Events

This section contains the following topics:

- Event handling overview, on page 1
- Define a Kafka event, on page 8

Event handling overview

The event handling mechanism enables CWM to interact with external brokers for handling outbound and inbound events. Workflows can act as either consumers or producers of events which can be used to initiate a new workflow, or signal an existing workflow. On top of that, for each event type that you define in CWM, you can add correlation attributes for filtering events and routing them to the workflow waiting for the event containing specific attribute values.

Event messages need to be defined according to Cloud Events specification. See the event format section for more details.

Brokers and protocols

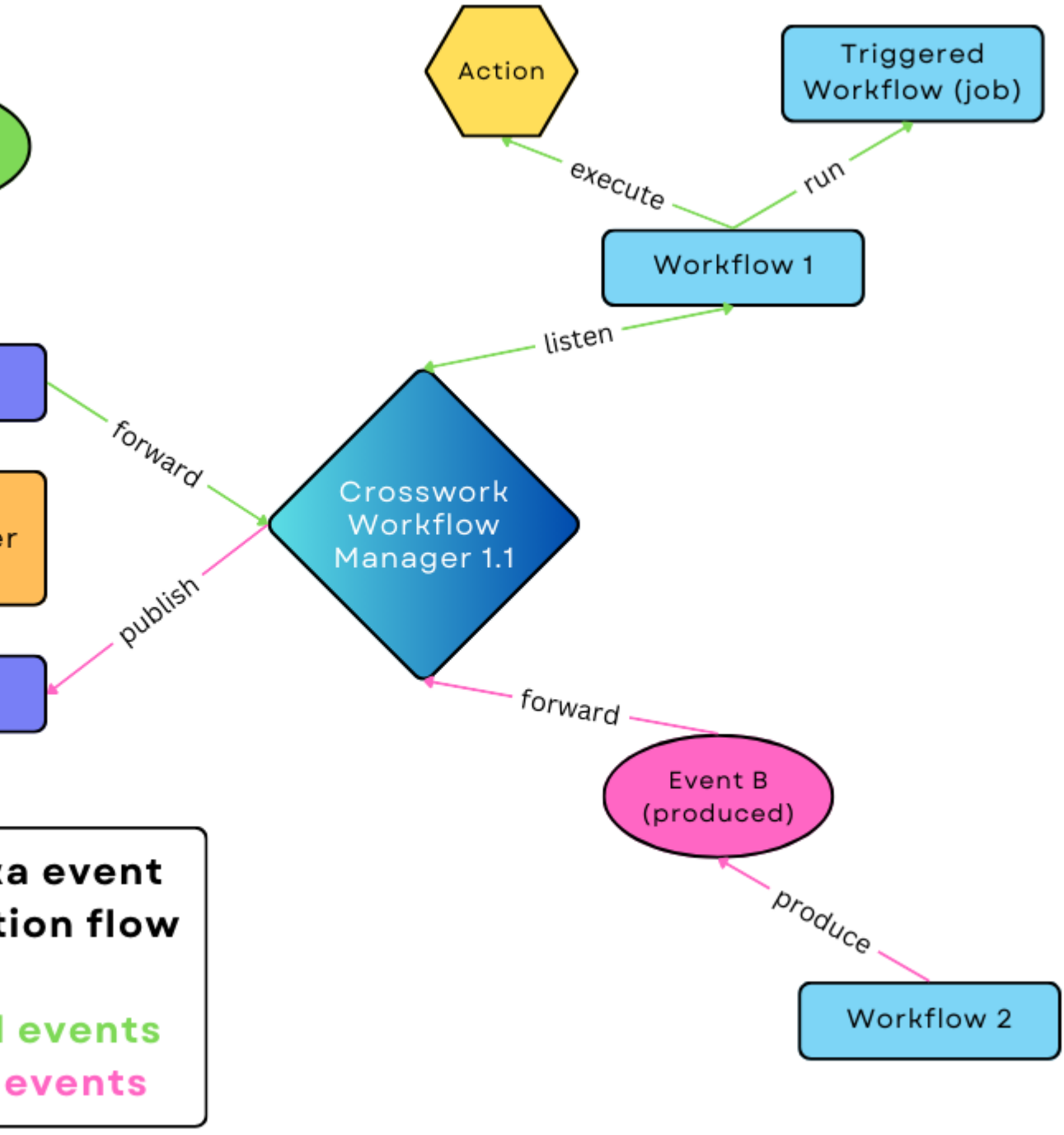
Crosswork Workflow Manager 1.1 supports the Kafka broker and the AMQP and HTTP protocols for handling events. Events can be either **consume** by a workflow running inside CWM (incoming events forwarded by a broker) or **produce** by a running workflow and forwarded to an external system (outgoing events received by a broker).

