pending member once. The ER diagram is attached below:

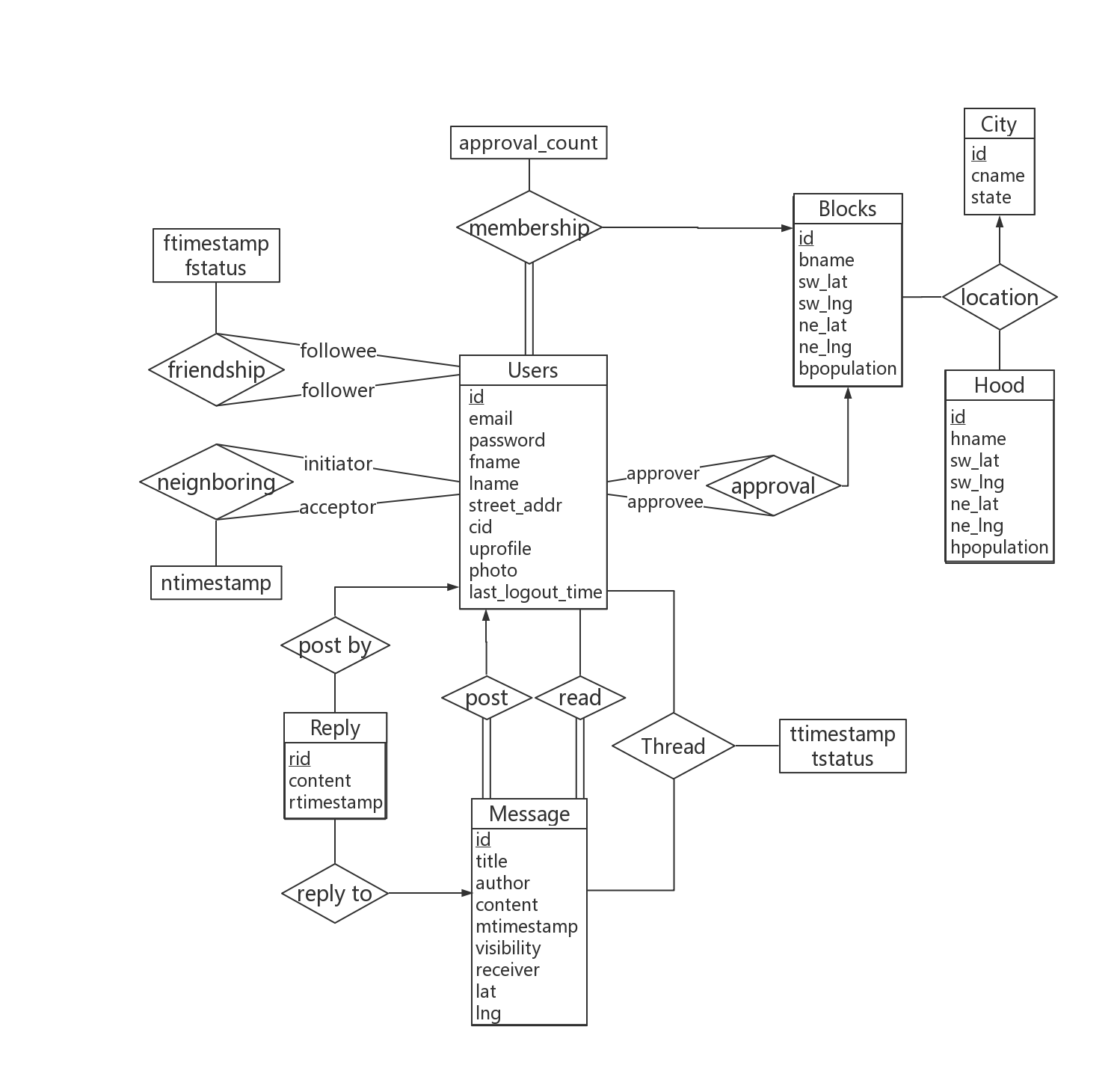


Figure 1 ER diagram

**Relational Schemas**

Then we translated the above ER schemas into relational schemas:

* Users (id, email, pword, fname, lname, street\_addr, *cid*, uprofile, photo, last\_logout\_timestamp)
* Hood (id, hname, sw\_lat, sw\_lng, ne\_lat, ne\_lng, hpopulation)
* Blocks (id, bname, sw\_lat, sw\_lng, ne\_lat, ne\_lng, bpopulation)
* Location (*bid*, *hid*, *cid*)
* Membership (*uid*, *bid*, approval\_count)
* Friendship (*follower*, *followee*, ftimestamp, fstatus)
* Neighboring (*initiator*, *acceptor*, ntimestamp)
* Message (id, *author*, title, content, mtimestamp, visibility, *receiver*, lat, lng)
* Reply (rid, *mid*, *author*, content, rtimestamp)
* Thread (*uid*, *mid*, tstatus, ttimestamp)
* City (id, cname, cstate)
* Approval (*approver*, *approvee*, *bid*)

In the schemas above, the primary key is labeled with an underline, and the foreign key is in *italics*.

Next, we are going to claim the constraints, assumptions, and justifications we made for the schemas.

|  |  |
| --- | --- |
| **Users** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| email | VARCAHR(45), NOT NULL, UNIQUE |
| pword | VARCAHR(45), NOT NULL |
| fname | VARCAHR(45), NOT NULL |
| lname | VARCAHR(45), NOT NULL |
| street\_addr | VARCAHR(45), NOT NULL |
| cid | INTEGER, FOREIGN KEY referencing cid in City |
| uprofile | TEXT |
| photo | TEXT |
| last\_logout\_timestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP, ON UPDATE CURRENT\_TIMESTAMP |

Table 1 User

* id is an auto-generated and auto-incremented number serves as a unique identifier of a user. It is generated upon user registration. The value is not recycled when a user is deleted.
* A user does not know his or her id. An email directly serves as a username. A user registers and logins via his or her email and password.
* fname and lname store the first name and last name of a user, respectively.
* street\_addr stores street address of a user, which be manually entered by the user or by selected on a map upon user registration.
* uprofile stores a profile, like a bio, for a user. It is nullable.
* photo stores an avatar for a user. It is nullable.
* last\_logout\_timestamp is a timestamp auto-generated and updated every time a user logouts.

|  |  |
| --- | --- |
| **Hood** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| hname | VARCAHR(45), NOT NULL |
| sw\_lat | FLOAT, NOT NULL |
| sw\_lng | FLOAT, NOT NULL |
| ne\_lat | FLOAT, NOT NULL |
| ne\_lng | FLOAT, NOT NULL |
| hpopulation | INTEGER, NOT NULL |

Table 2 Hood

* id is an auto-generated and auto-incremented number that serves as a unique identifier of a hood. The value is not recycled when a user is deleted. According to the project description, we can predefine hoods. Users register into a block and a hood predefined in the system. So, we assume that the addition or deletion of a hood is not common.
* hname stores the name of a hood.
* sw\_lat, sw\_lng, ne\_lat, ne\_lng store the coordinates of a rectangle defining the region of a hood on a map.
* hpopulation stores the population of a hood. It should be a sum of the population in all blocks within the hood. This will be implemented in our backend application.

|  |  |
| --- | --- |
| **Blocks** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| bname | VARCAHR(45), NOT NULL |
| sw\_lat | FLOAT, NOT NULL |
| sw\_lng | FLOAT, NOT NULL |
| ne\_lat | FLOAT, NOT NULL |
| ne\_lng | FLOAT, NOT NULL |
| bpopulation | INTEGER, NOT NULL |

Table 3 Blocks

* The setting of this schema is similar to Hood.
* bpopulation tracks the population of a block. According to the project description, a user’s request to join a block needs to be approved by three members in that particular block or by all members if this block has less than three members. We will need to compare approval\_count in Membership with bpopulation to achieve that.

|  |  |
| --- | --- |
| **Location** | |
| bid | INTEGER PRIMARY KEY, FOREIGN KEY referencing id in Blocks |
| hid | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Hood |
| cid | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in City |

Table 4 Location

* We use the composite primary key here to handle the problem of repetition of hood names or block names in different cities. For example, names like “High Street”, “Main Street”, and “Midtown” often appear in many cities.

|  |  |
| --- | --- |
| **Membership** | |
| uid | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| bid | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Blocks |
| approval\_count | INTEGER, NOT NULL, DEFAULT 0 |

Table 5 Membership

* The uid and the bid here represent a potential membership, which means a user is a member of a block. This membership is uniquely identified by the composite primary key.
* As we mentioned in Blocks, we will compare approval\_count with bpopulation to determine if this potential membership is pending or passed. We decided once the user is approved to join a block, his/her approval\_count is set to -1.

|  |  |
| --- | --- |
| **Friendship** | |
| follower | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| followee | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| ftimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |
| fstatus | VALUE IN (‘pending’, ‘rejected’, ‘accepted’), DEFAULT ‘pending’ |

Table 6 Friendship

* follower and followee here are foreign keys referencing to id in Users. A follower adds a followee to create a potential friendship at the ftimestamp, and a followee can accept the request to confirm the friendship or not.
* Because a friendship request can be made multiple times, we add ftimestamp to identify them. For example, when A adds B, B rejects it, then A adds B again.
* fstatus is set to ‘pending’ when a friendship request is created. A followee will receive a notification and be able to accept or reject the request. Upon acceptance, the follower will receive a notification. Also, when a fstatus is ‘accepted’, the follower and the followee will have access messages with visibility of ‘friend’ from each other.

