CUMULOCITY IoT
BY SOFTWARE AG

Cumulocity IoT
Event Language guide

IMPORTANT

The functionality described in this guide is deprecated. All new Cumulocity IoT installations will use the Apama CEP engine. Software AG has terminated support for using CEL (Esper) in Cumulocity IoT on 31 Dec 2020 following its deprecation in 2018.

For further information on using Apama's Event Processing Language in Cumulocity IoT refer to the Streaming Analytics guide.

For details on migration, refer to Migrating from CEL (Esper) to Apama in the *Streaming Analytics guide*.

October 2021

Table of Contents

Introduction

Overview

Using the Cumulocity IoT real-time event processing, you can add your own logic to your IoT solution. This includes data analytics logic but it is not limited to it. To define new analytics, you can use the Cumulocity Event Language. The language allows analyzing incoming data. It is using a powerful pattern and time window based query language. You can create, update and delete your data in real-time.

Typical real-time analytics use cases include:

- Remote control: Turn a device off if it’s temperature rises over 40 degrees.
- Validation: Discard negative meter readings or meter readings that are lower than the previous.
- Derived data: Calculate the volume of sales transactions per vending machine per day.
- Aggregation: Sum up the sales of vending machines for a customer per day.
- Notifications: Send me an email if there’s a power outage in one of my machines.
- Compression: Store location updates of all cars only once every five minutes (but still send real-time data for the car that I am looking at to the user interface).

In the following sections, we describe the basics for understanding how the Cumulocity Event Language works and how you can create your own analytics or other server-side business logic and automation.

CEP application variants

In Cumulocity IoT, there are two deployment scenarios for using CEP rules:

- MULTI_TENANT: This scope provides access to a shared instance of CEP container. All subscribed tenants share the resources of the same CEP instance. It is available if you are subscribed to the “Cep” application, a built-in application which comes with Cumulocity IoT.

- PER_TENANT: Each subscribed tenant has at least one own instance of CEP container. The container is isolated from other tenants, hence high CPU load or memory issues on other containers do not have any impact on the own one. This feature is available with the application “Cep-small” which is an optional service. Also, you need to be subscribed to the application" Feature-cep-custom-rules" to be able to upload your own CEP rules.

For details on application subscription refer to Administration > Managing tenants > Subscribing to applications in the *User guide*.

Using Cumulocity Event Language (CEL)

Cumulocity Event Language has a syntax similar to SQL language. In SQL a statement is run against a logically fixed database, produces a result and completes the task. In Cumulocity IoT, a statement is continuously running against a stream of input data (input events) and is continuously calculating its output (output events).

As an example, the following statement continuously retrieves new temperature sensor readings ranging above a particular temperature:

```
select *
from MeasurementCreated e
where getNumber(e, "c8y_TemperatureMeasurement.T.value") > 100
```

Here, *MeasurementCreated* is a stream containing an event for each measurement that is created in the system. Selecting a subset of these events is done using *where*, similar to SQL. *getNumber()* is a function to read out a numeric value from an event. In this example, “e” is the “MeasurementCreated” event and the property is “c8y_TemperatureMeasurement”. “T.value”, is a value in degrees Celsius of a temperature sensor (see the sensor library).

How can I create derived data from CEL?

There are special streams provided by the system to perform predefined operations (such as data storage or sending data by email). One stream is CreateAlarm, which can be used to store an alarm in Cumulocity IoT. Assume that an alarm should be generated immediately if the temperature of a sensor exceeds a defined value. This is done with the following statement:

```
insert into CreateAlarm
select
 e.measurement.time as time,
 e.measurement.source.value as source,
 "c8y_TemperatureAlert" as type,
 "Temperature too high" as text,
 "ACTIVE" as status,
 "CRITICAL" as severity
from MeasurementCreated e
where getNumber(e, "c8y_TemperatureMeasurement.T.value") > 100
```

Technically, this statement produces a new “AlarmCreated” event each time a temperature sensor reads more than 100 degrees Celsius and puts it into the “CreateAlarm” output stream. The property names in the selected clause have to match the properties of “AlarmCreated” (see the Cumulocity Event Language reference).

How can I control devices from CEL?

Remote control in CEL is just another type of derived data. Remote operations are targeted to a specific device. The following example illustrates switching a relay based on temperature readings:

```
insert into CreateOperation
select
"PENDING" as status,
<<heating ID>> as deviceId,
{
"c8y_Relay.relayState", "CLOSED"
} as fragments
from MeasurementCreated e
where getNumber(e, "c8y_TemperatureMeasurement.T.value") > 100
```

- *heating ID* is a placeholder for the ID of the heating that should be triggered.

- *fragments* defines the nested content of the operation a “c8y\Relay” that is “CLOSED”.

The syntax of the *fragments* part is a list of pairs of property names and values surrounded by curly braces: {?key1?, ?value1?, ?key2?, ?value2?, …}.

How can I query data from CEL?

It may be required to query information from the Cumulocity IoT database as part of the ongoing event processing. This is supported by a set of querying methods. Here is an example that shows how to summarize total sales for vending machines every hour. The sales report data created after a purchase is retrieved from the Cumulocity IoT database.

```
create window SalesReport.win:time_batch(1 hour)
(
    event com.cumulocity.model.event.Event,
    customer com.cumulocity.model.ManagedObject
)

insert into SalesReport
select
    e.event as event,
    findOneManagedObjectParent(e.event.source.value) as customer
from EventCreated as e

insert into CreateMeasurement
select
    "total_cust_trx",
    "customer_trx_counter",
    {
        "total", count(*),
        "customer_id", sales_report.customer.id.value
    }
from SalesReport as sales_report
group by sales_report.customer.id.value
```

Above we create a batch window first, which keeps data for one hour in order to calculate a total in this time frame. We store the prepared data into this window: Incoming events along with the parent managed object of the event source. This corresponds to the data model of our vending application: Sales reports are represented as events in Cumulocity IoT with a vending machine as source. Customers are represented as parent managed objects of vending machines.

The collection of sales reports is calculated through “insert into CreateMeasurement…” using a SQL-like syntax and is stored as a measurement. The difference to SQL is: In SQL, you calculate a result over a fixed, current content of a database. In Cumulocity Event Language, statements run endlessly and the process time has to be limited by the time window.

Event streams

In the Cumulocity Event Language data flows in streams. You can create events in streams and listen to events created in streams.

Predefined streams

There are some predefined streams to interact with several Cumulocity IoT APIs. For each input stream, Cumulocity IoT will automatically create a new event when the respective API call was made. If a measurement was created via REST API there will be a new event in the MeasurementCreated stream. For interacting with the Cumulocity IoT backend you can create an event on the respective output stream and Cumulocity IoT will automatically execute either the database query or create the API calls necessary for sending mails, sms, or similar. To create a new alarm in the database you can create a new event in the CreateAlarm stream.

API	Input streams	Output streams	Description
Inventory	ManagedObjectCreated ManagedObjectUpdated ManagedObjectDeleted	CreateManagedObject UpdateManagedObject DeleteManagedObject	This group of events represents creation, modification or deletion of a single ManagedObject.
Events	EventCreated EventUpdated EventDeleted	CreateEvent UpdateEvent DeleteEvent	This group of events represents creation or deletion of a single Event.
Measurements	MeasurementCreated MeasurementDeleted	CreateMeasurement DeleteMeasurement	This group of events represents creation or deletion of a single Measurement.
Device control	OperationCreated OperationUpdated	CreateOperation UpdateOperation	This group of events represents creation or modification of a single Operation.
Alarms	AlarmCreated AlarmUpdated	CreateAlarm UpdateAlarm	This group of events represents creation or modification of a single Alarm.
Emails	*(not used)*	SendEmail SendDashboard	This group of events represents sending of an email.
SMS	*(not used)*	SendSms	This group of events represents sending of a SMS.
HTTP	ResponseReceived	SendReqeust	This group of events represents sending http requests to external services.
Export	*(not used)*	SendExport	This group of events represents generating emails with exported data.

Look at the data model to see how the events for each stream are structured.

Creating events in a stream

Creating an event is done by the keywords `insert into` and `select`. First, you need to specify the “insert into” followed by the stream name for which stream you want to create an event. After that you can use the “select” clause to specify the parameters of the event. A parameter gets specified by the following syntax: `value as parameter`. You can specify multiple parameters by separating them by commas. The order of the parameters does not matter. Please notice that streams can have mandatory parameters you need to specify in the “select” clause.

Listening to events in a stream

The most common way to trigger the creation of an event in a stream is when something happens on another stream. Therefore you can listen to events from other streams. This is done by the keyword `from` followed by the name of the stream and (optional) followed by a variable name to reference the event in your statement at a later point.

Conditions

Adding conditions can be done with the keyword `where` to not trigger your event creation for every incoming event but only if these conditions are met. The `where` keyword is followed by an expression that results either in true or false. You can also have multiple expressions connected with `and` or `or`.

Example

As an example, we create a statement. It should listen to a stream and create a new event in another stream whenever the specified condition applies. As example we want to create an alarm for each temperature measurement that is created.

1. To create an alarm we need to `insert into` the stream `CreateAlarm`.
2. We need to specify all parameters for the event in the `select` clause.
3. We want the alarm to be created when an event `from` the stream `MeasurementCreated` is received.
4. We want the alarm only be created under certain conditions of the event from the `MeasurementCreated` stream which we specify in the `where` clause.

The resulting statement can look like this:

```
insert into CreateAlarm
select
  measurementEvent.measurement.time as time,
  measurementEvent.measurement.source.value as source,
  "c8y_TemperatureAlarm" as type,
  "Temperature measurement was created" as text,
  "ACTIVE" as status,
  "CRITICAL" as severity
from MeasurementCreated measurementEvent
where measurementEvent.measurement.type = "c8y_TemperatureMeasurement";
```

Troubleshooting

Error message

```
Real-time event processing is currently overloaded and may stop processing your events.
Please contact [product support](https://cumulocity.com/guides/welcome/contacting-support/).
```

Description

The CEP queue for the respective tenant is full. This might for example happen when more events are created than currently can be handled.

In this case, an alarm will be raised. To avoid losing incoming new events, the oldest events will be deleted, i.e. an incoming new event triggers the deletion of the queue head event.

