Tuiter Restful Web Service API Designs

Grant Chau

CS 5500: FSWE

The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server. The PUT method replaces all current representations of the target resource with the request payload. The DELETE method deletes the specified resource. The GET method is different in that it is only used to retrieve information and should not have any side-effects on the server. POST is meant for server-side data altering while PUT is used for data modification. If all of a specified resource is to be removed, we use DELETE. From these definitions, the Analysis of APIs below becomes simple:

1 Follows Restful Web Service APIs

1.1 A User follows another User

1. Analysis: Assume that a User A follows another User B. The followed User B will be added to the following User A's Following list, and A will be added to B's follower list. Data is submitted, and server-changes are required.
2. HTTP method: POST
3. Path Pattern: /api/users/:uid/follows/followingid
4. Request: Follower User's PK, Following User's PK
5. Response: New Follows JSON

1.2 User views a list of other Users that they are currently Following

1. Analysis: User A sees a list of Users that they are following without altering any data.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/follows
4. Request: User's PK
5. Response: User JSON Array

1.3 User views a list of other Users that are currently Following them

1. Analysis: User A sees a list of Users that are currently Following them without altering any data.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/follows/fans
4. Request: User's PK
5. Response: User JSON Array

1.4 User unfollows another User

1. Analysis: Assume that a User A unfollows another User B. The unfollowed User B will be removed from A's following list and the unfollowing User A will be removed from B's follower list.
2. HTTP method: DELETE
3. Path Pattern: /api/users/:uid/follows/followingid
4. Request: (un)Follower User's PK, (un)Following User's PK
5. Response: Delete status

1.5 User unfollows all Users in their Follow list

1. Analysis: Assume that a User A desires to unfollow everyone that they are currently following. They can clear their Following list rather than manually unFollowing all of them. This deletes their list.
2. HTTP method: DELETE
3. Path Pattern: /api/users/:uid/follows
4. Request: User's PK
5. Response: Delete status

1.6 User removes a specific Follower from their following list

1. Analysis: Assume User A suddenly changes their opinion on User B. User A removes User B from their follower list so that User B is no longer following User A.
2. HTTP method: DELETE
3. Path Pattern: /api/users/:uid/follows/fans/:followingid

4. Request: Follower User's PK, Following User's PK

5. Response: Delete Status

2 Bookmarks Restful Web Service APIs

2.1 A User Bookmarks a Tuit

1. Analysis: User A Bookmarks a Tuit. In the future, the User can view the Tuits that they have Bookmarked in a separate location from the feed of all Tuits on Tuiter. So, this is a modification rather than a creation action. Thus, we use the PUT method.

2. HTTP method: PUT

3. Path Pattern: /api/users/:uid/bookmarks/:tid

4. Request: User's PK, Tuit's PK

5. Response: New Bookmark JSON

2.2 A User unBookmarks a Tuit

1. Analysis: User A has previously Bookmarked Tuit P but does not wish to see Tuit P in thier list of Bookmarked Tuits. User A may unbookmark Tuit P, which removes the Tuit from the User's list of Bookmarked Tuits. Now, when the User is going through their list of Bookmarked Tuits, they will now longer be able to see Tuit P within these Tuits. Since this is a modification and not a deletion (modifying an already existing array), this is a PUT operation.

2. HTTP method: PUT

3. Path Pattern: /api/users/:uid/bookmarks/remove/:tid

4. Request: User's PK, Tuit's PK

5. Response: Update Status

2.3 A User clears their Bookmarked Tuits

1. Analysis: User A has some number (could be none) of Tuits Bookmarked and decides they are not interested in seeing any of them in their Bookmarked Tuits anymore. They may empty their bookmarks in one place. This is unlike the cases described in 2.1 and 2.2, where an existing array of Tuits is modified to be empty. Instead, the array of Tuits is deleted entirely.

2. HTTP method: DELETE

3. Path Pattern: /api/users/:uid/bookmarks

4. Request: User's PK

5. Response: Tuit JSON Array (should be empty)

2.4 A User views a list of Tuits they have Bookmarked

1. Analysis: User A has some number (could be none) of Tuits Bookmarked and wishes to see all of them in one place. They can use this to view the Tuits in an array of Tuits without manipulating the array.

2. HTTP method: GET

3. Path Pattern: /api/users/:uid/bookmarks

4. Request: User's PK

5. Response: Tuit JSON Array

2.5 A User views a single Bookmark

1. Analysis: User A desires to view a specific Tuit that they have Bookmarked. If they know the Tuit's ID but not the contents, they can view the Tuit's contents in their Bookmark list after identifying them by their PK without manipulating any data on the server.

2. HTTP method: GET

3. Path Pattern: /api/users/:uid/bookmarks/:tid

4. Request: User's PK, Tuit's PK

5. Response: Tuit JSON

3 Messages Restful Web Service APIs

3.1 A User sends a Message to another User

1. Analysis: User A uses the direct Message function to send a private Message to User B. This message can only be seen by them two but must be instantiated by one of them and must first cause a change in the server.

