Nirva EVENT Service

Document Version: 1.07

Table of Contents

Overview

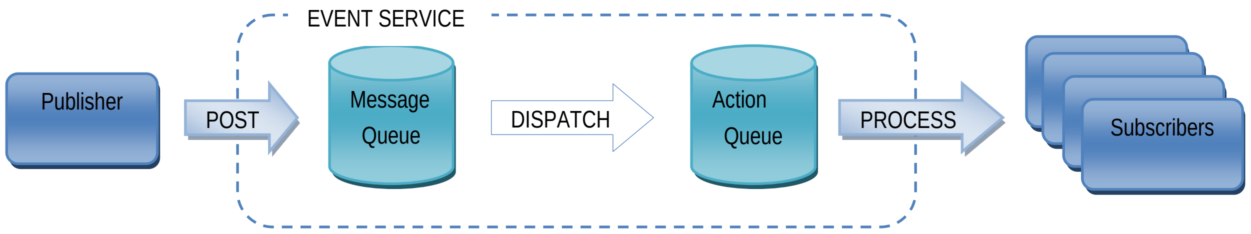
The EVENT service is a NIRVA external service which provides Nirva with a message publishing system allowing building Event Driven Architectures (EDA). It keeps track of messages (events) sent to a channel and the subscribers who are to receive the events. The service allows receiving multiple occurrences of the same messages allowing for escalading mechanisms. It gives the subscribers the possibility to inhibit further occurrences of a given message.

It has a minimal manipulation of the content of the messages in the form of a string for simple cases. For more complex situations, the management of the content can easily be externalized.

The EVENT service allows:

- Having multiple channels open to publishing and subscribing to messages
- Processing multiple occurrences of the same messages which allows setting up escalation mechanisms in case of multiple publications of the same messages (in particular in the case of events being alerts to be processed).
- Management of multiple subscribers per channel and generation of specific actions for each subscriber.
- Inhibiting occurrences of a message on a per-subscriber basis.
- Different possibilities of desynchronizing the posting of an event and its processing.

The EVENT service relies on two notions: messages and actions. From the services point of view, messages are events received by the service and actions are the copies of these messages to be distributed to each of the subscribers of the channel the message was sent to. Internally, the EVENT service stores the messages and actions in two queues. A dispatching phase consists in duplicating events in the message queue into events to be put in the action queue.



The service works in an environment composed of the following acting entities:

- one or more “publishers” posting messages (for example a production system)
- one or more “subscribers” wishing to receive messages sent to a given channel

- the EVENT service itself
- zero or more “dispatchers” allowing to externalize the dispatching of events
- one or more “processors” which process the pending “actions” the EVENT service has in its queue

The EVENT service only plays the role of receiving events and dispatching them to the appropriate subscriber queues. How events are generated is left to the publishers and how they are to be processed left to the processors.

The EVENT service only knows of the subscribers on a by-name basis (i.e. from the EVENT service’s point of view a subscriber is simply a name). It is the responsibility of the external system processing the actions generated by the service to know how subscriber names translate to real subscribers. The EVENT service does not internally know of producers and therefore does not handle this information explicitly. If required the content part of the EVENT messages may be used to contain information about producers.

Producers post messages to the EVENT service on a given channel by calling the “EVENT:MESSAGE:POST” command. Posting a message is asynchronous and once the message has been registered by the EVENT service the call returns.

A processor queries the EVENT service for pending actions to be taken by calling the “EVENT:ACTION:GET” command. This command returns a table with the information about a set of actions to be taken and the target subscribers. The processor does the expected action and informs the EVENT service that the action is complete by calling the “EVENT:ACTION:COMPLETE” command. An action is generated for each occurrence of a given message and for each subscriber which has not inhibited the message.

During processing (which is external to the EVENT service), it is the role of the processor to take the correct action for each subscriber.

Multiple occurrences of a message

In order to handle escalation, the EVENT service allows receiving the same message multiple times.

Each new reception is considered as a new occurrence of the message as long as it has not been cleared. The EVENT service considers two messages to be the same whenever they have the same id. Messages are kept in a list of active messages until they are cleared by an “EVENT:MESSAGE:CLEAR”. Each new reception of an active messages increments the occurrence counter for the message.

It is up to the producer to ensure that two occurrences of a same message have the same message id and to call the clear command when the message has become obsolete. An optional auto-clear mechanism clears the message once all the corresponding actions have been processed.

The feature allowing multiple occurrences can, in particular, be used to build an alert system allowing for alert escalation. In such a case, it is desirable for multiple occurrences of a same alert to “escalade” by, for example, enlarging the scope of people to be informed of the message (abnormal situation). Once the situation has returned to normal, the system should be informed of this.

The “email alert system” tutorial gives an example of how such a mechanism may be used.

Message inhibition

In some cases, a specific subscriber of a channel may wish to ignore further occurrences of a given message. In this case the “EVENT:MESSAGE:INHIBIT” command should be called and the EVENT service will not generate an action for the given subscriber for any new occurrence of the message. However, once the message is cleared, the inhibition state is removed.

In the alert case, this mechanism allows for a particular subscriber to ignore further warnings for a same abnormal situation, for example, once he is aware that it is being taken care of.

The “email alert system” tutorial gives an example of how such a mechanism may be implemented using the EVENT service.

Event-based synchronization

The EVENT service also allows waiting for an event to occur. This allows implementing event-based synchronization mechanisms. In this case, the subscriber issues an “EVENT:MESSAGE:WAIT” command which blocks him until the expected event is received or a timeout occurs.

The “user interface synchronization” tutorial gives an example of how such a mechanism may be implemented using the EVENT service.

Dispatching messages into actions

The EVENT service works with two queues:

- A “message” queue which stores the incoming messages before they are dispatched into actions.
- An “action” queue which stores actions to be transmitted to each subscriber. There is one copy of the message for each of the subscribers of the channel excluding subscribers which have inhibited the message.

Dispatching a message into one action per subscriber can be done at different steps depending on the mode of the channel. The EVENT service provides four dispatch modes: IMMEDIATE, DIFFERED, EXTERNAL and UNLOCK.