Data model

Input streams

General structure

All input streams share the same base structure.

Parameter	Data type	Description
_type	String	The type of the event. See the table below which value types can be used for different streams.
_mode	String	The processing mode in which the data was sent to Cumulocity IoT. See **Processing mode** in the Cumulocity IoT OpenAPI Specification.
_origin	String	The origin of the event. If the data was created by a cep rule the origin will be “cep”.
payload	Object	The actual data contained in the event

Types:

Stream	Type
ManagedObjectCreated	MANAGED_OBJECT_CREATE
ManagedObjectUpdated	MANAGED_OBJECT_UPDATE
ManagedObjectDeleted	MANAGED_OBJECT_DELETE
EventCreated	EVENT_CREATE
EventUpdated	EVENT_UPDATED
EventDeleted	EVENT_DELETE
MeasurementCreated	MEASUREMENT_CREATE
MeasurementDeleted	MEASUREMENT_DELETE

Stream	Type
OperationCreated	OPERATION_CREATE
OperationUpdated	OPERATION_UPDATE
AlarmCreated	ALARM_CREATE
AlarmUpdated	ALARM_UPDATE
ResponseReceived	REQUEST_RESULT

For simpler access you can receive the payload directly in the data type of the respective stream by accessing it via an API specific parameter:

API	Parameter	Data type
Inventory	managedObject	ManagedObject
Events	event	Event
Measurements	measurement	Measurement
Device control	operation	Operation
Alarms	alarm	Alarm

ManagedObject

class: com.cumulocity.model.ManagedObject

Parameter	Data type	Description
id	ID	ID of the ManagedObject
type	String	The type of the ManagedObject
name	String	The name of the ManagedObject
lastUpdated	Date	The time when the ManagedObject was last updated
owner	String	The owner of the ManagedObject
childAssets	Object[]	An array of the IDs of all child assets
childDevices	Object[]	An array of the IDs of all child devices
assetParents	Object[]	An array of the IDs of all parent assets
deviceParents	Object[]	An array of the IDs of all child devices

The Object[] for the references to the parents and children contains only IDs. You can use the cast function e.g. `cast(event.managedObject.childAssets[0], com.cumulocity.model.ID)` .

Example:

```
select
  event.managedObject.id,
  event.managedObject.type,
  event.managedObject.name,
  event.managedObject.lastUpdated,
  event.managedObject.owner,
  event.managedObject.childAssets,
  event.managedObject.assetParents,
  event.managedObject.deviceParents,
  event.managedObject.childDevices
from ManagedObjectCreated event;
```

Event

class: com.cumulocity.model.event.Event

Parameter	Data type	Description
id	ID	The ID of the Event
creationTime	Date	The time when the Event was created in the database
type	String	The type of the Event
text	String	The text of the Event
time	Date	The time when the Event was created (as sent by device)
source	ID	The ID of the device which created the Event

Example:

```
select
  event.event.id,
  event.event.creationTime,
  event.event.type,
  event.event.text,
  event.event.time,
  event.event.source
from EventCreated event;
```

Measurement

class: com.cumulocity.model.measurement.Measurement

Parameter	Data type	Description
id	ID	The ID of the Measurement
type	String	The type of the Measurement

Parameter	Data type	Description
time	Date	The time when the Measurement was created (as sent by device)
source	ID	The ID of the device which created the Measurement

Example:

```
select
  event.measurement.id,
  event.measurement.type,
  event.measurement.time,
  event.measurement.source
from MeasurementCreated event;
```

Operation

class: com.cumulocity.model.operation.Operation

Parameter	Data type	Description
id	ID	The ID of the Operation
creationTime	Date	The time when the Operation was created in the database
status	OperationStatus	The current status of the Operation
deviceId	ID	The ID of the device which should execute the Operation

Example:

```
select
  event.operation.id,
  event.operation.creationTime,
  event.operation.status,
  event.operation.deviceId
from OperationCreated event;
```

Alarm

class: com.cumulocity.model.event.Alarm

Parameter	Data type	Description
id	ID	The ID of the Alarm
creationTime	Date	The time when the Alarm was created in the database
type	String	The type of the Alarm
count	long	The number of times the alarm was reported while active

Parameter	Data type	Description
severity	Severity	The severity of the Alarm
status	AlarmStatus	The status of the Alarm
text	String	The text of the Event
time	Date	The time when the Event was created (as sent by device)
source	ID	The ID of the device which created the Alarm

Example:

```
select
  event.alarm.id,
  event.alarm.creationTime,
  event.alarm.type,
  event.alarm.count,
  event.alarm.severity,
  event.alarm.status,
  event.alarm.text,
  event.alarm.time,
  event.alarm.source
from AlarmCreated event;
```

Response received

Parameter	Data type	Description
status	Integer	Http response status
body	String	Http response body
creationTime	Date	The time when the response was created
source	Object	Source set in SendRequest output stream

Example:

```
select
  event.status,
  event.body,
  event.creationTime,
  getString(event.source, 'id.value') as source
from ResponseReceived event;
```

Output streams

General structure

Output streams contain the possibility to CREATE, UPDATE and DELETE data in Cumulocity IoT. When updating or deleting data it is necessary to provide the ID of the object that will be updated or deleted. When creating data, Cumulocity IoT will generate an ID if not set in the event processing. The creation of data also requires certain parameters to be set (the same as at our REST APIs). In addition to the predefined parameters listed, it is possible to add any custom fragment to the data. Please take a look at the custom fragments section for adding custom fragments.

Note: Creating your own ID will only work on ManagedObjects.

ManagedObjects

Available outputs
CreateManagedObject
UpdateManagedObject
DeleteManagedObject

Parameter	Data type	Description	Mandatory
id	ID or String	ID of the ManagedObject	UPDATE and DELETE
type	String	The type of the ManagedObject	No
name	String	The name of the ManagedObject	No
owner	String	The owner of the ManagedObject. If not set data created from event processing will have the owner “cep”	No
childAssets	Set<String> or Set<ID>	A set of IDs of all child assets	No
childDevices	Set<String> or Set<ID>	A set of IDs of all child devices	No

Example:

```
insert into CreateManagedObject
select
  "myManagedObject" as name,
  "myType" as type
from EventCreated event;

insert into UpdateManagedObject
select
  "12345" as id,
  "myNewManagedObject" as name
from EventCreated event;

insert into DeleteManagedObject
select
  "12345" as id
from EventCreated event;
```

Events

Available outputs
CreateEvent
UpdateEvent
DeleteEvent

Parameter	Data type	Description	Mandatory
id	ID or String	The ID of the Event	DELETE
type	String	The type of the Event	CREATE
text	String	The text of the Event	CREATE
time	Date	The time when the Event was created (as sent by device)	CREATE
source	ID or String	The ID of the device which created the Event	CREATE

Example:

```
insert into CreateEvent
select
  "copiedEventType" as type,
  "This event was copied" as text,
  event.event.time as time,
  event.event.source as source
from EventCreated event;

insert into DeleteEvent
select
  "12345" as id
from EventCreated event;
```

Measurements

Available outputs
CreateMeasurement
DeleteMeasurement

Parameter	Data type	Description	Mandatory
id	ID or String	The ID of the Measurement	DELETE
type	String	The type of the Measurement	CREATE
time	Date	The time when the Measurement was created (as sent by device)	CREATE
source	ID or String	The ID of the device which created the Measurement	CREATE

Example:

```
insert into CreateMeasurement
select
  "c8y_TemperatureMeasurement" as type,
  event.event.time as time,
  event.event.source as source,
  {
    "c8y_TemperatureMeasurement.T.value", 5
  } as fragments
from EventCreated event;

insert into DeleteMeasurement
select
  "12345" as id
from EventCreated event;
```

Operations

Available outputs
CreateOperation
UpdateOperation

Parameter	Data type	Description	Mandatory
id	ID or String	The ID of the Operation	UPDATE
status	OperationStatus or String	The current status of the Operation	CREATE
deviceId	ID or String	The ID of the device which should execute the Operation	CREATE

Example:

```
insert into CreateOperation
select
  OperationStatus.PENDING as status,
  event.event.source as deviceId,
  {
    "c8y_Restart", {}
  } as fragments
from EventCreated event;

insert into UpdateOperation
select
  "12345" as id,
  OperationStatus.EXECUTING as status
from EventCreated event;
```

Alarms

Available outputs
CreateAlarm
UpdateAlarm

Parameter	Data type	Description	Mandatory
id	ID or String	The ID of the Alarm	UPDATE
type	String	The type of the Alarm	CREATE
severity	Severity or String	The severity of the Alarm	CREATE
status	AlarmStatus or String	The status of the Alarm	CREATE
text	String	The text of the Event	CREATE
time	Date	The time when the Event was created (as sent by device)	CREATE
source	ID or String	The ID of the device which created the Alarm	CREATE

Example:

```
insert into CreateAlarm
select
  "c8y_HighTemperatureAlarm" as type,
  event.event.time as time,
  event.event.source as source,
  CumulocitySeverities.WARNING as severity,
  CumulocityAlarmStatuses.ACTIVE as status,
  "The device has high temperature" as text
from EventCreated event;

insert into UpdateAlarmn
select
  "12345" as id,
  CumulocityAlarmStatuses.ACKNOWLEDGED as status
from EventCreated event;
```

Special streams

The streams mentioned in this section do not interact with the Cumulocity IoT database but will create calls to external services.

SendMail

Parameter	Data type	Description	Mandatory
receiver	String	The mail address of the receiver	yes
cc	String	The mail address of the cc	no
bcc	String	The mail address of the bcc	no
replyTo	String	The mail address which should receive replies to the sent mail	yes
subject	String	The subject line of the mail	yes
text	String	The body of the mail	yes

It is possible to have more than one mail address in the parameters receiver,cc and bcc. Therefore create a string that contains all mail addresses separated by commas. “receiver1@mail.com,receiver2@mail.com”.

