Unit 7. Triggers

Estimated time

01:00

Overview

Triggers are an automated way to alter data, including event data, in the central Event Manager database. This unit teaches you how to create and use triggers.

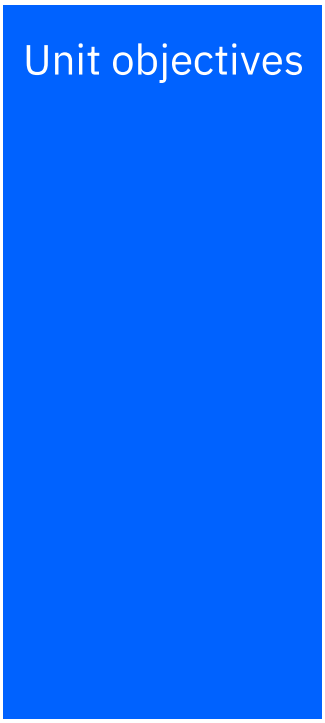


Figure 7-1. Unit objectives

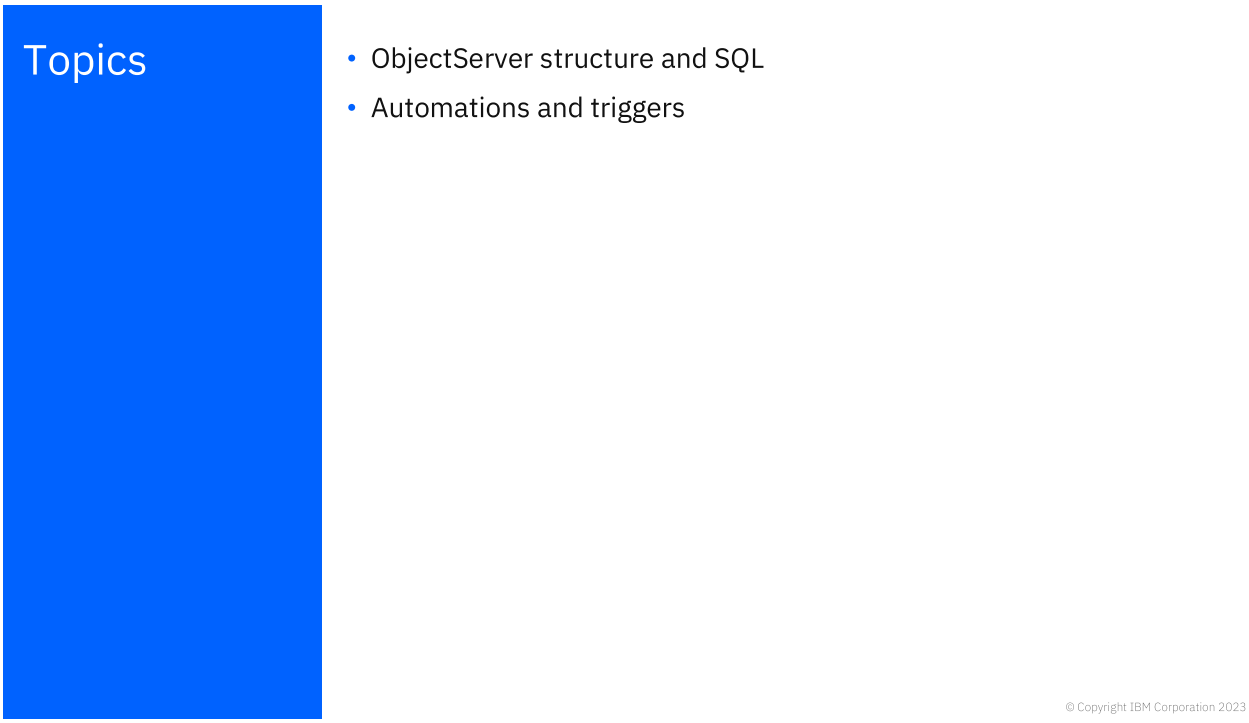
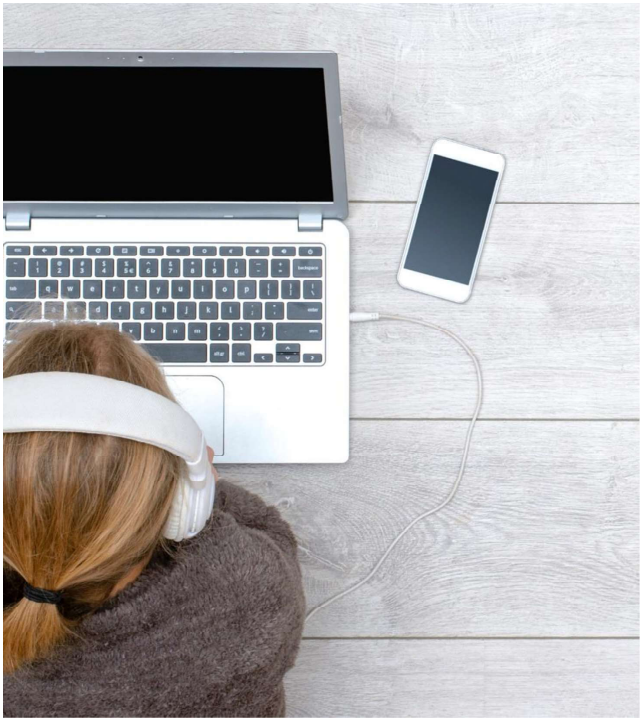


Figure 7-2. Topics

7.1. ObjectServer structure and SQL



ObjectServer structure and SQL

Figure 7-3. ObjectServer structure and SQL

What is an ObjectServer?

- An ObjectServer is the central database where Event Manager stores events
- An ObjectServer is a collection of multiple databases
- ObjectServers:
  - Receive event data from probes, webhooks, and other and monitors
  - Process event data using tools and automations
  - Transfers event data using gateways
  - Displays event data to the user
- ObjectServers come in pairs for redundancy: a primary ObjectServer and a backup ObjectServer
- ObjectServers can be deployed in tiers to process high volumes of events, for example a collection tier to receive events along with an aggregation tier to process events

Figure 7-4. What is an ObjectServer?

ObjectServers are the central data stores within Event Manager, including the database where all events are stored.