The usage or not of the message queues depends on the dispatching mode. This is summarized below:

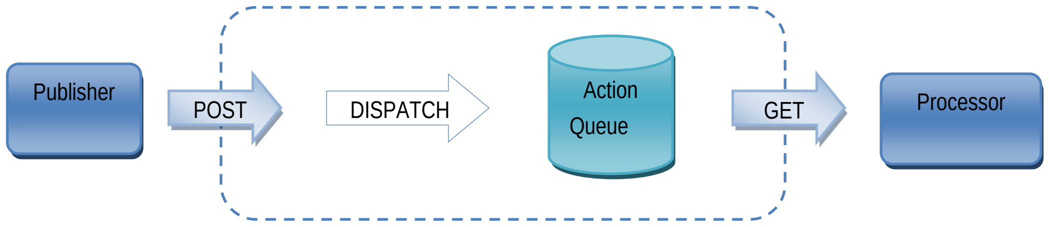
Dispatch Mode	Use Message Queue	Use Action Queue
IMMEDIATE	NO	YES
DIFFERED	YES	YES
EXTERNAL	YES	YES/NO
UNLOCK	NO	YES

When messages are sent to a channel whose dispatching mode is not IMMEDIATE or UNLOCK, the messages are stored into the “message” queue of the channel. In DIFFERED mode pending messages are processed when the action queue is empty and a request for new actions has come in. In EXTERNAL mode a subset of the message queue is retrieved by an external entity in the form of a file which does the processing of the raw events. It can re-inject the messages into the service by calling the MESSAGE:DISPATCH command.

For situations in which posting messages should not be blocking for high performance, the dispatch mode should not be set to IMMEDIATE since the generation of the actions for the incoming message will slow down the caller. The slowing down is linear in the number of subscribers to the channel.

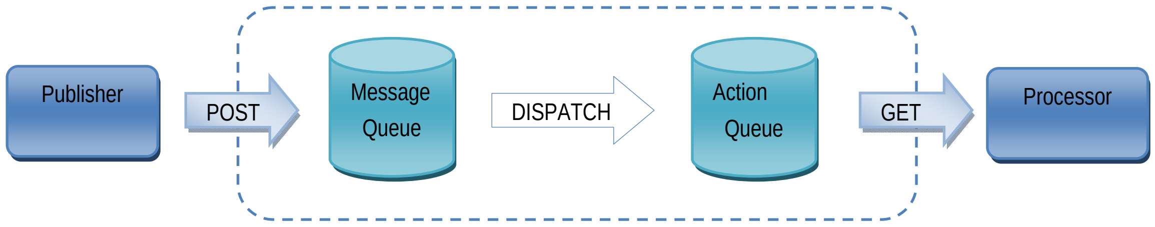
Immediat

Messages are automatically dispatched into actions as soon as the message is posted via the MESSAGE:POST command.



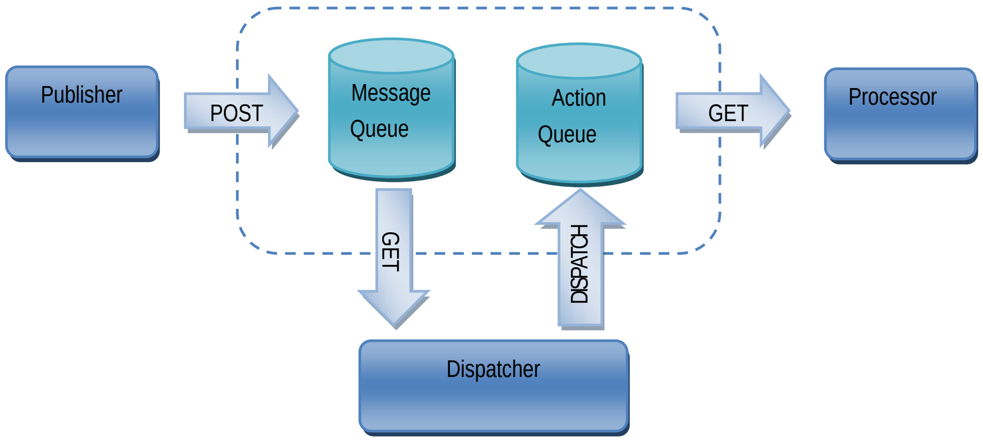
Differed

Messages are dispatched when no more actions are left in the action queue and more actions are required to respond to an ACTION:GET command.



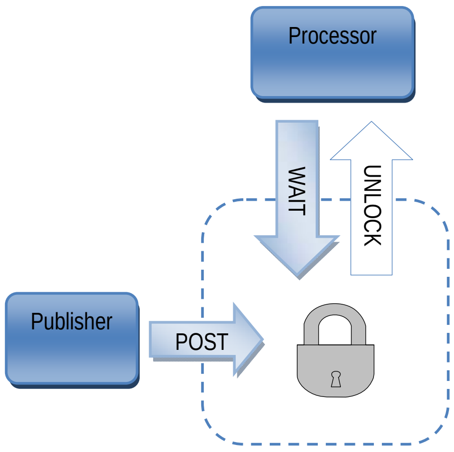
External

Dispatching is externalized. A completely separate entity comes into play to handle the dispatching of pending messages. This entity calls the MESSAGE:GET to retrieve a series of messages to be processed and dispatched. For each message it calls the MESSAGE:DISPATCH command



