EventMechanism

Event Mechanism for Modifications to DSpace Object Model

lcs 23-May-06

This wiki page proposes a new piece of the DSpace core API to monitor all object model modification events.

Introduction: What is it?

The *event mechanism* is a means to to notify any designated "receiver" (i.e. run some code) whenever there is a change in the state of the DSpace archive's object model. "Change" means any addition, modification, or deletion of the content and metadata objects such as Items, Bitstreams, Collections, etc.

An "event" is a formal description of a change to the state of the archive; it has a precise and detailed description of what changed. Events can be categorized and filtered

Sketches of "use cases": Where does Event Mechanism fit in?

The first three of these (history, search, and browse) are the only ones planned to be implemented at first.
*__New History system__ - The History recorder is a consumer of events. It only records changes to persistent objects (Item, Bitstream, etc) so it filters out events on e.g. `BitstreamFormat`s. Since the event itself contains all data needed for the History record, it does not need to look up anything in the database. History recording will probably run asynchronously.
*__Search Index updating__ - Automatically update the search indexes when content and/or metadata changes. Driving this function with events gives us the option of making it asynchronous, so applications that make many changes (e.g. ingesting many packages or making several changes to an Item) can complete without waiting to rebuild indexes. It may also want to coalesce all the changes to an object that are close in time, so it can perform one update. The event model would also give us the option of doing index updates in a separate JVM or thread.
*__Browse Index updating__ - just like search index updating, only maintaining the browse indexes. These are separate functional blocks with some subtle differences but their use of the event model is very similar.
*__AIP updating__ - Rebuild AIP manifest after an Item (or Collection, Community) has been changed. Doing this in response to events rather than synchronously has several advantages:

-
    - Removes a time-consuming operation from the original transaction.
    - Can coalesce a batch of changes to an object made close in time so the AIP is updated once, for a large savings in time.
      *__Object caches__ - Each webapp could keep a "persistent" cache of objects in memory, persistent between Web requests (unlike now), to speed performance. Receipt of an event instantly decaches the object that was modified. This could offer a large performance gain over the current object cache that only lasts for one Web request.
      *__Event log__ - Write a separate formal log of modification events including those too ephemeral to be recorded by History.
      *__Statistics__ - record statistics directly into a live data store. Runs in every webapp. NOTE: This would possibly require *dissemination* events to be reported as well!
      *__Automatic media filters__ - e.g. generate thumbnails for new images upon ingestion (or soon after).

Desireable Features in the Event Mechanism

- Events are produced in the object-model level of the core code, e.g. DSpace Object classes such as Item, Bitstream, Collection, etc. The object itself generates an event whenever it is modified, created, or destroyed. This lets *all* applications produce events automatically whenever they change anything in the object model.
- There are many producers of events (separate webapps, command-line apps), and also potentially many potential consumers of events. This points toward a publish/subscribe model of event delivery, although not exclusively.
- Allow clients to receive events asynchronously, even after a certain dormant period; so, e.g., an AIP-updating daemon could start up every hour and process queued modification events.
- Events are persistent until they expire after some configured time, so event queue survives across crashes and restarts of the server. Since History and the browse/search indexes will depend on them, events must not be lost.
- Post and receive events between separate JVMs, possibly on separate machines – e.g. for a local server cluster.
- Do *not* consider delivering events to a separate archives (i.e. site). Inter-site events require a stable protocol interface and should be the subject of a different development effort. It could be driven by the local event mechanism, and used to implement replication of assets.
- Subscriber Filtering: can select events by filtering along these "axes":
    - Object Type of event's Subject:
        - Item
        - Bitstream
        - Collection
        - Commnity
        - Site
        - EPerson
        - Workflow``Item
        - Workspace``Item
        - (maybe) Bitstream``Format
        - (maybe) Metadata``Schema, Metadata``Field
    - Event type:
        - create
        - modify content

            - modify metadata
            - destroy
            - add (to collection/community)
            - remove (from collection/community)
- Contents of an Event are similar to the history entry described in HistoryDiscussion:
    - Event type (as above in filters)
    - Subject (object) type
    - Subject Identifer (i.e. object that was acted on)
    - Date and Time it occurred.
    - Outcome - details of changes made.
    - Additional DSpace objects acted upon, e.g. Collection that Item was added to.
    - DSpace Context object, includes:
        - Name-and-versions of software modules involved (both DSpace-internal and "application", if available).
        - Identifier of a higher-level transaction grouping several events done on behalf of e.g. the same web request.
- Granularity of events: Mostly "to be decided". For now, assume each event describes activity on one DSpace Object, e.g. separate events for creating each Bitstream in a package ingest.
- Help application to avoid being triggered by its own events, e.g. a media filter that adds a thumbnail image must recognize the event _that_action generated and ignore it, instead of getting into an infinite loop. Same issue with the AIP-refresher. (This might be implemented by putting an "application name" in the Context, so application could sense and ignore its own events.)
- Preserve "stateless" nature of DSpace implementation: The application layer of the code has no state, it's all at the object model level and encapsulated in the database and asset store. Applications sharing the database and asset store all see the same content. Events must reflect this shared state.
- Event mechanism should not adversely affect the present performance.
- Provide a configurable "local" mode, which assumes the archive only runs one JVM at a time which includes all event producers and consumers. This would presumably allow substantially better performance for sites willing to accept the restriction. However, even command-line applications could not run with another JVM present, if they produce events that it would not see. Perhaps this can just be implemented as an automatic optimzation for producers and consumers in the same JVM.

Implementation Issues

- Build event generation into every object representing data in the archive, i.e. all the DSpace``Objects, Workspace``Item, Workflow``Item, EPerson, etc.
- Interaction between database transactions and Events: We don't want to fire any events associated with a transaction until the transaction is committed successfully. When one transaction generates several events, we have a problem:
    - Perhaps queue all events on the Context object and fire them after committing the transaction.
    - Ideally, use a JDBC-based event store sharing a database connection with the object model so we can commit the new events and the other changes concurrently, or roll them back together in the event of an error.
- How is DSpace Object represented in the Event schema? If it's just an identifier (e.g. Handle) then every event receiver has to hit the DB to create an Item object. However, if we attempt to pass an actual pre-loaded Item object, that has some potential problems:
    - Overhead of serializing all the contents for remote transmission.
    - DSpace Objects containing "dead" Context are problematic, since it may still need to load more data to fulfill "get" methods, but it cannot get to the DB with a dead Context.
- Could be based on Sun's Java Message Service (JMS) API. Perhaps hide the JMS API to be able to change it out later. Consider the Apache ActiveMQimplementation.
- Consider giving receivers a means to "coalesce" redundant events to help the ones that only care about whether an item got changed (e.g. AIP-refresher) and not about the details.

Next Steps

- Implement a minimal prototype:
    - Only Item (and maybe Bitstream) object.
    - Build on top of a message-passing library like ActiveMQ.
    - Test performance of local and remote event passing.
- Solicit Comments on this wiki page:

Prototypes

- A simple design and prototype is described in SimpleEventHandling
- Full prototype in EventSystemPrototype