ObjectServer databases

Initially, the ObjectServer has the following databases, among others:

- **alerts**: Alert data, and event list configuration
- catalog: System catalog containing Object Server metadata (can be viewed but not modified)
- custom: Database for tables added by users
- iduc_system: Channel setup for accelerated event notification (AEN)
- master: Compatibility with previous releases; Desktop ObjectServer tables
- persist: Triggers, procedures and signals
- security: Authentication information for users, roles, groups, permissions
- service: service.status table for service display (used mostly with monitors)
- tools: User tools and menu structure
- transfer: Used by the ObjectServer gateways

Figure 7-5. ObjectServer databases

The ObjectServer is typically referred to as a memory resident database. In reality, it is a collection of multiple databases. This slide contains a list of key databases that are defined in any ObjectServer. A user with the appropriate authority can add a database to the ObjectServer. Users can also add their own databases, database tables, and columns in a database table.

The ObjectServer is case-sensitive. All of the default databases are defined with names that contain all lowercase letters. Any use of the actual database name, in an SQL statement, for example, must reference the name in lowercase.

One important table is named alerts.status, which is used to store normalized event records. The `alerts.status` table defines the structure of the event records.

ObjectServer structure: `alerts.status`

Key fields in the alerts.status table:

- Identifier
- Serial
- ServerSerial
- ServerName
- StateChange
- FirstOccurrence
- LastOccurrence
- Type
- Acknowledged
- Node
- Tally
- ExpireTime
- Severity
- OwnerUID/GID
- AlertGroup
- AlertKey
- Manager
- Summary
- Class

Triggers

Figure 7-6. ObjectServer structure: alerts.status

The ObjectServer table `alerts.status` defines the structure of the Event Manager event record. Event Manager probes, webhooks, and other monitors populate the `alerts.status` table.

In the example of a probe, the probe collects information from some source, and breaks the information into pieces referred to as tokens. The probe assigns a token to a column in the `alerts.status` table. The probe populates the table by creating an `SQL INSERT` statement. That statement is forwarded to the ObjectServer, and causes a record to be created in the `alerts.status` table.

This slide lists some of the key fields in the alerts.status table:

- **Identifier**: This is the unique identifier for the alerts.status database and is key to event deduplication. It is essential that the identifier identifies repeated events appropriately.
- **Serial**: This is an automatically populated field and is a unique reference for an event within a particular ObjectServer. Event Manager automatically assigns a number to this field when a new event arrives in the ObjectServer.
- **ServerSerial** and **ServerName**: Unique values for the server that received the event first. This is important for architectures with multiple ObjectServers and gateways.
- **StateChange**: This is a time field updated by triggers. Event Manager updates this field with the current time each time the state of an event changes (either from the data source or the ObjectServer).
- **FirstOccurrence**: This is a time field that is updated by triggers. It contains a timestamp of when the event first arrives in the ObjectServer. It should not subsequently be changed.
- **LastOccurrence**: This is a time field updated by triggers. It contains a timestamp of the last occurrence of the event. Unlike FirstOccurrence, its value changes if another instance of the same event arrives.

- **Type**: An integer field and can generally take three values: 0, 1 and 2. A type of 0 means that the type has not been set. A type of 1 means that the event is a problem event (link down, for example). A type of 2 means that the event is a resolution event (link up).
- **Acknowledged**: Events in Event Manager can be acknowledged by a user. It is this field that represents the acknowledgement of an event. It is an integer field but behaves as a Boolean: 0=unacknowledged and 1=acknowledged.
- **Node**: Identifies the managed entity from which the alarm originated. This could be a device or host name, service name, application name, or other entity. The Node column must contain the name of the entity which allows direct communication, or can be resolved to allow direct communication, with the entity.
- **Tally**: The ObjectServer maintains a count (or tally) of the total number of recurrences of an event.
- **ExpireTime**: An integer field that can optionally be used to automatically remove events. When set to a non-zero value, it represents the number of seconds that the event remains in the ObjectServer. A trigger checks any event with a non-zero value. After the event has been in the system for longer than the configured number of seconds, it is removed.
- **Severity**: This field denotes the severity or priority of the event within the ObjectServer. It is an integer field and has these values by default:
  - 0: Clear
  - 1: Intermediate
  - 2: Warning
  - 3: Minor
  - 4: Major
  - 5: Critical
- **OwnerUID**: The owner ID of the event in alerts.status table.
- **OwnerGID**: The group ID of the event in alerts.status table.
- **AlertGroup**: The descriptive name of the failure type indicated by the alert. For example: `Interface Status` or `CPU Utilization`.
- **AlertKey**: Indicates the managed object instance referenced by the alert. For example, the disk partition indicated by a file system full alert or the switch port indicated by a utilization alert.
- **Manager**: Normally denotes the data source that processed the event.
- **Summary**: A summary of the problem associated with the event.
- **Class**: A way of classifying equipment types in events. Enables tools to be assigned against events of specific equipment types.

ObjectServer structure: `alerts.details`

Under certain circumstances you might be interested in the raw data from the probe:

- When details tracking is enabled, data is stored in the alerts.details table as token-value pairs
- You can view details from the **Details** tab in the Information window, which you access through the **Alerts** menu for selected events
- Details are linked to their respective events using the **Identifier** field, which is a primary key in both the alerts.status and alerts.details tables
- There is a 1-to-1 correspondence between events and details: one detail record (alerts.details) possible for each event record (alerts.status)

Figure 7-7. ObjectServer structure: alerts.details

One of the primary roles of a data source such as a probe is to convert machine information from the event source into human-readable text in the ObjectServer event record. There are times when it might be important to see the original machine data, typically when debugging some problem. You can configure the data source to send extra data to the ObjectServer whenever an event is created. This additional data is saved in the alerts.details table. There is a database link between alerts.status and alerts.details, which enables a user to view the details that are related to a specific event.

ObjectServer structure: alerts.journal

When working with events, you might want to track the history of a particular event:

- Who has owned it
- What severity levels it has passed through
- What automations have acted on it

Journals provide these functions:

- Journal information is held in the alerts.journal table
- Journals are linked to their respective events through the Serial field, which is a primary key in both alerts.status and alerts.journal tables

There is a 1-to-*n* correspondence between events and journals

- There can be *n* journal records (alerts.journal) for each event record (alerts.status)

Triggers

Figure 7-8. ObjectServer structure: alerts.journal

When working with events in Event Manager, you often want to track the history of an event. You want to know who owns the event, what severity levels it passes through, what automations act upon it, and more. You can use the journal to track the history. When a new journal entry is added, the data is stored in the alerts.journal table.

The journal entry contains the name of the user, the date, time, and text that describes the operation. This information provides an important chronological history of actions that are taken against the event.

You can add entries to the journal manually. You can also add to the journal with a trigger.