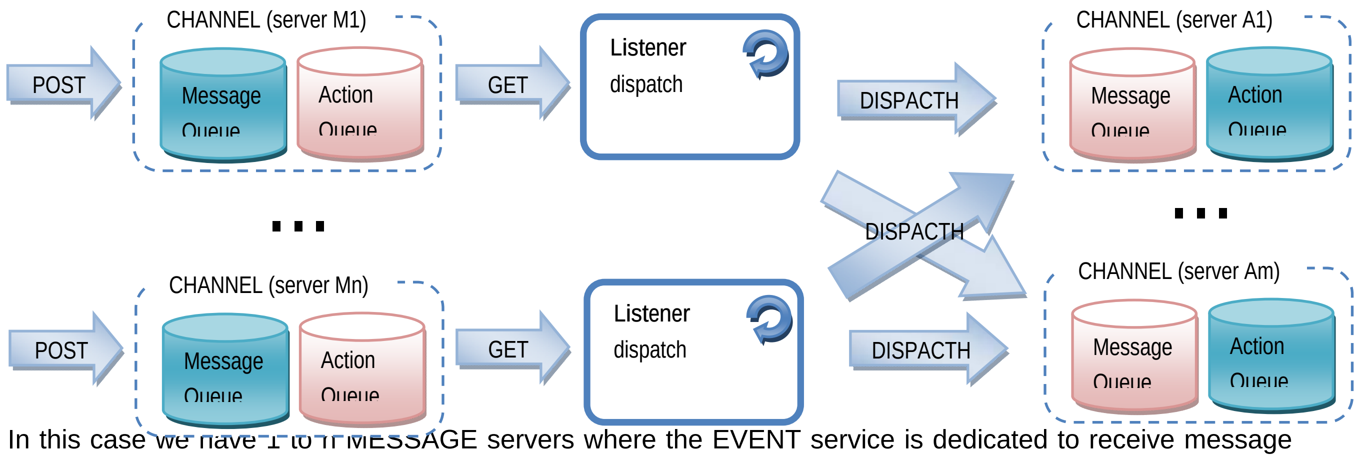
Unlock

In this mode, neither of the two queues is used. The processor calls the synchronous command MESSAGE:WAIT. Upon the reception of a message any waiting processors are unlocked with the information about the message.



Load-balancing message posting

High volume situations may require that multiple servers may be used to post events while. In this case, the EVENT service may be deployed on multiple servers. The following schema describes how this is done.



In this case we have 1 to n MESSAGE servers where the EVENT service is dedicated to receive message posts and 1 to m ACTION servers where the EVENT service is dedicated to deliver actions. All the event services in this case are in “EXTERNAL” mode. Dispatching events from the MESSAGE servers to the ACTION servers is done by implementing listeners which retrieve messages from the MESSAGE servers and dispatch them to the appropriate ACTION server.

For multiple occurrences and inhibition to work correctly the following constraints should be respected.

- If an occurrence of a message is dispatched on one server, future occurrences of the message must also be dispatched on that server.
- If subscriptions may be duplicated among different servers, then
  - inhibitions should also be duplicated on the same servers
  - messages should only be dispatched to one of the servers (otherwise, subscribers will receive multiple times the same occurrence of the same message)

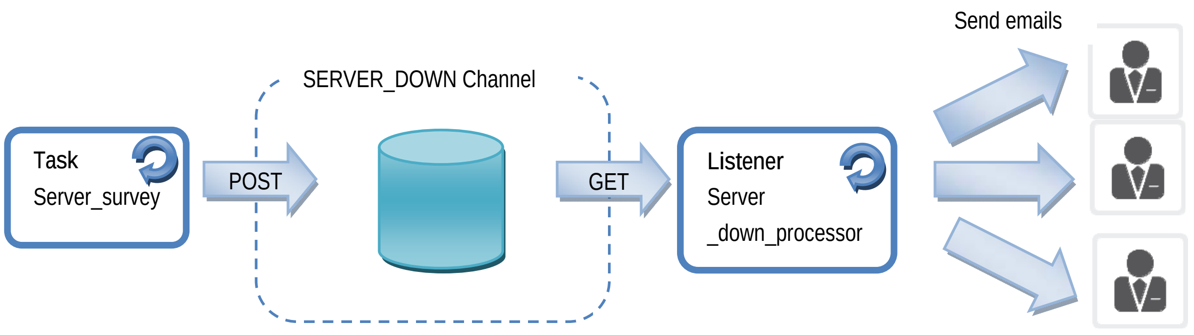
Tutorial Examples

This chapter presents tutorial use-cases of the different ways the EVENT service may be used.

An application package implementing this tutorial can be found in the Files directory of service.

Email alert system

The EVENT service is particularly well suited to set up an alert system. This tutorial will show how to set up a simple email alert system.



The system we wish to build in this tutorial is an application checking that a given server is alive and inform the subscribers by email that the server has come down.

To do this we will build an application consisting of the following components:

- One or more servers to survey. In the tutorial, we will consider the servers are Nirva servers and that the test to check if a server is up consists in successfully opening an NV_REQUEST
- An event channel "SERVER_DOWN" to which alert messages down should be posted when a server is found to be down
- For each server, a Nirva task checking every 10 minutes that the server is up and:
  - When the server is down, posts a message "<TARGET_SERVER>" to the SERVER_DOWN channel
  - When the server is up, clears the message "<TARGET_SERVER>" on the SERVER_DOWN (note that sending a complete for a non existing message is simply ignored by the event service).

- A listener which checks for any pending actions to be taken and for each will send an email to the target subscriber. The subscriber field of the action will be considered as a valid email.

The application will automatically benefit of the inhibition mechanism of the alert service. In the tutorial, this is simply set up by adding an inhibition link inside the alert email sent to the users. When the user clicks on the link he will no longer receive further occurrences of the same message until the message is cleared. For example, if the server being watched is 192.168.1.5 and has come down the survey task will detect this and post and EVENT on the SERVER_DOWN channel. Since the channel is in IMMEDIATE mode, the message will be dispatched as one ACTION per subscriber. If joe@example.com is one of the subscribers, an ACTION for him will eventually be processed by the “server_down_processor” listener which will send him an email. In our full implementation of this example, the email will contain two links: one to inhibit the message, one to clear the message. If Joe clicks on the “inhibit” button the next time the task runs and finds that 192.168.1.5 is still down Joe will not receive this second email. Joe can also click on the “clear” message indicating that he knows that 192.168.1.5 is back up. If the task finds that 192.168.1.5 is still down this will generate a new message which all the subscribers will receive again (even if they had inhibited the message before Joe clicked on clear).