Example:

```
insert into SendEmail
select
  "receiver1@cumulocity.com,receiver2@cumulocity.com" as receiver,
  "cc@cumulocity.com" as cc,
  "bcc@cumulocity.com" as bcc,
  "reply@cumulocity.com" as replyTo,
  "Example mail" as subject,
  "This mail was sent to test the SendEmail stream in Cumulocity" as text
from AlarmCreated;
```

SendSms

Parameter	Data type	Description	Mandatory
receiver	String	The phone number of the receiver	yes
text	String	The body of the sms. Max. 160 characters	yes

Parameter	Data type	Description	Mandatory
deviceId	String	The ID of the device generating the sms. A log event will be created for the device	no

It is possible to have more than one phone number in the parameter receiver. Therefore create a string that contains all phone numbers separated by commas e.g. “+49123456789,+49987654321”. Although it is technically not required by Cumulocity IoT to have the country code we recommend you to use it because the sms gateway might require it. You can use the notation like e.g. “0049” or “+49” (for Germany).

Note:

This feature will only work if your tenant is linked to a sms provider. For more information please contact product support.

Example:

```
insert into SendSms
select
  "+49123456789" as receiver,
  "This sms was sent to test the SendSms stream in Cumulocity" as text,
  "12345" as deviceId
from AlarmCreated;
```

SendPush

This stream enables the possibility to send push notifications from Cumulocity IoT via the Telekom push service to mobile applications.

Parameter	Data type	Description	Mandatory
type	String	Push Provider Type. Currently only TELEKOM is possible.	yes
message	String	The body of the push message.	yes
deviceId	String	The ID of the device generating the push message.	yes

Note:

This feature will only work if your tenant is linked to a push provider. For more information please contact product support.

Example:

```
insert into SendPush
select
"TELEKOM" as type,
"sample push message" as message,
a.alarm.source.value as deviceId
from AlarmCreated a;
```

SendRequest

This stream enables the possibility to send HTTP requests from Cumulocity IoT to external systems.

Parameter	Data type	Description	Mandatory
url	String	Url of external system	yes
method	String	Method of HTTP request	yes
body	String	Body of HTTP reqeust	no
authorization	String	HTTP Authorization header	no
contentType	String	HTTP Content-Type header	no
headers	Map<String,String>	HTTP headers	no
source	Object	Represents object which will be passed to ResponseReceived input stream	no

Example:

```
insert into SendRequest
select
  'post' as method,
  'http://some.external.service.com' as url,
  'application/json' as contentType,
  toJSON(m.payload) as body,
  m.payload as source
from MeasurementCreated m
```

SendExport

This stream enables the possibility to generate export.

Parameter	Data type	Description	Mandatory
enabledSources	List	Export configuration ids	true
subject	String	Subject of email	false
text	String	Text of email. Available placeholders: {host}, {binaryId}. Default message is: “File with exported data can be downloaded from {host}/inventory/binaries/{binaryId}”	false
receiver	String	Receiver of email	false

Example:

```
insert into SendExport
select
    'configurationExportId' as enabledSources,
    'subject' as subject,
    'text' as text,
    'receiver@example.com' as receiver
from
    pattern [every timer:at(5, *, *, *, *)]
```

Additional data models

ID

class: com.cumulocity.model.ID

Parameter	Data type	Description
value	String	The actual ID value
type	String	The type of the ID
name	String	The name of the device (only if the ID refers to a device like in measurement.source)

Example:

```
select
  event.measurement.source.value,
  event.measurement.source.type,
  event.measurement.source.name
from MeasurementCreated event;
```

OperationStatus

class: com.cumulocity.model.operation.OperationStatus

OperationStatus is an enum offering the following values: `PENDING` , `SUCCESSFUL` , `FAILED` , `EXECUTING`

Example:

```
insert into UpdateOperation
select
  event.operation.id.value as id,
  OperationStatus.FAILED as status
from OperationCreated event;
```

Severity

class: com.cumulocity.model.event.Severity

Severity is the interface for the enum implementation CumulocitySeverities. CumulocitySeverities offers the following values: `CRITICAL`, `MAJOR`, `MINOR`, `WARNING`

Example:

```
insert into UpdateAlarm
select
  event.alarm.id.value as id,
  CumulocitySeverities.MAJOR as severity
from AlarmCreated event;
```

AlarmStatus

class: com.cumulocity.model.event.AlarmStatus

AlarmStatus is the interface for the enum implementation CumulocityAlarmStatuses. CumulocityAlarmStatuses offers the following values: `ACTIVE`, `ACKNOWLEDGED`, `CLEARED`

Example:

```
insert into UpdateAlarm
select
  event.alarm.id.value as id,
  CumulocityAlarmStatuses.ACKNOWLEDGED as status
from AlarmCreated event;
```

Functions

With the Cumulocity Event Language it is possible to utilize functions. This section will cover the already built-in functions ready to use.

For guidance on how to write your own functions please check Creating own functions.

Java functions

Every module automatically imports the following libraries:

```
java.lang.*
java.math.*
java.text.*
java.util.*
```

You can use any of the functions located in those libraries.

Examples:

Using java.util.Random

```
create variable Random generator = new Random();

insert into CreateMeasurement
select
  "12345" as source,
  "c8y_TemperatureMeasurement" as type,
  current_timestamp().toDate() as time,
  {
    "c8y_TemperatureMeasurement.T1.value", generator.nextInt(12) + 18,
    "c8y_TemperatureMeasurement.T1.unit", "C",
    "c8y_TemperatureMeasurement.T2.value", generator.nextInt(12) + 18,
    "c8y_TemperatureMeasurement.T2.unit", "C",
    "c8y_TemperatureMeasurement.T3.value", generator.nextInt(12) + 18,
    "c8y_TemperatureMeasurement.T3.unit", "C",
    "c8y_TemperatureMeasurement.T4.value", generator.nextInt(12) + 18,
    "c8y_TemperatureMeasurement.T4.unit", "C",
    "c8y_TemperatureMeasurement.T5.value", generator.nextInt(12) + 18,
    "c8y_TemperatureMeasurement.T5.unit", "C"
  } as fragments
from pattern[every timer:at(*, *, *, *, *, */30)];
```

Using java.math.BigDecimal

```
select
  getNumber(m, "c8y_TemperatureMeasurement.T.value").divide(new
BigDecimal(3),2,RoundingMode.HALF_UP)
from MeasurementCreated m;
```

Database functions

To interact with your historical data you can use one of the following functions to directly query the database.

Most functions are available in several variants:

- findOne…(…): The function expects exactly one object as query result and fails otherwise.
- findFirst…(…): The function returns the first object in the query result or “null”, if the result is empty.
- findAll…(…): The function returns all objects in the query result.

Here is the full list of available functions. Replace the ellipses ("…") with “findOne”, “findFirst” or “findAll”.

Function name (with variants)	Return type	Alternative argument lists
findManagedObjectById	Managed Object	id*:String* id*:GId*
findFirstManagedObjectParent findOneManagedObjectParent	Managed Object	managedObjectId*:String* managedObjectId*:GId*
…ManagedObjectByFragmentType	List	fragmentType*:String*
…ManagedObjectByType	List	type*:String*
findEventById	Event	id*:String* id*:GId*
findFirstEventByFragmentType findOneEventByFragmentType	Event	fragmentType*:String*
…EventByFragmentTypeAndSource	List	fragmentType*:String*, source*:String*
…EventByFragmentTypeAndSourceAndTimeBetween	List	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*
…EventByFragmentTypeAndSourceAndTimeBetweenAndType	List	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*, type*:String*
…EventByFragmentTypeAndSourceAndType	List	fragmentType*:String*, source*:String*, type*:String*

Function name (with variants)	Return type	Alternative argument lists
…EventByFragmentTypeAndTimeBetween	List	fragmentType*:String*, from*:Date*, to*:Date*
…EventByFragmentTypeAndTimeBetweenAndType	List	fragmentType*:String*, from*:Date*, to*:Date*, type*:String*
findFirstEventByFragmentTypeAndType findOneEventByFragmentTypeAndType	Event	fragmentType*:String*, type*:String*
…EventBySource	List	source*:String*
findMeasurementById	Measurement	id*:String* id*:GId*
findFirstMeasurementByFragmentType findOneMeasurementByFragmentType	Measurement	fragmentType*:String*
…MeasurementByFragmentTypeAndSource	List	fragmentType*:String*, source*:String*
…MeasurementByFragmentTypeAndSourceAndTimeBetween	List	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*
…MeasurementByFragmentTypeAndSourceAndTimeBetweenAndType	List	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*, type*:String*
…MeasurementByFragmentTypeAndSourceAndType	List	fragmentType*:String*, source*:String*, type*:String*
…MeasurementByFragmentTypeAndTimeBetween	List	fragmentType*:String*, from*:Date*, to*:Date*
…MeasurementByFragmentTypeAndTimeBetweenAndType	List	fragmentType*:String*, from*:Date*, to*:Date*, type*:String*
findFirstMeasurementByFragmentTypeAndType findOneMeasurementByFragmentTypeAndType	Measurement	fragmentType*:String*, type*:String*
…MeasurementBySource	List	source*:String*
findLastMeasurementByFragmentTypeAndSourceAndTimeBetween	Measurement	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*
findLastMeasurementByFragmentTypeAndSourceAndTimeBetweenAndType	Measurement	fragmentType*:String*, source*:String*, from*:Date*, to*:Date*, type*:String*
findLastMeasurementByFragmentTypeAndTimeBetween	Measurement	fragmentType*:String*, from*:Date*, to*:Date*
findLastMeasurementByFragmentTypeAndTimeBetweenAndType	Measurement	fragmentType*:String*, from*:Date*, to*:Date*, type*:String*

Function name (with variants)	Return type	Alternative argument lists
findOperationById	Operation	id*:String* id*:GId*
findFirstOperationByAgent findOneOperationByAgent	Operation	agentId*:String*
...OperationByAgentAndStatus	List	agentId*:String*, status*:String*
findFirstOperationByDevice findOneOperationByDevice	Operation	deviceId*:String*
...OperationByDeviceAndStatus	List	deviceId*:String*, status*:String*
...OperationByStatus	List	status*:String*
...OperationByCreationTimeBetween	List	from*:Date*, to*:Date*
findAlarmById	Alarm	id*:String* id*:GId*
...AlarmBySource	List	sourceId*:String*
...AlarmBySourceAndStatus	List	sourceId*:String*, status*:String*
...AlarmBySourceAndStatusAndType	List	sourceId*:String*, status*:String*, type*:String*
...AlarmBySourceAndStatusAndTimeBetween	List	sourceId*:String*, status*:String*, from*:Date*, to*:Date*
...AlarmBySourceAndTimeBetween	List	sourceId*:String*, from*:Date*, to*:Date*
findFirstAlarmByStatus findOneAlarmByStatus	Alarm	status*:String*
...AlarmByStatusAndTimeBetween	List	status*:String*, from*:Date*, to*:Date*
...AlarmByTimeBetween	List	from*:Date*, to*:Date*

