Update Tracking

Update Tracking

This document describes how the backend services should use to view to detect changes in records.

Motivation

Given that we want to work on Views of data, we want to be able to monitor when an object View has changed. We say that an object View has changed if any of the objects in the View have changed. We use the term Record to denote the set of objects in a view.

States in Fedora get special treatment for Records. A Record is considered Active if the entry object in the record is Active. A Record is considered Deleted if the entry object is Deleted. And of course, a Record is Inactive if the entry object is inactive. A Record cannot formally be in several states at once, but it is more useful to consider the three states as three branches of the Record. Even if the Record is inactive, you can get the latest active version of the Record.

We want to be able to return all Records in a given Collection for a given State that have been modified after a given Time. To do this, we maintain a database of Views that is updated on all changes of an object.

Finding Changed Records

To find changed records we will ask for a set of entry objects with the following criteria

- collectionPid: The id of the collection containing the objects
- state: The state of the entry object
- viewAngle: The viewangle which the entry objects must be entries in
- offset: Disregard any entries not modified after this timestamp
- limit: The number of entry objects to return

We will then get a list of records, sorted by the timestamps with these fields

- entryPid: The pid of the entry object
- timestamp: The timestamp of the last change to this record
- collectionPid: The collection pid, as given in the parameter. This is not currently used for anything
- contentModelPid: The content model which denoted this object as a entry object for the given view. This is not currently used for anything

. For each entry pid, we can construct the Record by the method CalculateView, as it looked at the given timestamp.

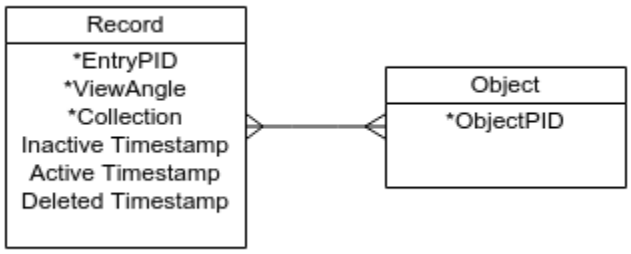
Maintaining state

Whenever one of the objects in a Record is changed, the whole Record counts as updated. As such, any services that subscribe to the Repository in any way need to be notified. If there is a search index for the Records, and one is updated, its state in the index must be recomputed.

The problem arrives when trying to do this. The View system is designed to ease the computing of a Record when knowing the Entry object. The reverse is finding the Records, ie. the Entry objects, that have this data object in their View. Rather than encoding this information in the model, we chose to keep an external record of all the views.

The external record will be a database. It will have two tables.

Database Schema



The first table, RECORDS, will have these columns

- entryPid: This is the pid of the entry object
- viewAngle: This is the viewangle
- Collection: The is the pid of the collection that this record belong to

- Inactive Timestamp: This is the latest timestamp, where anything in the record have changed
- Active Timestamp: This is the latest timestamp, where anything changed in the record while the entry object was active
- Deleted Timestamp: This is the latest timestamp, where the entry object was deleted

EntryPid, viewAngle and Collection will form a key.

The second table, OBJECTS, will have these columns

- objectPid: The pid of this object

The two tables will have a many-to-many relation, linking objectPid to (EntryPid,ViewAngle,Collection)

Changing an object and marking the view as updated

Basically, we need three kinds of operations to handle updates:

- We need to update the time for when a bundle was last updated. We'll call this "updateTimestamps"
- We need to update which bundles exists in which states. We'll call this "modifyState"
- We need to update which objects are part of the view. We'll call this "reconnectObjects"

There are a fixed number of operations that can be done on objects in doms.

For each of these, this is what should be done on the index as a result

1. Object Created: The Object was created in DOMS
   Fedora operations:
   - ingest
   Action:

   ```
   modifyState(Inactive)
   reconnectObjects()
   updateTimestamps()
   ```

2. Object Deleted: The Object was purged from DOMS
   Fedora operations:
   - purgeObject
   Action:

   ```
   modifyState(Deleted)
   updateTimestamps()
   if content model
         for all objects of this class
                 reconnectObjects()
                 updateTimestamp()
   ```

