**DESIGN PHASE**

**Follows**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** | **Method /resource/path** | **Request** | **Response Body** |
| User follows another user | **POST**  /users/:uid/follows/:other-uid | User’s PK and PK of the user they want to follow | New Follows JSON |
| User unfollows another user | **DELETE**  /users/:uid/follows/:other-uid | User’s PK and PK of the user they want to unfollow | Delete Status |
| User views a list of other users they are following | **GET**  /users/:uid/follower | User’s PK | JSON array filled only with follows where the user is the value for **userFollowing** |
| User views a list of other users that are following them | **GET**  /users/:uid/following | User’s PK | JSON array filled only with follows where the user is the value for **userFollowed** |
| Delete all follows associated with a given user | **DELETE**  /users/:uid/follows | User’s PK | JSON array filled with Delete Statuses for all of the follows that were deleted |
| User ranks a follow relationship (e.g. Myspace Top Friends) | **PUT**  /follows/:fid | Follows PK | Update Status |

Diagram

Description automatically generatedThe table above was created based on the class diagram to the right. The class diagram describes the relationship between a Follow and a User as a **many to many** relationship. A user can **follow** an infinite amount of people and a user can in turn be **followed** by an infinite amount of people. Given this, let’s walk through the RESTful web services we have devised.

Starting with the first use case, given that we are saying that this relationship is **many to many**, in order for a follow to be created, we need to make a **POST** request and we need the primary key for each user. When the record is created, we now have a direct reference to each user which is a good thing! If a change ever occurs to the Follow, we will already have the primary for each user so we can quickly access and update both users accordingly. Now once the Follow is created a JSON of that new Follow record should be returned so that the information within that new record can be used for other tasks as necessary.

Some of the same rationale above applies to the next use case which is the deletion of a follow relationship. If a user decides that they no longer want to follow someone, we will be making a **DELETE** request, but we will still need the primary key for both users. In this case we use the primary key of the user doing the unfollowing to filter through all follows records and then we use the primary key of the user being unfollowed to find the exact follow relationship within the filtered data to delete. Once the deletion occurs, we should return a status update as to whether the unfollow was successful or not.

The next two use cases, viewing all users you follow and viewing all users who follow you, are **GET** requests and they both rely on the primary key for a given user. The user’s primary key will allow us to filter through all of the follows data and find only follows records that the user is associated with. Now after this initial filter is when each of the use cases differs. If a user wants to see who they follow, then, as the class diagram denotes, we need to find the follows records where the user is listed as **userFollowing**. On the other hand, if a user wants to see who is following them then we need to find the follows records where the user is listed as **userFollowed**. In both cases the response from the server should be a JSON array filled with the appropriate Follows records.

Our next use case would never be used by a user but would be very helpful for an administrator. In the event that a user account had to be deleted, it would be extremely helpful to remove every Follows record that the user was associated with. To do this we would need to make a **DELETE** request and provide the primary key for a given user. When the request is made the primary key will allow the data to be filtered appropriately. Once all the records were found we could thus delete all of them and return an array of the status of each deletion so that the administrator could verify that allow Follows records were appropriately deleted.

Finally, the last use case is a callback to the days of Myspace. On that social media site, a user could, for better or worse, rank their friends. Thus, if we wanted to implement that feature in our system and allow a user to rank the people that follow them, we would need to make a **PUT** request to the database, and we would only need to know the primary key for the appropriate Follows record. Since we already have a method for finding all the people who follow a given user, we would already have pre-filtered data and would thus just need to know what record we were altering. Once changes to the Follows record occurred, the system should return a status update so that the user knew their change was successful.

**Bookmarks**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** | **Method /resource/path** | **Request** | **Response Body** |
| User bookmarks a Tuit | **POST**  /users/:uid/bookmarks/:tid | Users PK and Tuits PK | New Bookmarks JSON |
| User unbookmarks a Tuit | **DELETE**  /users/:uid/bookmarks/:tid | Users PK and Tuits PK | Delete Status |
| User views a list of Tuits they have bookmarked | **GET**  /users/:uid/bookmarks | Users PK | JSON array containing all Bookmarks that the user has created |
| Get all bookmarks | **GET**  /bookmarks | No information needed | JSON array containing all Bookmarks ever created |
| Update a specific bookmark | **PUT**  /bookmarks/:bid | Bookmarks PK | Update Status |

Following the analysis for the Follows RESTful API, we move right along to the Bookmarks RESTful API. The table above was created using the class diagram shown below. The class diagram shows that Bookmarks are associated with both Users and Tuits. When a bookmark is created it references both the Tuit that was bookmarked as well as the User that bookmarked said Tuit. Additionally, the cardinality shows that a user can have many bookmarks and a Tuit can be bookmarked by many users. Given these ideas, let’s walk through the use cases.