Introduction to ObjectServer SQL

- Subset of American National Standards Institute (ANSI) SQL
  - Includes some proprietary extensions
- Used throughout Event Manager in automations, tools, and filters
- Three general functional areas:
  - Data definition: ObjectServer structure and behaviour
  - Data manipulation: Triggers, tools
  - System administration: Command line
- This unit focuses on *data manipulation*
- ObjectServer data is case sensitive

Triggers

Figure 7-9. Introduction to ObjectServer SQL

ObjectServer SQL is a subset of ANSI SQL, which is used widely throughout Event Manager. For example, it is used in the SQL file that creates the ObjectServer, and is used in automations in the ObjectServer. ObjectServer SQL commands can be roughly divided into three functional areas:

- **Data Definition**: Used in SQL files to define and create databases and tables
- **Data Manipulation**: Used by triggers, tools, and filters to retrieve, modify, and delete data
- **System Administration**: Used on the command line to manage the system

The data ObjectServer is case sensitive, including database names, table names, and column names.

This unit covers data manipulation commands, which are useful for ObjectServer triggers.

Command-line access

- The **nco_sql** command provides access to the ObjectServer's SQL interface
  `$OMNIHOME/bin/nco_sql -server <ObjectServer Name>`
  - Requires ObjectServer user name and password
  - If not specified, root is the assumed user
  - The nco_sql command is available inside of ObjectServer containers and on computers where ObjectServers run

Figure 7-10. Command-line access

Use the nco_sql tool to access the command-line SQL interface of an ObjectServer.

```
nco_sql –server <OBJECTSERVER_NAME> –user <NAME> –password <password>
```

If no user name is specified, the system assumes root.

Viewing data with SELECT

Selecting events example:

```
SELECT * FROM alerts.status WHERE Severity = 5;
```

`SELECT *`	retrieve all columns
`FROM alerts.status`	from the alerts database and the status table
`WHERE Severity = 5`	that meet this condition

Triggers

Figure 7-11. Viewing data with SELECT

Use the SELECT command to retrieve one or more rows or partial rows of data from an existing table.

You can select an event with the following command:

```
select * from database.table where FieldName=condition;
```

Select all non-hidden columns:

```
select *
```

From the database and table specified:

```
from database.table
```

Using the following condition:

```
where FieldName=condition
```

In this example, the command retrieves all columns from the alerts.status table and displays every record with a Severity of critical.

```
select * from alerts.status where Severity=5;
```

Fields and operators

Selecting specific fields with multiple conditions

```
SELECT Summary, Class FROM alerts.status
WHERE Severity = 5 AND Node = 'batman';
```

Inequality comparisons

```
SELECT * FROM alerts.status WHERE Grade >= 3;
```

LIKE operator for string comparisons

```
SELECT * from alerts.status
   WHERE Node LIKE '^bat.*';
```

LIKE is only used with char fields, often with regular expressions

Figure 7-12. Fields and operators

ObjectServer SQL supports the ability to retrieve specific fields or columns. The following example only retrieves the Summary and Class fields.

```
select Summary, Class from alerts.status
```

ObjectServer SQL supports logical operators:

```
AND
OR
```

ObjectServer SQL supports comparison operators.

> greater than[or equal to] >=

< less than[or equal to] <=

<> Not equal to

LIKE and NOT LIKE are typically used in string comparisons and often in conjunction with regular expressions. The following metacharacters are the most commonly used:

. Match any single character (for example, link.n matches link2n, not link21n)

***** Match none or more of the previous characters (for example, link* matches lin, link, or linkkk)

+ Match one or more of the previous characters (for example, link+ matches link, linkkk but not lin or linxk)

[ ] Match any single character within the given range (for example, link[0-5] matches link2 but not link9)

^ Ensure that the pattern matches at the beginning of the string (for example, ^link.* matches linknorth, but not northlink)

**$** Ensure that the pattern matches at the end of the string (for example, .*link$ matches northlink but not linknorth)

In regular expressions, a backslash (\) escapes special characters (match literal value).

The NOT keyword inverts the result of any comparison.

IN operator and subqueries

Use IN to compare a value to a list of values

```
SELECT * FROM alerts.status WHERE Node IN ('batman', 'catwoman', 'robin');
```

Can also use IN with subqueries

```
SELECT * FROM alerts.status WHERE Serial IN (( SELECT Serial FROM
  alerts.journal ));
```

The example retrieves all events with a journal entry

Figure 7-13. IN operator and subqueries

The IN list comparison operator compares a value to a list of values.

Example:

```
select * from alerts.status where Severity IN (1,3,5)
```

The query returns the rows in which Severity is equal to the number 1, 3 or 5.

Subqueries are complex conditions that match a value with the results of another select statement, possibly on another database.

Modifying data with UPDATE and SET

The UPDATE command changes table data

- Single field

```
UPDATE alerts.status SET Severity = 5
WHERE Severity = 4 AND Acknowledged = 0;
```

- Multiple fields

```
UPDATE alerts.status
SET Severity = 5, Service = 'Web Host‘
WHERE Grade = 4 AND Customer like 'ISP';
```


Figure 7-14. Modifying data with UPDATE and SET

The UPDATE command updates columns in an existing row of data in a table.

Update statement updates a database.table set assignment where condition.

Update the records:

```
update
```

Set the following assignment:

```
set assignment
```

Where the following condition is true:

```
where condition
```

In this case, the asterisk or field name is not required. The statement updates the table that is defined, using the set assignment, based on the where condition.

For example, this statement first locates any record in the alerts.status table with a Severity of 3. It changes the Severity to 4 for every record found.

```
update alerts.status set Severity=4 where Severity=3;
```

Creating data with INSERT

The INSERT INTO command adds a row to a table

```
INSERT INTO alerts.status (Node, Severity, Summary, Identifier)
  VALUES ('was1', 5, 'Disk Util 95%', 'was1DiskUtil95')
```

Removing data using DELETE

The `DELETE` command removes one or more rows from an existing table

```
DELETE FROM alerts.status WHERE Severity=0;
```

Deleted events cannot be recovered

Figure 7-15. Creating data with INSERT

The INSERT command creates a new row of data in an existing table. If you are not inserting values for every column in the row, you can specify a comma-separated list. This list has columns that are inserted within parentheses, followed by the VALUES keyword, followed by a comma-separated list of values within parentheses.

INSERT statement

```
insert into database.table (IntegerField, StringField, InetgerField2,
StringField2) values (3, 'text', 3, 'more text');
```

String field values are single-quoted. You must specify a value for the primary key columns in the INSERT command.

UPDATING keyword

The optional UPDATING keyword forces the specified columns to be updated if the insert is deduplicated.

