



PUBLIC

SAP Data Hub 2.7

Document Version: 2.7.7 – 2021-Apr-14

Modeling Guide for SAP Data Hub

Content

- 1 Modeling Guide for SAP Data Hub. 5**
- 2 Introduction to the SAP Data Hub Modeler. 7**
 - 2.1 Log in to SAP Data Hub Modeler. 8
 - 2.2 Navigating the SAP Data Hub Modeler. 9
- 3 Creating Graphs. 11**
 - 3.1 Execute Graphs. 16
 - Parameterize the Process Execution. 17
 - 3.2 Schedule Graph Executions. 18
 - Cron Expression Format. 19
 - 3.3 Maintain Resource Requirements for Graphs. 21
 - 3.4 Groups, Tags, and Dockerfiles. 23
 - 3.5 Execution Model. 25
- 4 Monitoring the SAP Data Hub Modeler. 27**
 - 4.1 Monitor the Graph Execution Status. 28
 - Graph Execution. 29
 - Graph States. 30
 - Process States. 31
 - 4.2 Trace Messages. 31
 - Trace Severity Levels. 33
 - 4.3 Using SAP Data Hub Monitoring. 33
 - Log in to SAP Data Hub Monitoring. 34
 - Monitoring with the SAP Data Hub Monitoring Application. 35
 - Schedule Graph Executions with SAP Data Hub Monitoring. 39
 - 4.4 Downloading Diagnostic Information for Graphs. 42
 - Diagnostic Information Archive Structure and Contents. 43
- 5 Tutorials. 48**
 - 5.1 Create a Graph to Execute a TensorFlow Application. 48
 - Example Graphs for TensorFlow. 50
 - 5.2 Creating a Graph to Perform Text Analysis. 52
 - Example Graph for Simple Text Analysis. 52
 - Example Graph for Text Analysis of Files. 54
- 6 Using Scenario Templates. 59**
 - 6.1 Load Data from Data Lake to Database (HANA/VORA). 59

6.2	E(T)L from Database.	60
6.3	ABAP with Data Lakes.	61
6.4	BW with Data Lakes.	62
	Register New Data from Data Lake to Vora Table.	63
	Setup BW to Access Vora Table.	65
	Create an Example Scenario.	66
	Query Vora Table Using Query Objects.	71
	Insert New Data from ODS View to a Data Store (Advanced) Object.	72
	Retrieve Data from BW.	75
6.5	Data Processing with Scripting Languages.	75
7	Creating Operators.	78
7.1	Operators.	82
	Operator Details.	83
7.2	Port Types.	89
	Compatible Port Types.	91
	Table Messages.	93
8	Working with the Data Workflow Operators.	97
8.1	Execute an SAP BW Process Chain.	101
8.2	Execute an SAP HANA Flowgraph Operator.	102
8.3	Transform Data	104
	Configure the Data Source Node	107
	Configure the Data Target Node.	108
	Configure the Projection Node.	110
	Configure the Aggregation Node.	112
	Configure the Join Node.	114
	Configure the Union Node.	116
	Configure the Case Node.	117
8.4	Execute an SAP Data Hub Pipeline.	118
8.5	Execute an SAP Data Services Job.	120
8.6	Transfer Data	122
	Transfer Data from SAP BW to SAP Vora or Cloud Storage.	123
	Transfer Data from SAP HANA to SAP Vora or Cloud Storage.	130
8.7	Control Flow of Execution.	133
8.8	Control Start and Shut Down of Data Workflows.	134
8.9	Send E-mail Notifications.	135
9	Integrating SAP Cloud Applications with SAP Data Hub.	137
10	Service-Specific Information.	140
10.1	Alibaba Cloud Object Storage Service (Alibaba Cloud OSS).	140
10.2	Amazon S3.	144

10.3	Azure Data Lake (ADL)	147
10.4	Google Cloud Storage (GCS)	149
10.5	Hadoop File System (HDFS)	152
10.6	Local File System (File)	155
10.7	Windows Azure Blob Storage (WASB)	155
10.8	WebHDFS	158
11	Change Data Capture (CDC)	162
12	Subengines	163
12.1	Working with the C++ Subengine to Create Operators	164
	Getting Started	165
	Creating an Operator	166
	Logging and Error Handling	168
	Port Data	169
	Setting Values for Configuration Properties	171
	Process Handlers	172
	API Reference	174
12.2	Working with Python 2.7 and Python 3.6 Subengines to Create Operators	186
	Normal Usage	188
	Advanced Usage	190
12.3	Working with the Node.js Subengine to Create Operators	203
	Node.js Operators and OS Processes	203
	Use Cases for the Node.js Subengine	204
	The Node.js Subengine SDK	205
	Data Types	207
	Additional Type Specific Constraints	208
	Develop a Node.js Operator	209
	Project Structure	210
	Project Files and Resources	211
	Logging	213
12.4	Working with Flowagent Subengine to Connect to Databases	214
13	Create Dockerfiles	218
13.1	Docker Inheritance	220
14	Create Types	221
15	Creating Scenes for Graphs	224
15.1	Understand Key Concepts	224
15.2	Use the Scene Editor	229
15.3	Develop Controls	233
	Write Scripts	236

