EventSystemPrototype

Contents

Prototype Implementation of New Event System

This page describes and links to the code of a prototype implementation of the EventMechanism proposal. It is similar to the SimpleEventHandling prototype proposal, indeed, it started from Richard Rodgers' codebase.

The rest of this page assumes you've read EventMechanism and SimpleEventHandling, so review them now for the basic terminology and architecture.

Purpose of Event System

- Improve modularity by consolidating code that reacts to changes in the data model, e.g. updating of search and browse indexes, history, etc.
  - Event system is configurable at runtime, aiding experiments with new event "consumers".
  - Sites can choose to turn off unused consumers, such as the History System.
- Includes a default configuration that mimics the current system's behavior precisely: search and browse indexes are updated synchronously.
- The asynchronous event mechanism is disabled by default, but can be enabled in the configuration, to explore the utility of asynchronous updates in various contexts.

About the Prototype

Data Structures

An **Event** is defined as a record containing the following fields:

1. **Type**, or *Action*, describes the action that caused the event. One of `Create, Delete, Modify, ModifyMetadata, Add, Remove`.

2. **Subject**, the archival object which was primarily affected by the action described by this event. For example, in an **Add** event adding an Item to an Collection, the Collection is the subject. **NOTE:** The Subject and Object are represented by pairs of integers naming the object type and DB id of the DSpace objects in question, for efficiency in serializing the event. There is a convenience method in the event implementation to fetch the actual `DSpaceObject` when needed.
3. **Object**, the other archival object (if any) that took part in the action, e.g. the Item which was added to the Collection in the previous example.
4. **Detail**, an arbitrary string supplied by the application code giving some useful detail about the event, e.g. naming which metadata fields were modified.
5. **Timestamp**, the time at which the event occurred.
6. **TransactionID**, an opaque string value which is identical for all of the events spawned by one Context-based transaction.
7. **Context** The context that spawned the events (or a reproduction in the case of asynchronous events) is also passed to consumers; its significant fields are:
8. ***CurrentUser** the authenticated EPerson who caused the event.
9. ***ExtraLogInfo** another arbitrary string set by the application.

Contents of an Event

The Detail field of an event (accessed by *event.getDetail()* returns a String whose significance varies by the type of event and the type of subject it is about, as follows:

Contents of the Detail field in an Event

Object Type	Create	Delete	Add	Remove	Modify	ModifyMetadata
Bitstream	*null*,"REGISTER"	*subject.getSequenceID()*	*n/a*	*n/a*	*null*	field list
Bundle	*null*	*subject.getName()*	*object.getSequenceID()*	*object.getSequenceID()*	*null*	*null*
Item	*null*	*subject.getHandle()*	*object.getName()*	*object.getName()*	*null*, "WITHDRAW", "REINSTATE"	DC field list
Collection	*subject.getHandle()*	*subject.getHandle()*	*object.getHandle()*	*object.getHandle()*	"remove_template_item",*null*	*metadata field names*
Community	*subject.getHandle()*	*subject.getHandle()*	*object.getHandle()*	*object.getHandle()*	*null*	*metadata field names*
EPerson	*null*	*subject.getEMail()*	*n/a*	*n/a*	*null*	field names
Group	*null*	*subject.getName()*	*object.getEMail(), object.getName()*	*object.getEMail(), object.getName()*	*n/a*	"name"
Site	*n/a*	*n/a*	*object.getHandle()*	*n/a*	*n/a*	*n/a*

Key:

- *n/a* means the event does not occur; for example, a Bitstream cannot add or remove members.
- *null* means the field is set to the *null* value.
- A comma-separated list describes several different possibilities.

About the Code

The prototype code includes new classes and interfaces making up the event system itself, some event consumer classes, and a large set of changes to the codebase. The changes are to implement the event system, and also to remove the old calls search and browse index updates, since they are now handled through event consumers.

It adds event consumers for search and browse systems which will be the only means to automatically update the search and browse indexes.

It also includes an event-based implementation of the Subscribe function email sent to an EPerson about new/changed Items in subscribed collections) to demonstrate what can be done with asynchronous event processing.

The prototype supports multiple configured event dispatchers, so each application (or even each Context within an application) can choose a dispatcher appropriate for its needs – e.g. interactive apps can process search updates immediately, while batch imports defer them to improve performance.