Configuration

To make this example work, you will need the following information:

- The name or IP of a server on which Nirva is installed (TARGET_SERVER in the following)
- The address of an SMTP server not requiring authentication (SMTP_SERVER in the following)
- One or more email addresses to which alert messages are to be sent

Below are the steps to make the tutorial work:

- Install the EVENT service and enable it.
- Configure Nirva to set the SMTP server to SMTP_SERVER. Also fill in the “user” part with the prefix of the sender of the emails.
- Install the EVENT_TUTORIAL application and start it. Upon starting the application will do a minimal configuration where possible (see perl:init for details).
- In the EVENT service configuration, enable the SERVER_DOWN channel.
- For each email address add a subscriber to the SERVER_DOWN channel using the email address as the name of the subscriber.
- In the EVENT_TUTORIAL application, update the “server_survey” task and modify the SERVER parameter of the task’s procedure so that it is set to <TARGET_SERVER>. For example, in the case the server to be watched is 192.168.1.5, the task’s procedure should be:

```
perl:Survey/survey_task[CHANNEL='SERVER_DOWN' SERVER='192.168.1.5']
```

- Start the survey task
- In the EVENT_TUTORIAL application, start the “server_down_processor” listener.

Reference

The “Survey” module of the EVENT_TUTORIAL application contains the code for this example. It is composed of 4 files described below.

Files

Survey/confirm_clear.xsl

This file generates the HTML confirmation obtained whenever a user clicks on the “Clear message” link in the alert email. (This file is used with the full version of the example packaged in the EVENT_TUTORIAL application, not the example code given in this document).

Survey/confirm_inhibit.xsl

This file generates the HTML confirmation obtained whenever a user clicks on the “Inhibit message” link in the alert email. (This file is used with the full version of the example packaged in the EVENT_TUTORIAL application, not the example code given in this document).

Procs

---

init.pl

To work correctly, this tutorial requires some configuration. To make this easier a default configuration is set up by the perl:init procedure called when the application starts. This procedure does the following for this example:

- Create a SERVER_DOWN channel where messages indicating that a server is down will be posted
- Create a listener to process the actions received on the SERVER_DOWN channel
- Create an example scheduler task checking regularly the availability of a given server. This task requires further configuration since it expects that the called procedure be passed a SERVER parameter. The task can be duplicated for each server to be watch.
- Setup the required security for the nvdef user (used by the listener) to give him the right to send emails.

Survey/mail_action.pl

This file contains Perl code implementing the listener taking the necessary action. In our example, each action is one email to send to one of the subscribers. We retrieve them one by one and send emails individually. The code for this procedure is recalled below (for readability, some parts of the code have been simplified compared to the complete version of the application).

```
# Action listener procedure for the EVENT tutorial
#
# PARAMETERS:
#   - CHANNEL : Channel to listen to
#   - FROM : Email to use as the from part of the email

NV::GetParameter("CHANNEL");
my $channel = $NV::RESULT;
my $from = "admin@example.com";
my $server = "localhost:1081";

while(true) {
  # Cleanup the container
  NV::Command("NV_CMD=|CONTAINER:REMOVE|");

  # Quit if session is terminating
  NV::Command("NV_CMD=|SESSION:CHECK_CLOSE_REQUEST|");
  if($NV::RESULT eq "YES") {
          last;
  }

  # Ask for an action to process
  NV::Command("NV_CMD=|EVENT:ACTION:GET| CHANNEL=|$channel|");

  # Quit if no action is to be processed
  NV::Command("NV_CMD=|OBJECT:EXIST| NAME=|ACTION|");
  if($NV::RESULT eq "NO") {
          last;
  }

  NV::Command("NV_CMD=|DEBUG:DISPLAY_OBJECT| NAME=|ACTION|");

  # Get the information about the action as variables
  NV::Command("NV_CMD=|OBJECT:TABLE_GET_ROW| NAME=|ACTION| ROW=|1|");
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|SUBSCRIBER|");
  my $subscriber = $NV::RESULT;qc
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|MESSAGE|");
  my $message = $NV::RESULT;
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|CONTENT|");
  my $content = $NV::RESULT;

  my $body = "Server $message is down !!”;
  # Send the email
  NV::Command("NV_CMD=|MAIL:SEND| AUTH=|NO| SUBJECT=|[EVENT Tutorial ALERT] Server
$message is down (|+|#OCCURRENCE|+|)| FROM=|$from| TO=|$subscriber| BODY=|$body|");

  # Mark the action as finished
  NV::Command("NV_CMD=|EVENT:ACTION:COMPLETE| CHANNEL=|$channel| MESSAGE=|#MESSAGE|
OCCURRENCE=|#OCCURRENCE| SUBSCRIBER=|#SUBSCRIBER|");
}
```