Example:

```
insert into status (Identifier, Severity, Tally, Serial) values
('ConrolMachineStats15', 5, 12, 21) updating (Severity);
```

In the preceding example, a new record is inserted into the alerts.status table. The new record will have the following four fields specified.

- Identifier: ControlMachineStats15
- Severity: 5
- Tally: 12
- Serial: 21

If a record already exists in alerts.status with the same identifier, then deduplication occurs. During deduplication, only specific fields are updated, and Severity is not one of them. By including the `updating (Severity)` text, the ObjectServer is forced to update the Severity field. Without that text, the Severity field does not change.

Delete statement

This statement deletes the rows from the table using the specified condition, for example:

```
delete from database.table where condition;
```

This statement removes every record from alerts.status that is Green/Clear (Severity=0):

```
delete from alerts.status where Severity=0;
```

Note

The action takes place immediately, and the result is permanent.

The administrator tool and the SQL workbench

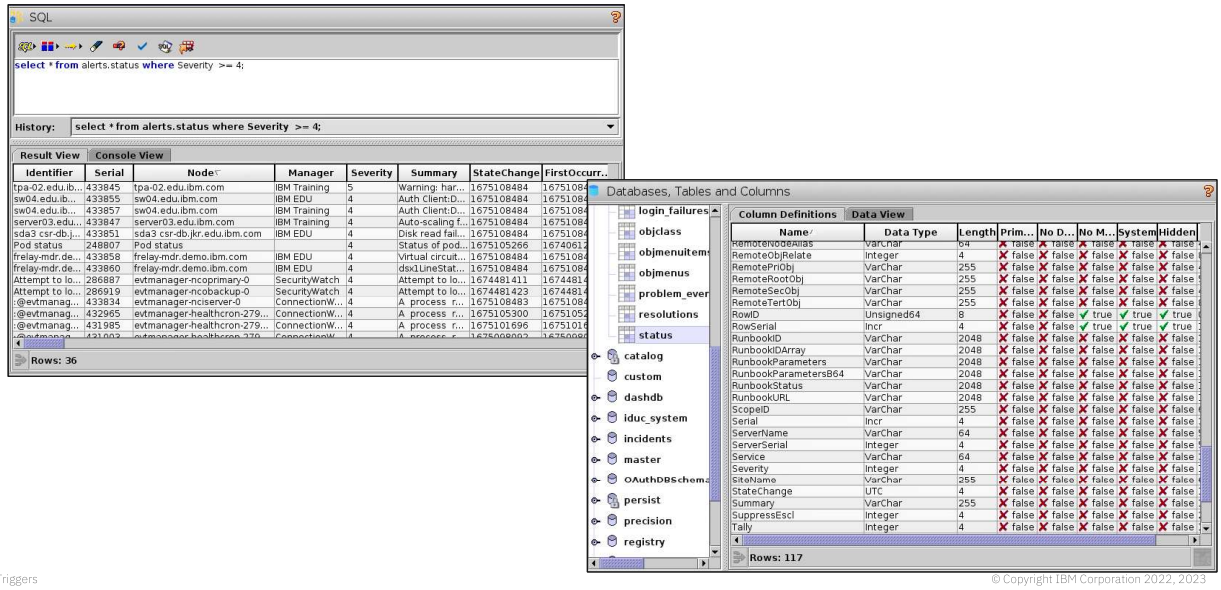


Figure 7-16. The administrator tool and the SQL workbench

The Event Manager administrator tool is a powerful configuration tool that helps you customize and manage ObjectServer databases. The administrator tool is a thick client, which you download and install with the ObjectServer on-premises software. The administrator tool connects to ObjectServers in your environment, whether your ObjectServers are on-premises or running in Red Hat OpenShift.

The administrator tool includes a tool called the SQL workbench. This tool helps you create and validate SQL commands.

This slide shows an SQL statement in the SQL workbench, along with a list of columns in the alerts.status database.

7.2. Automations and triggers

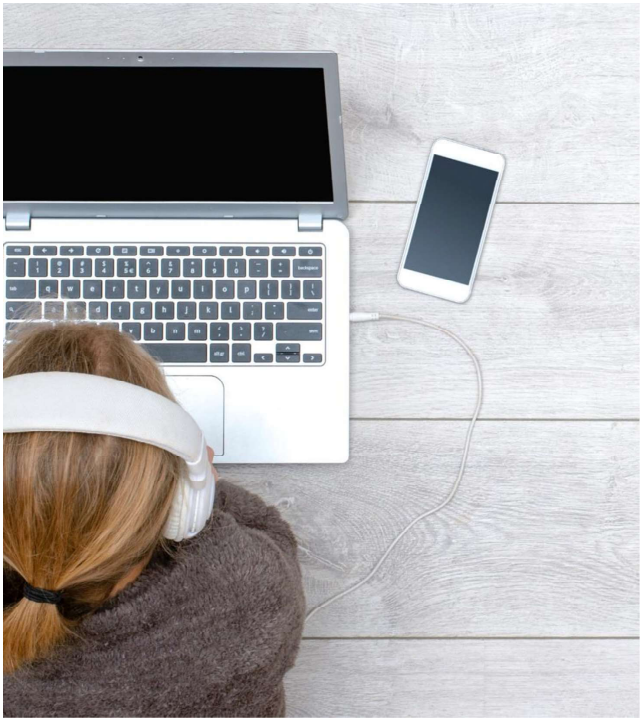


Figure 7-17. Automations and triggers

Triggers (automations)

Triggers are a way to respond to events that happen within the ObjectServer. Triggers are used for the following tasks, among others:

- Automate management of events
- Perform actions automatically on receipt of certain events
- Incorporate escalation procedures
- Correlate events

Figure 7-18. Triggers (automations)

Triggers detect changes in the ObjectServer and run automated responses to these changes. This enables the ObjectServer to process alerts without requiring an operator to take action.

Triggers are also called automations.

Trigger types

There are three types of triggers:

- Database: A database condition exists in ObjectServer
- Temporal: This trigger runs on a timed basis
- Signal: A system or user-defined signal was raised

Figure 7-19. Trigger types

There are three types of triggers.

- **Database**: A database trigger activates based on a change to a database table. The trigger defines the name of the database table, and what type of change activates the trigger.
- **Temporal**: A temporal trigger is activated based on a frequency. For example, a temporal trigger can activate once every minute.
- **Signal**: A signal trigger activates based on the occurrence of a special event, which is called a signal. The ObjectServer includes a collection of system signals. An example of a system signal is the connect signal. The connect signal is raised when a component connects to the ObjectServer. You can define a user signal. When you create a user signal, you define the conditions that are related to the signal. The ObjectServer raises system signals automatically. A user signal is typically raised manually with a tool.