In the first use case, we are covering how a bookmark is created. First off since we are creating a new record, we know this is a **POST** request. Now, following the class diagram we know that in order to make a Bookmark record we will need a reference to the User and the Tuit. This is accomplished by including the primary key for each resource in the request. Once the request is made, we should, as a result, receive a JSON of the new record so that the information within that new record can be used for other tasks as necessary.

Diagram

Description automatically generatedThe next use case is deleting a bookmark and since we are removing data, this is a **DELETE** request. Similar to the creation of a bookmark, we are interacting with both the User and Tuit resources so in order to complete our request we need to provide the respective primary keys for each resource. When the deletion request is made having access to the primary keys for each resource allows us to update everything accordingly. As a response to the request, we should simply receive a status update regarding the success of the deletion.

To accomplish the use case of a user viewing all their bookmarks, we will need to perform a **GET** request. Within that request we will also have to include the primary key for the user in question. The database will have a record of all the bookmarks ever created and in order to get the bookmarks pertaining that user we will need to use their primary key to filter accordingly. As a response this request, we should expect to receive a JSON of Bookmarks associated with our user.

For this next use case, we would only ever expect an administrator to use it. To get all of the bookmarks ever created we would need to make a **GET** request, but we would not need to include anything else in our request. This request would ping the database and as a response we should expect to receive a JSON array containing every bookmark record that the database had on file. This request would be very useful for things like data analysis or migration.

Finally, the last use case would again only really be useful for an administrator. In the event that a specific bookmark record needed to be updated we would need to make a **PUT** request and since we are targeting a specific record, we would need to know its primary key. Unlike earlier use cases, we do not need to include the User primary key or Tuit primary key seeing as how we are only updating the record as opposed to trying to delete it or create a new one. The use case for something like this would be maybe incorporating new “quality of life” information into the bookmark in which case again we need only use the primary key of the bookmark itself. As a direct result of this request, we should expect to receive a status update from the system regarding the changes.

**Messages**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** | **Method /resource/path** | **Request** | **Response Body** |
| User sends a message to another user | **POST** users/:uid/messages/:other-uid | Users PK and Primary key of another user | New Message JSON |
| User views a list of messages they have sent | **GET** users/:uid/messages/sent | Users PK | JSON array filled only with Messages where the user is the value for **from** |
| User views a list of messages sent to them | **GET** users/:uid/messages/received | Users PK | JSON array filled only with Messages where the user is the value for **to** |
| User deletes a message | **DELETE** /messages/:mid | Messages PK | Delete Status |
| User edits a message | **PUT**  /messages/:mid | Message PK | Update Status |
| Get all messages | **GET**  /messages | No information needed | JSON array containing all Messages ever created |

For our final resource, you will again find that the table above was created based on the class diagram to the right. The relationship between a Message and a User is described as a **many to many**. A User can both receive and send an infinite number of Messages.

Diagram

Description automatically generatedFor the first use case, we are interested in a user sending a message to another user. Since the message does not yet exist, we will need to perform a **POST** request. Additionally, within that request we will need to include both the primary key for the user sending the message and the primary key of the user they are sending a message to. Assuming the request is successful, the response should be a JSON of the new Message. As with all the other POST requests, having a copy of the created record will help with any housekeeping that the application will need to do.

Now the next two use cases, viewing all Messages a user has sent and viewing all Messages a user has received, will both require a **GET** request. These use cases both involve simply fetching information, but to fetch the correct information we will need to include the primary key for the user in question. The primary key let’s us filter through all Messages data and find only the Messages associated with our user. Once those are found, this is where our use cases require different final steps. For the case where a user wants to see Messages they have sent, the system’s final filter should be to look for only the Messages where the **from** field is pointing to our user in question. On the other hand, to find the Messages that the use has received, the system’s final filter should be to look for only Messages where the **to** field is pointing to our user. Finally in both cases, our expected response should be a JSON array filled with only the appropriately filtered Messages records.

The use case of deleting a Message requires a **DELETE** request. Now there are a few ways to go about deleting a message. For the path we could first filter on the given user then filter on what message they want to delete (sent or received), but that seems a little convoluted. Chances are that if a user is deleting a message, they are doing so from a screen where they already have the messages associated with them, be it sent and/or received. Given that idea, to delete a given message all we need to include is the primary key for the Message. With that in hand we can make our request and the expected response should be a status update on the success of the deletion.

The use case of editing a Message follows the same idea as our previous use case. If a user wishes to edit a Message, that will involve a **PUT** request, but as explained above, we only need to include the primary key for the Message they wish to edit. Once the request is fulfilled, we should expect to receive a status update in the response.

Similar to a few of the other use cases we have covered, this final use case would only be used by an administrator. Retrieving all the messages ever created on the site would involve a **GET** request and because we want all of them, we don’t need to provide any additional information with our request. The given request would find all the Messages stored in the database and we should expect a JSON array containing all of the Messages records.

**Implementation PHASE**

**Follows**

**Bookmarks**

**Messages**