Note It is important to remember that CWM doesn't act as an event broker itself. It provides means to connect to external brokers to forward messages/events.

Kafka broker

For **consume** event kind, CWM connects to a Kafka broker and listens for a specific event type on a topic. Once an event of the specific type registers to the right topic, CWM retrieves the event data and forwards it to the running workflow. Then, the workflow executes actions defined inside the Event State and/or runs another workflow execution (if selected).

For **produce** event kind, a running workflow produces a single event or a set of events which CWM then forwards to the broker and they get published in the right



The Kafka broker will accept every event message format supported by the language-specific SDK as long as a valid content-type is sent. The list of supported formats is here: https://github.com/cloudevents/spec?tab=readme-ov-file.

AMQP protocol (e.g. RabbitMQ broker)

For **consume** event kind, CWM connects to an AMQP broker and listens for a specific event type on a queue. Similarly to the Kafka broker, when an event of the specific type registers to the right queue, CWM retrieves the event data and forwards it to the running workflow. Then, the workflow executes actions defined inside the Event State and/or runs another workflow execution (if selected).

For **produce** event kind, a running workflow produces a single event or a set of events which CWM then forwards to the broker and they get published in the right queue.

AMQP brokers will accept every event message format supported by the specific SDK as long as a valid content-type is sent. The list of supported formats is here: https://github.com/cloudevents/spec?tab=readme-ov-file.

HTTP protocol

For **consume** event kind, CWM exposes an HTTP endpoint that listens for any incoming events. If an event of specific type comes, it is forwarded to the running workflow that waits for this event type.