Trigger general settings

- Trigger name
- Trigger group
- When: valid SQL condition
- Action, which can be one or more of these types:
  - SQL action
  - SQL procedure
- Comment: description of trigger
- Declare local variables
- State
  - Debug: Logging to message log (ObjectServer log file)
  - Enabled: Enable or disable trigger. Both the trigger and trigger group must be enabled for the trigger to fire
- Priority
  - Set trigger fire order (1 –20), 1 being highest

Figure 7-20. Trigger general settings

When you create a trigger, you must configure some settings. Some of the setting values are common across all trigger types. Some triggers do not use some settings.

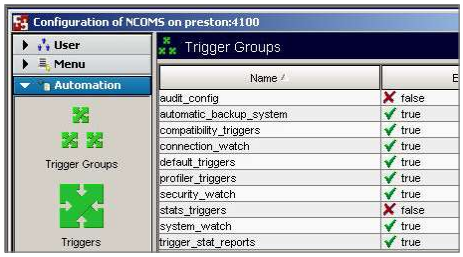
Trigger names are character strings. A trigger name cannot contain spaces or special characters except for the underscore. The trigger name must be unique.

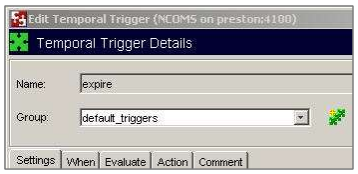
Triggers are organized into trigger groups. You must choose a group when you create a trigger.

The WHEN setting is used to configure an optional condition that must be met before a trigger can activate. The condition that is defined in the WHEN setting is in addition to the type of trigger. For example, you create a temporal trigger with a frequency of every hour. The trigger activates every hour. If you create a WHEN condition, the trigger runs every hour only if the WHEN condition is met.

The ACTION setting contains the commands that run when the trigger is activated. The commands are typically one or more SQL commands.

Trigger groups





- Use Trigger Groups to manage multiple triggers
- Each trigger must belong to only one Trigger Group**,** but can be moved between groups
- A Trigger Group can only be deleted if empty

Figure 7-21. Trigger groups

Triggers are organized into trigger groups. Trigger groups are used to organize and control one or more triggers.

A trigger group has a name, and the name has the same constraints as a trigger name. A trigger group can be used to control the activation of multiple triggers. If you disable a trigger group, you prevent the activation of all triggers that belong to that group.

WHEN clause

You can determine when the trigger fires:

- Day of week
- Time of day
- Severity at a certain level:
  `new.Severity>3`
- Deduplication period is below a certain amount:
  `(new.LastOccurrence - new.FirstOccurrence) < 60`

Figure 7-22. WHEN clause

The WHEN setting is used to define a condition that must be met before a trigger activates. A user might create a trigger to automatically delete certain events from the ObjectServer that runs once every hour. After the trigger is enabled, the trigger activates every hour, on every day of the week. The user might not want the trigger to remove events on Saturday or Sunday. The user can add a WHEN condition to the trigger to test for the day of the week. In the WHEN condition, the user specifies that the trigger does not activate on Saturday or Sunday. After the WHEN condition is added, the trigger activates once every hour on every day of the week except Saturday or Sunday.

Trigger actions

- Trigger actions are SQL code blocks
- They can be designed to manipulate data in the action statement itself
- They can run a predefined procedure

Figure 7-23. Trigger actions

The ACTION contains the commands that run when the trigger activates. The ACTION can contain a single SQL command, a block of commands, or a command that runs a procedure.

Commonly used SQL commands

A body statement can contain one or more SQL commands:

```
ALTER <ObjectServer Object>
UPDATE
INSERT
DELETE
WRITE INTO
RAISE SIGNAL{ EXECUTE | CALL } PROCEDURE
CANCEL
BREAK
IF / THEN / ELSEIF
CASE / WHEN
FOR / FOR EACH ROW
SET <TypedObject> = <TypedValue>
```


Figure 7-24. Commonly used SQL commands

This slide contains a list of some of the SQL commands that you can use within an SQL code block. These commands are commonly used in triggers.

SQL code blocks: IF THEN ELSE

The IF THEN ELSE statement performs one or more actions based on the specified conditions.

Syntax:

```
IF condition THEN
        action_command_list
        [ELSEIF condition THEN optional
                        action_command_list
...]
[ELSE optional
        action_command_list]
END IF;
```


Figure 7-25. SQL code blocks: IF THEN ELSE

ObjectServer SQL is used to implement logical controls within any code block. The most common way is by using an IF() statement.

For instance, you might want to implement the logic that if Grade is 99, do one thing; else if Grade is 98, do something else. You can use IF() blocks for this task.

SQL code blocks: FOR EACH ROW

The FOR EACH ROW loop performs actions on a set of rows that match a certain condition

Syntax:

```
FOR EACH ROW variable_name in database_name.table_name
[ WHERE condition ]
BEGIN
        action_command_list;
END;
```


Figure 7-26. SQL code blocks: FOR EACH ROW

You can use the FOR EACH ROW loop to perform actions on a set of rows that match a certain condition. The following example increases the severity of all alerts in the alerts.status table that have a severity of 3 to a severity of 4.

```
FOR EACH ROW alert_row in alerts.status WHERE alert_row.Severity=3
   BEGIN
      SET alert_row.Severity = 4;
   END;
```

Triggers that use FOR EACH ROW are also known as a row-level triggers.

About the EVALUATE clause

Generally, use of the EVALUATE clause is relatively inefficient and its use should be avoided whenever possible. When a trigger contains an EVALUATE clause, a temporary table is created to hold the results of the SELECT statement in the EVALUATE clause. The amount of time and resources that this temporary table consumes depends on the number of columns that are selected and the number of rows matched by the condition in the WHERE clause.

In most cases, you can replace an EVALUATE clause with a FOR EACH ROW clause which cursors over the data and does not incur the processor usage of creating a temporary table.

A suitable use for an EVALUATE clause is when a GROUP BY clause is being applied to an SQL query.

