Sharing data in Orchestrator

Overview

Orchestrator can use a variety of methods to share data between workflows. For example a workflow can write in a queue via the queue stager plugin and another workflow can read from this queue with a resource manager plugin. Or a workflow can write in a shared file and another workflow can read from this file. In this article, we'll see how to use the pre-defined shared_states table to share data.

A use case is periodic cleanup. For example, a workflow would detect files then exit without deleting the detected files as they are still being processed by an external application. These files are stored in shared_states. Another workflow would periodically scan shared_states for entries older than 2 days and delete the found files from the disk.

Another use case is workflow synchronization. For example, a workflow would detect files then pass them to an external application for processing. It would then wait for the application to complete processing by scanning shared_states on a given correlation id. The external application would call another workflow when done with its processing. This workflow would add an entry into shared_states for the given correlation id. Thus the first workflow would be unblocked and continue its processing.

shared_states table definition

This MySQL table is created with the Orchestrator installation and is defined as:

#	Name	Datatype	Length/Set	Unsig...	Allow NULL	Zero...	Default
1	**id**	**INT**	**11**	☐	☐	☐	**AUTO_INCREME...**
2	path	VARCHAR	255	☐	☑	☐	NULL
3	name	VARCHAR	255	☐	☑	☐	NULL
4	entry	LONGBLOB		☐	☑	☐	No default
5	entity	VARCHAR	255	☐	☑	☐	NULL
6	aggregate_type	VARCHAR	255	☐	☑	☐	NULL
7	aggregate_id	INT	11	☐	☑	☐	NULL
8	created_at	DATETIME		☐	☑	☐	NULL
9	updated_at	DATETIME		☐	☑	☐	NULL

Indexes have been created for the ‘path’ ,‘name’, ‘aggregate_type’ and ‘aggregate_id’ fields so that they can be used for lookups.
‘path’ and ‘name’ are both limited to 255 characters length. For example, you could put a file path or a correlation id into ‘path’ and a workflow name in ‘name’. Anything fitting your workflow logic can be used.
The ‘entry’ field is a LONGBLOB. Some user data like an XML content could be put into this field.

shared_states utilization

Adding an entry

An entry can be added in the shared_states table with the `SharedState.persist_to_db()` method in a custom Ruby plugin.

With this syntax:

```
states = SharedState.persist_to_db(entry,path/name)
```

The first parameter is a hash holding the 'entry' field.

The second parameter is a string holding 2 values separated by a '/':

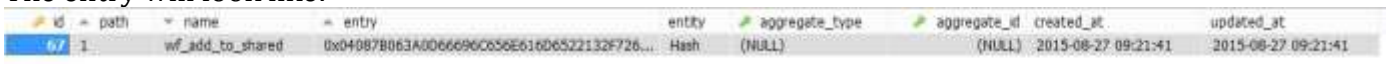
- the value for the 'path' field
- the value for the 'name' field

Example: if filepath and id are input parameters of the workflow "wf_add_to_shared", I can add an entry into shared_states with this code:

```
s_id = SharedState.persist_to_db({:filename=>inputs['filepath']},
"#{inputs['id']}/wf_add_to_shared")
```

In this example, the 'entry' hash has only 1 key/value pair with :filename as the key name and the input filepath value as the value, 'path' holds a unique id equal to the input id value and 'name' holds the name of the workflow "wf_add_to_shared".

The entry will look like:



id	path	name	entry	entity	aggregate_type	aggregate_id	created_at	updated_at
67	1	wf_add_to_shared	0x04087B063A0066696C656E616D6522132F726...	Hash	(NULL)	(NULL)	2015-08-27 09:21:41	2015-08-27 09:21:41

Using a unique id guarantees that a new entry is created in the table for each call to this method. Otherwise a new entry will override an old entry, for example if this was used:

```
s_id = SharedState.persist_to_db({:filename=>inputs['filepath']},
"1/wf_add_to_shared")
```

The filepath could also be used in the path or name field like:

```
s_id = SharedState.persist_to_db({},
"#{inputs['filepath']}/wf_add_to_shared")
```

Looking up for an entry

An entry can be searched for repeatedly with the `SharedState.find()` method in a custom Ruby trigger plugin such as:

```
states = SharedState.find(:all,:conditions=>["path = '#{inputs['id']}'"])
states.each do |state|
  debug "entry id = #{state.path}, filename = #{state.data[:filename]}"
  state.destroy #remove DB entry
end
if states.size > 0
  @status_details = "Found #{states.size} entries"
  @status = STATUS_COMPLETE
end
```

The method second parameter holds a SQL statement to express the search criteria such as: `" path = '1' AND name = 'wf_add_to_shared' "`

The method returns an array of entries matching the criteria if any.

The 'entry' field can be accessed on each array element via the `element.data` hash.

To remove an entry from the table, use the `element.destroy` syntax. Note that if you don't remove the element from the table, it will be detected again.