3. Object State Changed: The Object changed state in DOMS
   Fedora operations:
   - modifyObject
   Action:

   ```
   modifyState(state)
   updateTimestamp()
   ```

4. Datastream Changed: The Object datastreams changed. Handled differently depending on whether this is the relations datastream
   Fedora operations:
   - addDatastream
   - modifyDatastreamByReference
   - modifyDatastreamByValue
   - purgeDatastream
   - setDatastreamState
   - setDatastreamVersionable
   - updateTimestamp
   Action:

```
if RELS-EXT
  reconnectObjects(this)
fi
updateTimestamp(this)
if VIEW and Content Model
      for all objects of this class
              reconnectObjects(object)
              updateTimestamp(object)
fi
```

5. Object Relations Changed: The Object changed in a fashion that DOES require the view to be recomputed.
   Fedora operations:
   - addRelationship
   - purgeRelationship
   Action:

```
reconnectObjects(this)
updateTimestamp(this)
if this is a content model
        for all objects of this class
              reconnectObjects(object of this class)
              updateTimestamp(object of this class)
```

Each of these operations will be elaborated below

modifyState

Param new state (one of Active, Inactive or Deleted)

if new state is not Deleted

- If the object was not previously known in RECORDS
  - if the object is an Entry object # Check content models vs. content model cache
    - Create row in RECORDS and OBJECTS denoting this Record, with the Inactive Timestamp set
    - if the new state is Active
      - also set the Active Timestamp
- else
  - For each row in RECORDS with entryPid = this pid
    - set Inactive Timestamp
    - if new state is Active
      - set Active Timestamp

If the new state is Deleted

- Get all Records containing this pid from OBJECTS and not this pid as entryPid #This gets all the other records that could be affected by this delete
- For each of these Records
  - reconnectObjects(record.entryPid) # Recalculate the records
- Delete all rows with objectPid=this pid from OBJECTS # Remove reference to this object
- For each Record with entryPid = this pid # And mark is as deleted if it is an entry
  - set Deleted Timestamp
  - unset Active and Inactive Timestamp

Update Timestamps

select Records from OBJECTS where Deleted Timestamp <= Inactive Timestamp

for each

- If Active Timestamp >= Inactive Timestamp
  - set Active Timestamp
- set Inactive Timestamp

Reconnect Objects

An object's relations changed or an object was deleted. This could change which objects are in which entry's views.

Get the view Information about this object (Which viewAngles is this object entry for)

get the Collection information about this object (which collections is it in)

Create Records in RECORDS corresponding to all these view angles and collections (if they do not exist already) with the Inactive Timestamp set

For each Record in RECORDS with this entry pid and not in this set of view angles or not in this set of collections

- Set Deleted Timestamp
- unset Inactive and Active Timestamp
- remove all objects from OBJECTS linked to this Record

for each Record this object is part of (query OBJECTS with objectPid = this pull)

- remove all Objects relating to this Record from OBJECTS
- recalculate view of entryPid/viewAngle
- update OBJECTS

There are a number of cases, which are better to discuss now

1. A view relation is added, meaning that some other object will now be included in the view. We recalculate the view and update OBJECTS, so this will be noticed.
2. A non-view relation is added, meaning that the view will not change We recalculate the view and update OBJECTS, but this will be a no-change.
3. A view relation is removed, meaning that the view will now contain one object less. We recalculate the view and update OBJECTS, so this will be noticed.
4. A non-view relation is added, pointing to an object in another view, which have this relation as an inverseViewRelation, meaning that this object will now be part of that other view. This will not currently be noticed, which is a problem
5. A content model relation is added. If this content model makes the object an Entry object, this will be noticed. If this object was already part of a view, the view will be recalculated, and thus the change will be noticed. If the object was not part of a view and the change did not make it an entry, it should not be noticed.
6. A collection relation is added. If this object is or becomes an entry, this will be noticed. Otherwise, it should not be noticed.

Implementation considerations

- For performance reasons, it might be a VERY GOOD IDEA to cache some of the content models lookups, as recalculateView could be way to slow.
- Especially update Timestamps will have to complete is milliseconds if this design is to work, as otherwise the doms operations will be faster, and with the current load the update tracker would get further and further behind. Whereas DOMS can be multithreaded, the update tracker will have to complete operations sequentially or implement very advanced locking.

Purge of content models are meaningful. Mark as deleted does not matter, as no content model states matter

A content model table, to quickly check if you are a content model and what you are entry for could speed things up a lot