2. HTTP method: POST

3. Path Pattern: /api/users/:uid/messages/:receiveid

4. Request: User's PK, Message in body, Recieve's PK

5. Response: New Message JSON

3.2 A User unsends a Message

1. Analysis: User A uses the direct Message function to send a private Message called Message M to User B but decides to unsend it. User A may delete Message M, and this Message cannot be seen from that point on for either User A or User B. The Message M is deleted.
2. HTTP method: DELETE
3. Path Pattern: /api/users/:uid/messages/:msgid
4. Request: User's PK, Message's PK
5. Response: Update Status

3.3 A User views a previously received Message

1. Analysis: User A received some Message M. User A can view a specific message from the list of messages that User A has received or sent if they search for M's PK in their list of Messages. This requires no changes on the server.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/messages/:msgid
4. Request: User's PK, Message's PK
5. Response: Message as JSON

3.4 A User views all Messages they previously sent in any conversation

1. Analysis: User A can view a list of Messages that they have sent to others without manipulating any data on the server.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/messages/sent
4. Request: User's PK
5. Response: Messages JSON Array

3.5 A User views all Messages they previously sent in a specific conversation

1. Analysis: User A can view a list of Messages [M_1, M_2, ..., M_n] that they have sent to User B without manipulating any data on the server.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/messages/sent/:receiveid
4. Request: User's PK, Receive's PK
5. Response: Messages JSON Array

3.6 A User views all Messages previously sent to them

1. Analysis: User A can view a list of Messages that they have received from others without manipulating any data on the server.
2. HTTP method: GET
3. Path Pattern: /api/users/:uid/messages
4. Request: User's PK
5. Response: Messages JSON Array

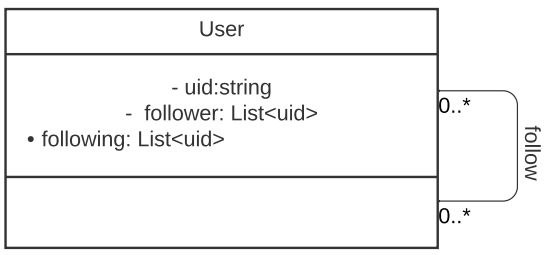
Tuiter Design Alternatives; Grant Chau

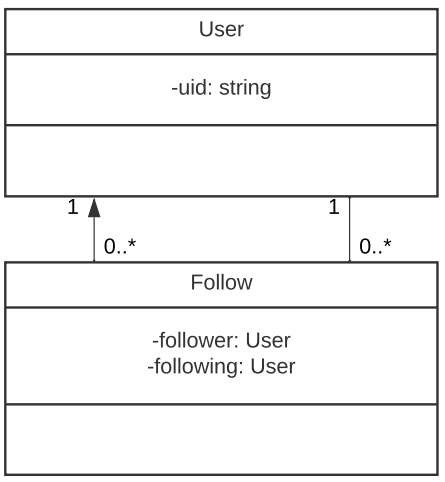
Follow function alternate designs (Option 1 above, Option 2 below):

Option 1:

Use a dual-set struture where every User posesses a list of References to Users who follow them while also maintaining a list of references to Users who are followed by him.

Pros and Cons:
While this solution is feasible in an OOP design paradigm, it is a difficult task to maintain membership consistency in both relationships (follows and following). It is also not as extensible as its alternatives and requires refactoring if changes are to be made to the Follow relationship. It also cannot be implemented in relational databases, since it requires multiple values in its relational column.





Option 2:

Use a single-set based design where a lookup table is created and it stores information regarding the Follow relationship between Users.

Pros and Cons:
This design is extremely versatile, as it allows for great extensibility by keeping information regarding User follower/following data in one place. That is, it can be implemented in both relational and non-relational databases and is viable in OOP design. Though it might find difficulty in non-relational databases, maintaining a single dataset is much more doable than maintaining two separate arrays that must be synchronous with data, as found in Option 1. Furthermore, updating in this design is done much smoother and in one place rather than two.

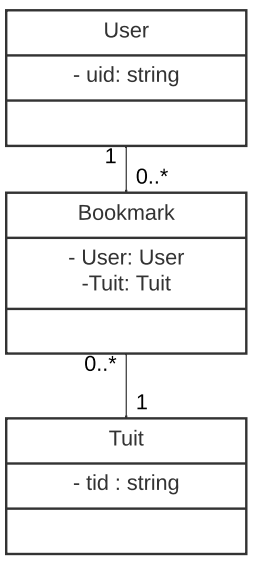
Follow: The implementation I will choose is the second one, Option 2. Since it allows for more efficient management of the Follow relationship between Users that is easier to understand and work with. I also considered the test cases when making this decision. Option 1 would be much more efficient when Users check their following and follower lists quite often, and Option 2 is much less efficient, as it requires many more queries in this regard. However, it seems that a User doesn't check their following list too often (at least in the test cases), instead focusing more on the addition and removal of other Users in their following/ follower list while only checking the contents of these respective lists every few operations. This is where Option 2's design really shines: maintaining synchronous list contents and using less transactions to make more updates to the database. Thus, Option 2 proves to be more useful in the context of Tuiter.

