Evernote Synchronization via EDAM

v1.0.4

October 18, 2010

Revision History

Date	Ver	Author	Changes Summary
9/7/07	1.0	Dave Engberg	Principal author / editor Split from EN3 Arch Overview doc
9/8/07	1.0.1	Dave Engberg	Fix inconsistent example data in the interaction diagrams.
10/16/07	1.0.2	Dave Engberg	Fix language around tag deletion on full sync
9/21/09	1.0.3	Dave Engberg	Reformat for external consumption
10/22/10	1.0.4	Dave Engberg	Add details about synchronizing linked (shared/public) notebooks

Table of Contents

1. EDAM Overview

Evernote Data Access and Management (EDAM) is a protocol for exchanging Evernote data with the Evernote service. The data structures and remote procedures supported by EDAM are expressed using the Thrift interface definition language. These Thrift IDL files can be used to generate language-specific data and programming interfaces for communication. The generated classes are used to perform marshalling and unmarshalling of the data and procedure calls using an abstract notion of "protocols" and "transports".

The Thrift runtime libraries include specific protocol and transport implementations to allow Thrift RPC calls using a binary encoding over HTTP and/or raw TCP/IP sockets. These mechanisms can be relatively easily extended to (for example) marshal the data using an ASCII encoding based on XML, JSON, etc.

The EDAM Thrift IDL files include a definition of the following data types from the core Evernote data model:

- User
- UserAttribute
- Notebook
- Tag
- Note
- NoteAttribute
- Resource
- ResourceAttribute
- SavedSearch
- LinkedNotebook

The IDLs also express a set of remote procedures, broken down into two different logical "services":

UserStore
- authenticate
- refreshAuthentication

NoteStore
- createNote
- createNotebook
- createSearch
- createTag
- deleteNote
- deleteNotebook
- deleteSearch
- deleteTag

- getNote
- getNoteContent
- getNotebook
- getResource
- getResourceData
- getSearch
- getSyncChunk
- getSyncState
- getTag
- listNotebooks
- listNotes
- listSearches
- listTags
- updateNote
- updateNotebook
- updateSearch
- updateTag

The full documentation for these data structures and functions is available at: http://www.Evernote.com/about/developer/api/ref/

The interfaces defined in these Thrift IDLs are intended to be used to serve three different purposes:

1. Internal EN service components may query the UserStore or NoteStore service within the data center.
2. Full caching clients use the interface to perform data synchronization over HTTP via an external-facing "sync gateway" service. These clients use the standard binary Thrift encoding.
3. Thin clients make direct use of the API over the HTTP gateway to view or manipulate the state of the user's account directly. This may include the clipper(s), JavaScript web clients (via an ASCII transport), or external partners/integrations.

As a result of this multi-use property of the API, there are some functions that are only relevant for one or two of the above scenarios, while others would be used by all three. For example, the "getSyncChunk" function is specific to the caching clients performing synchronization, while "authenticate" or "getResourceData" may be used by all three.

This document is limited to the synchronization case where a client with a local cache of notes must synchronize against the Evernote service.

2. EDAM Synchronization Overview

The Evernote synchronization scheme is designed around a particular set of requirements:

1. Synchronization follows a “client-server” model where the central service is the ultimate arbiter of the state of the account.
2. Clients are built using a variety of local storage mechanisms, so the synchronization scheme must not assume a particular low-level storage representation: data must be transferred at a “logical” level rather than a “block/record” level.
3. Evernote must support both full and incremental synchronization. It’s not acceptable to send the entire DB with each sync.
4. Synchronization must be possible over unreliable networks without significant retransmission. Even for initial sync, clients must be able to resume transfer after a network error.
5. Synchronization cannot “lock” the Evernote service to make the process atomic/synchronous. The scheme must tolerate modifications from other clients during the synchronization process.