Utility functions

access fragments

Fragments are accessible through the following helper functions:

- Object getObject(Object event, String path[, Object defaultValue])
- String getString(Object event, String path[, String defaultValue])
- Number getNumber(Object event, String path[, Number defaultValue])
- Boolean getBoolean(Object event, String path[, Boolean defaultValue])
- Date getDate(Object event, String path[, Date defaultValue])
- List getList(Object event, String path[, List defaultValue])

You can use JsonPath (without the root element $) to navigate in the object structure Example:

```
select
  getNumber(m, "c8y_TemperatureMeasurement.T.value")
from MeasurementCreated m;

select
  e.event as event
from EventCreated e
where getObject(e, "c8y_Position") is not null;
```

cast

The cast() function gives you the possibility to transform data to the correct data type if you receive it e.g. as Object. Casting to a basic Java type:

```
cast(myVariable, long)
```

For other types you need to specify the fully-qualified class name

```
cast(event.managedObject.childAssets[0], com.cumulocity.model.ID)
```

current_timestamp

The current_timestamp() function will give you the current server time. You can easily transform it to the required Date data type with the toDate() function to be used in the Cumulocity streams.

Example:

```
insert into CreateAlarm
select
  "c8y_HighTemperatureAlarm" as type,
  current_timestamp().toDate() as time,
  event.event.source as source,
  CumulocitySeverities.WARNING as severity,
  CumulocityAlarmStatuses.ACTIVE as status,
  "The device has high temperature" as text
from EventCreated event;
```

inMaintenanceMode

The inMaintenaceMode() function is a fast way to check if the device is currently in maintenance mode. It takes an ID as parameter and will return a boolean value.

Example:

```
insert into SendEmail
select
  "receiver1@cumulocity.com,receiver2@cumulocity.com" as receiver,
  "cc@cumulocity.com" as cc,
  "bcc@cumulocity.com" as bcc,
  "reply@cumulocity.com" as replyTo,
  "Example mail" as subject,
  "This mail was sent to test the SendEmail stream in Cumulocity" as text
from EventCreated e
where not inMaintenanceMode(e.event.source);
```

replaceAllPlaceholders

To enrich your texts you can either use concatenation

```
insert into SendEmail
select
  "receiver1@cumulocity.com,receiver2@cumulocity.com" as receiver,
  "cc@cumulocity.com" as cc,
  "bcc@cumulocity.com" as bcc,
  "reply@cumulocity.com" as replyTo,
  "Example mail" as subject,
  "An event with the text " || e.event.text || " has been created." as text
from EventCreated e;
```

If the texts get longer and have more values that are dynamically set from the data you can use the replaceAllPlaceholders() function. Another advantage of this function is that you can not only use the current object but also access all information of the device that created the alarm, measurment, event.

In your text string you mark the placeholders with the JsonPath to the value (without the root element $) and surround it by #{}. If you want to access data from the device you start the JsonPath with source.

The function gets called with the string which contains the placeholders and the object which you want to use for filling the placeholders. The source device will then be automatically queried.

Example:

```
create variable string myMailText =
"The device #{source.name} with the serial number #{source.c8y_Hardware.serialNumber}
created an event with the text #{text} at #{time}. The device is located at #
{source.c8y_Address.street} in #{source.c8y_Address.city}.";

insert into SendEmail
select
  "receiver1@cumulocity.com,receiver2@cumulocity.com" as receiver,
  "cc@cumulocity.com" as cc,
  "bcc@cumulocity.com" as bcc,
  "reply@cumulocity.com" as replyTo,
  "Example mail" as subject,
  replaceAllPlaceholders(myMailText, e.event) as text
from EventCreated e;
```

toNumberSetParameter

The toNumberSetParameter() function helps you to configure timer patterns outside of the module. When deploying a module with timer patterns the pattern has to be fixed to the point of deployment and cannot be changed without redeploying the module. It is possible to configure timer patterns with variables if the variables get resolved immediately on deployment. This enables you to store the timer pattern in a ManagedObject. On deployment you load it and fill it with the timer pattern. The toNumberSetParameter() function transforms strings to the NumberSetParameter type which is the input for timer patterns. For more information about timer patterns please check here.

Example:

```
create variable ManagedObject device = findManagedObjectById("12345");
create variable string minuteVal = getString(device, "config.minute");
create variable string hourVal = getString(device, "config.hour");
create variable string dayOfMonthVal = getString(device, "config.day");
create variable string monthVal = getString(device, "config.month");
create variable string dayOfWeekVal = getString(device, "config.weekday");

insert into CreateOperation
select
  "PENDING" as status,
  "12345" as deviceId,
  { "c8y_Restart", {} } as fragments
from
 pattern [every timer:at(toNumberSetParameter(minuteVal),
 toNumberSetParameter(hourVal),
 toNumberSetParameter(dayOfMonthVal),
 toNumberSetParameter(monthVal),
 toNumberSetParameter(dayOfWeekVal))];
```

Advanced use cases

Custom fragments

Cumulocity IoT APIs give you the possibility to structure your data freely. In the Cumulocity Event Language this is also the case. Each of the output streams can be extended with custom fragments. You can add fragments by setting the fragments field in the stream with a list of key, value pairs. The key is the full JsonPath to the value.

```
{
  key1, value1,
  key2, value2,
  key3, value3
} as fragments
```

Example 1:

```
insert into CreateMeasurement
select
  "12345" as source,
  "c8y_TemperatureMeasurement" as type,
  current_timestamp().toDate() as time,
  {
    "c8y_TemperatureMeasurement.T1.value", 1,
    "c8y_TemperatureMeasurement.T1.unit", "C",
    "c8y_TemperatureMeasurement.T2.value", 2,
    "c8y_TemperatureMeasurement.T2.unit", "C",
    "c8y_TemperatureMeasurement.T3.value", 3,
    "c8y_TemperatureMeasurement.T3.unit", "C",
    "c8y_TemperatureMeasurement.T4.value", 4,
    "c8y_TemperatureMeasurement.T4.unit", "C",
    "c8y_TemperatureMeasurement.T5.value", 5,
    "c8y_TemperatureMeasurement.T5.unit", "C"
  } as fragments
from EventCreated;
```

This will result in the following json structure:

```
{
  "type": "c8y_TemperatureMeasurement",
  "time": "...",
  "source": {
    "id": "12345"
  },
  "c8y_TemperatureMeasurement": {
    "T1": {
      "value": 1,
      "unit": "C"
    },
    "T2": {
      "value": 1,
      "unit": "C"
    },
    "T3": {
      "value": 1,
      "unit": "C"
    },
    "T4": {
      "value": 1,
      "unit": "C"
    },
    "T5": {
      "value": 1,
      "unit": "C"
    },
  }
}
```

Example 2:

```
insert into CreateManagedObject
select
  "MyCustomDevice" as name,
  "customDevice" as type,
  {
    "c8y_IsDevice", {},
    "c8y_SupportedOperations", {"c8y_Restart", "c8y_Command"},
    "c8y_Hardware.serialNumber", "mySerialNumber",
    "c8y_Hardware.model", "myDeviceModel",
    "com_cumulocity_model_Agent", {},
    "c8y_RequiredAvailability.responseInterval", 30
  } as fragments
from EventCreated e;
```

This will result in the following json structure:

```
{
  "name": "MyCustomDevice",
  "type": "customDevice",
  "c8y_IsDevice": {},
  "c8y_RequiredAvailability": {
    "responseInterval": 30
  },
  "c8y_SupportedOperations": [
    "c8y_Restart",
    "c8y_Command"
  ],
  "com_cumulocity_model_Agent": {},
  "c8y_Hardware": {
    "model": "myDeviceModel",
    "serialNumber": "mySerialNumber"
  }
}
```

Advanced trigger

Triggering a statement by an arriving event in some stream is not the only possibility. The following sections will cover other ways to trigger statements and combining triggers.

Pattern

Patterns enable you to trigger by combinations or sequences of other triggers. If you have a trigger like this

```
from EventCreated e;
```

the functionality is identical with this trigger using a pattern.

```
from pattern [every e=EventCreated];
```

It is also possible to add filters in the pattern.

```
from pattern [every e=EventCreated(event.type = "myEventType")];
```

You can trigger by joining streams.

```
from EventCreated e unidirectional, AlarmCreated.std:lastevent() a
where e.event.source = a.alarm.source;
```

This will trigger on every EventCreated (defined through the keyword unidirectional) and add the latest AlarmCreated if it is from the same device.

Note: it will not add the latest AlarmCreated of the device but the latest AlarmCreated overall if it is from the same device

You can also trigger by sequences.

```
from pattern[every (e=EventCreated -> a=AlarmCreated(alarm.source = e.event.source))];
```

This will trigger for every pair EventCreated followed by AlarmCreated. It will start on an arriving EventCreated and then finally trigger on an AlarmCreated from the same device. Afterwards it is going to wait for the next EventCreated.

Timer

Instead of using streams for triggering a statement there is also the possibility to trigger by timers. You can either trigger in a certain interval

```
from pattern [every timer:interval(5 minutes)];
```

or as a cron job.

```
// timer:at(minutes, hours, daysOfMonth, month, daysOfWeek, (optional) seconds)
// minutes: 0-59
// hours: 0-23
// daysOfMonth: 1-31
// month: 1-12
// daysOfWeek:  0 (Sunday) - 6 (Saturday)
// seconds: 0-59

from pattern [every timer:at(*, *, *, *, *)]; // trigger every minute
from pattern [every timer:at(*, *, *, *, *, *)]; // trigger every second
from pattern [every timer:at(*/10, *, *, *)]; // trigger every 10 minutes
from pattern [every timer:at(0, 1, *, *, [1,3,5])]; // trigger at 1am every monday, wednesday
and friday
from pattern [every timer:at(0, */2, (1-7), *, *)]; // trigger every 2 hours on every day in
the first week of every month
```

You can also combine timer patterns with other patterns. For example you can check if there was an event within a certain time after another event.

```
from pattern [every e=EventCreated -> (timer:interval(10 minutes) and not a=AlarmCreated)];
```

This will trigger if there is an EventCreated and within 10 minutes there is no AlarmCreated.