|  |  |
| --- | --- |
| **Neighboring** | |
| initiator | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| acceptor | INTEGER, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| ntimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |

Table 7 Neighboring

* The setting of this schema is similar to Friendship.
* Adding of a neighbor is uniliteral according to the project description. An initiator can add an acceptor to his/her neighbor directly without permission. So, we do not need to mark its status like ‘pending’ or ‘accepted’.

|  |  |
| --- | --- |
| **Message** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| author | INTEGER, FOREIGN KEY referencing uid in Users |
| title | TEXT, NOT NULL |
| content | TEXT, NOT NULL |
| mtimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |
| visibility | VARCAHR(45), NOT NULL, VALUE IN (‘direct’, ‘friend’, ‘neighbor’, ‘block’, ‘hood’) |
| receiver | INTEGER, FOREIGN KEY referencing id in Users, NULLABLE, DEFAULT NULL |
| lat | FLOAT, NULLABLE |
| lng | FLOAT, NULLABLE |

Table 8 Message

* id is an auto-generated and auto-incremented number, serves as a unique identifier of a message. It is generated upon posting a message. The value is not recycled when a reply is deleted.
* mtimestamp here is for displaying when the message is posted.
* A message has visibility of ‘direct’, ‘friend’, ‘neighbor’, ‘block’, and ‘hood’. When a user chooses to post a direct message, a text field for entering a message receiver is shown. This is done by the front end. Then the receiver is recorded in receiver attribute. However, when visibility is set to be ‘friend’, ‘neighbor’, ‘block’, or ‘hood’, the receiver text field is hidden, and receiver attribute is set to be NULL.
* lat and lng are coordinates where the message is pinned on a map. Users can choose to not pin the message.

|  |  |
| --- | --- |
| **Reply** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| mid | INTEGER, FOREIGN KEY referencing id in Message |
| author | INTEGER, FOREIGN KEY referencing id in Users |
| content | TEXT, NOT NULL |
| rtimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |

Table 9 Reply

* rid is an auto-generated and auto-incremented number, serves as a unique identifier of a reply. It is generated upon posting a reply. The value is not recycled when a reply is deleted.
* mid represents that this reply is replying to the message with the particular message.
* rtimestamp here is for displaying when the reply is posted.
* According to the project description, a replay has the same visibility as the original message, so we do not need a visibility attribute here.
* According to the project description, we can decide the reply depth on our own judgment. So, we decided the reply depth to be 1, which means all replies are replying to the original message. They are organized by chronological order with the aid of rtimestamp.

|  |  |
| --- | --- |
| **Thread** | |
| uid | INTEGER, FOREIGN KEY referencing id in Users, PRIMARY KEY |
| mid | INTEGER, FOREIGN KEY referencing id in Message, PRIMARY KEY |
| tstatus | VARCAHR(45), NOT NULL, VALUE IN (‘read’, ‘unread’), DEFAULT ‘unread’ |
| ttimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |

Table 10 Thread

* A thread here is defined as a piece of information consisting of a message and corresponding replies. In essence, we decide to pre-compute what information a user should be able to see when he or she accesses the timeline news feed. Such pieces of information are identified by uid and mid. Twitter uses pre-computed information for its timeline too. But the information is stored in memory by the aid of Redis. Definitely, Twitter’s algorithm will be much more sophisticated. We would like to try a simplified approach here by storing pre-computed timeline in Thread schema.
* By pre-computing the timeline, we mean that when a message is posted, we immediately determine who can read this message. For example, when A post a direct message to B, with A’s uid=1, B’s uid=2, and mid=3, we add (2, 3, ‘unread’, CURRENT\_TIMESTAMP) to Thread schema. So, when B loads his or her timeline news feed, a message with mid=3 will be displayed. Similarly, when A post a message to his or her friends, entries with A’s friends’ id will be added. Anyone who is not his or her friend will not even know the message is posted by the querying Thread table, and hence the thread is not going to be in his or her timeline. In this way, we pre-compute the timeline for each user. Automatically computing uids and inserting entries into the Thread table are done by a server application.
* tstatus determines whether a user has read the message. It helps when we need to display the unread message only.
* Each time a reply is posted for a message, we automatically set tstatus back to ‘unread’. This is done by the same server application. So, the audience will be about to notice the reply.
* ttimestamp can help when we need to display messages or replies after the last login.

|  |  |
| --- | --- |
| **City** | |
| id | INTEGER, UNIQUE, PRIMARY KEY |
| cname | VARCAHR(45), NOT NULL |
| state | VARCAHR(45), NOT NULL |

Table 11 City

* id is an auto-generated and auto-incremented number, serves as a unique identifier of a city. The value is not recycled when a reply is deleted. According to the project description, blocks and hoods are predefined for simplicity. So, we assume cities and states are also predefined. We will retrieve cities and states information from open-source data on the internet. Users select them upon user registration.
* By using id as a primary key, we can distinguish the cities with the same name but in different states. For example, the name ‘Portland’ appears in both Oregon and Maine.

|  |  |
| --- | --- |
| **Approval** | |
| approver | INTEGER, NOT NULL, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| approvee | INTEGER, NOT NULL, PRIMARY KEY, FOREIGN KEY referencing id in Users |
| bid | INTEGER, NOT NULL, PRIMARY KEY, FOREIGN KEY referencing id in Blocks |

Table 12 Approval

* approver and approvee here are foreign keys referencing to id in Users. An approver approves an approvee’s pending membership to join his/her block.
* bid is a primary key and a foreign key referencing to id in Blocks. It specifies which blocks approval is for.

**Function Implementation**

We create our schema by using SQLite. The required functions are done using the following SQL queries:

1. User Y lives in city X and becomes a member of block Z

**Sign up:**

INSERT INTO Users(email, pword, fname, lname, street\_addr, cid) VALUE (‘zl1477@nyu.edu’, ‘pword’, ‘Vin’, ‘Liu’, ‘110 1st St.’, X);

**Become a member of a block:**

INSERT INTO Membership(uid, bid) VALUE (X,Y);

**Create or edit profile:**

UPDATE Users SET uprofile = 'I love SQL!'

WHERE id = Y;

1. User X posts a message U at coordinate (Y, Z). User V replies it.

**Post an initial message:**

INSERT INTO Message(author, title, content, mtimestamp, visibility, receiver, lat, lng) VALUE (X, 'Posting test case', 'Hi there! I am testing the c2 by posting a new message', CURRENT\_TIMESTAMP, 'hood', NULL, Y, Z);

**Reply to a message:**

INSERT INTO Reply(mid, author, content, rtimestamp) VALUE

(U, V, 'Hi there! I am testing the c2 by replying a new message', CURRENT\_TIMESTAMP);

1. User X adds Y as a friend. User X adds Y as a neighbor. User Y accepts the friendship.

**Add friend:**

INSERT INTO Friendship VALUE

(X, Y, CURRENT\_TIMESTAMP, 'pending');

**Add neighbor:**

INSERT INTO Neighboring VALUE

(X, Y, CURRENT\_TIMESTAMP);

**Accept friendship:**

UPDATE Friendship SET fstatus = 'accepted'

WHERE followee = Y AND follower = X;

**List all friends:**

(SELECT follower FROM Friendship

WHERE followee = X)

UNION

(SELECT followee FROM Friendship

WHERE follower = X);

**List all neighbors:**

SELECT accepter FROM Neighboring

WHERE initiator = X;

1. User X

**List all threads that have new message since last access:**

SET @llt = (SELECT last\_logout\_timestamp FROM Users AS u

WHERE u.id = X);

SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X;

SELECT \* FROM Message

WHERE id IN (SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X);

SELECT \* FROM Reply

WHERE mid IN (SELECT id FROM Message

WHERE mid IN (SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X));

**List all unread thread in friend feed:**

SELECT \* FROM Message AS w,

(SELECT id FROM Message

WHERE author IN

(SELECT follower FROM Friendship

WHERE followee = X AND fstatus = 'accepted'

UNION

SELECT followee FROM Friendship

WHERE follower = X AND fstatus = 'accepted')

AND visibility = 'friend') AS m

WHERE w.id IN

(SELECT mid FROM Thread

WHERE uid = X AND tstatus = 'unread') AND m.mid = w.id;

**List all “bicycle accident” threads:**

SELECT \* FROM Thread

WHERE uid = X AND mid IN

(SELECT id FROM Message

WHERE title LIKE '%bicycle accident%' OR content LIKE '%bicycle accident%');

**Sample Data**

We list some sample data below:

Users:

A screen shot of a computer

Description automatically generated

Figure 2 sample data for User

City:

A picture containing scoreboard, text

Description automatically generated

Figure 3 sample data for City

Hood:

A close up of a black background

Description automatically generated

Figure 4 sample data for Hood

Blocks:

A close up of a white wall

Description automatically generated

Figure 5 sample data for Blocks

Location:

A screenshot of a cell phone

Description automatically generated

Figure 6 sample data for Location

Membership:

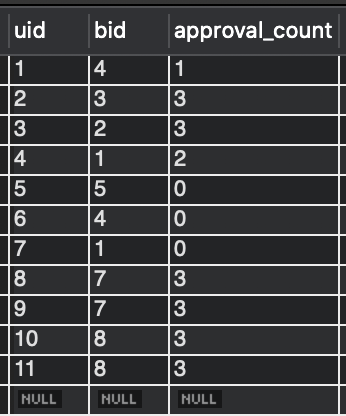


Figure 7 sample data for Membership

Friendship:

A close up of a piece of paper

Description automatically generated

Figure 8 sample data for Friendship

Neighboring:

A close up of a logo

Description automatically generated

Figure 9 sample data for Neighboring

Message:

A screenshot of a computer screen

Description automatically generated

Figure 10 sample data for Message

Reply:

A screenshot of a video game

Description automatically generated

Figure 11 sample data for Reply

Thread:

A close up of a logo

Description automatically generated

Figure 12 sample data for Thread

**Function Test**

In this part, we test the required function asked in project 1

(1) Account

Sign up:

A screenshot of a cell phone screen with text

Description automatically generated

Become a member of a block:

A picture containing screenshot, indoor

Description automatically generated

Create or edit profile:

A picture containing indoor

Description automatically generated

(2) Message

Post an initial message:

A screenshot of a flat screen television

Description automatically generated

Reply a message:A screenshot of a cell phone screen with text

Description automatically generated

(3) Add friend:

A screenshot of a computer screen

Description automatically generated

(4) Add neighbor:

A picture containing screenshot

Description automatically generated

(5) Accept friendship:

A screenshot of a computer screen

Description automatically generated

(6) List all friends:

A screenshot of a cell phone

Description automatically generated

(7) List all threads that have new message since last access:

A screenshot of a computer

Description automatically generated

(8) List all neighbor:

A screenshot of a video game

Description automatically generated

(9) replies:

A screenshot of a cell phone

Description automatically generated

(10) List all unread thread in friend feed:

A screenshot of a video game

Description automatically generated

(11) List all “bicycle accident” threads: (we omit the actual message content, just showing thread)

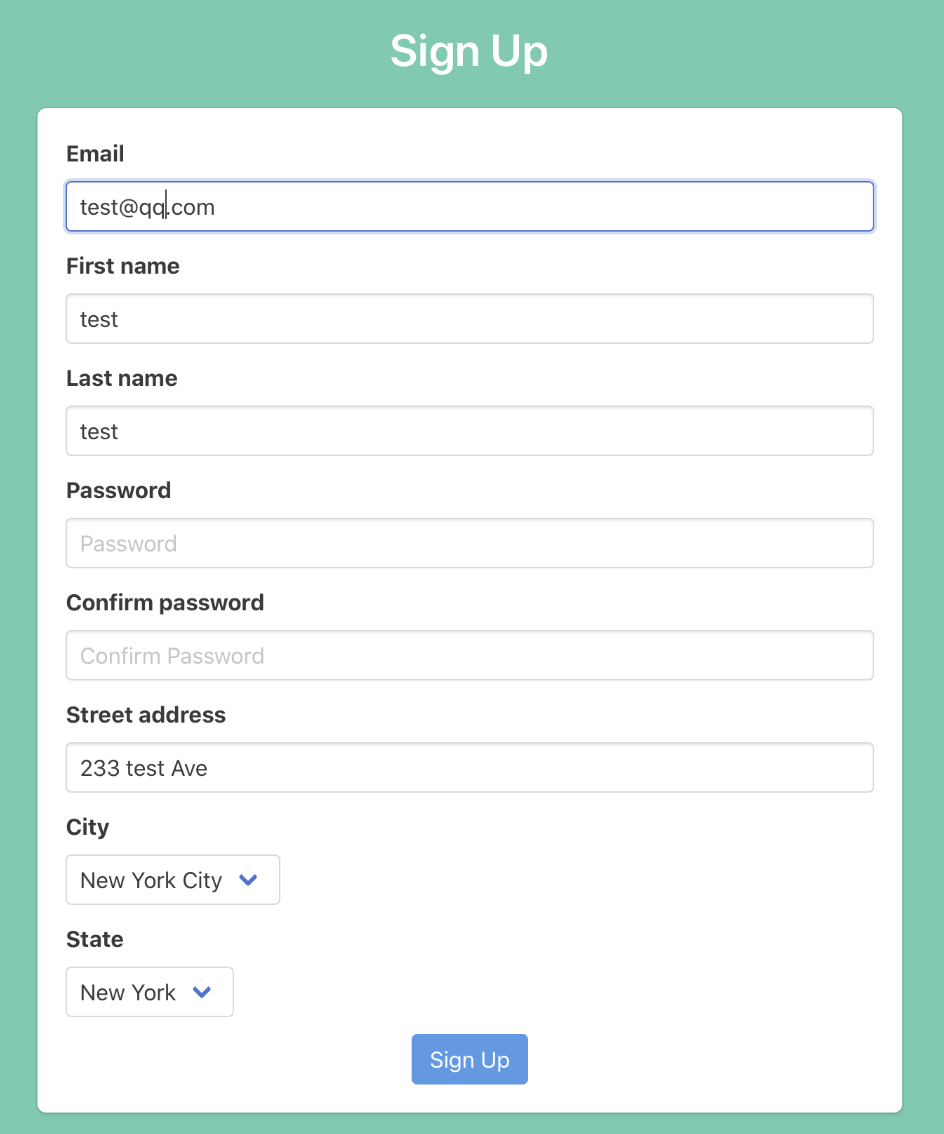
A screenshot of a video game

Description automatically generated

**2. Web Application**

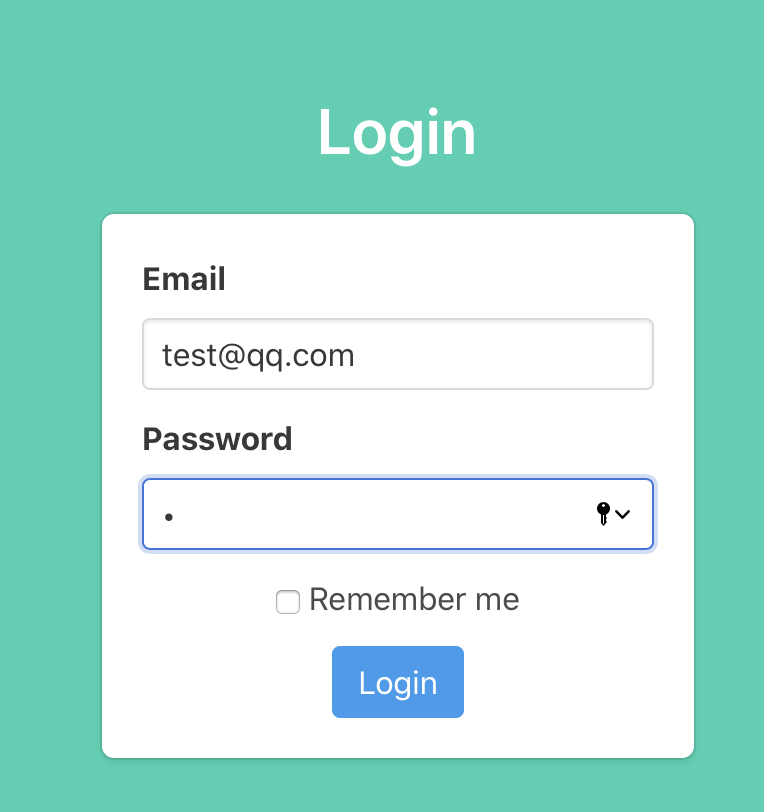
In this part, we design a web-based user interface for the database designed in the first project. We use Python for the web application design, with the help of Flask and SQLAchemy to communicate with the database. The following are examples of how to use this application. (For the installation of the environment, please see README.md in the project folder)

* Creating a New User



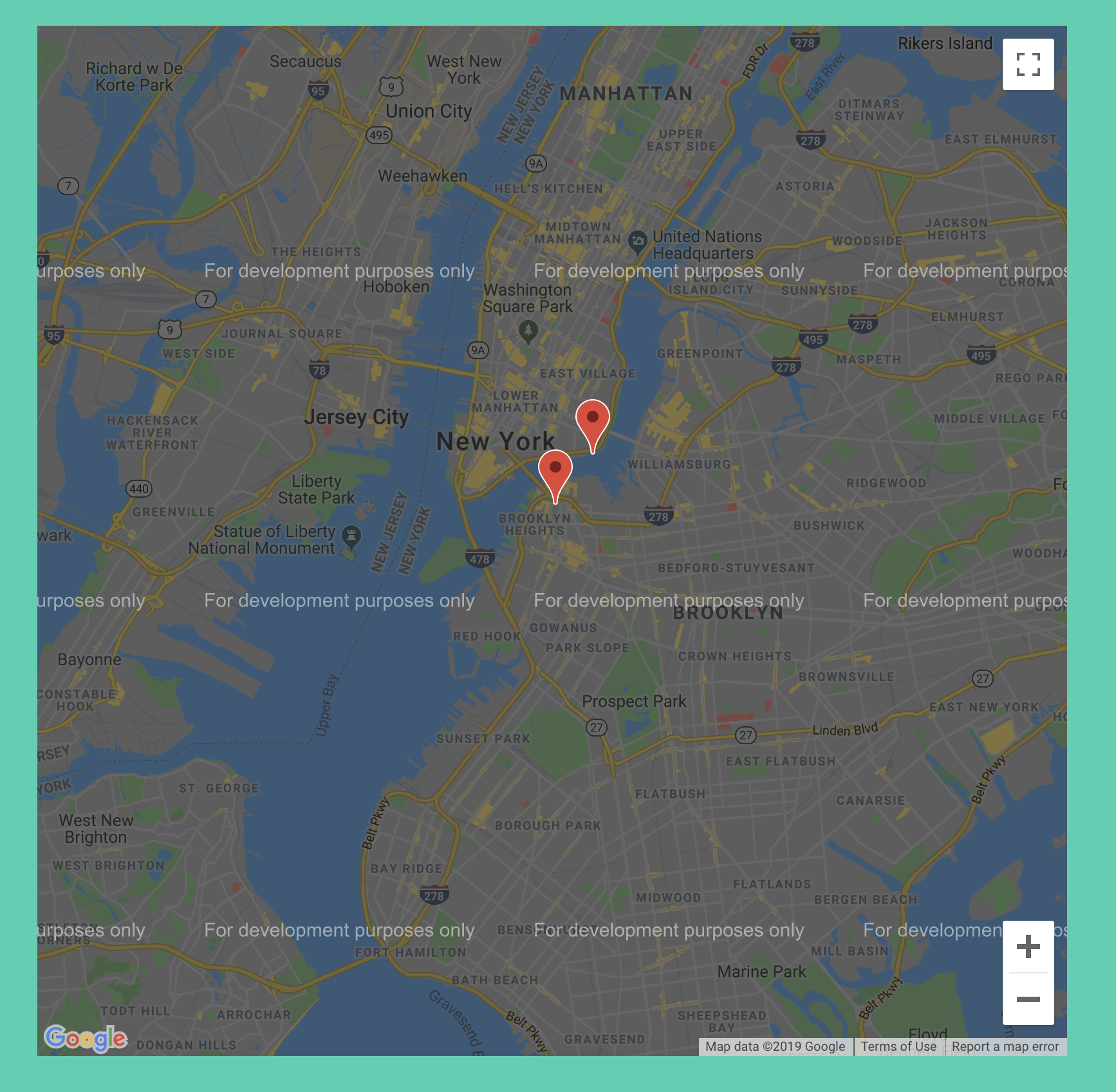
Here we are signing up a test user.