EDAM Synchronization addresses these requirements using a “state based replication” scheme that treats the central service as a simple data store and pushes most decisions into the clients. This is intended to be similar to the model used by mail systems such as IMAP or MS Exchange, which have been able to address a similar set of requirements in a robust and scalable manner.

In this scheme, the Evernote service does not keep track of the current state of individual clients, and it does not keep a fine-grained transaction log to implement “log-based replication”. Instead, it stores a set of data elements (notes, tags, etc.) for each user. Each data element has an “Update Sequence Number” (USN) that identifies the order in which it was last modified in the account. The system can use these numbers to determine whether one object in the account was modified more recently than another.

The USNs within an account start at “1” (for the first object created in the account) and then increase monotonically every time an object is created, modified, or deleted. The server keeps track of the “update count” for each account, which is identical to the highest USN that has been assigned.

At any point, the service is capable of ordering the objects in the account using their USN values. To synchronize, a client must only receive the objects that were changed after the last client sync … i.e. only objects that have a USN that is higher than the server’s “update count” at the last successful sync.

This goal is somewhat complicated by requirements #3-5, above. In order to meet these requirements, the protocol must allow the client to request small "blocks" of objects without locking the service during the sync. The protocol must handle the situation where the state of the service changes in the middle of a sequence of blocks being sent to the client … either due to the size/speed of transmission, or due to a network interruption.

As mentioned above, the synchronization scheme pushes all of the record keeping and conflict resolution work onto the client so that the service can perform synchronization in a scalable "stateless" manner. This means that the client needs to keep track of the state of the server during each sync, and then use this information to send and receive updates on the next sync. At a high level, the client performs the following steps:

- Get the list of new/modified objects from the service (since the last sync)
- Reconcile the server's changes with the local database
- Send the client's unsynchronized updates to the service
- Record the server's state for the next sync

In order to distinguish data that is in sync with the service from data that has been created/modified by the client since the last sync, the client must maintain an internal "dirty" flag on every modified object in the local store. This constitutes the list of objects that must be pushed to the service (after conflict resolution).

The client should make it possible for a user to perform a full synchronization even if the client is theoretically capable of an incremental sync.

3. Synchronization pseudo-code

The following pseudo-code expresses the steps that would be performed by a client in order to synchronize with the service.

Service variables:

- `updateCount` – the highest USN given out within the account
- `fullSyncBefore` – the cut-off date for old caching clients to perform an incremental (vs. full) synchronization. This value may correspond to the point where historic data (e.g. regarding expunged objects) was wiped from the account, or possibly the time of a serious server issue that would invalidate client USNs.

Client variables:

- `lastUpdateCount` – the server's `updateCount` at the last sync
- `lastSyncTime` – the time of the last sync (as given by the service)

Authentication:

1. Authenticate to the service using `UserStore.authenticate(username,pwd,key,secret)` over HTTPS.
    a. Receive opaque `authenticationToken` string to use for all other operations.
    b. Record expiration time of the `authenticationToken`. If token is near expiration before any request to the server, get a new token (with a later expiration) via `UserStore.refreshAuthentication(…)` (over HTTPS only).

Sync Status:

2. If the client has never synched with the service before, continue to **Full Sync.**
3. `NoteStore.getSyncState(…)` to get the server's `updateCount` and `fullSyncBefore` values.
    a. If (`fullSyncBefore > lastSyncTime`), continue to **Full Sync**.
    b. If (`updateCount = lastUpdateCount`) , the server has no updates. Skip to **Send Changes**.
    c. Otherwise, perform an incremental update sync (go to **Incremental Sync**)

Full Sync:

4. `NoteStore.getSyncChunk(…, afterUSN=0, maxEntries)` to get the first block of data objects from the service. The server will return the meta-data for at most `maxEntries` objects (of any type), starting from the least-recently-modified object in the account. This includes the full state of "small" objects like Tags and SavedSearches, but only includes the metadata for Notes and Resources. The length and MD5 hash of these objects' large data fields (note contents, binary resources, etc.) must be requested separately, later. Expunged (deleted permanently) objects are included by reference (GUID) only.