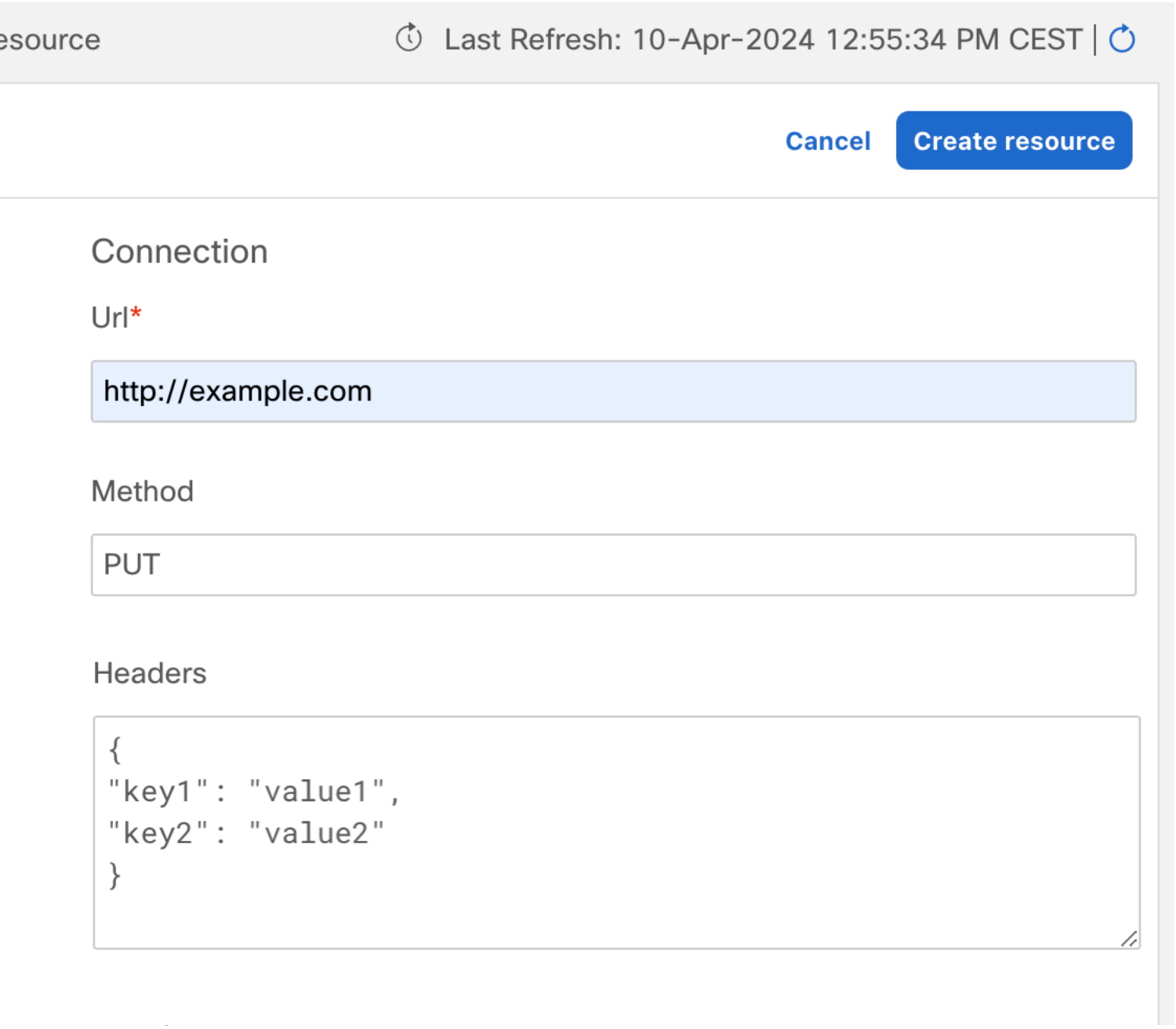
Note When events are consumed, CWM functions as the destination HTTP server. Therefore, the URL of the CWM server is what you effectively provide as the resource for the given HTTP event type.

Event messages need to be HTTP *POST* requests, and message body needs to be in JSON format representing a Cloud Event:

Note Example: `{ "specversion": "1.0", "id": "2763482-4-324-32-4", "type": "com.github.pull_request.opened", "source": "/sensors/tn-1234567/alerts", "datacontenttype": "text/xml", "data": "<test=\"xml\"/>", "contextAttrName": "contextAttrValue" }`

For **produce** kind events, a workflow produces an event in the Cloud Event format and CWM forwards it as an HTTP *POST* request to an HTTP endpoint exposed by an external system. The HTTP endpoint address is a concatenation of the host **URL** defined in the Resource configuration in CWM and the **End point** field of the Event definition inside the workflow definition. Inside the resource configuration, you can change the

request method to *PUT* or other, and add key and value pairs as header (in JSON



Event system configuration

Secret

In event configuration, secrets store credentials needed to enable connection to a broker or endpoint exposed by a third party service that sends or receives events. This includes basic authentication: username and

password. The Secret ID that you provide when creating a secret will be referenced when creating a resource, so you need to add a secret beforehand. To learn how to do it, see the section on adding secrets.

Resource

The resource is where you provide all the connection details (including the secret) needed to reach an event broker or endpoint exposed by a third party service. Depending on the broker/protocol you want to use, you can choose among three default event resource types:

- `system.event.amqp.v1.0.0`
- `system.event.kafka.v1.0.0`
- `system.event.http.v1.0.0`

Notice that there is a different set of configuration fields for each of them:

- For AMQP, provide the **ServerDSN** in the following format: `amqp://localhost:5723`.
- For Kafka:
  - **KafkaVersion**: provide your Kafka version. The standard way to check Kafka version is to run `bin/kafka-topics.sh --version` in a terminal.
  - **Brokers**: provide your Kafka broker addresses in the following format: `["localhost:9092", "192.168.10.9:9092"]`.
  - **OtherSettings**: an editable list with default Kafka setting values. You can modify the values if needed.
- For HTTP:
  - **Produce** event kind: fill in the **URL** field and optionally, **Method** and **Headers** (for example, Client-ID header name and value as a JSON object).