* Login Page



After login, a session will be created for the user to grant access until the user logouts.

* Embedded Map



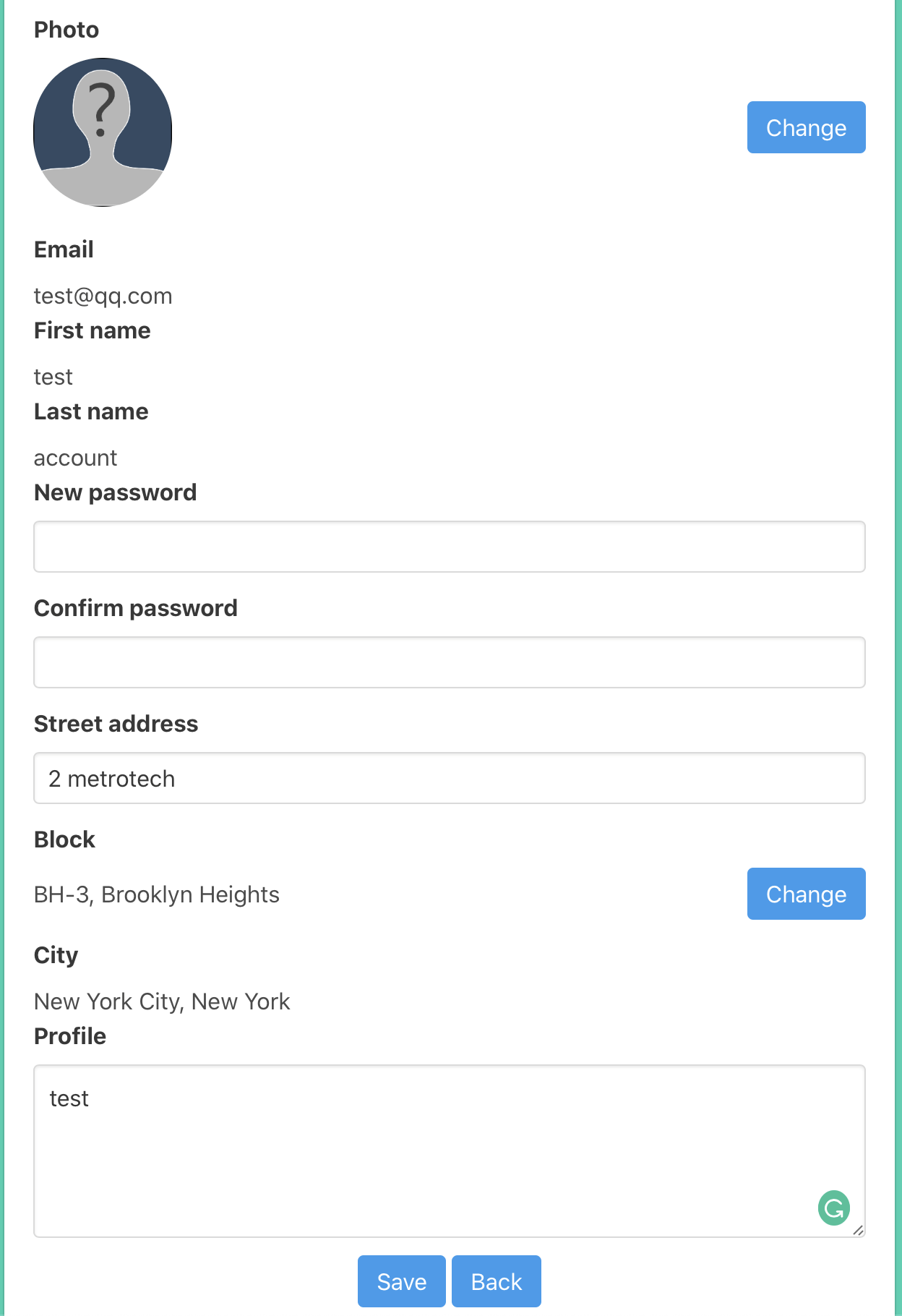
Notifications involving geological data are pinned on the map.

* Navigation Banner



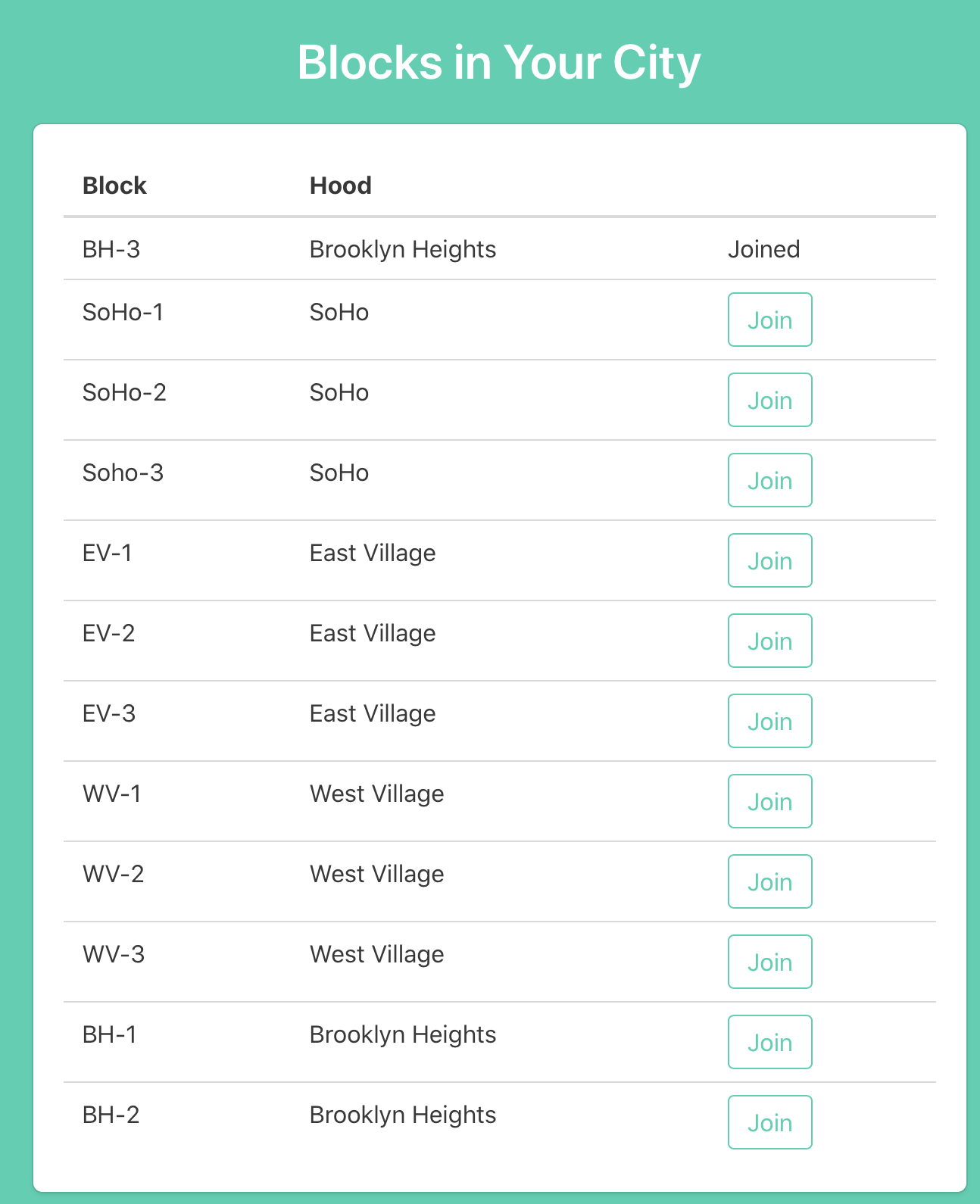
A navigation banner on the top of the page provides a menu for the user to explore the application.