    a. If chunk's `chunkHighUSN` is less than chunk's `updateCount`, buffer the chunk and request the next chunk by repeating Step #4 with `afterUSN = chunkHighUSN`. (This can be done safely in spite of a time gap between chunks).
5. Process the list of buffered chunks in order to construct the current state of the service:
    a. Build the list of server tags (identified by GUID) contained within the sync blocks. Search the blocks, in order, for tags to add to this list while removing tags from the list if their GUID is "expunged".
        i. If a tag exists in the server's list, but not in the client, add to the client DB. If a tag with the same name (but different GUID) already exists in the account:
            1. If the existing tag has the "dirty" flag, the user has created a tag in the service and in the client with the same name while offline. Perform a field-by-field merge or report the conflict for resolution.
            2. Otherwise, rename the existing client tag (e.g. appending "(2)")
        ii. If a tag exists on the client, but not on the service:
            1. If the client's tag has no "dirty" flag, or if it has previously been uploaded to the server, then delete the tag from the client.
            2. Otherwise the tag is new on the client, it will be uploaded later
        iii. If a tag exists in both the client and the server:
            1. If they have the same USN and no "dirty" flag, then they're in sync.
            2. If they have the same USN, but the client's has a "dirty" flag, then it will be uploaded to the server later.
            3. If the server's tag has a higher USN and the client has no "dirty" flag, then update the state of the client's tag with the server's object. (Resolve name conflicts as above)
            4. If the server's tag has a higher USN and the client has a "dirty" flag, then the object has been modified on both the server and the client. Perform a field-by-field merge if possible, or report the conflict for resolution.
    b. Perform the same algorithm for Saved Searches in the account.
    c. Perform the same algorithm for the list of Notebooks in the account. If a notebook is deleted from the client, all of its Notes and Resources are also deleted.
    d. Perform the same algorithm for LinkedNotebooks in the account. These are just pointers to notebooks in other user's accounts, so there are no direct relationships to any of the other data elements in this account.
    e. Perform the same algorithm for Notes in the account. The content of Notes is not transmitted as part of the sync block, so it must be requested separately via `NoteStore.getNoteContent(…)` for new notes, or if the note content has been modified (as determined by the MD5 checksum and length included in the Note metadata). The same is true for any embedded Resources data block and recognition text.
6. On completion of the server data merge, the client stores the server's `updateCount` to `lastUpdateCount` and the server's current time to `lastSyncTime`.
7. Go to **Send Changes.**

Incremental Sync

8. Perform Step #4 to build a list of sync blocks, but starting with `afterUSN=lastUpdateCount`.
9. Process the list of buffered chunks in order to add/update objects from the server to the client:
    a. Build the list of server Tags (identified by GUID) contained within the sync blocks. Search the blocks, in order, for tags to add to this list while removing tags from the list if their GUID is "expunged".
        i. If a tag exists in the server's list, but not in the client, add to the client DB. If a tag with the same name (but different GUID) already exists in the account:
            1. If the existing tag has the "dirty" flag, the user has created a tag in the service and in the client with the same name while offline. Perform a field-by-field merge or report the conflict for resolution.

2. Otherwise, rename the existing client tag (e.g. appending "(2)")

ii. If a tag exists in both the client and the server:

1. If the client's object has no "dirty" flag, then update the state of the client's tag with the server's object. (Resolve name conflicts as above)
2. If the client's object has a "dirty" flag, then the object has been modified on both the server and the client. Perform a field-by-field merge if possible, or report the conflict for resolution.

b. Perform the same algorithm for Saved Searches in the account.
c. Perform the same algorithm for the list of Notebooks in the account.
d. Perform the same algorithm for the list of LinkedNotebooks in the account.
e. Perform the same algorithm for Notes in the account. The content of Notes is not transmitted as part of the sync block, so it must be requested separately via `NoteStore.getNoteContent(…)` for new notes, or if the note content has been modified (as determined by the MD5 checksum and length included in the Note metadata). The same is true for any embedded Resources data block and recognition text.