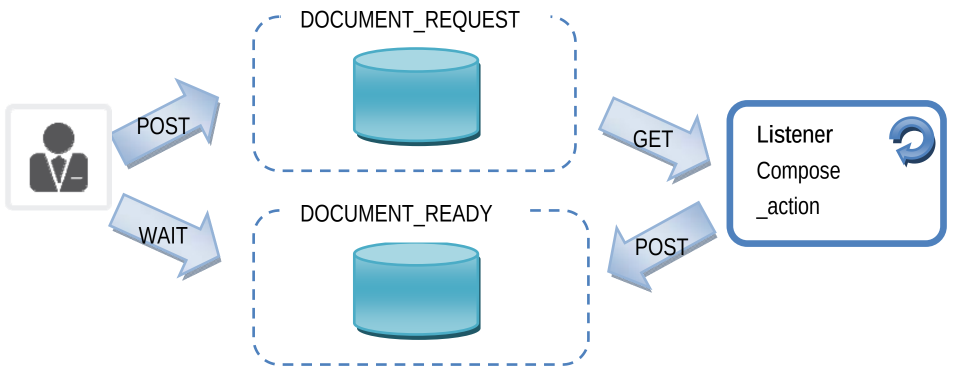
Survey/survey_task.pl

This file contains Perl code implementing the task checking whether a given server is up and running. If this is not the case, it posts and alert message to the specified channel. If the server responds, it clears any eventual previous warnings.

```
# Example survey task for the EVENT tutorial
#
# PARAMETERS:
#   - CHANNEL : Channel to listen to
#   - SERVER  : Address of the Nirva server to watch

NV::GetParameter("SERVER");
my $server = $NV::RESULT;

NV::GetParameter("CHANNEL");
my $channel = $NV::RESULT;


NV::SetErrorMode("SCRIPT");
my $ok = (NV::Command("NV_CMD=|REQUEST:OPEN| NAME=|distant| SERVER=|$server|
APPLICATION=|nvdef| USER=|nvdef|") == 1);
if(!$ok) {
  NV::Command("NV_CMD=|EVENT:MESSAGE:POST| CHANNEL=|$channel| MESSAGE=|$server|
CONTENT=|Can't connect|");
} else {
  NV::Command("NV_CMD=|EVENT:MESSAGE:CLEAR| CHANNEL=|$channel| MESSAGE=|$server|");
}
NV::Command("NV_CMD=|REQUEST:CLOSE| NV_REQUEST=|distant|");
NV::SetErrorMode("Nirva");
```

Making a user interface wait on an asynchronous process

This example shows how to use the EVENT service in a situation where a user interface requires waiting for a backend asynchronous process to terminate. In this example the user will submit a form which will generate XML data to be composed into a PDF file. The schema below illustrates this situation:



This example is composed of the following components:

- The user via an HTML form sending the a new document generation request
- An synchronous Nirva procedure which will do the following
  - Generate a job id for the request (MISC:UNIQ)
  - Saving the data it to the storage as a Nirva container serialized in XML
  - Posting an message “<job_id>” to the “DOCUMENT_REQUEST” channel with the storage reference as the content of the message
  - Wait for a message “<job_id>” event on the “DOCUMENT_READY” channel which will contain the storage reference of the PDF file.
  - Retrieve the PDF file
- A Nirva listener which processes any pending actions from the DOCUMENT_REQUESTS by:
  - Retrieving the data from the storage reference
  - Composing the PDF using the data
  - Saving the PDF document to the storage
  - Posting an event “<job_id>” to the “DOCUMENT_PROCESSED” channel

Configuration

Below are the steps to make the tutorial work:

- Install the EVENT service and enable it.
- Install the EVENT_TUTORIAL application and start it. Upon starting the application will do a minimal configuration where possible (see perl:init for details).
- Install the XSLFO service and enable it. It is used to compose an example document.
- In the EVENT service configuration, add as many subscribers to the DOCUMENTS_READY as those which may appear as submitters in the tutorials submission screen. (A default “Example” submitter will have been added by the applications “perl:init” procedure.
- In the EVENT service configuration, enable the DOCUMENT_REQUEST and DOCUMENT_READY channels.

- Enable the “composition” listener

Reference

Files

Compose/form.xsl

This XSL file generates a simple HTML form allowing the user to give the composition data. The form calls the perl:Compose/post_form procedure and retrieves a FILE object named PDF with the composed document to be displayed.

Compose/make_xslfo.xsl

This XSL stylesheet is an example document model in XSL which transforms the user data into an XSL-FO document which will be rendered to PDF by the Nirva XSLFO service.

Procs

---

init.pl

To work correctly, this tutorial example requires some configuration. To make this easier a default configuration is set up by the perl:init procedure called when the application starts. This procedure does the following for this example:

- Create the DOCUMENT_REQUEST and DOCUMENT_READY channels
- Create a listener to process the actions received on the DOCUMENT_REQUEST channel
- Create a STORAGE volume and level to store the XML data and composed PDF files.
- Setup the required security for the nvdef user (used by the listener) to give him the right to send access the storage.

Compose/post_form.pl

This procedure takes care of the processing on the frontend-side. This consists in:

- Generating a new unique document identifier.
- Retrieving the data submitted by the user
- Saving the data as an XML file to Nirva's STORAGE
- Posting a message to the DOCUMENT_REQUEST channel (using the document's id as the message id and the storage reference as its content)
- Waiting for an action on the DOCUMENT_READY channel indicating that the document has been composed
- Sending the document back to the user.

Below is given example code for this procedure is:

```
#
# Submit a composition request and retrieve the document composed asynchronously
#

# Create a document id
NV::Command("NV_CMD=|MISC:UNIQ|");
my $docid = $NV::RESULT;

# Create an XML of data and save it to the storage
NV::Command("NV_CMD=|OBJECT:CREATE| REPLACE=|YES| NAME=|document| TYPE=|STRING|
VALUE=|$docid|");
NV::Command("NV_CMD=|OBJECT:CREATE| REPLACE=|YES| NAME=|data| TYPE=|TABLE|
COLUMNS=|requester;client_name;client_address| LINESEP=|\n| ROWSEP='|'
VALUE=|#REQUESTER|+|;|+|#CLIENT_NAME|+|;|+|#CLIENT_ADDRESS| COLSEP=|;|");
NV::Command("NV_CMD=|XML:SET_XML| XMLOBJ=|XML|");
NV::Command("NV_CMD=|STORAGE:DOCUMENT:WRITE| NAME=|EVENT_TUTORIAL| FILE=|XML|");
NV::Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|REFERENCE|");
my $xml_reference = $NV::RESULT;

# Post a document request message
NV::Command("NV_CMD=|EVENT:MESSAGE:POST| CHANNEL=|DOCUMENT_REQUEST| MESSAGE=|$docid|
CONTENT=|$xml_reference|");

# Wait for the document to be ready
NV::Command("NV_CMD=|EVENT:MESSAGE:WAIT| CHANNEL=|DOCUMENT_READY| MESSAGE=|$docid|
SUBSCRIBER=|#REQUESTER|");

# Retrieve the storage reference from the content column
NV::Command("NV_CMD=|OBJECT:TABLE_GET_CELL_LINE| NAME=|ACTION| COLNAME=|CONTENT|");
my $pdf_reference = $NV::RESULT;
NV::Command("NV_CMD=|STORAGE:DOCUMENT:READ| REF=|$pdf_reference| FLAT=|YES| FILE=|PDF|
REPLACE=|YES|");
```

Compose/compose_action.pl

This procedure takes care of the backed processing (in our example composing a document). It implements a listener which on each run processes any pending compositions as long as there is one.