* Changing Account Info

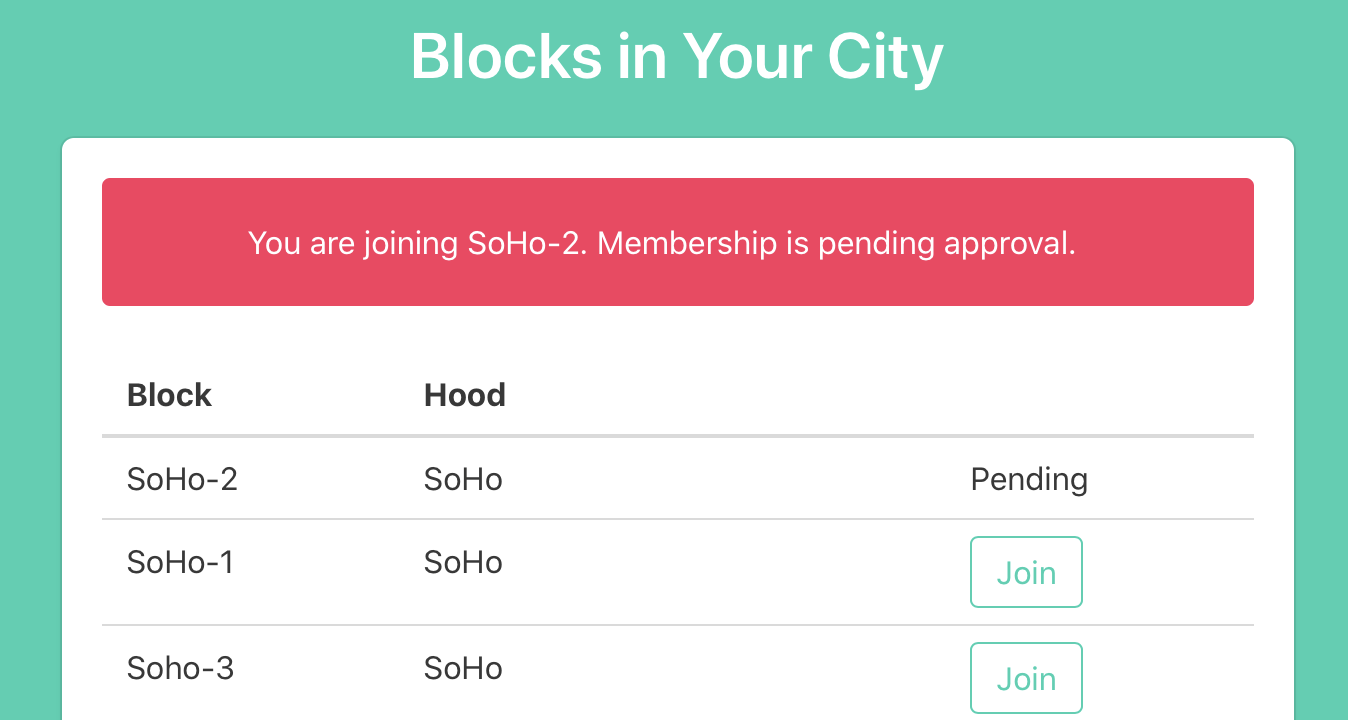


Users can change his/her account information on the Account page.

* Change to another Block or Join a Block

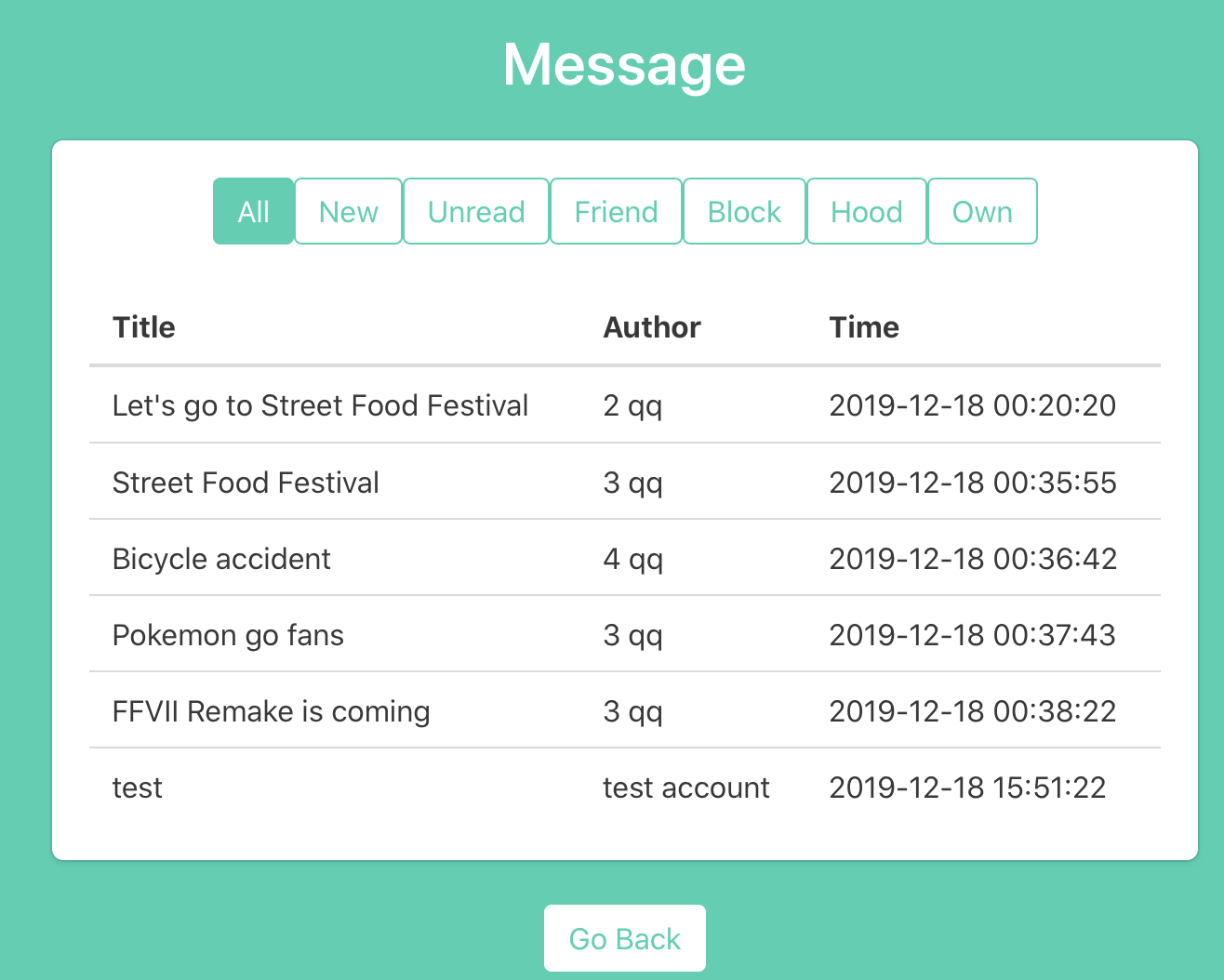


Click the “ change block” button on the Account page lead the user to explore a block list in his/her city. The user can apply to join a block here.