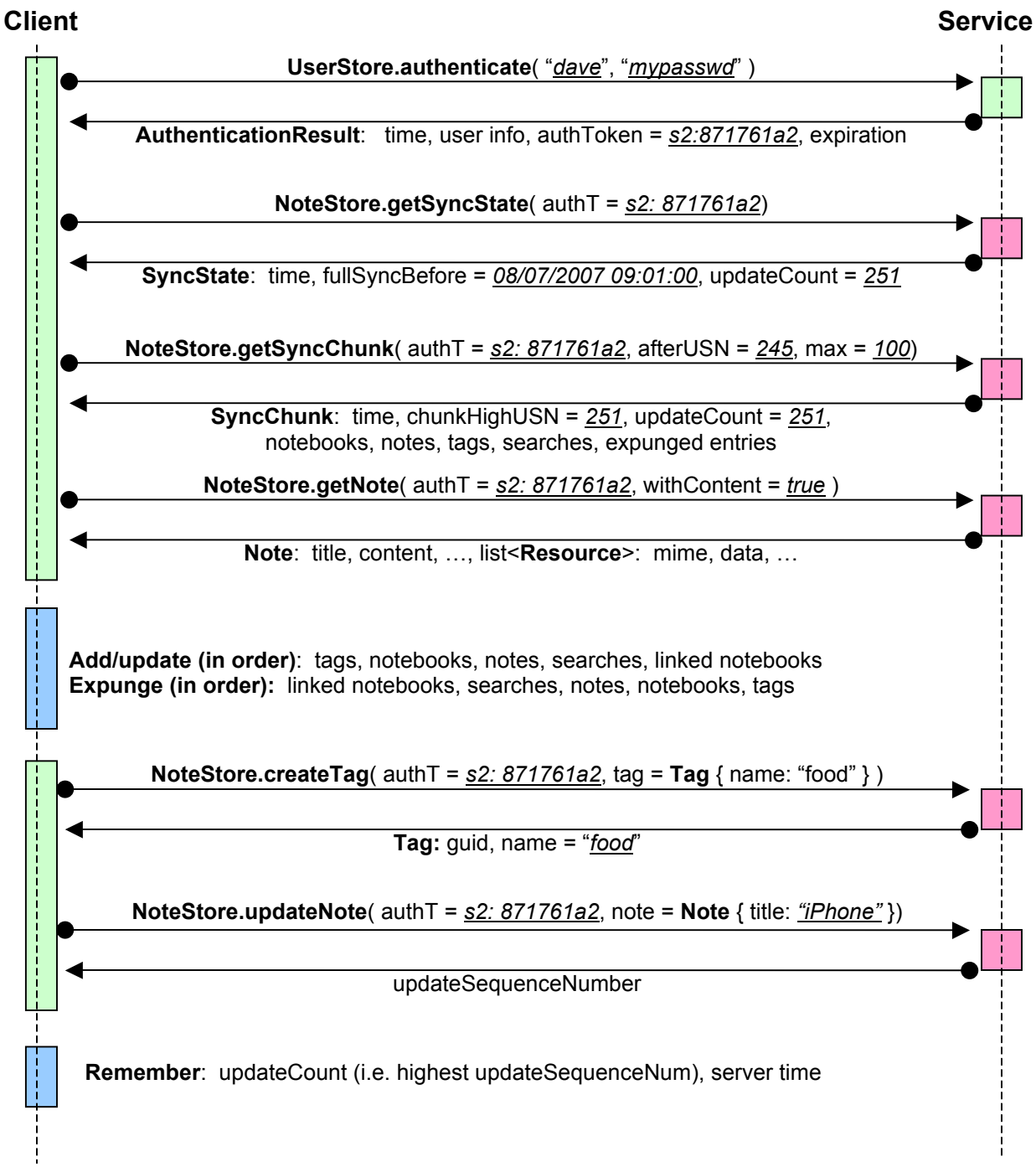
10. Process the list of buffered chunks in order to perform deletions from the server to the client:
    a. Assemble the set of expunged Note GUIDs from the sync blocks. For each GUID in the list, remove the corresponding Note from the client, if present.
    b. Same for Notebooks. Expunging a notebook implies the removal of all of its Notes and Resources.
    c. Same for Saved Searches.
    d. Same for Tags.
    e. Same for LinkedNotebooks.
11. On completion of the server data merge, the client stores the server's `updateCount` to `lastUpdateCount` and the server's current time to `lastSyncTime`.
12. Continue to **Send Changes.**