Trigger response sequence

An automation, or trigger, fires when certain conditions occur in the ObjectServer:

- Time
- Database action
- Signal

The WHEN clause determines whether the action should run:

- Condition to meet before running the action
- Can determine time of day or day of week first

The EVALUATE statement builds a read-only temporary table to be used in the action

- Use of the EVALUATE clause is relatively inefficient and its use should be avoided whenever possible
- In most cases, you can replace an EVALUATE clause with a FOR EACH ROW clause

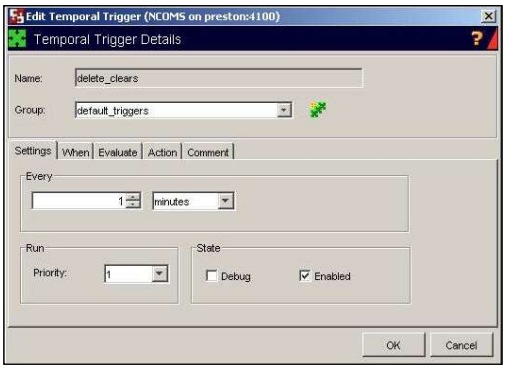
Action runs the SQL code block

Figure 7-27. Trigger response sequence

In summary, triggers activate based on three types of conditions: time, database change, and signal. When the trigger activates, an optional WHEN clause is evaluated. If the WHEN condition is met, the trigger continues. The optional EVALUATE setting creates a temporary table that contains records that meet some condition. The ACTION setting contains the commands that run.

Temporal trigger example

The `delete_clears` trigger:



SETTINGS — every 1 minute

WHEN — [not used]

EVALUATE — [not used]

ACTION

```
begin
     delete from alerts.status
     where Severity = 0 and
     StateChange < (getdate() - 120);
end
```

COMMENT — Delete clear (Green) alerts over 2 minutes old in alerts.status every 60 seconds

Triggers

Figure 7-28. Temporal trigger example

A temporal trigger activates based on a time frequency. Do not confuse time in this case with time of day. The frequency defines how frequently the trigger activates, not when it activates. For example, a temporal trigger with a frequency of 1 hour activates every hour. The trigger might not activate on even hour boundaries, for example, 8:00, 9:00, 10:00. The activation is based on when the ObjectServer starts. After the ObjectServer starts, the trigger activates every hour.

Example: the delete_clears trigger

The `delete_clears` trigger performs a basic housekeeping function, ensuring that clear (Severity = 0) events are removed from the ObjectServer alerts.status table after a period of inactivity.

This trigger is a temporal trigger, so the Settings tab indicates that it runs every minute.

The When tab is empty in this trigger, indicating that it always runs.

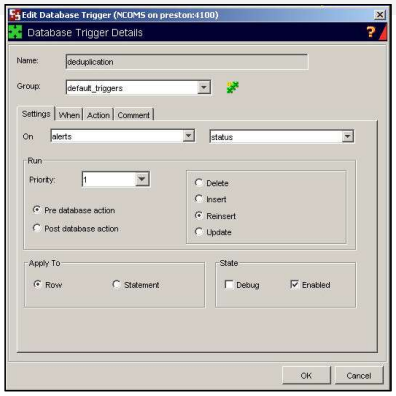
The Evaluate tab permits building a temporary result set. This set is not needed in this trigger, so it is empty.

The Action tab contains the SQL statement that accomplishes the work of this trigger:

```
begin
   delete from alerts.status
   where Severity = 0 and
   StateChange < (getdate() - 120);
end
```

Database trigger example

The deduplication trigger:



SETTINGS: before a reinsert action into alerts.status

WHEN: [not used]

ACTION:

```
begin
   set old.Tally = old.Tally  +  1;
   set old.LastOccurrence = new.LastOccurrence;
   set old.StateChange = getdate();
   set old.InternalLast = getdate();
   set old.Summary = new.Summary;
   set old.AlertKey = new.AlertKey;
   if  ((old.Severity = 0)and(new.Severity > 0))
   then
        set old.Severity = new.Severity;
   end if;
end
```

COMMENT: Deduplication processing for alerts.status

Triggers

Figure 7-29. Database trigger example

One of the key features of the ObjectServer is data deduplication. This feature provides for significant event volume reduction by storing a single copy of an event regardless of how many times it repeats.

In a database trigger, the ObjectServer looks for a database operation to occur against a table, rather than a time interval. The database operation can be Delete, Insert, Reinsert, or Update. The Pre/Post Action selector determines whether the action executes before or after the specified database operation.

Apply to Row/Statement, if set to row (the default), means that the contents of the Action tab run as many times as there are rows selected. When set to Statement, the action runs only once, regardless of how many rows were affected.

On the Action tab, triggers also have access to the implicit variables `new` and `old`, whose values are automatically set by the system.

- Row fields before change: old.fieldname (for example, `old.Severity`)
- Row fields before change: new.fieldname (for example, `new.Severity`)

In some operations, new or old row variables might not be available. For example, if a row is deleted, there is no new row to read or modify. The availability of implicit variables depends on the database operation performed.

Example: the deduplication trigger

The deduplication trigger is an example of a database trigger. The Settings tab indicates that it fires on a reinsert (an attempted insert into a table where the unique key already exists) into alerts.status.

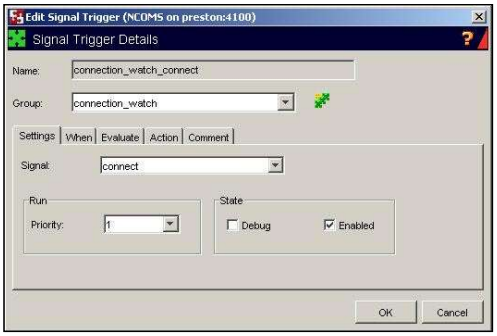
The When tab is empty, so it always runs.

The Evaluate tab is not used in this trigger.

The Action tab updates fields on the existing event selectively (replacing the existing old values with the incoming new values). It also increments the Tally, and only updates the Severity if it had already been set to clear (0).

Signal trigger example

The `connection_watch_connect` trigger:



SETTINGS — Trigger on a connect signal

WHEN — [not used]

EVALUATE — [not used]

ACTION

```
begin
     if( %signal.description = '' )  then
         insert into alerts.status (Identifier, Summary, … OwnerUID)
             values (%signal.process+'@'+%signal.node+' connected '+
             to_char(%signal.at),  … 65534);
          else
         insert into alerts.status (Identifier, Summary, … OwnerUID)
            values (%signal.process+':'+%signal.description+'@'+%signal.node+
            'connected '+to_char(%signal.at), …  65534);
         end if;
end
```

COMMENT — Create an alert when a new client connects

Triggers

Figure 7-30. Signal trigger example

A signal is an occurrence in the ObjectServer that can be detected and acted upon. Signals can have triggers attached to them. The ObjectServer can then respond with a specific action when a signal is raised.

System signals are raised by the ObjectServer on changes to the system, for example:

- System startup, system shutdown
- Client connect, client disconnect, connection failure
- Backup success or failure

When a system signal is raised, attributes that identify the cause of the signal are attached to the signal:

```
%signal.at, %signal.server, %signal.node
```

These attributes cannot be deleted or modified.