Outputs

Outputs give you the possibility to not take every event on a stream into account and to directly control when a statement should output its result. If you have a measurement that is taken every 10 seconds and you want to do calculations with it maybe it is not necessary to calculate with all measurements but only a subset.

```
// will output the last measurement arrived every 1 minute
from MeasurementCreated e
where e.measurement.type = "c8y_TemperatureMeasurement"
output last every 1 minutes;

// will output the first of every 20 measurements arriving
from MeasurementCreated e
where e.measurement.type = "c8y_TemperatureMeasurement"
output first every 20 events;

// will output all 20 measurements after the 20th arrived
from MeasurementCreated e
where e.measurement.type = "c8y_TemperatureMeasurement"
output every 20 events;
```

If you need to take all measurements into account because e.g. you want to calculate the sum of your measurements and you do not want to update it for every new measurement.

```
select
    sum(getNumber(e, "myCustomMeasurement.mySeries.value")),
    last(*)
from MeasurementCreated e
where e.measurement.type = "myCustomMeasurement"
output last every 50 events;
```

Every 50 measurements this statement will output the sum (of all measurements since deployment not just of the 50) and the latest measurement.

Event windows

Event windows give you the possibility to batch together multiple events in a stream for further analysis. There are mainly two ways to create windows:

1. Windows for a certain time

   select avg(getNumber(e, “myCustomMeasurement.mySeries.value”)), last(*) from MeasurementCreated.win:time(1 hours) e where e.measurement.type = “myCustomMeasurement”;

   select avg(getNumber(e, “myCustomMeasurement.mySeries.value”)), last(*) from MeasurementCreated.win:time(1 hours) e where e.measurement.type = “myCustomMeasurement” output last every 1 hours;

The difference between the two statements is that the first one will trigger on every MeasurementCreated and then output the average of the last hour. The second statement only triggers every hour and will only output the last average (calculated when the last MeasurementCreated was received).

2. Windows with a certain amount of events:

   select avg(getNumber(e, “myCustomMeasurement.mySeries.value”)), last(*) from MeasurementCreated.win:length(100) e where e.measurement.type = “myCustomMeasurement”;

   select avg(getNumber(e, “myCustomMeasurement.mySeries.value”)), last(*) from

MeasurementCreated.win:length(100) e where e.measurement.type = “myCustomMeasurement” output last every 100 events;

Windows can also be globally declared:

```
create window MeasurementCreated.win:length(20) as MyMeasurementWindow;

select
  avg(getNumber(e, "myCustomMeasurement.mySeries.value")),
  last(*)
from MyMeasurementWindow e
where e.measurement.type = "myCustomMeasurement";
```

Declaring a window gives you also the possibility of clearing the window.

```
on AlarmCreated delete from MyMeasurementWindow
```

Creating own streams

Creating complex modules is not doable in a single statement. While Cumulocity IoT already provides certain streams it is possible to create additional ones to control your event flow. It is not required to define a stream. If you use a unknown stream name it will automatically be created and defined with the input you set.

```
insert into MyEvent
select
  e.event as e
from EventCreated e;

select e.type from MyEvent e;
```

If you now try to add:

```
insert into MyEvent
select
  e as e
from AlarmCreated e;
```

You will not be able to deploy the statement because the stream MyEvent has already been declared with one variable e of type Event. This statement tries to set a value of type AlarmCreated to e.

You can also explicitly create a new stream.

```
create schema MyEvent(
  e Event
);
```

The general syntax is:

```
create schema StreamName(
  var1Name var1Type,
  var2Name var2Type,
  var3Name var3Type
);
```

You can use every basic Java data type, data types from the imported Java libraries, Cumulocity IoT data types (like Event, Measurement, ManagedObject, …) and other streams.

```
create schema TwoMyEvents(
  firstEvent MyEvent,
  secondEvent MyEvent
);
```

Note: Stream names are unique and once declared (regardless if implicit or explicit) the stream is available in all your modules

Creating own functions

If you want to make more complex calculation than e.g. sum or average you can create your own helper functions and expressions. For writing the function you can use JavaScript as the scripting language. You can also import Java classes into your expressions using importClass.

Examples:

Increasing the given severity (using JavaScript):

```
create expression CumulocitySeverities js:increaseSeverity(severity) [
    importClass (com.cumulocity.model.event.CumulocitySeverities);
    if(severity == CumulocitySeverities.WARNING) {
        CumulocitySeverities.MINOR;
    } else if(severity == CumulocitySeverities.MINOR) {
        CumulocitySeverities.MAJOR;
    } else if(severity == CumulocitySeverities.MAJOR) {
        CumulocitySeverities.CRITICAL;
    } else {
        severity
    }
];
```

Calculating the distance between two geo-coordinates:

```
create expression distance(lat1, lon1, lat2, lon2) [
  var R = 6371000;
  var toRad = function(arg) {
    return arg * Math.PI / 180;
  };
  var lat1Rad = toRad(lat1);
  var lat2Rad = toRad(lat2);
  var deltaLatRad = toRad(lat2-lat1);
  var deltaLonRad = toRad(lon2-lon1);

  var a = Math.sin(deltaLatRad/2) * Math.sin(deltaLatRad/2) +
    Math.cos(lat1Rad) * Math.cos(lat2Rad) * Math.sin(deltaLonRad/2) *
Math.sin(deltaLonRad/2);

  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

  var d = R * c;
  d;
];
```

Variables

You can define variables in your modules.

```
create variable String myEmailText = "Hello World";
create variable List supportedOperationsList = cast({"c8y_Restart", "c8y_Relay"},
java.util.List);
```

You can also dynamically change variable values during runtime

```
create variable String latestEventType;

on EventCreated e set latestEventType = e.event.type;
```

Contexts

Contexts give you the possibility to handle and sort events based on defined values. If you want to create a calculation for some measurements you usually want this to be done for all the devices having this measurement and more importantly separated for each device.

Taking this example

```
select
  avg(getNumber(e, "myCustomMeasurement.mySeries.value")),
  last(*)
from MeasurementCreated.win:length(100) e
where e.measurement.type = "myCustomMeasurement";
```

It will work perfectly for a single device. But as soon as you have two devices the average calculation would be over both devices because all measurements end up in MeasurementCreated. The statement is not aware of how to distinguish the measurements by device. Creating a context is like telling the statement where it can find the information by which it should separate the incoming events.

```
create context DeviceAwareContext
  partition by measurement.source.value from MeasurementCreated;
```

This context definition declares that in the stream MeasurementCreated the context key (by which we want to separate the events) can be found at measurement.source.value which is the ID of the device.

Now we can add the context to the statement:

```
context DeviceAwareContext
select
  avg(getNumber(e, "myCustomMeasurement.mySeries.value")),
  last(*)
from MeasurementCreated.win:length(100) e
where e.measurement.type = "myCustomMeasurement";
```

Now the average will be calculated for each device separately.

The context can only be applied to statements that have an input that is declared in the context. If you have multiple statements that need to be context aware and have different inputs you need to configure each input in the context and where to find the context key.

```
create context DeviceAwareContext
  partition by
    measurement.source.value from MeasurementCreated,
    alarm.source.value from AlarmCreated,
    event.source.value from EventCreated,
    operation.deviceId.value from OperationCreated;
```

You can also create context keys of multiple values:

```
create context DeviceAwareContext
  partition by measurement.source.value and measurement.type from MeasurementCreated;
```

This context will not only create an own partition for each device but also for each measurement type.

Best practises and troubleshooting

CEP statements and Esper scripts

Symptom: Your event processing rules are disabled automatically

Cumulocity IoT monitors the memory usage and workload generated by event processing rules, as well as any errors that are occurring while running those rules. If an event processing rule consumes too much memory or generates too many errors, Cumulocity IoT automatically disables this rule.

Troubleshoot from having rules disabled due to memory consumption:

Make sure that your event processing rules do not keep too many events in windows. For example, if you use “win:keepall()” and many events enter the rule, it will be disabled within a short time. Instead of using “win:keepall()”, try using statements such as “std:lastevent()” that consumes less memory.

Troubleshoot from getting disabled due to too high amount of errors:

Monitor the amount of statements for errors. To see errors you will need to read through the details. In case of an error, correct the statement. An example of a wrong statement: ‘insert into UpdateAlarm select “CLEARED” as status from…’ is regarded as an error statement. An example of a corrected statement is: “insert into UpdateAlarm select “10201” as ID, “CLEARED” as status from…”

Naming statements

The @Name annotation gives you the possibility to name your statements in the module. A name needs to be unique within a single module. This will have a direct effect on the channels in the realtime notifications. It will also help to debug the module in administration UI because the channel name (and therefore the statement name) is printed in the list. If you do not name a statement it will automatically be named “statement_{number of statement}”.

Using device contexts

If you need a device context, it is usually not necessary to put every statement into context. If you use aggregation of measurements most of the time you only need the context in the statement that executes the actual aggregation. It is a useful concept to develop the module completely without the context first and add it at the end to those statements where the context applies.

Splitting modules

If your module gets really big it might be helpful to split it into multiple modules. If you declare schematas or functions they will be available in all modules of your tenant. A good approach can be:

- Module 1: filtering incoming data and load additional data from database
- Module 2: Calculation
- Module 3: Creating data in database

Keep in mind that this will create dependencies within the modules (e.g. module 2 needs a schema defined in module 1). You must avoid circular dependencies.