This consists in:

- Fetching a pending action on the DOCUMENT_REQUEST channel.
- Retrieving the XML data from the storage for the target action
- Composing the document (in our example this consists in applying the XSL Compose/make_xslfo.xsl and calling the Nirva XSLFO service).
- Saving the composed PDF document to the storage
- Posting a DOCUMENT_READY message to the channel

Below is given an example code for this procedure is:

```
#
# Process any pending actions on the DOCUMENT_REQUEST  channel
#
while(1) {
  # Cleanup the container
  NV::Command("NV_CMD=|CONTAINER:REMOVE|");

  # Quit if session is terminating
  NV::Command("NV_CMD=|SESSION:CHECK_CLOSE_REQUEST|");
  if($NV::RESULT eq "YES") { last; }

  # Ask for an action to process (quit if none : leave them for the next call)
  NV::Command("NV_CMD=|EVENT:ACTION:GET| CHANNEL=|DOCUMENT_REQUEST|");
  NV::Command("NV_CMD=|OBJECT:EXIST| NAME=|ACTION|");
  if($NV::RESULT eq "NO") { last; }

  # Get the information about the action as variables
  NV::Command("NV_CMD=|OBJECT:TABLE_GET_ROW| NAME=|ACTION| ROW=|1|");
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|SUBSCRIBER|");
  my $subscriber = $NV::RESULT;
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|MESSAGE|");
  my $message = $NV::RESULT;
  NV::Command("NV_CMD=|VARIABLE:GET| NAME=|CONTENT|");
  my $content = $NV::RESULT;

  # Get the data form the storage (message content is a storage reference)
  NV::Command("NV_CMD=|STORAGE:DOCUMENT:READ| REF=|$content| FLAT=|YES| FILE=|DATA|
REPLACE=|YES|");

  # Compose the document
  NV::Command("NV_CMD=|XML:TRANSFORM| XMLSRC=|DATA| XMLDEST=|XSLFO|
XSL_NAME=|Compose/make_xslfo|");
  NV::Command("NV_CMD=|XSLFO:DOCUMENT:COMPOSE| XSLFO=|XSLFO| OUTPUT=|PDF|
FORMAT=|PDF|");

  # Save the file to the storage
  NV::Command("NV_CMD=|STORAGE:DOCUMENT:WRITE| NAME=|EVENT_TUTORIAL| FILE=|PDF|");
  NV::Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|REFERENCE|");
  my $pdf_reference = $NV::RESULT;

  # Post an event indicating that the document is ready
  NV::Command("NV_CMD=|EVENT:MESSAGE:POST| CHANNEL=|DOCUMENT_READY| MESSAGE=|$message|
CONTENT=|$pdf_reference|");
}
```

Installation

The EVENT service is delivered as a NIRVA package and can be installed like any NIRVA service directly from the NIRVA configuration web site. Please see the NIRVA configuration chapter in the NIRVA user's guide for further information.

Configuration

The EVENT service configuration allows to:

- List the available channels
- Define new the channels
- Enable/disable the channels
- Delete channels
- List the active messages of a channel
- Clear an active message
- List the subscribers of a channel

Reference

This chapter gives the complete reference of all the EVENT service commands.

Classes

Here are the available EVENT service classes:

Class	Description
EVENT	Main class
CHANNEL	Class concerning EVENT channels
MESSAGE	Class concerning messages
ACTION	Class concerning actions to be taken

Error codes

EVENT Class

Value	Description
400	Bad or missing parameter
401	Command not allowed from this source
402	Maximum number of sessions reached
404	Unknown command

CHANNEL Class

Value	Description
400	Command is not valid for the channel's mode

Value	Description
404	Channel doesn’t exist
409	Channel already exists
410	Channel is disabled
500	Error accessing the channel
503	The channel is not available

SUBSCRIBER Class

Value	Description
404	Subscriber doesn’t exist
409	Subscriber already subscribed to this channel

MESSAGE Class

Value	Description
408	Timeout occurred when waiting for message

Permissions

Value	Description
EVENT_ADMIN	Permission required to modify service parameters
CHANNEL_SUBSCRIBE	Permission required to subscribing and unsubscribing to channels
CHANNEL_ADMIN	Permission required to create or delete channels
CHANNEL_LIST	Permission required to list the available channels
MESSAGE_POST	Permission required to post and clear messages
MESSAGE_INHIBIT	Permission required to inhibit messages
MESSAGE_WAIT	Permission required to wait for messages
MESSAGE_DISPATCH	Permission required to get and dispatch messages
MESSAGE_LIST	Permission required to list messages
ACTION_PROCESS	Permission required to get and complete actions

Commands

For each command, the reference gives the command name, the sources for which the command may be used, the command description, the eventual command permissions, the parameter list and the eventual list of objects created by the command.

The parameters described in this chapter are command specific parameters. For general parameters, please refer to the Nirva command syntax chapter in the Nirva Application Platform documentation.

The available sources are:

- Client for all Nirva client interfaces including Nirva client library (nvc).
- Web for commands from a web browser.
- Procedure for commands from a Nirva procedure.
- Service for commands from service to service

EVENT class

This is the standard service class that provides service scope commands.

NOP

EVENT:EVENT:NOP

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command does nothing but allows to test that the event service is on line and answers correctly.

If the service is not on line, this command returns an error.

Parameters

None

Permissions

None

GET_CHANNEL_DIRECTORY

EVENT:EVENT: GET_CHANNEL_DIRECTORY

Source	Use Input Container	Use Output Container
Client Procedure Service	No	No

Description

This command allows getting the channel directory path.