Note Note that **URL** needs to be the address of destination HTTP server, but without the URL path. You will provide the URL path as **End point** when configuring the event type.

  - **Consume** event kind: fill in the **URL** field with the server URL of your CWM instance, for example, `192.168.10.9:9092`.

Note Remember to provide the URL of your CWM instance without the URL path (`/event/http`). You will later use the URL path as the **End point** when configuring the event type.

Event type

Note To create a new event type, you need to have a resource and secret added to CWM.

The following fields are available when adding an event type:

- **Event type name**: the name of your event type. It's later referred to inside the workflow definition.
- **Resource**: a list of resources previously added to CWM.
  - **Event source**: a fully user-defined entry that will be referenced in the workflow definition. Required for `produce` event kind.
  - **End point**: the name of Kafka topic (event stream), AMQP endpoint (terminus), or HTTP URL (Host) path.

Note For HTTP **consume** event kind, provide `/event/http` as your **End point**.

- **Select kind**: a list consisting of two options: `consume` or `produce` event kind.

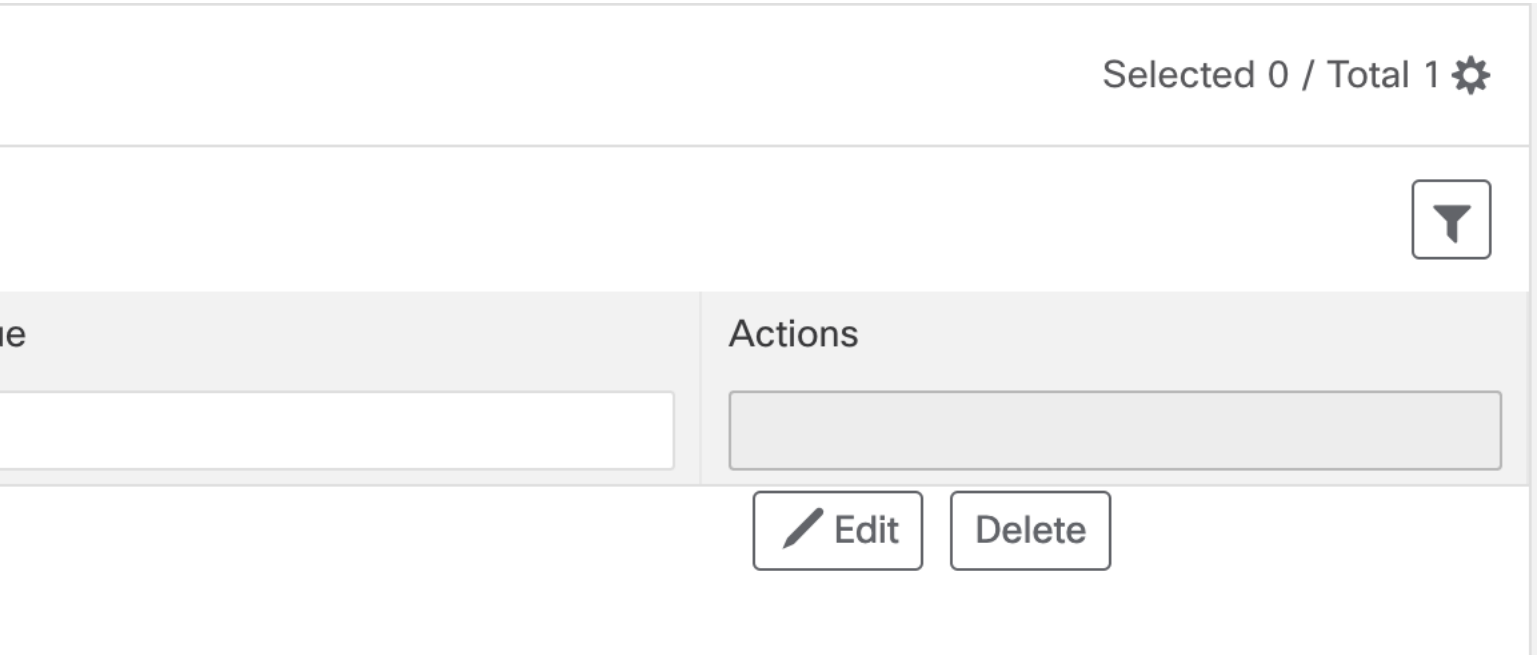
Note The `both` option is not yet supported for {{ version.CWM }}.