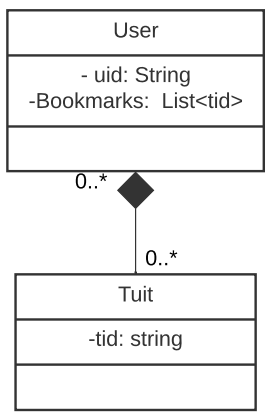
Bookmark function alternate designs (Option 1 left, Option 2 right):

Option 1:

Similar to Option 2 in the Follow design, we can create a lookup table that stores data regarding the relationship between a User and a Tuit with respect to the Bookmark functionality. Of course, this means that we must first create a Bookmark class then use this class to relate Users and Tuits.

Pros and Cons:
This option is the most versatile, and it is viable in both relational and non -relational databases. Though it is doable following the OOP design, it complicates cases ii nvolving unions in a non-relational database. Overall though, it is highly favorable.





Option 2:

Each User will posess a list which represents the Tuits that are bookmarked by the User. This means that there is no need for a Bookmark class; Bookmarked Tuits are contained in Users' list of Bookmarks.

Pros and Cons:
Though this option is highly unvialbe in a relational database, we are using OOP, so this design is still usable. It is also much simpler to understand than Option 1's design, though it is less extensible. However, since Bookmarks do not have any other relations with any other objects other than Users and Tuits, extensibility may not be an issue. The only issue is that this design is ambiguous regarding the nature of how exactly the Tuits are stored in the User's list of Bookmarked Tuits.

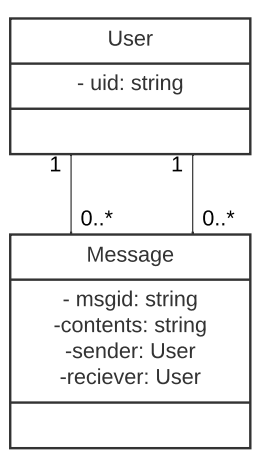
Bookmark: Option 1's strengths include the more efficient addition and deletion of Bookmarks, but it will prove to be relatively slow when many Tuits must be retrieved from a large set of Tuits. This is because many more queries within the design (not between the client and the system) are required to retrieve this information. Option 2 is faster in all regards, but there is the issue of how to add and delete Tuits to a User's set of Bookmarks. Option 1 is also undoubedtly more versatile, but after seeing the test cases, Option 2 seems like the better bet. There isn't too much functionality to add to the Bookmark functionality in the future (the only thing I can think of is sorting by tag, topic, location, date, etc.), so a simple list should be sufficient to maintain this relationship between Users and Tuits (and the given test cases seem to support this).

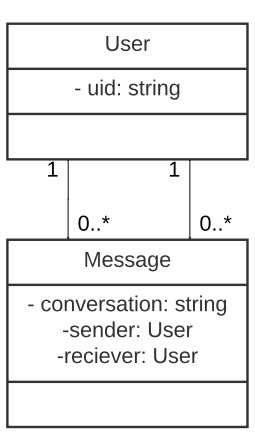
Message function alternate designs (Option 1 left, Option 2 right):

Option 1:

Similar to Option 2 in the Follow design, we can create a Message dictionary that stores information regarding sent and recieved messages between Users. Each time a Message is sent, a new Message object is created and stored in this manner.

Pros and Cons:
This option can be implemented readily in both relational and non-relational databases, and it is very viable in OOP deisgn practices. However, it may prove to be troublesome when used in a non-relational database due to the sheer number generated by User interaction and the constant synchronization requirements when new Messages are created. However, it is very easy to delete a message and to view a specific message by its id.





Option 2:

We can follow a similar design pattern except, instead of maintaining a lookup table that stores information regarding sent and recieved messages between Users and storing them as individual message objects in a database, we represent only the messages that a User sends to another User by maintaining a list of sent messages from a User in a single object.

Pros and Cons:
This option is easier to implement than Option 1, since only a single messaging conversation per Message object needs to be stored. Though, this may cause difficulties when adding them to a relational database due to the multi-valued nature of the conversation. While searching for a specific message is easy (just look through the conversation), finding more details about this message might be difficult.

Message: Though Option 2 has a clear strength in implementation, it is not very extensible, as it trusts that messages are going to always be Strings. Furthermore, the homework requirements lead me to believe that making and deleting Messages is more important than viewing the list of messages sent and recieved by Users. This means that Option 1 is a better choice on these ends, since creating and deleting specific messages is more simple with this design. Option 2 is honestly a close second though, since it is an excellent choice for simply viewing messages between two Users which is more common than sending and recieving them. However, since message creation/deletion is more difficult to synchronize, the implementation I select here should favor this process. It is also more common on the test cases provided, so I'll go with Option 1.