Permissions

EVENT_ADMIN

Output buffer

Name of the channel directory path

SET_CHANNEL_DIRECTORY

EVENT:EVENT: SET_CHANNEL_DIRECTORY

Source	Use Input Container	Use Output Container
Client Procedure Service	No	No

Description

This command sets the channel directory path. When changing this setting the EVENT service all channels should have been disables. The EVENT service leaves the channel data in the previous directory. If it is required that this data bee kept, it should be moved to the new location. Otherwise, any pending messages and actions will be lost.

Permissions

EVENT_ADMIN

Parameters

DIRECTORY	Name of the channel directory path

CHANNEL class

CREATE

EVENT:CHANNEL:CREATE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command creates a new channel.

Permissions

CHANNEL_ADMIN

Parameters

CHANNEL	Name of the channel to create
DESCRIPTION	Human readable description
MODE	Dispatch mode which will be used by this channel. The value may be one of: ▪ IMMEDIATE ▪ DIFFERED ▪ EXTERNAL ▪ UNLOCK

UPDATE

EVENT:CHANNEL:UPDATE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command updates an existing channel.

Permissions

CHANNEL_ADMIN

Parameters

CHANNEL	Name of the channel to update
DESCRIPTION	Human readable description
MODE	Dispatch mode which will be used by this channel. The value may be one of: ▪ IMMEDIATE ▪ DIFFERED ▪ EXTERNAL ▪ UNLOCK

REMOVE

EVENT:CHANNEL:REMOVE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command removes channel.

Permissions

CHANNEL_ADMIN

Parameters

CHANNEL	Name of the channel to remove

ENABLE

EVENT:CHANNEL:ENABLE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command enables the specified channel.

Permissions

CHANNEL_ADMIN

Parameters

CHANNEL	Name of the channel to enable

DISABLE

EVENT:CHANNEL:DISABLE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command disables the specified channel.

Permissions

CHANNEL_ADMIN

Parameters

CHANNEL	Name of the channel to disable

LIST

EVENT:CHANNEL:LIST

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command lists all the available channels.

Permissions

CHANNEL_LIST

Parameters

None

Output

CHANNELS A table containing the following columns:

- Name : Name of the channel
- Description : Description of the channel
- Mode : Dispatching mode of the channel
- Enabled : Whether the channel is enabled or not

SUBSCRIBE

EVENT:CHANNEL:SUBSCRIBE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command adds a subscriber to the list of subscribers of the channel.

Permissions

CHANNEL_SUBSCRIBE

Parameters

CHANNEL	Name of the channel to subscribe to
SUBSCRIBER	Identifier of the subscriber to add to this channel

UNSUBSCRIBE

EVENT:CHANNEL:UNSUBSCRIBE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command removes a subscriber from the list of subscribers of the channel.

Permissions

CHANNEL_SUBSCRIBE

Parameters

CHANNEL	Name of the channel to unsubscribe from
SUBSCRIBER	Identifier of the subscriber to remove from this channel

---

LIST_SUBCRIBERS

EVENT:CHANNEL:LIST_SUBSCRIBERS

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	Yes

Description

This command lists all subscribers of this channel.

Permissions

CHANNEL_LIST

Parameters

CHANNEL	Name of the channel for which to list the subscribers

Output

SUBSCRIBERS	A table containing the following columns: ■ NAME : Name of the subscriber

MESSAGE class

POST

EVENT:MESSAGE:POST

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command receives a new message for the given channel. How it is processed depends on the dispatch mode of the channel.

If the dispatch mode is “EXTERNAL” or “DIFFERED” the message is stored to the message queue. No further processing is done. In “DIFFERED” mode the processing of messages will be done when the action queue is empty and an ACTION:GET request cannot be satisfied. In “EXTERNAL” mode, it is left up to and external entity to call the “MESSAGE:GET” method to retrieve a set of messages to be processed in a file and call the “MESSAGE:DISPATCH” for each message found.

When processing an active message, the occurrence of the message is increased. Otherwise a new message is created with the given id. In both cases a pending action is generated for each subscriber of the channel. These can then be processed asynchronously and eventually distributed via single calls to the EVENT:ACTION:GET command.

Permissions

MESSAGE_POST

Parameters

CHANNEL	Name of the channel the message should be posted to
MESSAGE	This parameter contains the identifier of the message. This identifier must be the same for every occurrence of the same message.
CONTENT	The content of the message.
AUTO_CLEAR	Automatically clear the messages once all the actions generated have been processed. Default is “YES”. When this option is set to “NO” the message will be kept and the message’s occurrence counters incremented each time a new post arrives. This until MESSAGE:CLEAR command is called.

WAIT

EVENT:MESSAGE:WAIT

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	Yes

Description

This command allows for event-based synchronization. It blocks the caller until a publisher posts a message with the specified id.

Permissions

MESSAGE_WAIT

Parameters

CHANNEL	Name of the channel the expected message will be posted to
MESSAGE	This parameter contains the identifier of the expected message.
SUBSCRIBER	Name of the subscriber we are waiting for.
TIMEOUT	Time to wait for the message before a timeout occurs (default is 0, i.e. wait forever).
AUTOCOMPLETE	This parameter determines whether the action should be automatically marked as completed once the expected message has been received.

Output

ACTION A TABLE object is returned with the following columns:

- MESSAGE: Identifier of the message to be processed
- OCCURRENCE: Occurrence of the message
- CONTENT: Content of the message
- SUBSCRIBER : Subscriber of the message
- DATE: Date the message was posted (in the format YYYY-MM-DD HH:MM:SS)
- DELAY: Delay in seconds between this occurrence of the message and the previous occurrence. 0 if this is the first occurrence.

INHIBIT

EVENT:MESSAGE:INHIBIT

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command inhibits the future generation of actions for the specified message and subscriber, by marking the message as inhibited for the specified subscriber. If a message is marked as “inhibited” for a given subscriber, when the message is dispatched into actions, no new action for the given subscriber will be generated. A message stays inhibited as long as the message has not been “cleared”. Therefore, this command is mostly useful in cases where the messages are not cleared automatically.