- **Start listener** (only for `consume` kind): check it to start listening for the defined event type.
- **Run job** (only for `consume` kind): tick this checkbox if you want to trigger a workflow upon receiving the event. Then select the desired workflow from the list.

Correlation attributes

Optionally, you can set context attributes for your event. They apply only to the `consume` event kind and are used to trigger workflows selectively. You can view them as a kind of custom filters that refine the inbound event data and route them to the right workflows that listen on event types with specific values of correlation attributes.

To add an attribute to your event type, click **Add attribute**, provide attribute name and value, and click



Note Correlation attributes are fully user-defined. They need to match the JSON key and value pair stated inside the Cloud event message that is to be routed to a given workflow.

Event message format

Event messages need to follow the Cloud Events specification format. Minimum viable event message contains the following parameters:

```
{
  "specversion": "1.0",
  "id": "00001",
  "type": "com.github.pull_request.opened",
  "source": "/sensors/tn-1234567/alerts"
}
```

The message can carry additional parameters like `"datacontenttype"`, `"data"`, and correlation context attribute name (`contextAttrName` in this example) :

```
  {
  "specversion": "1.0",
  "id": "2763482-4-324-32-4",
  "type": "com.github.pull_request.opened",
  "source": "/sensors/tn-1234567/alerts",
  "datacontenttype": "text/xml",
  "data": "<test data=\"xml\"/>",
  "contextAttrName": "contextAttrValue"
}
```

Workflow event definition and state

In the workflow definition, there are two major syntactical elements that you use to handle the events that the workflow will be waiting on. These are:

- Event definition: used to define the event type and its properties:

```
{
   "name": "applicant-info",
   "type": "org.application.info",
   "source": "applicationssource",
   "correlation": [
    {
      "contextAttrName": "applicantId"
    }
   ]
}
```

- Event state: used to define actions to be taken when the event occurs:

```
{
"name": "MonitorVitals",
"type": "event",
"onEvents": [
        {
          "actions": [
            {
              "functionRef": {
                "refName": "uppercase",
                "arguments": {
                  "input": {
                    "in": "patient ${ .patient } has high temperature"
                  }
                }
              }
            }
          ],
          "eventRefs": [
            "HighBodyTemperature"
          ]
        }
      ]
}
```

Define a Kafka event

Prerequisites

- A set-up Kafka service (or RabbitMQ with AMQP 1.0 plugin in case of AMQP, or any HTTP Client).
- CWM 1.1 installed using OVA.

Step 1: Create Kafka secret and resource

To enable a secure connection to the Kafka service, you need to create a secret with Kafka credentials and a resource with connection details. Here's how to do it:

Create secret

Procedure

Step 1 In CWM, navigate to the **Admin** -> **Secrets** tab.

Step 2 Click **Add Secret**.

Step 3 In the **New secret** view, specify the following:

a) Secret ID: `KafkaSecret`

b) Secret type: `basicAuth`

Step 4 After selecting the secret type, a set of additional fields is displayed under the Secret type details section. Fill in the fields:

a) password: password used for logging in to Kafka.

b) username: username used for logging in to Kafka.

Step 5 Click **Create Secret**.

Create resource

Procedure

Step 1 In CWM, navigate to the **Admin** -> **Resources** tab.

Step 2 Click **Add Resource**.