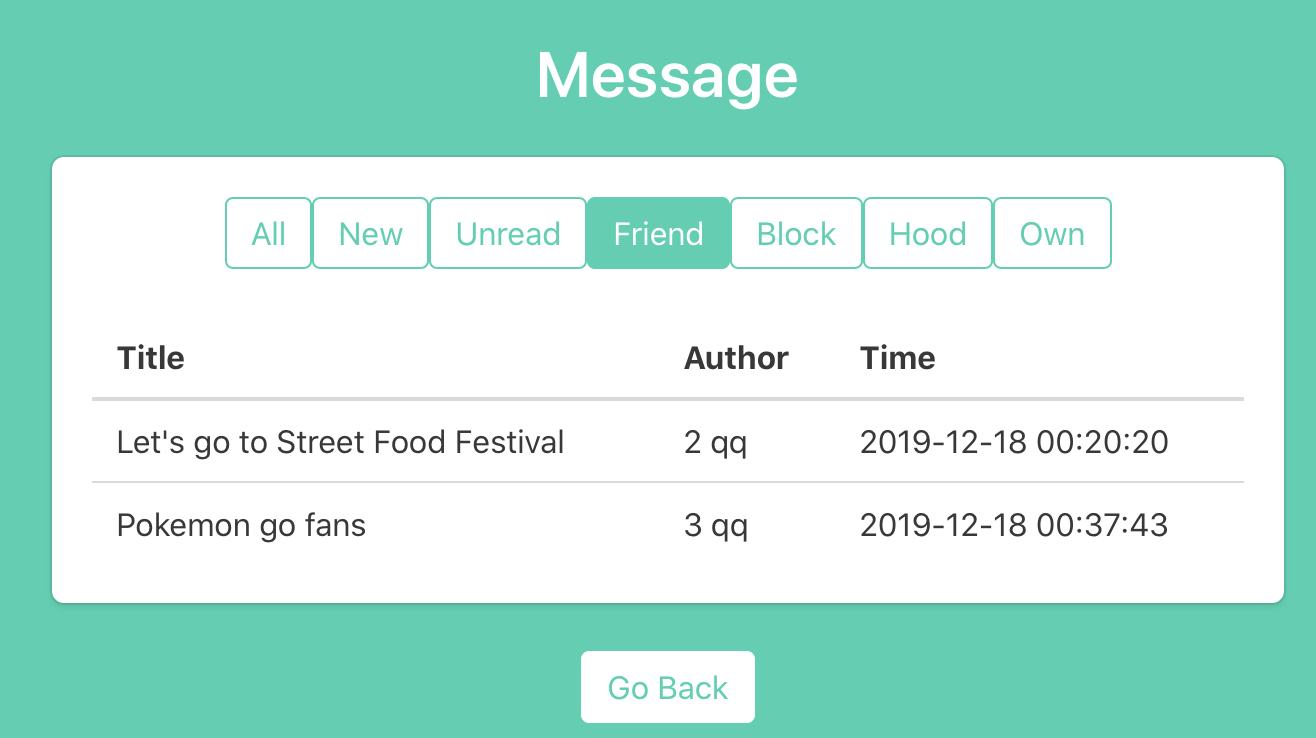


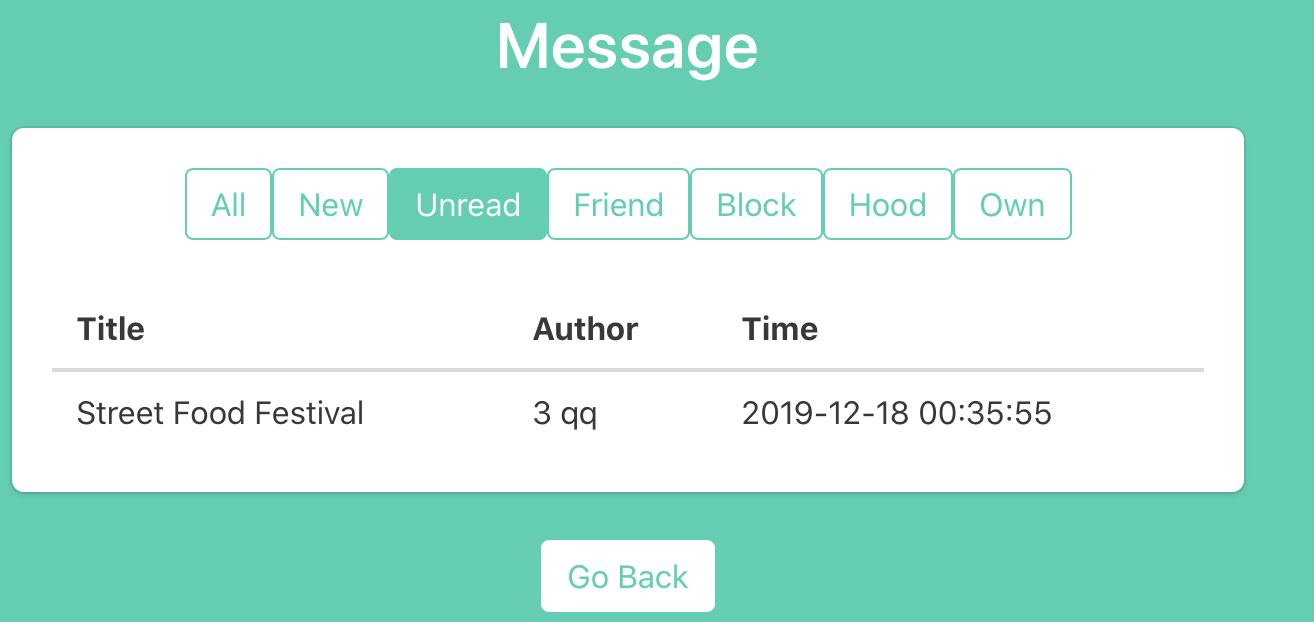
Once a joining application is sent, the membership is pending under approval by existing members in that block. If there the user is the first member in this block, the application will be accepted at once.

* Exploring the Timeline Messages



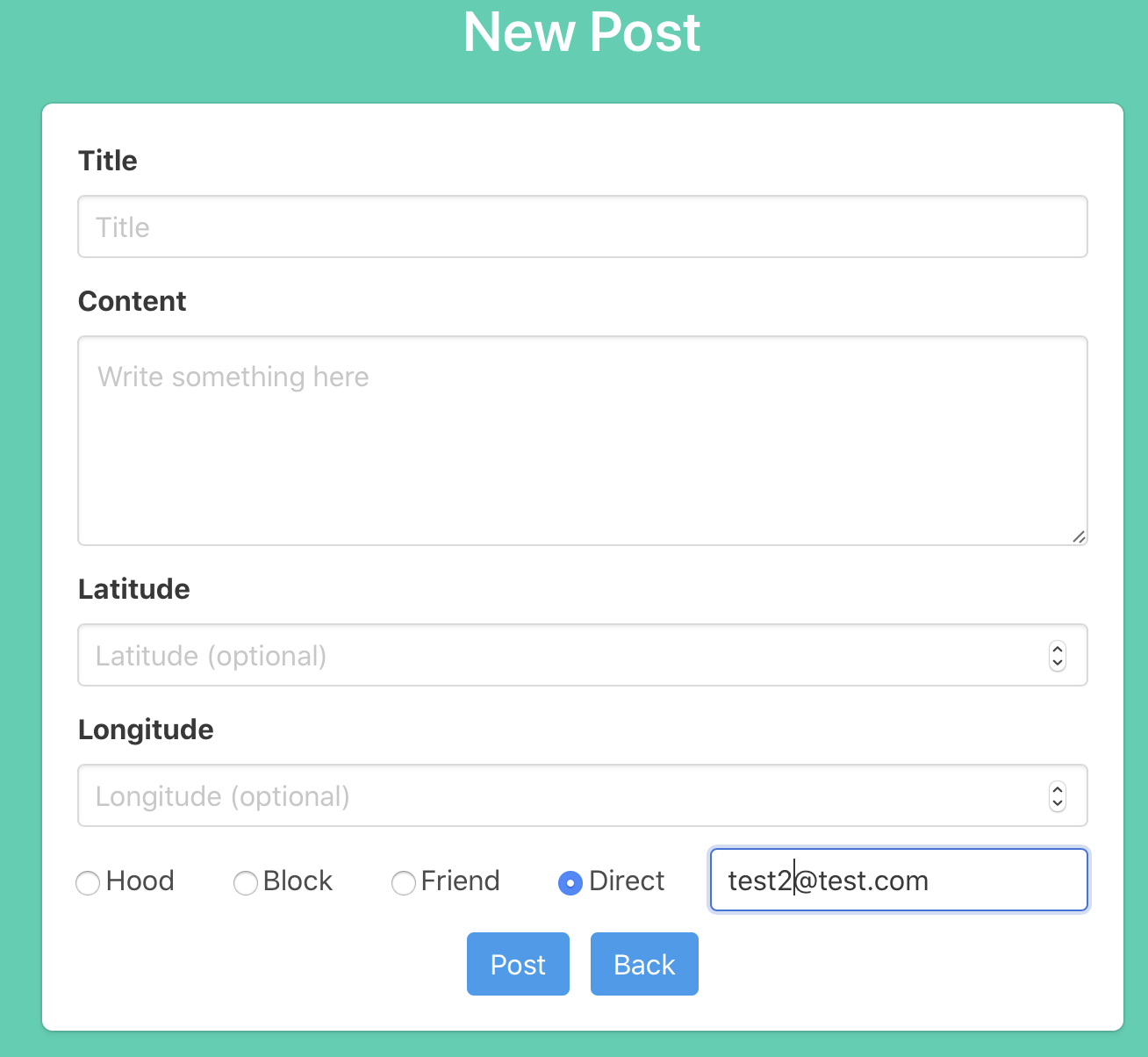
Click Timeline -> View on the navigation to see messages in the user’s timeline.