The old HistoryManager is removed and is not replaced by this implementation. See the History Prototype page for a new implementation.

This code has only be tested against the PostgresQL 7.34 and 8.1 databases; I would appreciate information about experiences with other databases.

Sample Source Code

The Event Prototype is now maintained under Subversion at MIT Libraries located here http://libstaff.mit.edu/svn/repos/projects/dspace-messaging-v2-prototype

The DSpace patch queue tracker item is currently out-of-date. Please use the subversion repository branch as the canonical source for the Event prototype.

Installing and Configuring

Installing the Prototype

SVN Checkout

To install the event prototype, start with an SVN checkout of the dspace-messaging-prototype project. This code is currently undergoing testing and debugging.

Requirements

DSpace 1.5 and later requires Java 1.5 or later to build and run. Be sure your default java invocation or *$JAVA_HOME* is running Sun's JVM of at least that version. It was tested under Sun JRE Standard Edition 1.5.0_08.

Optional Asynchronous Support

There is a "dspace-jms" addon which will support the addition of Asynchronous JMS based messaging via ActiveMQ, it is available under http://libstaff.mit.edu/svn/repos/projects/dspace-messaging-v2-prototype/dspace-jms dspace-jms] in the prototype branch. This can be added to your build of dspace by adding it to the modules listed in the dspace/pom.xml.

Now you should be able to build and install the DSpace source as usual.
See the next section to configure it before starting a server.

Configuration

Here is a list of all the configuration keys, topic, followed by an example (default) fragment of your configuration file.

Consumers

event.consumer.name.class - Creates a consumer named *name*, the value is a fully-qualified Java class name that implements the consumer.There must be a corresponding filters configuration.

event.consumer.name.filters - Defines a set of event filters for the named Consumer. The value is a list of "filters" which select the events this consumer will see, selected by combinations of DSpace object type and action. The filter list value consists of a set of filter clauses separated by colons (:). Each clause is a set of DSpace Object types, a plus sign (**+**), and a set of actions. The object and action lists are
separated by vertical-bar (**|**). Here is a rough grammar:

```
filter-list ::= filter [ ":" filter ]..

filter ::= object-set "+" action-set

object-set ::= object [ "|" object]..

action-set ::= action [ "|" action]..

object ::= "All" | "Bitstream" | "Bundle" | "Item" | "Collection" | "Community" | "Site" | "Group" | "Eperson"

action ::= "All" | "Create" | "Modify" | "Modify_Metadata" | "Add" | "Remove" | "Delete"
```

The special value "All" denotes all available objects or actions. The filter All+All allows all events through.

The filters in a list are logically ORed together, although they should be distinct.

Whitespace and case are ignored in the filter list, so e.g. *ALL* is as good as *all*.

Dispatchers

event.dispatcher.name.class - Creates a dispatcher named *name*, the value is a fully-qualified Java class name that implements the dispatcher. There must be a corresponding

```
consumers
```

configuration.

event.dispatcher.name.consumers - List of consumers to which this dispatcher sends events. The value is a list of consumer clauses, separated by comma (**,**). Each clause contains the name of the consumer, which must correspond to a

```
event.consumer
```

configuration as described above, followed by a colon (**:**) and a declaration of whether it is *synchronous* or *asynchronous*. The words may be abbreviated *sync* and *async*, and case is not important.

There must always be a dispatcher named *default*. This is the dispatcher used when the application does not set any specific dispatcher in the *Context*.

Choosing Dispatcher in Applications

To demonstrate configurable dispatchers, the *org.dspace.app.packager.Packager* application has a configurable dispatcher. The key is *packager.dispatcher*, the value is the name of a dispatcher. Default is, of course, *default*.