Step 3 In the **New resource** window, specify the following:

a) Resource name: `KafkaResource`

b) Resource type: `cisco.cwm.kafka.v1.0.0` (or `cisco.cwm.amqp.v1.0.0` or `cisco.cwm.http.v1.0.0` if you use these protocols instead)

c) Secret ID: `KafkaSecret`

d) Connection:

- **KafkaVersion**: provide your Kafka version. The standard way to check this is to run `bin/kafka-topics.sh --version` in a terminal.
- **Brokers**: provide your Kafka broker address in the following format: `["localhost:9092"]`.
- **OtherSettings**: an editable list with default Kafka setting values. You can modify the values if needed.

Note
Connection setting differ in case of **AMQP** and **HTTP** resource types:

- For AMQP, provide the **ServerDNS** in the following format: `amqp://localhost:5723`. - For HTTP, provide the **URL** and additional **headers** (for example, Client-ID header name and value). Note that URL needs to be your host address but without the URL path. This you will specify as **End point** when configuring the resource type.

Step 4 Click **Create resource**.

Step 2: Add event type to CWM

When you have the secret and resource in place, it's time to specify the type of event that will be consumed or produced by CWM.

Procedure

Step 1 In the CWM UI, select the **Admin** tile from the navigation menu on the left.

Step 2 In the **Event system** panel, click **Add event type**.

Step 3 In the **New event type** modal, provide the required input:

a) **Event type name**: provide name for your event type. You will later refer to it inside the workflow definition.

b) **Resource**: from the list, select `KafkaResource`.

c) **Event source**: define your event source. It's fully user-defined and will be referenced in the workflow definition. Required for `produce` event kind.

d) **End point**: for Kafka, provide your Kafka topic (event stream). For AMQP, provide endpoint (terminus). For HTTP, provide URL (Host) path.

e) **Select kind**: from the list, select `consume`.

Note
Use `Produce` to define an event to be produced by a workflow and consumed by another system. In this case, the remaining **Step 2** settings presented below this point won't apply. The `both` option is not yet supported for CWM 1.1.

f) **Start listener**: click it to start listening for the defined event type.

g) **Run job**: tick this checkbox if you want to trigger a workflow upon receiving the event. Then select the desired workflow from the list.

h) **Correlation context attributes**: optionally, you can set context attributes for your event. They apply only to the `consume` event kind and are used to trigger workflows selectively. You can view them as a kind of custom filter that refines the inbound event data and triggers actions defined inside a "listening" workflow on the basis of your context attributes.

i) Click **Add attribute** and provide attribute name and value (fully user-defined).

Step 4 Click **Create Event type**.

Step 3: Define event in a workflow

Now that we have the event type added, we can create a workflow that registers for this event type and executes an action when the event is received by CWM. For this purpose, we'll need to define the event using an Event definition and specify the Event state and define actions to be taken when the event occurs. For example purposes, let's take a scenario where a router overheating alarm (inbound event) triggers the workflow event state and two remediation actions are executed:

```
{
  "id": "HighRouterTempWorkflow",
  "name": "Router Overheating Alarm Workflow",
  "start": "RemediateHighTemp",
```

```
  "events": [
    {
      "kind": "consumed",
      "name": "HighRouterTemp",
      "type": "HighRouterTemp",
      "source": "monitoring.app"
    }
  ],
  "states": [
    {
      "end": {
        "terminate": true
      },
      "name": "RemediateHighTemp",
      "type": "event",
      "onEvents": [
        {
          "actions": [
            {
              "functionRef": {
                "refName": "DispatchTech",
                "contextAttributes": {
                  "RouterIP": "${ .RouterIP }"
                },
                "resultEventTimeout": "PT30M"
              }
            },
          ],
          "eventRefs": ["HighRouterTemp"],
           },
           {
           "actions": [
            {
              "functionRef": {
                "refName": "MoveTraffic",
                "contextAttributes": {
                  "RouterIP": "${ .RouterIP }"
                },
                "resultEventTimeout": "PT30M"
              }
            }
          ],
          "timeouts": {
            "actionExecTimeout": "PT60M"
          }
        }
      ]
    }
  ],
  "version": "1.0.0",
  "description": "Remediate router overheating",
  "specVersion": "0.8"
}
```

Note Note that the example is not a complete workflow. It presents a sample of how you can define an event inside a workflow and act on it.