Send Changes

13. For each tag in the local account with a "dirty" flag:
    a. If the tag is new (i.e. its local USN isn't set), send it to the service via `NoteStore.createTag(…)`. If the service reports a conflict (due to an update from another client after this client received the last block from the service – should be unlikely), the client must resolve the conflict locally. If the server replies with a GUID for the tag that is different than requested, then change the local tag's GUID to match.
    b. If the tag is modified (i.e. its local USN is set), send it to the service via `NoteStore.updateTag(…)`, with the same conflict resolution requirement.
    c. In either case, check the USN in the response:
        i. If the USN = `lastUpdateCount` + 1, then the client is still in sync with the service. Update `lastUpdateCount` to match the USN.
        ii. If the USN > `lastUpdateCount` + 1, then the client is not in sync with the service, so will want to do an **Incremental Sync** after sending changes.
14. Same algorithm for each Saved Search in the local account with a "dirty" flag.
15. Same algorithm for each Notebook in the local account with a "dirty" flag.
16. Same algorithm for each Note in the account with a "dirty" flag. The client is required to transmit the full data (note contents, resource data, recognition data) for each note sent using `NoteStore.createNote(…)`, and is required to transmit such data if modified as part of any call to `NoteStore.updateNote(…)`. I.e. the note must be uploaded as one message … its parts may not be transmitted later.

4. Initial (Full) Sync Interaction Example

5. Incremental (Update) Sync Interaction Example



6. Linked Notebooks and Synchronization

The algorithm, above, describes the process necessary to maintain a consistent state between a rich client and a full Evernote account. If a client also wants to access the contents of one or more shared notebooks from other Evernote accounts, the process becomes a bit more complicated.

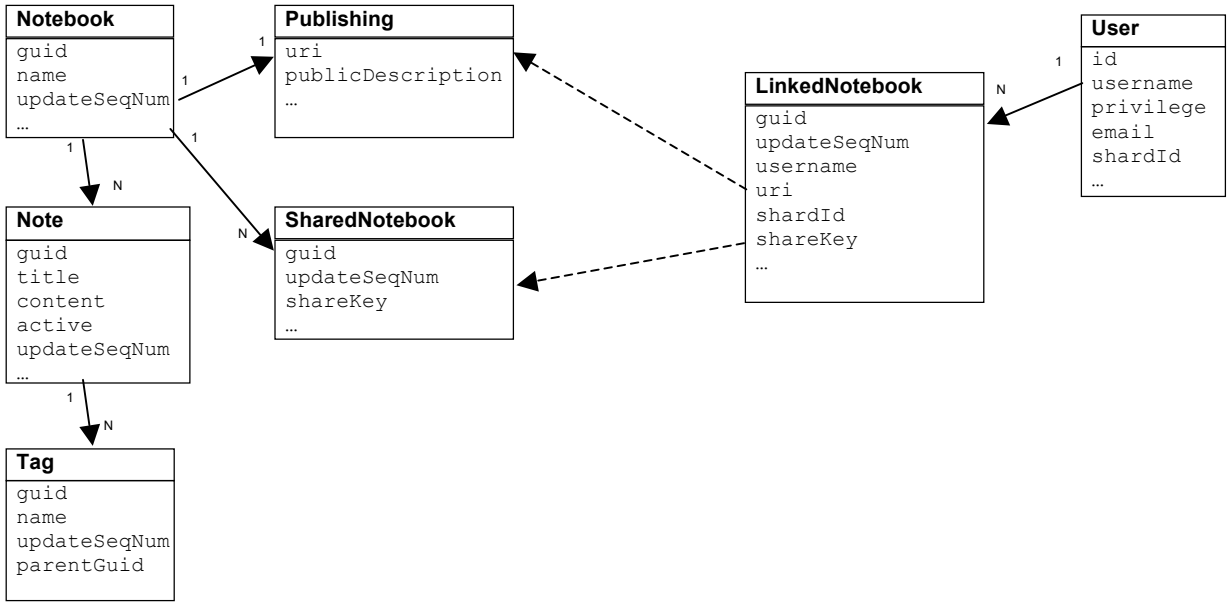
Imagine that Alice (a long-time user with an account on shard "s2") shares her Cooking notebook with Bob (a newer user with an account on shard "s45"). Bob receives the invitation to access this shared notebook and decides to "link" it into his own account so that he can access it again in the future.