Number formats

When interacting with measurements the values will be in BigDecimal (if you use getNumber()). When calculating with BigDecimal there will be an error if the result is a repeating decimal. This will result into a null return from built-in functions like avg(). There are two ways to prevent this issue:

1. If you are using built-in functions the easiest way is to cast the BigDecimal to a double value

   avg(cast(getNumber(e, “c8y_TemperatureMeasurement.T.value”), double))

2. If you are calculating yourself (e.g. in a expression) make sure to round or cut the number if you want to stay with BigDecimal.

   getNumber(e, “c8y_TemperatureMeasurement.T.value”).divide(new BigDecimal(3), 5, RoundingMode.HALF_UP)

Examples

Calculating an hourly median of measurements

We are assuming the input data looks like this:

```
{
  "c8y_TemperatureMeasurement": {
    "T": {
      "value": ...,
      "unit": "C"
    }
  },
  "time":"...",
  "source": {
    "id":"..."
  },
  "type": "c8y_TemperatureMeasurement"
}
```

To create the median we need the following parts in the module:

- A context to separate the measurements correctly per device
- A time window over one hour
- An output that returns only the last average calculation every hour
- Everything created as a new measurement

Module:

```
create context HourlyAvgMeasurementDeviceContext
  partition measurement.source.value from MeasurementCreated;

@Name("Creating_hourly_measurement")
context HourlyAvgMeasurementDeviceContext
insert into CreateMeasurement
select
  m.measurement.source as source,
  current_timestamp().toDate() as time,
  "c8y_AverageTemperatureMeasurement" as type,
  {
    "c8y_AverageTemperatureMeasurement.T.value", avg(cast(getNumber(m,
"c8y_TemperatureMeasurement.T.value"), double)),
    "c8y_AverageTemperatureMeasurement.T.unit", getString(m,
"c8y_TemperatureMeasurement.T.unit")
  } as fragments
from MeasurementCreated.win:time(1 hours) m
where getObject(m, "c8y_TemperatureMeasurement") is not null
output last every 1 hours;
```

Creating alarm if the operation was not executed

Operations usually run to a fixed sequence when handled by the device.

- PENDING (after creation)
- EXECUTING (once device received the operation and starts the handling)
- SUCCESSFUL or FAILED (depending on the execution result)

An operation that does not reach SUCCESSFUL or FAILED within a certain time usually indicates an issue (like device lost connection or device got stuck while handling). Even if the operation was not handled successfully the device should update the operation as FAILED. For this example we will use 10 minutes as a acceptable duration for operation handling. We will check for the following sequence:

- OperationCreated
- OperationUpdated for the same operation within 10 minutes that sets the status to either SUCCESSFUL or FAILED

If the second part does *not* appear we will create a new alarm:

```
@Name("handle_not_finished_operation")
insert into CreateAlarm
select
    o.operation.deviceId as source,
    CumulocitySeverities.MAJOR as severity,
    CumulocityAlarmStatuses.ACTIVE as status,
    "c8y_OperationNotFinishedAlarm" as type,
    current_timestamp().toDate() as time,
    replaceAllPlaceholders("The device has not finished the operation #{id} within 10
minutes", o.operation) as text
from pattern [
    every o = OperationCreated
        -> (timer:interval(10 minutes)
        and not OperationUpdated(
            operation.id.value = o.operation.id.value
            and (operation.status in (OperationStatus.SUCCESSFUL, OperationStatus.FAILED))
        ))
];
```

Creating alarms from bit measurements

Devices often keep alarm statuses in registers and can not interpret the meaning of alarms. In this example, we assume that a device just sends the entire register as a binary value in a measurement. A rule must identify the bits and create the respective alarm.

We create three expressions to resolve alarm text, type, and severity for each of the bits.

```
create expression String getFaultRegisterAlarmType(position) [
    switch (position) {
        case 0:
          "c8y_HighTemperatureAlarm";
          break;
        case 1:
          "c8y_ProcessingAlarm";
          break;
        case 2:
          "c8y_DoorOpenAlarm";
          break;
        case 3:
          "c8y_SystemFailureAlarm";
          break;
        default:
          "c8y_FaultRegister" + position + "Alarm";
          break;
    };
];

create expression CumulocitySeverities getFaultRegisterAlarmSeverity(position) [
    importClass(com.cumulocity.model.event.CumulocitySeverities);
    switch (position) {
        case 0:
          CumulocitySeverities.MAJOR;
          break;
        case 1:
          CumulocitySeverities.WARNING;
          break;
        case 2:
          CumulocitySeverities.MINOR;
          break;
        case 3:
          CumulocitySeverities.CRITICAL;
          break;
        default:
          CumulocitySeverities.MAJOR;
          break;
    };
];

create expression String getFaultRegisterAlarmText(position)[
    switch(position) {
        case 0:
          "The machine temperature reached a critical status";
          break;
        case 1:
          "There was an error trying to process data";
          break;
        case 2:
          "Door was opened";
          break;
        case 3:
          "There was a critical system failure";
          break;
        default:
          "An undefined alarm was reported on position " || position || " in the binary fault
register";
          break;
    };
];
```

To analyze the binary measurement value we will interpret it as a string value and loop through each character. The getActiveBits() function will do that and return a list of the bit positions at where the measurement had a “1”. It will not return it as a List but instead as a List where the map structure matches the scheme BitPosition so we can handle it as if it is a stream. This is used as an option to join the stream and trigger an alarm by individual measurement values listed.

```
create schema BitPosition(
  position int
);

create schema MeasurementWithBinaryFaultRegister(
  measurement Measurement,
  faultRegister String
);

create expression Collection getActiveBits(value) [
    importPackage(java.util);
    var bitOnNumbers = new ArrayList();
        var size = value.length;
    for(var no = 0; no < size; no++) {
        if(value.charAt(no) == "1") {
        bitOnNumbers.add(Collections.singletonMap('position', size - no - 1));
            }
    }
    bitOnNumbers;
];

@Name("extract_fault_register")
insert into MeasurementWithBinaryFaultRegister
select
  m.measurement as measurement,
  getString(m, "c8y_BinaryFaultRegister.errors.value") as faultRegister
from MeasurementCreated m
where getObject(m, "c8y_BinaryFaultRegister") is not null;

@Name("creating_alarm")
insert into CreateAlarm
select
    m.measurement.source as source,
        getFaultRegisterAlarmSeverity(bit.position) as severity,
        CumulocityAlarmStatuses.ACTIVE as status,
    m.measurement.time as time,
    getFaultRegisterAlarmType(bit.position) as type,
    getFaultRegisterAlarmText(bit.position) as text
from
    MeasurementWithBinaryFaultRegister m unidirectional,
    MeasurementWithBinaryFaultRegister[getActiveBits(faultRegister)@type(BitPosition)] as
bit;
```

Creating a measurement like this

```
{
    "c8y_BinaryFaultRegister": {
    "errors": {
        "value": 10110
    }
  },
  "time":"...",
  "source": {
    "id":"..."
  },
  "type": "c8y_BinaryFaultRegister"
}
```

will trigger the last statement three times.

- measurement and bit position 1
- measurement and bit position 2
- measurement and bit position 4

and therefore create three alarms.

Consumption measurements

Assuming we have a sensor which measures the current fill level of something and sends the values in a regular basis to Cumulocity IoT we can easily create additional consumption values. Calculating the absolute difference between two measurements can be useful but it will only give you a clear view if the measurements are send always in the same interval. Therefore we will put the absolute difference in relation to the time difference and calculate as a per hour consumption.

We will compare the value and time difference of two adjacent measurements for a device (we will need a context for that).

```
create schema FillLevelMeasurement(
  measurement Measurement,
  value double
);

create schema AdjacentFillLevelMeasurements(
    firstValue double,
    lastValue double,
    firstTime Date,
    lastTime Date,
    source String
);

create context ConsumptionMeasurementDeviceContext
      partition measurement.source.value from FillLevelMeasurement;

create expression double calculateConsumption(firstValue, lastValue, firstTime, lastTime) [
  if (lastTime == firstTime) {
    0;
  } else {
    ((firstValue - lastValue) * 3600000) / (lastTime - firstTime);
  }
];

@Name("filter_fill_level_measurements")
insert into FillLevelMeasurement
select
  m.measurement as measurement,
  cast(getNumber(m, "c8y_WaterTankFillLevel.level.value"), double) as value
from MeasurementCreated m
where getObject(m, "c8y_WaterTankFillLevel") is not null;

@Name("combine_two_latest_measurements")
context ConsumptionMeasurementDeviceContext
insert into AdjacentFillLevelMeasurements
select
  first(m.value) as firstValue,
  first(m.measurement.time) as firstTime,
  last(m.value) as lastValue,
  last(m.measurement.time) as lastTime,
  context.key1 as source
from FillLevelMeasurement.win:length(2) m;

@Name("create_consumption_measurement")
insert into CreateMeasurement
select
  m.lastTime as time,
  m.source as source,
  "c8y_HourlyWaterConsumption" as type,
  {
    "c8y_HourlyWaterConsumption.consumption.value", calculateConsumption(m.firstValue,
m.lastValue, m.firstTime.toMillisec(), m.lastTime.toMillisec()),
    "c8y_HourlyWaterConsumption.consumption.unit", "l/h"
  } as fragments
from AdjacentFillLevelMeasurements m;
```

Using Zementis analytic models

The CEP rule/module below shows how to use Zementis analytic models inside Cumulocity IoT.

We are assuming the input data looks like this:

```
{
  "c8y_SteamMeasurement": {
    "Temperature": {
      "value": ...,
      "unit": "C"
    }
  },
{
  "c8y_TemperatureMeasurement": {
    "Pressure": {
      "value": ...,
      "unit": "bar"
    }
  },
{
  "c8y_TemperatureMeasurement": {
    "Steamoutput": {
      "value": ...,
      "unit": "%"
    }
  },
  "time":"...",
  "source": {
    "id":"..."
  },
  "type": "c8y_TemperatureMeasurement"
}
```

First, a predictive model is created and uploaded via Zementis console. Assume, the model becomes available for data scoring on https://myadapa.zementis.com:443/adapars/apply/model_name endpoint.