Permissions

MESSAGE_INHIBIT

Parameters

CHANNEL	Name of the channel the message should be posted to
MESSAGE	This parameter contains the identifier of the message. This identifier should the same among all occurrences of the same message.
SUBSCRIBER	Subscriber wishing to inhibit the message

CLEAR

EVENT:MESSAGE:CLEAR

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command allows marking a message as obsolete. It removes the memorized information for the message from the action queue. The service will also remove any inhibitions for the message. However, the unprocessed actions for the message will be kept.

Permissions

MESSAGE_POST

Parameters

CHANNEL	Name of the channel the message should be posted to
MESSAGE	This parameter contains the identifier of the message. This identifier should the same among all occurrences of the same message.

GET

EVENT:MESSAGE:GET

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

Messages to be processed are returned in the form of a file. Multiple message files may exist depending on the frequency they are processed and the frequency at which new posts arrive. In order to process the messages in the same order they have been received, this command returns the oldest message file available for the given channel first. If a non empty file is available it is returned in a file object named MESSAGES. If the file is empty the output will not return the MESSAGES object. In both situations any previously existing MESSAGES object is removed. This way a simple OBJECT:EXIST command allows checking the presence or absence of a message file.

A message file contains one message per line. Each line contains the message_id, followed by a ‘;’ and completed by the content of the message. The content of the original message having been escaped by replacing carriage returns (\r) by the string ‘\r’, line feeds by the string (‘\n’) and backspaces by the string (‘\\’).

Permissions

MESSAGE_DISPATCH

Parameters

CHANNEL	Name of the channel from which to retrieve messages.

Objects

MESSAGES	This file object contains the latest set of messages for the channel if there is one. No object is returned if there are no more messages to be processed.

DISPATCH

EVENT:MESSAGE:DISPATCH

Source	Use Input Container	Use Output Container
Client Web Procedure Service	Yes	No

Description

This command is only available when the channel's dispatch mode is "EXTERNAL". It takes the input messages and dispatches them to the action queue. If a message active (i.e. the action queue has a memorized reference to a previous occurrence of the message), the occurrence counter of the message is increased. Otherwise a new message is created with the given id. In both cases a pending action is generated for each subscriber of the channel. These can then be processed asynchronously and eventually distributed via single calls to the EVENT:ACTION:GET command

Permissions

MESSAGE_DISPATCH

Parameters

CHANNEL	Name of the channel the message should be posted to
MESSAGE	This parameter contains the identifier of the message to be dispatched in "STRING" mode.
CONTENT	The content of the message to be dispatched in "STRING" mode.
AUTO_CLEAR	Automatically clear the messages once all the actions generated have been processed. Default is "YES". When this option is set to "NO" the message will be kept and the message's occurrence counters incremented each time a new post arrives. This until MESSAGE:CLEAR command is called.

MODE	Can either be “FILE” or “STRING”. In “STRING” mode (the default), only one event is dispatched by using the MESSAGE and CONTENT parameters. In “FILE” mode, a file object which has the same format as the one obtained by the “MESSAGE:GET” command is expected. All the messages found in the file are dispatched.
FILE	Name of the file object containing the messages to be dispatched in “FILE” mode.

LIST

EVENT:MESSAGE:LIST

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	Yes

Description

The action queue keeps a list of the messages which have been dispatched into actions. This command allows returning this list. The list may be limited to the active messages which have not been inhibited by a specified subscriber.

Permissions

MESSAGE_LIST

Parameters

CHANNEL	Name of the channel the message should be posted to (mandatory)
SUBSCRIBER	This optional parameter allows limiting the list to the active messages of the given subscriber. Otherwise said, specifying a subscriber does not return the active messages he has inhibited from the returned list.

Objects

MESSAGES

This table object contains the list of active messages with the following columns:

- MESSAGE : Identifier of the message
- CONTENT : Content of the message
- DATE : Date of the last occurrences

- DELAY : Delay between the last occurrence
- OCCURRENCES : Number of times the messages has occurred.

ACTION class

GET

EVENT:ACTION:GET

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	YES

Description

This command allows retrieving an action to be processed from the action queue. It allows extracting any available action, any action for a given subscriber, any action for a given message or any action for a given subscriber and message identifier. It is highly recommended that the way messages are extracted from a channel be globally consistent (e.g. it should be fixed that all the actions for channel “X” are to be processed by subscriber).

If any actions to be processed are available, it returns a table object containing one row with the information about the action in the different columns. If no action is available, then the ACTION object is not created (if any action object existed it will have been removed).

Permissions

ACTION_PROCESS

Parameters

CHANNEL	Name of the channel to be processed.
SUBSCRIBER	Optional target subscriber of the actions to retrieve
MESSAGE	Optional message id of the actions to retrieve
MAX_COUNT	Maximum number (between 1 and 1000) of actions to be retrieved in one call (the default is to return 1 action per call)
AUTO_COMPLETE	Option indicating that the action should be auto-completed (value of “YES”) or not (value of “NO”). The default is “YES”. Auto-completing is equivalent to calling ACTION:COMPLETE just after the ACTION:GET command.

Output

ACTION	A TABLE object is returned with the following columns:

- MESSAGE: Identifier of the message to be processed
- OCCURRENCE: Occurrence of this message
- CONTENT: Content of the message
- SUBSCRIBER: Subscriber the message should be sent to.
- DATE: Date the message was received (in the format YYYY-MM-DD HH:MM:SS)
- DELAY: Delay in seconds between this occurrence of the message and the previous occurrence. 0 if this is the first occurrence.

COMPLETE

EVENT:ACTION:COMPLETE

Source	Use Input Container	Use Output Container
Client Web Procedure Service	No	No

Description

This command indicates that the given message occurrence was processed correctly for the given subscriber.

Permissions

ACTION_PROCESS

Parameters

CHANNEL	Name of the channel the action processed belongs to.
MESSAGE	Identifier of the message the action was taken for.
OCCURRENCE	The occurrence count of action.
SUBSCRIBER	The target subscriber of the action