In this scenario, Alice has a "Notebook" object in her account.

For each sharing "invitation" that she has sent to a friend, there is a "SharedNotebook" object in her account. I.e. there is a 1:N relationship between the actual Notebook that you see in her account, and the SharedNotebook objects. Since this is part of her account, the "SharedNotebook" data is located on her shard (s2). If she stops sharing that notebook with Bob, then one of the SharedNotebook objects would be removed, but her own "Notebook" object would persist, and any other SharedNotebook invitations that she'd sent to this Notebook would continue to work.

When Bob accepted that invitation and linked it into his own account, he created a "LinkedNotebook" object in his account on his shard (s45). Think of this as a "hyperlink" pointing to someone else's SharedNotebook, rather than a tightly bound reference. If Alice rescinds her invitation, her SharedNotebook will be removed, but Bob's LinkedNotebook may not be automatically removed.

In addition to these direct SharedNotebook invitations, Alice may also share her notebook "with the world" by publishing it at a public URI. These "public" notebooks don't require SharedNotebook entries in the owner's account, but they can also be linked into another user's account so that they may view the public notebook contents at a later date. As a result, a LinkedNotebook object in a user's account may either contain information about connecting to a private SharedNotebook, or may contain the URI fields to access a public Notebook. Accessing these two different types of LinkedNotebooks will be slightly different.



The diagram, above, shows the relationship between the owner's content from the owner's shard (on the left) and the LinkedNotebook in the recipient's account on the right. The LinkedNotebook may reference the owner's notebook via either a URI or a shareKey, extracted from either the Publishing or SharedNotebook information from the owner's account.

Synchronizing information about other people's notebooks that are linked into your account requires two high-level additions to the basic synchronization scheme.

First, the client must keep an up-to-date list of all of the LinkedNotebook objects that exist in the user's account. Bob's client must know when a LinkedNotebook entry has been added to his account, and then must know when it has disappeared. That process is described, above, in steps 5d, 9d, and 10e. The SyncChunk object will contain references for new and modified LinkedNotebooks, and will contain the GUID of any expunged LinkedNotebooks.

Second, the client must use the information in user's LinkedNotebooks to access the corresponding shared Notebooks from the accounts and shards of the people who own those notebooks. In the scenario, above, Bob's client must use the information in the LinkedNotebook to talk to Alice's shard (s2) to synchronize the current contents of that Notebook from her account. To do so, Bob's client will keep separate synchronization "state" information for each LinkedNotebook to maintain efficient incremental synchronization.

I.e. you must synchronize the list of LinkedNotebooks in *your* account to find the pointers to other people's notebooks. Then you use those pointers to synchronize the content of the shared notebooks within other people's accounts.

The synchronization process for a LinkedNotebook is similar to the process for synchronizing to your own account, with a few changes.