Alternatively, you can use the `SharedState.retrieve_from_db()` method, taking a string as argument with the same syntax as the second argument of SharedState.persist_to_db():

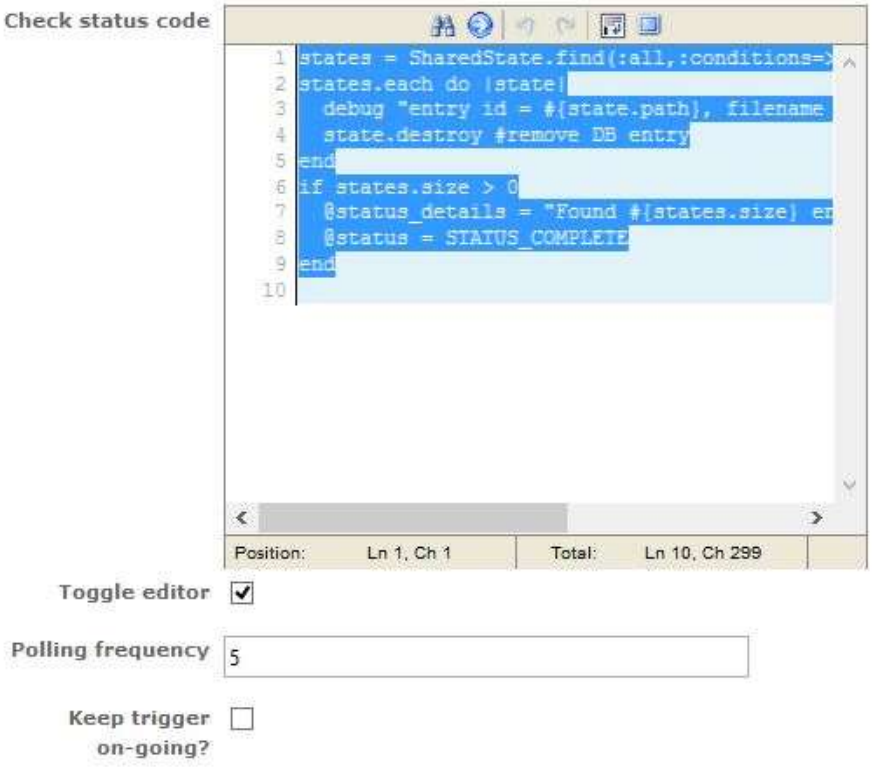
```
states = SharedState.retrieve_from_db(path/name)
```

Example:

```
state = SharedState.retrieve_from_db("#{inputs['id']}/wf_add_to_shared")
debug "entry filename = #{state[:filename]}"
```

Note that only one entry is returned by this method (the first found). The result is a hash holding the 'entry' field value. Also, there is no way to remove an entry from the table with the return of this method.

Custom trigger configuration:



The search is performed for every polling period (e.g. 5s) and the whole “check status code” code is executed at each polling period.

The step keeps calling the code in a loop unless you set a final status in the code (via the @status variable holding either STATUS_COMPLETE or STATUS_FAILURE or STATUS_ERROR).

If you don’t set “Keep trigger on-going?", the step will exit when a final status is met.

If setting “Keep trigger on-going?", be careful to have only 1 step with this setting in a given workflow otherwise you may end up with extra cloned instances. With the code above, when something matching your criteria is detected, the current step exits and a workorder clone is created executing the same code again. Thus you have one workorder per successful DB lookup.

More complex SQL statements can be made for lookups with the find() method such as:

```
states = SharedState.find(:all, :conditions=>["name =
'wf_add_to_shared' AND created_at < (DATE_SUB(NOW(), INTERVAL
aggregate_id MINUTE))"])
```

Here aggregate_id is a value used to determine which entries are older than a certain number of minutes. The entry with aggregate_id could be inserted via a third parameter of the persist_to_db() method such as:

```
s_id = SharedState.persist_to_db({:filename=>inputs['filepath']},
"#{inputs['id']}/wf_add_to_shared",{:aggregate_id=>60})
```

Rest listener

A Rest listener can be implemented in a workflow via the use of the Orchestrator Rest API and a custom trigger or database trigger waiting for a new record in shared_states.

The workflow is waiting for the event triggered externally via the API.

Here is an example of the API Rest call:

```
http://<Orchestrator IP address>
/aspera/orchestrator/workflow_reporter/custom_api_call?login=admin&p
assword=aspera&identifier=testing&uniq_id=321&job_id=123&job_result=
success
```

login/password are the credentials to log into Orchestrator

identifier is a required string to identify (not necessarily uniquely) the record created in the Orchestrator DB.

uniq_id is a required integer to identify uniquely the record created in the Orchestrator DB.

You can add custom optional key=value fields like job_id and job_result as above.

Calling this API will produce a new record in shared_states. You can then use a custom Ruby trigger or database trigger waiting for new records in shared_states.

Here is an example of custom Ruby trigger code based on identifier lookup, waiting for all records matching your identifier:

```
states = SharedState.find(:all,:conditions=>["aggregate_type = ?",
inputs['id']])
states.each do |state|
  #Handle each DB entry
  debug "entry data = #{state.data.inspect}"
  state.destroy #remove DB entry
end
if states.size > 0
  @status_details = "Found #{states.size} entries"
  #Return a complete status to end this workorder
  @status = STATUS_COMPLETE
end
```

In this case, the id declared in input of the custom trigger must be a string and should match the identifier from your API call.

state.data is a hash with the input information such as:

```
data = {"job_result"=>"success", "uniq_id"=>"321", "job_id"=>"123",
"identifier"=>"testing"}
```

Here is an example of custom Ruby trigger code based on uniq_id lookup, waiting for a unique record:

```
states = SharedState.find(:all,:conditions=>[aggregate_id = ?",
inputs['id']])
```

In this case, the id declared in input of the custom trigger must be an integer and should match the uniq_id from your API call.