The Settings tab lets you choose the signal to execute this trigger. The signal can be a System or User signal.

The When, Evaluate, and Action tabs function as in previous types of triggers covered.

The ObjectServer includes several system triggers, which are signal triggers. System triggers are raised automatically based on some condition. For example, the connect system signal is raised when a component connects to the ObjectServer. A disconnect signal is raised when a component disconnects from the ObjectServer. If a signal trigger is configured based on the `connect` signal, the trigger activates and creates a new event when a component connects to the ObjectServer.

Signal triggers are used to automate numerous system functions, for example:

- **Auditing**: The triggers generate ObjectServer events when administrative changes are made, such as the addition of new fields in an ObjectServer table.
- **Profiling**: Triggers collect and report statistics regarding the performance of the ObjectServer.
- **Connections:** Triggers generate events that are based upon connects, disconnects, and connection failures.
- **Failover and Failback**: Triggers are used to implement controlled failover and failback in ObjectServer high availability configurations

You can create user signals. For user signals, the signal must be raised in some fashion. The signal can be raised with a tool, or from within another trigger. For example, a database trigger can be configured to activate based on a DELETE to the alerts.status table. In the ACTION section of the database trigger, you can configure a statement to RAISE a user signal.

Example: the connection_watch_connect trigger

The connection_watch_connect trigger inserts a new event into the ObjectServer when client components connect.

The Settings tab indicates that it executes based upon a connection signal.

The When and Evaluate tabs are empty. The trigger always runs, and no temporary table is built. The Action tab might look complicated (the example in the slide has been shortened). However, the action inserts one type of event if the %signal.description attribute is null, and another type if it is not.

Procedures

Procedures are executable code called to perform common operations.

- Two types of procedures:
  - SQL procedures manipulate data in an ObjectServer database
  - External procedures run an executable on a remote system
- Can call procedures from nco_sql, a trigger, or a tool
- Syntax:

```
{EXECUTE|CALL} [ PROCEDURE ] procedure_name(expr,...);
```

- Example:

```
EXECUTE PROCEDURE myproc();
```

or with parameters:

```
EXECUTE PROCEDURE myproc("text",@Node);
```


Figure 7-31. Procedures

Procedures are objects that can be called in SQL to perform an SQL operation or an external operation. A procedure is similar to a macro in a programming language. It is a prebuilt collection of code (SQL statements) that can be called from another process. The behavior of the procedure can be adjusted, based upon variables (parameters) that are passed to the procedure when called.

The procedures are stored in an appropriate table in the ObjectServer. The tables are catalog.sql_procedures or catalog.external_procedures.

An External Procedure calls a script to run on the ObjectServer machine or any other machine defined in process activity (PA). The process must be in the form of a script, and can accept command-line parameters passed as variables from the caller. External procedures can accept only IN type parameters. After the script is launched, it cannot return parameters back to the caller.

You must run process activity (PA) to use external procedures. The external procedure is called within the ObjectServer. The ObjectServer notifies process activity that a command must be run on a host. The ObjectServer passes the host, user, and command name to process activity. The process activity daemon runs the command.

Adding journal entries with a trigger

- Create a journal entry with the JINSERT stored procedure:

```
call jinsert( old.Serial, %user.user_id, getdate, 'This is my journal entry');
```

  - Must change `old.Serial` to whatever is appropriate

- Example using the `mail_on_critical` trigger:

```
begin
    for each row critical in criticals
        begin
            execute send_email( critical.Node, critical.Severity, 'Critical Event Email',
  'root@localhost', critical.Summary, 'localhost');

            update alerts.status via critical.Identifier set Grade=2;

            execute jinsert( critical.Serial, %user.user_id, getdate, 'Email sent via
  mail_on_critical Automation');
        end;
end
```


Figure 7-32. Adding journal entries with a trigger

This slide shows the definition for the jinsert procedure. The jinsert procedure creates a journal record. The parameters values identify the corresponding event record, and the message to place in the journal entry. You can use this procedure to create a journal entry when a trigger modifies an event record.

In this example, the mail_on_critical trigger has been revised to call the jinsert procedure to produce a journal entry.

Trigger best practices

- In a WHERE clause, compare integers first, then characters. Leave intensive comparisons and regular expressions for last
- Ensure trigger does not catch events already managed
- Stagger trigger priorities so that they do not all run at the same time
- Add a description for all automations
- Event Manager comes with standard triggers that you can change to meet customer needs
  - The *best practice* for modifying a trigger is to create a copy, disable the original, and modify the copy
  - **ATTENTION:** You can modify existing triggers, but changes can affect the running of the ObjectServer

Figure 7-33. Trigger best practices

When constructing the expression in the where clause, pay attention to type of field. Perform all integer testing first (Severity=5), character processing next (Node=XYZ), and regular expression testing last.

When creating the filter condition, include criteria that ensures that the trigger does not select the same event over and over. This is important because triggers are designed to perform automated actions.

If multiple temporal triggers are defined to run at the same frequency (for example, every five minutes), stagger the priority settings among them. Staggering them ensures that they do not all run at the same time.

Triggers are a powerful and useful feature. Most teams introduce new triggers periodically to resolve some operational situation. Over time, it might be difficult to determine why a trigger exists and what it was designed to accomplish. Take the time to add a short description to any new triggers. Consider capturing details, such as when the trigger was created, the author, and why the trigger was created.

Avoid editing the standard triggers that come with Event Manager. Instead, make a copy of the existing trigger and edit the copy.

Unit summary

- Learn to interact with Event Manager using an SQL interface
- Understand how triggers change events in the Event Manager database

Figure 7-34. Unit summary

Review questions

1. True or False: A temporal trigger runs on a timed basis.
2. Which is the best way to insert a journal entry when an ObjectServer trigger fires?
   a. The JINSERT stored procedure.
   b. The AUTO_TRIGGER stored procedure.
   c. With the topology service.
   d. With the runbook service.

Figure 7-35. Review questions

Write your answers here:

1.

2.

Review answers

1. True or False: A temporal trigger runs on a timed basis.
   The answer is True.
2. Which is the best way to insert a journal entry when an ObjectServer trigger fires?
   a. The JINSERT stored procedure.
   b. The AUTO_TRIGGER stored procedure.
   c. With the topology service.
   d. With the runbook service.
   The answer is A.