CEP module:

```
create constant variable string model_name = "model_name";
create constant variable string model_url =
"https://myadapa.zementis.com:443/adapars/apply/";
create constant variable string auth = "Basic ...";
create constant variable string source_device = "12345";

create expression string js:getLabel(stringObj)[
var zemOutputs = JSON.parse(stringObj).outputs;
output = zemOutputs.pop().Predicted_label;
];

@Name("inputData")
insert into inputDataAll
select
     m.source as source,
     getNumber(m, "c8y_SteamMeasurement.Temperature.value") as `steam.temperature`,
     getNumber(m, "c8y_SteamMeasurement.Pressure.value") as `steam.pressure`,
     getNumber(m, "c8y_SteamMeasurement.Steamoutput.value") as `steam.steamoutput`
from MeasurementCreated m
where
     measurement.source.value = source_device;

@Name("requestZementis")
insert into SendRequest
select
     "GET" as method,
     model_url || model_name || "?record=" || toJSON(m.*) as url,
     auth as authorization,
     "application/json" as contentType,
     m.source as source
from inputDataAll m;

@Name("responseZementis")
insert into CreateEvent
select
     "response_received_" || getString(response, "status") as type,
     getLabel(response.body) as text,
     response.creationTime as dateTime,
     getString(response, "source.value") as source
from ResponseReceived response
where
     getString(response, "source.value") = source_device;

@Name("generateAlarm")
insert into CreateAlarm
select
     response.creationTime as dateTime,
     getString(response, "source.value") as source,
     "cepFailureAlarm" as type,
     "Zementis Test Failure" as text,
     "ACTIVE" as status,
     "MAJOR" as severity
from ResponseReceived response
where
     getString(response, "source.value") = source_device
     and getLabel(response.body) = "0";
```

The Cumulocity IoT CEP module works as follows:

- The data from a specific device is filtered. The measurement values that should be passed for analysis are selected.
- In order to apply the analytic model, an outbound HTTP request is performed to the above Zementis endpoint. The measurement values that need to be analyzed are passed in request URL parameters.
- Depending on the score returned from the model, an alarm is raised.

Study: Circular geofence alarms

This section will give an in-depth example how you can create more complex rules. It will use multiple of the features explained before in the other guide sections.

If you are just starting with the Cumulocity Event Language please take a look at these examples.

Prerequisites

Goal

We want our tracking devices that are continuously sending location events to automatically generate alarms if they move outside a geofence. This geofence will be a circle and should be configurable for each device separately. The alarm will be created at the moment the device moves outside the geofence. While it is moving outside it should not create new alarms because the first one will keep active. As soon as the device moves back into the geofence the alarm will be cleared.

Data model

Location event structure (the part we need):

```
{
  "id": "...",
  "source": {
    "id": "...",
  },
  "text": "...",
  "time": "...",
  "type": "...",
  "c8y_Position": {
    "alt": ...,
    "lng": ...,
    "lat": ...
  }
}
```

How do we store the geofence configuration in the device (the radius will be configured in meters):

```
{
  "c8y_Geofence": {
    "lat": ...,
    "lng": ...,
    "radius": ...
  }
}
```

Additionally we want to enable/disable the geofence alarms for each device without removing the configuration entirely. We will do that by adding/removing “c8y_Geofence” to c8y_SupportedOperations in the device:

```
{
  "c8y_SupportedOperations": [..., "c8y_Geofence", ...]
}
```

Calculation

The device is outside of the geofence if the distance between the current position and the center is bigger than the configured radius of the geofence. What we need is a function that can calculate the difference between to geo-coordinates:

```
create expression distance(lat1, lon1, lat2, lon2) [
    var R = 6371000;
    var toRad = function(arg) {
        return arg * Math.PI / 180;
    };
    var lat1Rad = toRad(lat1);
    var lat2Rad = toRad(lat2);
    var deltaLatRad = toRad(lat2-lat1);
    var deltaLonRad = toRad(lon2-lon1);

    var a = Math.sin(deltaLatRad/2) * Math.sin(deltaLatRad/2) +
        Math.cos(lat1Rad) * Math.cos(lat2Rad) * Math.sin(deltaLonRad/2) *
Math.sin(deltaLonRad/2);

    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

    var d = R * c;
    d;
];
```

The above function will return the distance in meters.

Step 1: Filtering the input

The main input for this module will be events. To discard non- matching events as early as possible we will create a filter in one statement that only matching events will pass. These will be put to a new stream.

```
create schema LocationEvent(
  event Event
);

@Name('Location_event_filter')
insert into LocationEvent
select
  e.event as event
from EventCreated e
where getObject(e, "c8y_Position") is not null;
```

We do not need the additional information of EventCreated and just take the payload (the event) to the next stream.

Step 2: Collecting necessary data

In the next step we need the configuration of the geofence for the calculation and grab it. Together with the event we will forward it into the next stream.

```
create schema LocationEventAndDevice (
    event Event,
    device ManagedObject
);

@Name('fetch_event_device')
insert into LocationEventAndDevice
select
    e.event as event,
    findManagedObjectById(event.source.value) as device
from LocationEvent e;
```

Step 3: Checking if the device supports c8y_Geofence

With the device available we will now check if there is a geofence configured for the device and if it is activated (contains “c8y_Geofence” in c8y_SupportedOperations). To check the c8y_SupportedOperations array we will extract it from the device and use the anyOf() function. This function will loop through all elements and return true if the expression returns true for an element. For the configuration we will just check if the device contains the fragment “c8y_Geofence”

```
create schema LocationEventWithGeofenceConfig (
    event Event,
    eventLat Number,
    eventLng Number,
    centerLat Number,
    centerLng Number,
    maxDistance Number
);

@Name('parse_event_and_device_fragments')
insert into LocationEventWithGeofenceConfig
select
    c.event as event,
  getNumber(e.event, "c8y_Position.lat") as eventLat,
  getNumber(e.event, "c8y_Position.lng") as eventLng,
  getNumber(e.device, "c8y_Geofence.lat") as centerLat,
  getNumber(e.device, "c8y_Geofence.lng") as centerLng,
  getNumber(e.device, "c8y_Geofence.radius") as maxDistance
from LocationEventAndDevice e
where
    getList(e.device, "c8y_SupportedOperations", new ArrayList()).anyOf(el => el =
"c8y_Geofence") = true
    and getObject(e.device, "c8y_Geofence") is not null;
```

Step 4: Creating the trigger

As mentioned earlier the device is outside of the fence if the distance between the current device position and the geofence center is bigger than the configured geofence radius. To trigger the alarm we need 2 events so we can check if within these two events the device entered or left the geofence.

In the first step we calculate the distance with the function mentioned earlier:

```
create schema LocationEventWithDistance (
    event Event,
    maxDistance Number,
    distance Number
);

@Name('calculate_current_distance')
insert into LocationEventWithDistance
select
    e.event as event,
    e.maxDistance as maxDistance,
    cast(distance(centerLat, centerLng, eventLat, eventLng), java.lang.Number) as distance
from LocationEventWithGeofenceConfig e;
```

Now we create a window which holds the last two events

```
create schema LocationEventWithDistancePair (
    firstPos LocationEventWithDistance,
    secondPos LocationEventWithDistance
);

@Name('last_two_positions')
insert into LocationEventWithDistancePair
select
    first(*) as firstPos,
    last(*) as secondPos
from LocationEventWithDistance.win:length(2);
```

The stream LocationEventWithDistancePair now holds all data for checking if we should create the alarm or not.

Step 5: Creating the alarm

To create the alarm we now need two events where the first one has a distance smaller than the radius and the second one has a distance bigger than the radius. This would mean that the device just left the geofence.

```
@Name('create_geofence_alarm')
insert into CreateAlarm
select
    pair.firstPos.event.source as source,
    "ACTIVE" as status,
    current_timestamp().toDate() as time,
    "c8y_GeofenceAlarm" as type,
    "MAJOR" as severity,
    "Device moved out of circular geofence" as text
from LocationEventWithDistancePair pair
where pair.firstPos.distance.doubleValue() <= pair.firstPos.maxDistance.doubleValue()
and pair.secondPos.distance.doubleValue() > pair.secondPos.maxDistance.doubleValue();
```

Step 6: Clearing the alarm

To clear the alarm we just need to switch the condition at the bottom and additionally grab the currently active alarm to get its ID. We do not need to care about whether there is an existing alarm at this point. If there is none this statement will not evaluate successfully because the function would return null.

```
@Name('clear_geofence_alarm')
insert into UpdateAlarm
select
    findFirstAlarmBySourceAndStatusAndType(pair.firstPos.event.source.value, "ACTIVE",
"c8y_GeofenceAlarm").getId().getValue() as id,
    "Device moved back into circular geofence" as text,
    "CLEARED" as status
from LocationEventWithDistancePair as pair
where pair.firstPos.distance.doubleValue() > pair.firstPos.maxDistance.doubleValue()
and pair.secondPos.distance.doubleValue() <= pair.secondPos.maxDistance.doubleValue();
```

Step 7: Creating a device context

Our rule is already working now but there is one issue left: where to send the location event. If a device A sends a location event which is inside its geofence and the following event is from a device B which is outside the geofence it would create an alarm. The alarm would be generated for device A because when creating the alarm we regard the source of the first arriving event as source for the alarm creation. We need to configure that the window which holds the latest two events should only hold events of the same device. If there is an event from another device a new window should be created so there is one window for each device.