Optional Asynchronous Support

JMS Configuration

The following keys configure the Event system's use of JMS. Note that these are only needed if your chosen Dispatchers have any asynchronous consumers.

jms.timeToLive - sets the "time to live", for persistent messages bearing asynchronous events, which is the time they will be kept around. It is an integer measured in milliseconds. Default is 2 days.

jms.messageType - type of JMS message to use for asynch events. Not used by ActiveMQ, but it might be needed by a different JMS provider, so the configuration mechanism is ready.

jms.broker.uri - URI of broker to use when creating the first *ConnectionFactory*. See ActiveMQ 4.0 Documentationfor details about how to configure this.

jms.configuration - URI of XML configuration file for JMS java beans.Used by Spring to configure ActiveMQ. The default value, *xbean:/activemq.xml* looks for a file *activemq.xml* in the DSpace config directory.See ActiveMQ 4.0 Documentationfor instructions on the contents of this file, or follow the example.

ActiveMQ XML Configuration

The default version of the ActiveMQ configuration file will be in your install directory under *config/activemq.xml*. Copy it to the runtime config directory (e.g. *dspace/config*) and modify it if necessary, consult the ActiveMQ 4.0 documentation for details.

The version supplied works with a PostgreSQL database.

Example

```
# This default dispatcher preserves the status quo, all synchronous
# consumers of search, browse, and history:

event.dispatcher.default.class = org.dspace.event.BasicDispatcher
event.dispatcher.default.consumers = \
search:sync, \
browse:sync, \
history:sync

event.consumer.search.class = org.dspace.search.SearchConsumer
event.consumer.search.filters = Item|Collection|Community|Bundle+Create|Modify|Modify_Metadata|Delete:Bundle+Add|Remove

event.consumer.browse.class = org.dspace.browse.BrowseConsumer
event.consumer.browse.filters = Item+Create|Modify|Modify_Metadata:Collection+Add|Remove

event.consumer.history.class = org.dspace.history.HistoryConsumer
event.consumer.history.filters = all+*

# email to subscribers – run this asynchronously once a day.
event.consumer.mail.class = org.dspace.eperson.Subscribe
event.consumer.mail.filters = Item+Modify|Modify_Metadata:Collection+Add|Remove

# example of a configuration with a couple of async consumers
event.dispatcher.with-async.class = org.dspace.event.BasicDispatcher
event.dispatcher.with-async.consumers = \
search:sync, \
browse:sync, \
mail:async, \
testALL:async, \
history:async

event.consumer.testALL.class = org.dspace.event.TestConsumer
event.consumer.testALL.filters = All+All

# dispatcher chosen by Packager main()
packager.dispatcher = batch

# ActiveMQ JMS config:

jms.configuration = xbean:/activemq.xml

# local TCP-based broker, must start
jms.broker.uri = tcp://localhost:61616
```

Operation

Start and run applications as usual.

To see events in action, alter the default dispatcher configuration to include the **testALL** consumer, and make sure your DSpace log is recording at the INFO level (at least). Then, watch the log while doing anything that changes the data model; look for messages from the **TestConsumer** class.

You can also run the test consumer as an asynch consumer in a separate process to observe how asynchronous events are passed along in real time, or accumulated between polls.

Broker

If you configure any *asynchronous* dispatchers, you'll have to run the **ActiveMQ broker** on your server as well. There is a script to start and stop it easily which has been add to the *bin* directory of the source; it should get installed in the *bin* subdirectory of the DSpace runtime hierarchy.

Configuration

Check the default **ActiveMQ** configuration in *dspace-install/_config/activemq.xml*. The PostgreSQL login in particular may need to be configured for your site. ActiveMQ uses the database to keep tables of persistent events. They are automatically maintained to discard expired events.

Startup and Shutdown

To start the broker, run the command

```
dspace-install/bin/asynch-broker start
```

e.g.

```
/dspace/bin/asynch-broker start
```

To stop the broker (perhaps to restart it), run the command

```
dspace-install/bin/asynch-broker stop
```

You can make these commands part of the regular startup and shutdown procedure of your server; they are designed to be invoked from System-V-type "rc.d" scripts as are used on Solaris and Linux. Justbe sure to run it as the same user who owns your DSpace processes.

Processing asynchronous events

To process asynchronous events, you can run one consumer at a time with the command:

```
/dspace/bin/dsrun org.dspace.event.AsynchEventManager -c CONSUMER
```

e.g.

```
/dspace/bin/dsrun org.dspace.event.AsynchEventManager -c mail
```

..substituting the consumer name for *CONSUMER*, of course. Give the command with the **-h** option for help on other arguments and options, or see the comments in the source.