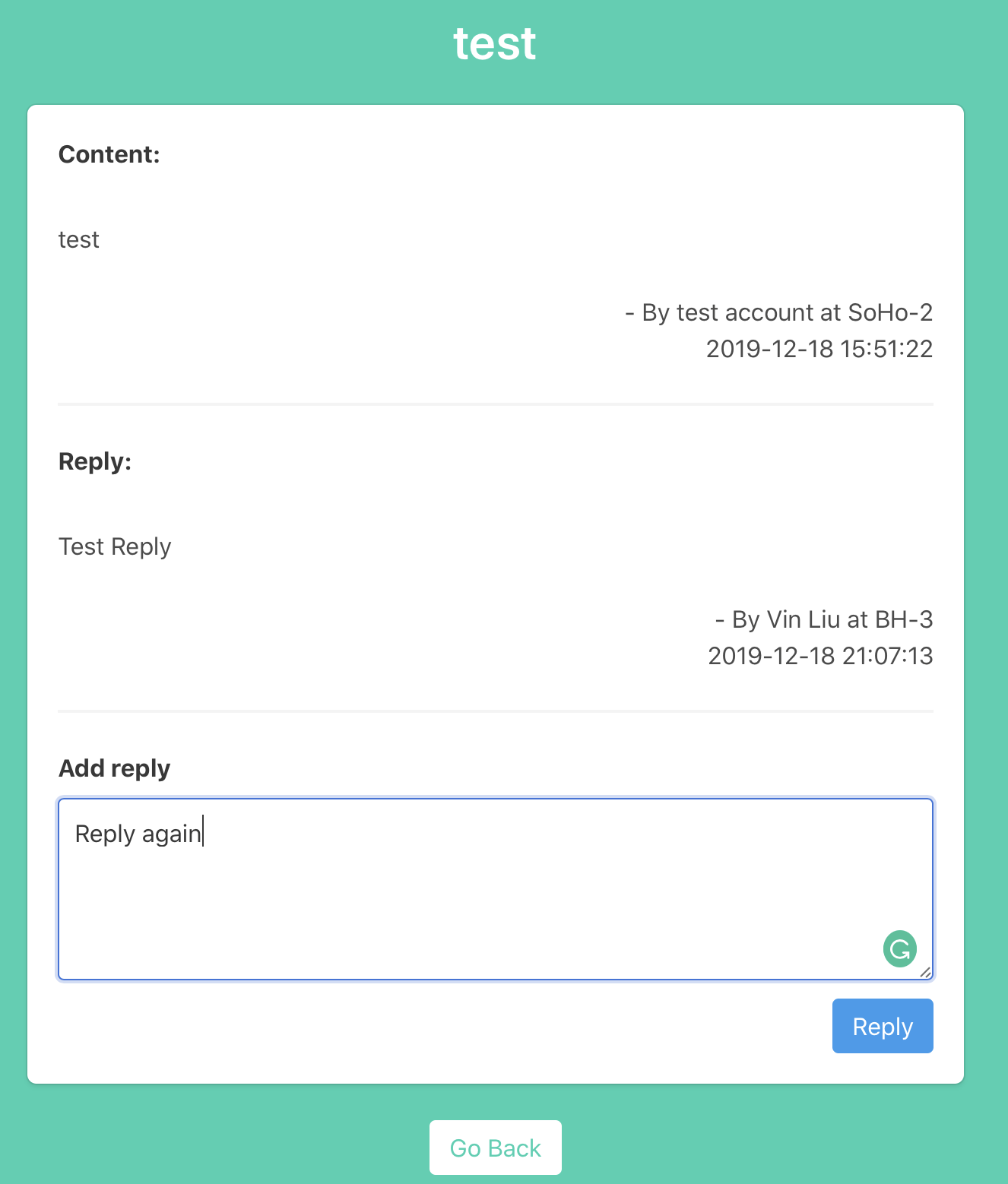
Messages can be categorized into different scopes. Click the tabs to filter the messages. `New` displays all messages posted after the time when the user last logout. `Unread` displays all messages have not been read by the user. `Friend` displays all messages from the user’s friends. `Block` and `Hood` display messages within the geographical scope. The messages posted by the user will be displayed under the `Own` tab.

* Create a Message



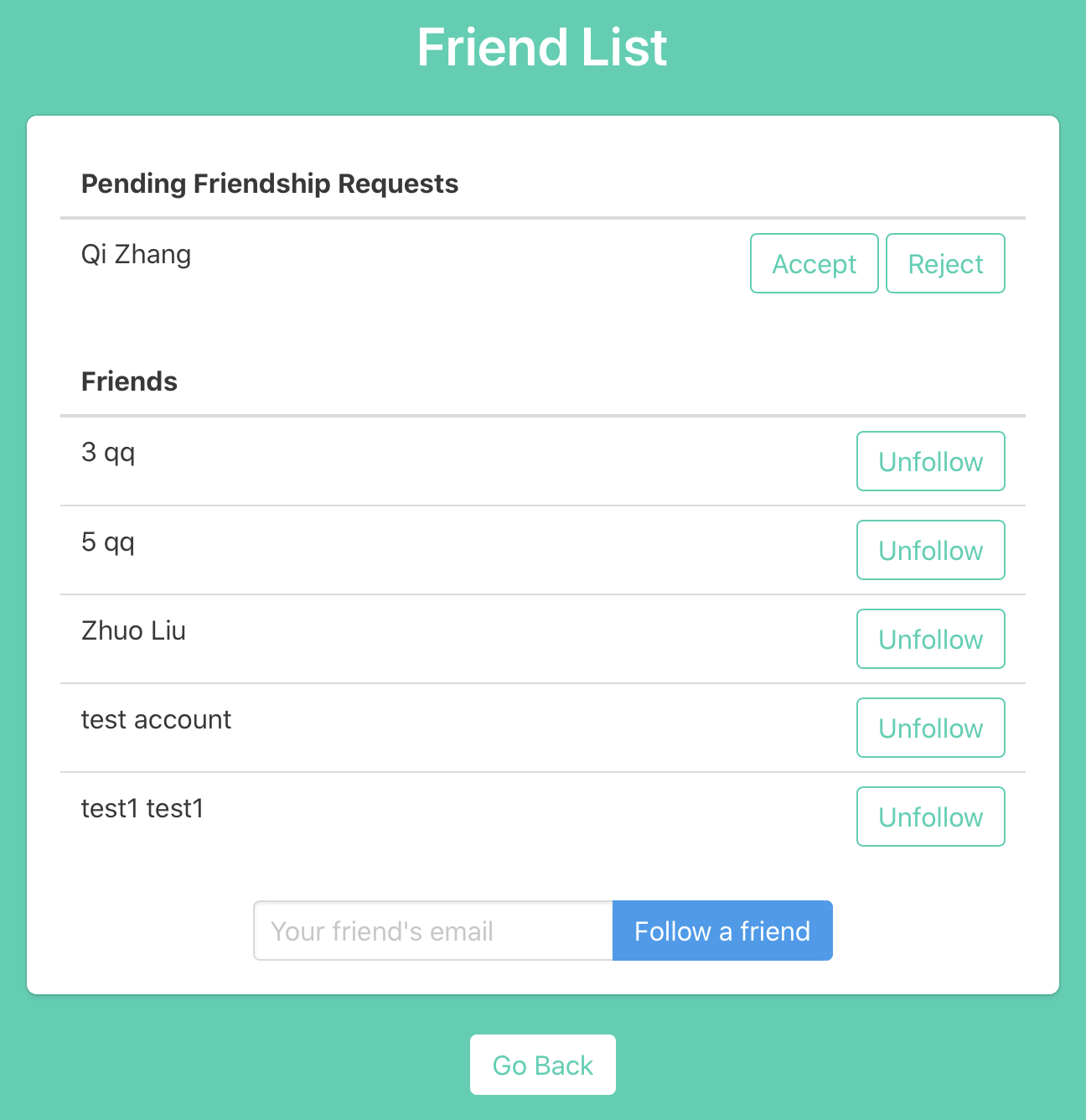
Click Timeline -> Create to compose a new post. The user can choose who can see the post. If the user wants to send a direct message, an email of the recipient user is needed.

* View a Message Detail and Reply a Message

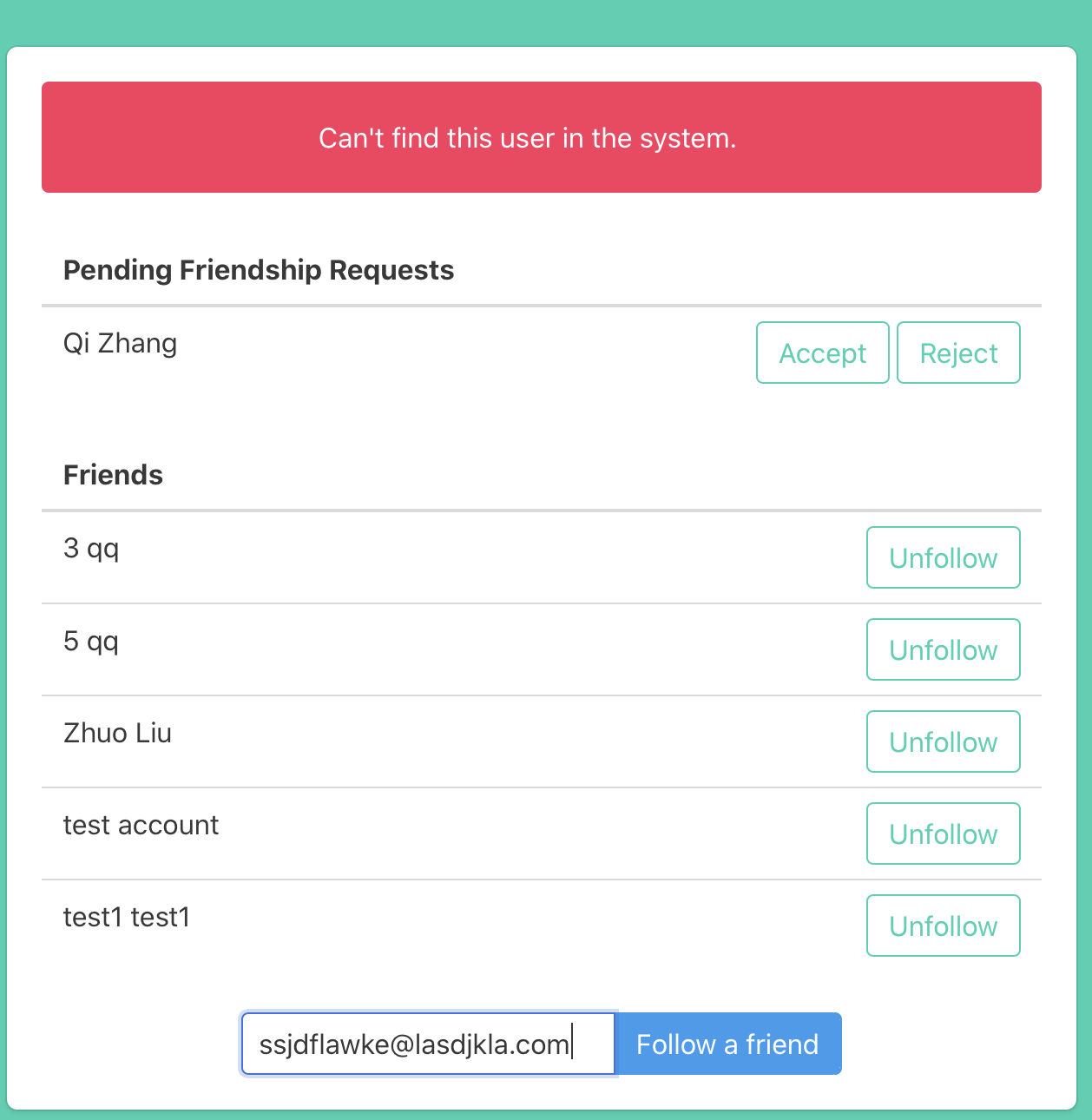


Click the title of any message on the timeline to see the content of the message, along with all existing replies. The user can also reply below the message.