This can be achieved with a context. We only need the context at the point where we create the window. The partition for the context should be the device id so that the engine automatically creates a separate context partition for every device.

```
create context GeofenceDeviceContext
   partition by event.source.value from LocationEventWithDistance;
```

Now we can add the context to the statement where we create the window. A context can only be applied to statements where the input of the statement is configured in the context. Otherwise the engine would not know which value to take for creating context partitions.

```
@Name('last_two_positions')
context GeofenceDeviceContext
insert into LocationEventWithDistancePair
select
  first(*) as firstPos,
  last(*) as secondPos
from LocationEventWithDistance.win:length(2);
```

Putting everything together

We can now combine all the parts into one module. The order of the statements does not matter. The only exception is that if you use custom models (like schemas, functions, contexts, variables, ...) you need to declare them before using them.

```
create expression distance(lat1, lon1, lat2, lon2) [
  var R = 6371000;
  var toRad = function(arg) {
    return arg * Math.PI / 180;
  };
  var lat1Rad = toRad(lat1);
  var lat2Rad = toRad(lat2);
  var deltaLatRad = toRad(lat2-lat1);
  var deltaLonRad = toRad(lon2-lon1);

  var a = Math.sin(deltaLatRad/2) * Math.sin(deltaLatRad/2) +
    Math.cos(lat1Rad) * Math.cos(lat2Rad) * Math.sin(deltaLonRad/2) *
Math.sin(deltaLonRad/2);

  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

  var d = R * c;
  d;
];

create schema LocationEvent(
  event Event
);

create schema LocationEventAndDevice (
  event Event,
  device ManagedObject
);

create schema LocationEventWithGeofenceConfig (
  event Event,
  eventLat Number,
  eventLng Number,
  centerLat Number,
  centerLng Number,
  maxDistance Number
);

create schema LocationEventWithDistance (
  event Event,
  maxDistance Number,
  distance Number
);

create schema LocationEventWithDistancePair (
  firstPos LocationEventWithDistance,
  secondPos LocationEventWithDistance
);

create context GeofenceDeviceContext
 partition by event.source.value from LocationEventWithDistance;

@Name('Location_event_filter')
insert into LocationEvent
select
  e.event as event
from EventCreated e
where getObject(e, "c8y_Position") is not null;

@Name('fetch_event_device')
insert into LocationEventAndDevice
select
  e.event as event,
  findManagedObjectById(event.source.value) as device
from LocationEvent e;

@Name('parse_event_and_device_fragments')
insert into LocationEventWithGeofenceConfig
select
  c.event as event,
  getNumber(e.event, "c8y_Position.lat") as eventLat,
  getNumber(e.event, "c8y_Position.lng") as eventLng,
  getNumber(e.device, "c8y_Geofence.lat") as centerLat,
  getNumber(e.device, "c8y_Geofence.lng") as centerLng,
  getNumber(e.device, "c8y_Geofence.radius") as maxDistance
from LocationEventAndDevice e
where
  getList(e.device, "c8y_SupportedOperations", new ArrayList()).anyOf(el => el =
"c8y_Geofence") = true
  and getObject(e.device, "c8y_Geofence") is not null;

@Name('calculate_current_distance')
insert into LocationEventWithDistance
select
  e.event as event,
  e.maxDistance as maxDistance,
  cast(distance(centerLat, centerLng, eventLat, eventLng), java.lang.Number) as distance
from LocationEventWithGeofenceConfig e;

@Name('last_two_positions')
context GeofenceDeviceContext
insert into LocationEventWithDistancePair
select
  first(*) as firstPos,
  last(*) as secondPos
from LocationEventWithDistance.win:length(2);

@Name('create_geofence_alarm')
insert into CreateAlarm
select
  pair.firstPos.event.source as source,
  "ACTIVE" as status,
  current_timestamp().toDate() as time,
  "c8y_GeofenceAlarm" as type,
  "MAJOR" as severity,
  "Device moved out of circular geofence" as text
from LocationEventWithDistancePair pair
where pair.firstPos.distance.doubleValue() <= pair.firstPos.maxDistance.doubleValue()
and pair.secondPos.distance.doubleValue() > pair.secondPos.maxDistance.doubleValue();

@Name('clear_geofence_alarm')
insert into UpdateAlarm
select
    findFirstAlarmBySourceAndStatusAndType(pair.firstPos.event.source.value, "ACTIVE",
"c8y_GeofenceAlarm").getId().getValue() as id,
    "Device moved back into circular geofence" as text,
    "CLEARED" as status
from LocationEventWithDistancePair as pair
where pair.firstPos.distance.doubleValue() > pair.firstPos.maxDistance.doubleValue()
and pair.secondPos.distance.doubleValue() <= pair.secondPos.maxDistance.doubleValue();
```

API reference: Real-time statements

Info: This section has formerly been part of the *Reference guide* but moved here as it only applies to the deprecated Esper CEP engine.

Overview

The API below is not yet published in “/platform” but can be reached using the URL “/cep”.

The real-time statements interface consists of five parts:

- The *cep* API resource returns a URI to a module collection.
- The *module collection* resource retrieves modules and enables creating new modules.
- The *module* resource represents an individual module that can be queried, modified, deployed or undeployed.

Note that for all PUT/POST requests accept header should be provided, otherwise an empty response body will be returned.

Module API

CepApi [application/vnd.com.nsn.cumulocity.cepApi+json]

Name	Type	Occurs	Description
self	URL	1	Link to this resource.
modules	ModuleCollection	1	Collection of all modules.

GET the CepApi resource

Response body: CepApi

Required role: ROLE_CEP_MANAGEMENT_READ

Example request: Retrieve the CepApi resource collection

```
GET /cep
Host: ...
Authorization: Basic ...
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.cepapi+json;ver=...
Content-Length: ...
{
   "self":"<<CepAPI URL>>",
   "modules":{
      "self":"<<ModuleCollection URL>>"
   }
}
```

Module collection

ModuleCollection [application/vnd.com.nsn.cumulocity.cepModuleCollection+json]

Name	Type	Occurs	Description
self	URI	1	Link to this resource.
modules	Collection	0..n	List of modules, see below.
statistics	PagingStatistics	1	Information about paging statistics.
prev	URI	0..1	Link to a potential previous page of modules.
next	URI	0..1	Link to a potential next page of modules.

GET a module collection

Response body: ModuleCollection

Required role: ROLE_CEP_MANAGEMENT_READ

Example request: Get collection of all modules

```
GET /cep/modules
Host: ...
Authorization: Basic ...
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.cepmodulecollection+json;ver=...
Content-Length: ...
{
   "id":"1",
   "self":"CURRENT URL",
   "name":"CepModule 1",
   "application":{
      "application":{
         "id":"3",
         "key":null,
         "name":"energyapp",
         "self":"<<this module application URL>>"
      },
      "self":"<<this module application reference URL>>"
   },
   "lastModified":"2012-01-10T17:15:24+01:00",
   "self": "<<URL to this module>>"
}
```

POST - Create a new Module with statements

Request body: module file Response body: Module (if “Accept” header is provided)
Required role: ROLE_CEP_MANAGEMENT_ADMIN.

Example request:

```
POST /cep/modules
Host: ...
Authorization: Basic ...
Content-Length: ...
Content-Type: multipart/form-data
```

Note: “POST /cep/modules” is a multipart message.

Example file:

```
module testmodule;
@Name('test1')select * from EventCreated.win:time(1 hour)
```

Annotation @Name can be skipped - in this case Cumulocity IoT platform will assign default name to the statement.

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.cepmodule+json;ver=...
{
   "id":"3",
   "lastModified":"2013-06-27T15:37:51.091+02:00",
   "name":"management",
   "self":"http:\/\/localhost:8181\/cep\/modules\/3",
   "status":"DEPLOYED"
}
```

The “id” and “lastModified” of the new module are generated by the server. Response contains also status of module deployment operation.

Module name is considered to be also application name.

Module

Module [application/vnd.com.nsn.cumulocity.cepModule+json]

Name	Type	Occurs	Description	PUT/POST
id	String	1	Uniquely identifies a module.	No
self	URI	1	Link to this resource.	No
lastModified	String	1	Time when module was created or modified.	No
name	String	1	The module name.	POST: Mandatory PUT: Optional
status	String	1	The module status: DEPLOYED, NOT_DEPLOYED (default).	POST: No PUT: Optional

GET Module

Response body: Module

Required role: ROLE_CEP_MANAGEMENT_READ

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.cepmodule+json;ver=...
Content-Length: ...
{
   "id":"1",
   "lastModified":"2013-04-08T14:35:29.879+02:00",
   "name":"the_module",
   "self":"<<URL of cepModule>>",
   "status":"NOT_DEPLOYED"
}
```

GET Module file with statements

Response body: text/plain

Required role: ROLE_CEP_MANAGEMENT_READ

Example response:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: ...

@Name('test1')select * from EventCreated.win:time(1 hour)@Name('test2')select id, count(*)
from MyOffOnStream.win:time(1 hour) group by id;
```

Warning: if given statement has default name assigned by the Cumulocity IoT platform, annotation @Name will not appear.

Update Module

Request body: Module

Response body: Module (only if “Accept” header is provided)

Required : ROLE_CEP_MANAGEMENT_ADMIN

Example Request:

```
PUT /cep/modules/<<moduleId>>
Host: ...
Authorization: Basic ...
Content-Type: application/vnd.com.nsn.cumulocity.cepmodule+json;ver=...
{
  "name" : "the_module",
  "status" : "DEPLOYED"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.cepmodule+json;ver=...
```

Update module file - Modify a Module with statements

Request body: module file Response body: Module (if “Accept” header is provided)
Required role: ROLE_CEP_MANAGEMENT_ADMIN.

Example request:

```
PUT /cep/modules/<<moduleId>>
Host: ...
Authorization: Basic ...
Content-Length: ...
Content-Type: text/plain
```

Example file:

```
module testmodule;
@Name('test1')select * from EventCreated.win:time(1 hour)@Name('test2')select id, count(*)
from MyOffOnStream.win:time(1 hour) group by id;
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.cepmodule+json;ver=...
{
   "id":"3",
   "lastModified":"2013-06-27T15:37:51.091+02:00",
   "name":"management",
   "self":"http:\/\/localhost:8181\/cep\/modules\/3",
   "status":"DEPLOYED"
}
```

During module modification old module is deleted and undeployed and new module is created and deployed, therefore module id changes.

DELETE a module

Request Body: N/A. Required : ROLE_CEP_MANAGEMENT_ADMIN

Example Request: Delete a module

```
DELETE /cep/modules/<<moduleId>>
 Host: [hostname]
 Authorization: Basic xxxxxxxxxxxxxxxxxxx
```

Example Response:

```
HTTP/1.1  204 NO CONTENT
```

Notifications

The real-time notifications allow for receiving almost immediately outputs from statements. They are available on URL

"/cep/notifications", the usage is described in a separate document.

Required role: ROLE_NOTIFICATION_READ

The subscription channel name format

The subscription channel contains the name of the module in which the real-time statement is defined and the name of the real-time statement itself. It has the following structure:

```
/<<moduleName>>/<<statementName>>
```

For example, to subscribe on notifications from a statement “overHeatAlarms” in the module “alarms”, the subscription channel should be the following string:

```
/alarms/overHeatAlarms
```