Pitfalls

Here are some unresolved issues and problems in the prototype. Your comments and proposed solutions are welcome!

1. Removing duplicate events

The code in *Context.addEvent()* pre-filters the events by removing any events which are *duplicates* – that is, identical to an event already in the queue for this transaction in all respects except for timestamp. The rationale is that a duplicate soaks up processing resources and does not convey any additional information, even to the History system, since events are so fine-grained. Furthermore, the way in which the current applications (e.g. WebUI) use the data model API seems to produce a lot of extraneous duplicate events so this filtering does a lot of good.

2. Consumer execution environment

Consumer code runs in a somewhat strange environment:

- Before calling *consume()*, the Context object will have already called its *commit()* method to commit its changes to the RDBMS, although the DB connection is still open.
- Any DB changes made by the consumer code must be committed by explicitly calling the JDBC commit() method on the DB connection.

- Consumers *must not* call the Context's commit() since it would run the event dispatch again, causing an infinite loop.
- Given the above restrictions, consumer code *can* modify data model objects and update them; for example, it is possible to run the MediaFilter from an event consumer to update e.g. thumbnails whenever an object is changed.
- Since the consumer may be run asynchronously (and it has no way to tell), it must allow for the possibility that the DSpace Objects referenced in the event may no longer exist. It should test the results of every find().

3. Handling failures

Since the Event system forms a vital part of the core DSpace server, any failure in event processing should register as a fatal error in the transaction. Unfortunately, the events are, necessarily, triggered *after* the database transaction commits. (This is necessary because otherwise there would be a race condition between asynch event processors looking at the data model and the process generating events, the asynch consumer might see a pre-transaction view of the DB.)

Also, failures late in the cycle of a WebUI transaction do not get rendered correctly as an error page because the servlet has already generated some output and the attempt to send a different status and an error page just sets off an illegal state exception. This problem is inherent in the current implementation of the Web UI and would be very hard to change.

Since the asynchronous event mechanism is the part most susceptible to errors, depending as it does on network resources and complex configuration, the Context code makes an attempt to exercise the dispatcher and asynch delivery as much as possible *before* committing the transaction, to flush out some fatal problems in time to abort it.

4. JMS requires explicit call to System.exit()

The ActiveMQ implementation uses shutdown hooks to terminate its internal state, and if they are not called the result is a JVM that hangs instead of shutting down because an ActiveMQ thread is still waiting for input from a network peer.

The solution is simple: command-line applications must always call *System.exit()* before terminating, and *not* just run off the end of the *main()* method. (Though the fact that there is this distinction counts as a flaw in the Java runtime, IMHO; the Unix system call they are aping has no such restriction.)

This prototype includes patches to add *System.exit()* calls to all command-line applications that can generate events. It's a good idea to fix any application that generates events at all, whether or not you anticipate any of those events being asynchronous.

5. ActiveMQ needs a separate broker process

The current configuration of ActiveMQ requires a separate "broker" listening at a well-known TCP port. For the prototype, it is started manually.

I'll investigate other ActiveMQ broker options (some of them simply don't work for this application, however), and also code to start it automatically or at least through a more friendly DSpace application.

6. Context current user in Consumer

When an asynch Consumer runs, its Context has the same CurrentUser set that was set in the code that generated the event. Since Consumers should not be doing anything that requires special privileges, this probably won't be an issue, but it's worth noting.

7. Browse not all in a Consumer

One part of the Browse index updating cannot be put into a consumer because of an architectural problem: The Browse tables have foreign keys into the Item table. Deleting an Item thus breaks those foreign key references, so the Browse tables must be updated first. By the time a Browse update would be running in an event consumer, the Item table would have already have had to be updated to reflect the delete, which is impossible.

See the *delete()* method in *org.dspace.content.Item*.

This is the *only* case in the DSpace core code where a search or browse index update could not be moved into an event consumer.

Future Work

The next tasks planned around the Event prototype are:

- Write a consumer to generate and refresh AIPs automatically.
- Integrate the media filters into a consumer so derived media get updated automatically.
- Investigate better configuration and management of ActiveMQ JMS Provider.

See Also

- EventMechanism - early sketch of event handling
- SimpleEventHandling - design that was the basis for this prototype
- New History System prototype - event-driven history implementation