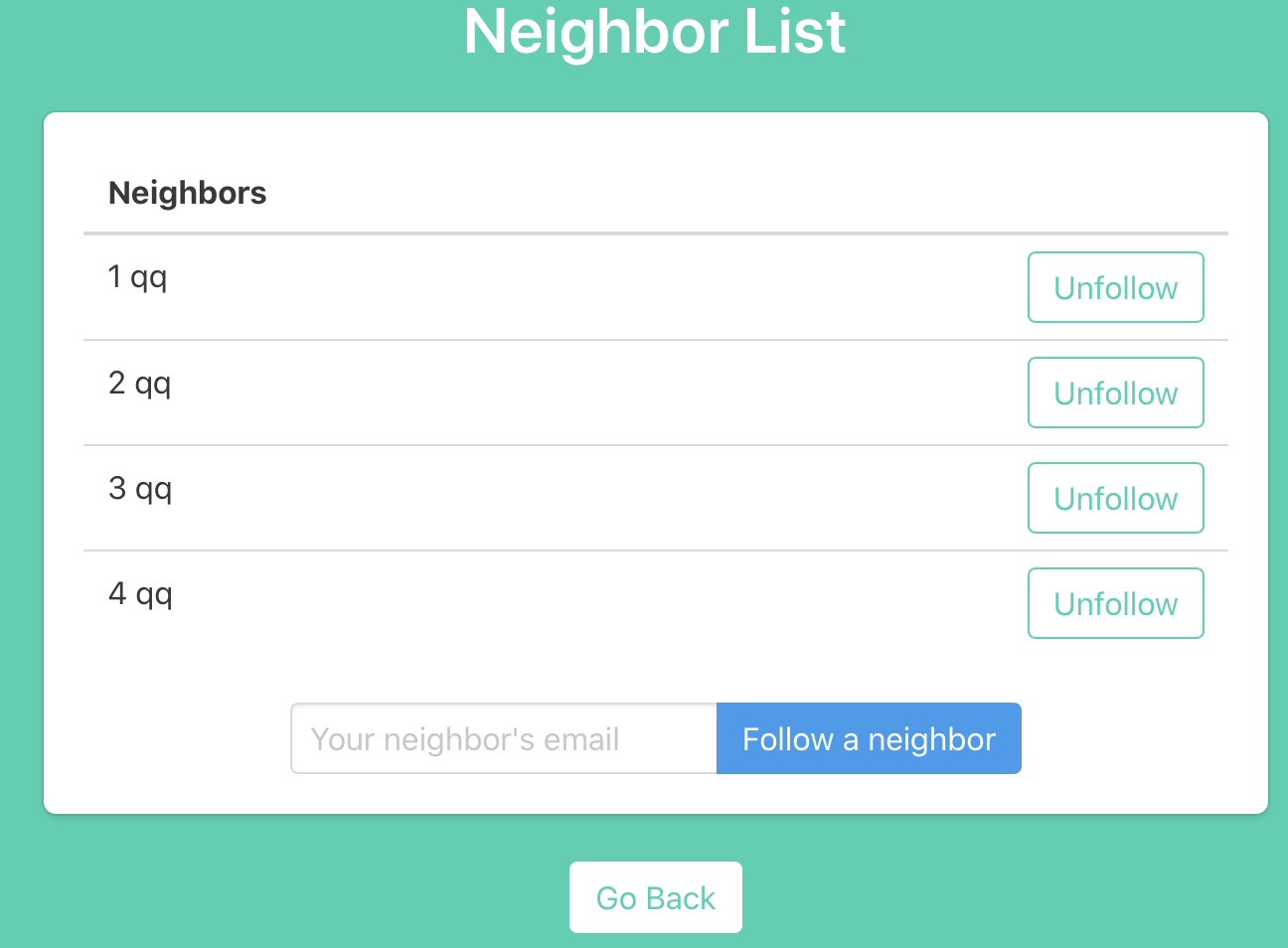
* Friend List



Click Follow -> Friend to see a friend list. Pending friendship request is listed on the top in the friend list page. The user can also add a new friend via email. Here we assume that if one user wants to send a friendship request to others, they have to know each other and their email addresses. A friendship request will be sent to the followee. Also, adding an unregistered user is not allowed.

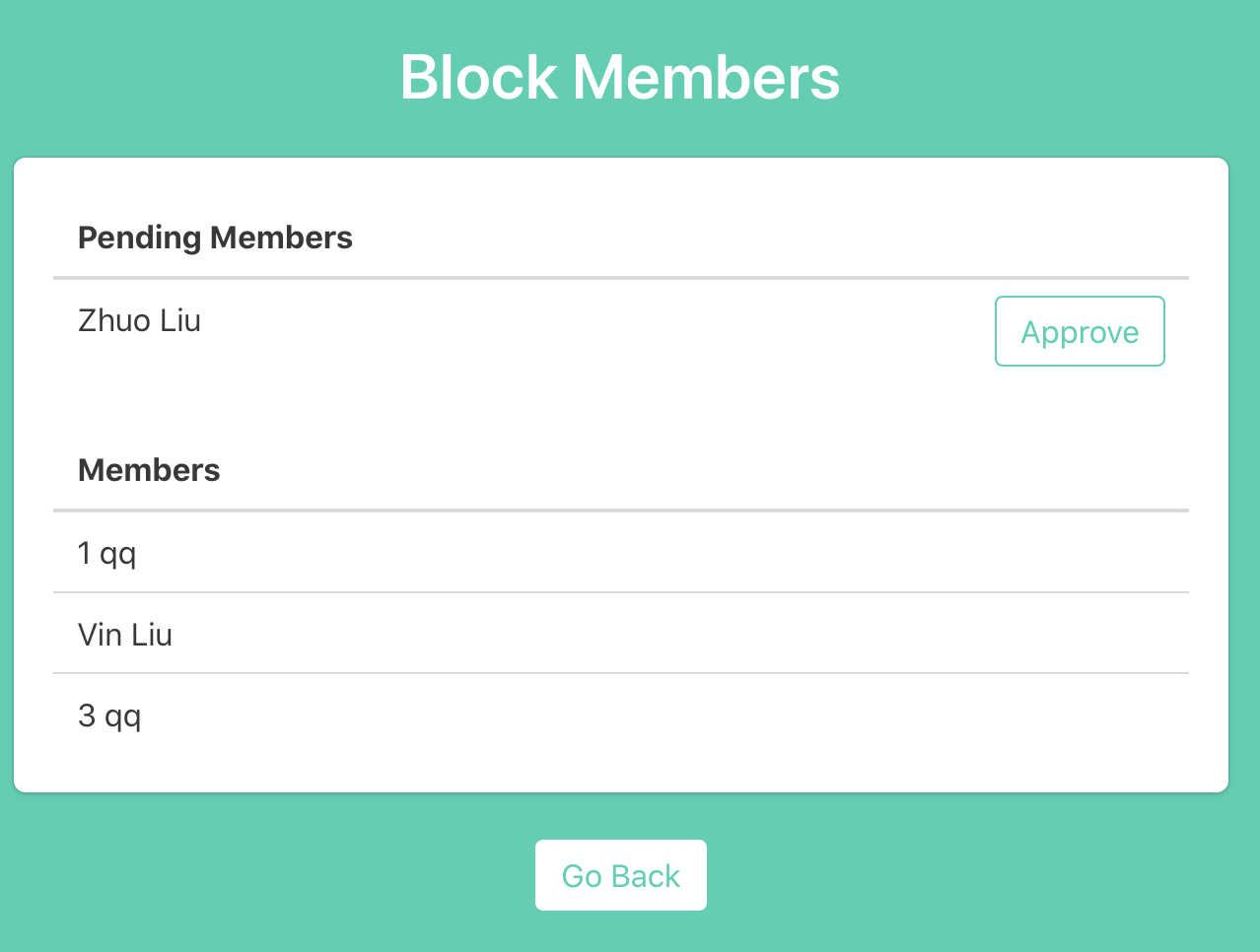


* Neighbor List



Click Follow -> Neighbor to view a neighbor list. Functions are similar to Friend List.

* View or Approve Block Members



Click Follow -> Block Member to view a list of members in the user’s block. Pending membership is listed at the top part. The user can approve the pending membership.

**3. Addressing Security Concerns**

* The SQL injection is prohibited by using flask-sqlalchemy module to handle all queries. We are not writing raw SQL statements to communicate with the database. Any special characters are automatically escaped and quoted by the SQL engine object. Also, the SQL statement is also automatically prepared by SQLalchemy.
* XSS is handled by flask-login module. (<https://flask.palletsprojects.com/en/1.0.x/security/>)
* All passwords are hashed before storing them into the database.
* The application is divided into Unauth & Auth parts. Any action in the Auth part can only be performed with a valid user session.