Authentication:

In addition to the authenticationToken to access the user's own data, they will need to receive separate authenticationTokens to access a SharedNotebook in another user's account. This is done by making a call to the NoteStore.authenticateToSharedNotebook() on the **owner's** shard. This call takes the shareKey from the SharedNotebook (i.e. the LinkedNotebook.shareKey value) along with the **recipient's** own authenticationToken from their own account (from step 1, above).

For example, Bob's client would make a call to this function on shard "s2" using his own authenticationToken in order to gain access to Alice's SharedNotebook on that system.

For LinkedNotebooks based on public notebook URIs, this step is not needed, since no authentication is required to access any data in a public notebook. NoteStore calls to access Notes or Resources in a public notebook can use any authentication token, or even an empty string.

Sync Status:

This should work similarly to steps 2 and 3, above, except that the getSyncState call should be replaced with a call to NoteStore.getLinkedNotebookSyncState(). This call should be made on the owner's shard. The caller should provide the LinkedNotebook information to identify the notebook.

This call will return a SyncState object that will show the state of the notebook owner's account's synchronization state.

Full and Incremental Sync:

Performing the synchronization against another user's notebook uses the NoteStore.getLinkedNotebookSyncChunk() call, which is identical to the getSyncChunk() call described in step 4, above, except that the LinkedNotebook structure must be supplied to the service.

This will return SyncChunks that will be filtered to only include:
1) Notes that are contained within the shared/published notebook (and are active, not in the Trash).
2) Only the Notebook object that has been published/shared.
3) Tags that are applied to at least one of the Notes in that Notebook.
4) Expunged GUIDs for Notes that are no longer in that Notebook. (I.e. they were either moved to a different notebook, moved to the Trash or expunged.)

The SyncChunk will not contain any other information from the owner's account, including SavedSearches, LinkedNotebooks, etc. I.e. the recipient is just seeing a slice of the total data in the user's account.

Otherwise, the synchronization algorithm (and order of operations) is similar to synchronizing to the user's own account. The SyncChunks for the account should be retrieved in order and then processed as described, above.

There are two ways that the SyncChunks for a LinkedNotebook will behave differently from the chunks of the user's own account:

The client will not be informed of Tags that have been expunged. Instead, any Tag from a shared notebook should be removed from the client's list of Tags if there are no Notes left that reference that Tag. I.e. in a SharedNotebook, only tags that are applied to at least one Note are considered valid. The client is not permitted to know the updated state of Tags that are not applied to at least one Note in a shared/public notebook, so they must assume that any unused Tags have been removed.

When a Note is moved from one notebook to another within a user's own account, they will receive a notification of the new state of that Note. However, when a Note is moved from a Linked notebook into a

different notebook (or into the Trash), any guests viewing that Notebook will be informed that the relevant Note has been “expunged” via the SyncChunk.expungedNotes list. Later, if that Note is ever restored to the Linked notebook, the client may see it “reappear” on the Notebook, with the exact same GUID. This behavior is different from how expunged objects behave within a user’s own account, where expunged objects are never resurrected. So the client should correctly handle this sequence:

1) Note with GUID abc-1234-def is added to a shared notebook and the client synchronizes to see it.
2) Later, the client synchronizes and is informed that note abc-1234-def has been expunged
3) In another, later, synchronization, the client is informed that note GUID abc-1234-def has been added to the shared notebook.

The service will guarantee that an individual SyncChunk will never include both a Note and its own expunged GUID, but two subsequent SyncChunks may contain this sequence, and the client must handle it correctly.

Send Changes

The client should track and submit new and changed notes to the service in the same manner as with their own account, but using the authenticationToken provided for that shard and linked notebook.

If a client has “write” permissions to the shared notebook, then it will be able to invoke either createNote or updateNote for any note within that linked Notebook.

The client will not be permitted to expunge a Note from a shared notebook. Since the client doesn’t have visibility on the Trash for a shared notebook, it’s recommended that the client doesn’t offer a way to “delete” notes from a shared notebook, either.