Triggers

Figure 7-36. Review answers

Exercise: Triggers

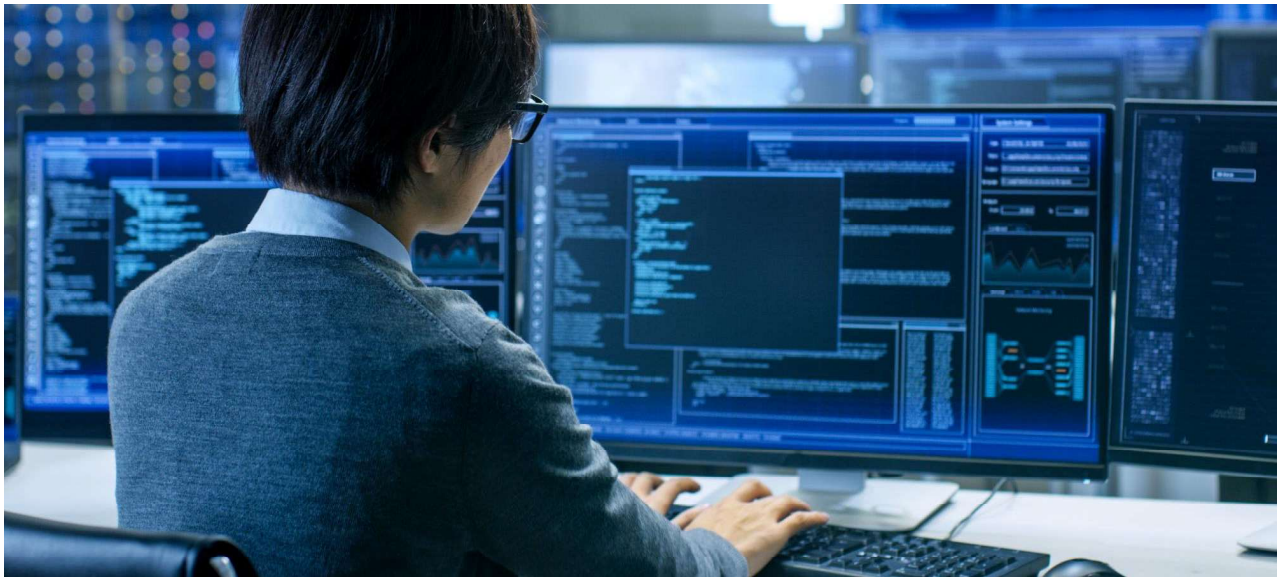


Figure 7-37. Exercise: Triggers

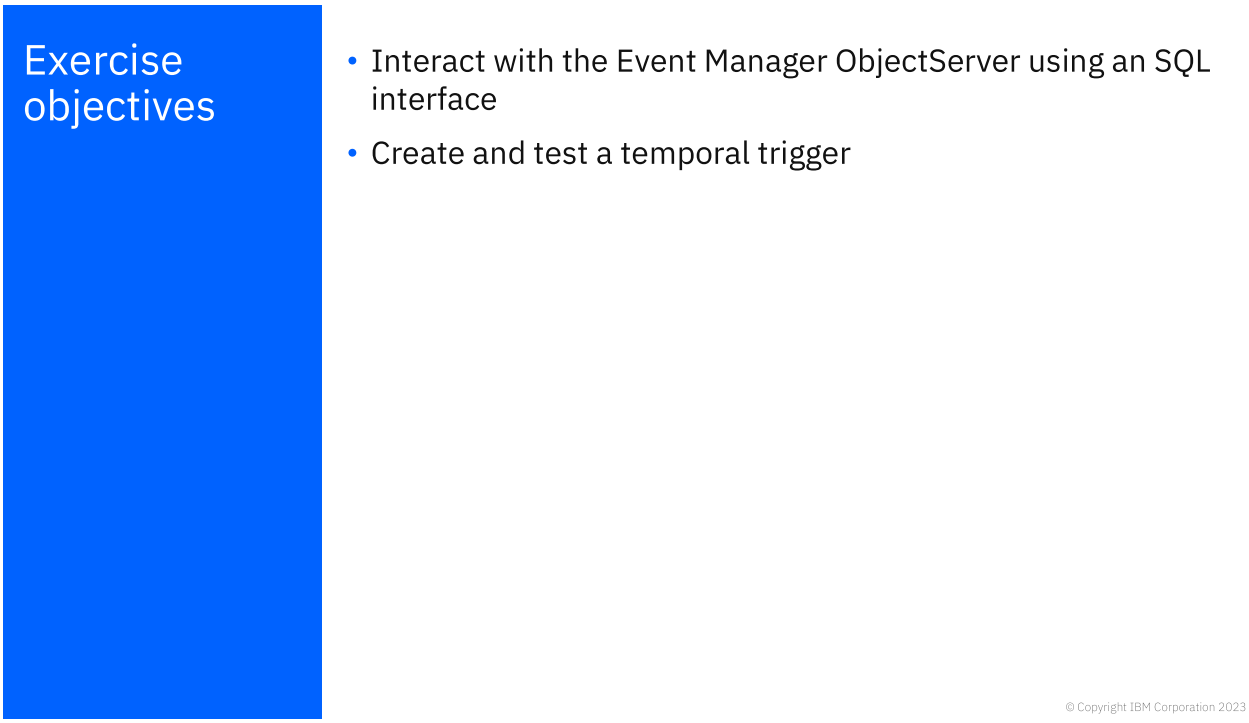


Figure 7-38. Exercise objectives

These are the tasks you complete during the lab exercises for this unit.

Lab tips

- This exercise depends on steps you completed earlier in labs for Unit 2: “Incoming integrations.” Before you continue, be sure to complete the labs for the “Incoming integrations” unit of this course.
- After 10 minutes of inactivity, the Event Manager Administrator tool logs you out. If this happens, double-click the ObjectServer to open the tool again. If you are logged out, you also see errors in the terminal. You can safely ignore errors like these:

```
ERROR : Code-0 : Mon Jan 30 14:52:57 EST 2023 : bastion.labs.ihost.com/10.100.1.8 :
TextEditorPanel.getFullWordAt :   : Exception: Invalid location
ERROR : Code-0 : Mon Jan 30 14:52:58 EST 2023 : bastion.labs.ihost.com/10.100.1.8 :
TextEditorPanel.getFullWordAt :   : Exception: Invalid location
ERROR : Code-0 : Mon Jan 30 14:52:59 EST 2023 : bastion.labs.ihost.com/10.100.1.8 :
TextEditorPanel.getFullWordAt :   : Exception: Invalid location
ERROR : Code-0 : Mon Jan 30 14:53:00 EST 2023 : bastion.labs.ihost.com/10.100.1.8 :
TextEditorPanel.getFullWordAt :   : Exception: Invalid location
```


Figure 7-39. Lab tips