1 Modeling Guide for SAP Data Hub

The Modeling Guide contains information on using the SAP Data Hub Modeler.

The SAP Data Hub Modeler helps create data processing pipelines (graphs), and provides a runtime and design time environment for data-driven scenarios. The tool reuses existing coding and libraries to orchestrate data processing in distributed landscapes. This guide is organized as follows:

Title	Description
Introduction to the SAP Data Hub Modeler	Provides an overview of the tool and describes certain business use cases.
Creating Graphs	Explains the steps for creating, configuring, and executing a graph in the modeler.
Monitoring the SAP Data Hub Modeler	This section describes the different monitoring options available within the tool to monitor the working of the SAP Data Hub Modeler.
Tutorials	Includes tutorials and example graphs to help users understand and work with the SAP Data Hub Modeler.
Using Scenario Templates	Describes a selection of common scenarios and how they can be implemented using operators and graphs provided by SAP Data Hub.
Create Operators	Describes operators and explains the steps for creating, configuring, and using the operators in a graph.
Working with the Data Workflow Operators	SAP Data Hub Modeler categorizes certain operators as Data Workflow operators. This section describes the data workflow operators and how to model graphs with the data workflow operators.
Subengines	Describes working with the C++, Python2.7, and Python3.6 subengines.
Create Dockerfiles	Explains the steps for creating Dockerfiles, defining tags, and building docker images.
Create Types	Explains the steps for creating and defining the types and type properties.
Creating Scenes for Graphs	Describes how to create Scenes, which are visualizations for graphs, using the Scene Editor.

Creating operators, types, Dockerfiles, or to work with the subengines in the SAP Data Hub Modeler requires some expertise and programming skills. If you are new to the topic, we recommend that you start your learning

journey by creating graphs with only the built-in (predefined) operators that the modeler provides. You can also use the tutorials in this guide to get started and learn more about the different capabilities of the SAP Data Hub Modeler.

Related Information

[Introduction to the SAP Data Hub Modeler \[page 7\]](#)

2 Introduction to the SAP Data Hub Modeler

The SAP Data Hub Modeler tool is based on the Pipeline Engine that uses a flow-based programming paradigm to create data processing pipelines (graphs).

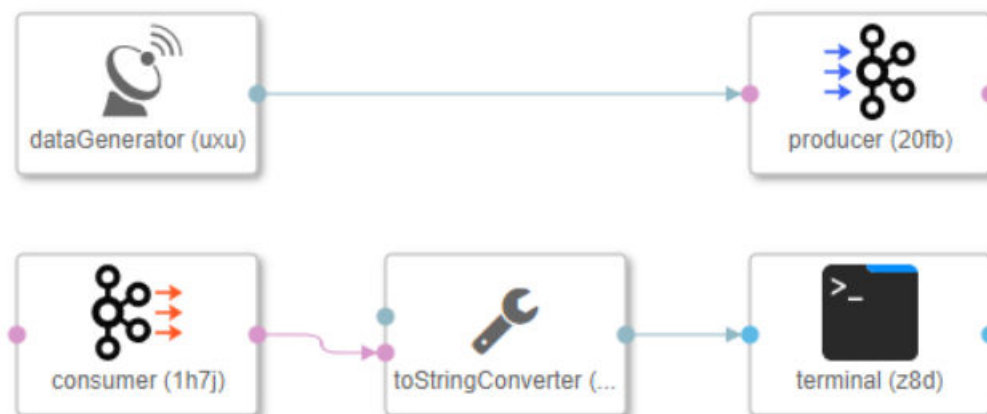
Big Data applications require advanced data ingestion and transformation capabilities. Some common use cases are to:

- Ingest data from source systems. For example, database systems like SAP HANA, message queues like Apache Kafka, or data storage systems like HDFS or S3.
- Cleanse the data.
- Transform the data to a desired target schema.
- Store the data in target systems for consumption, archiving, or analysis.

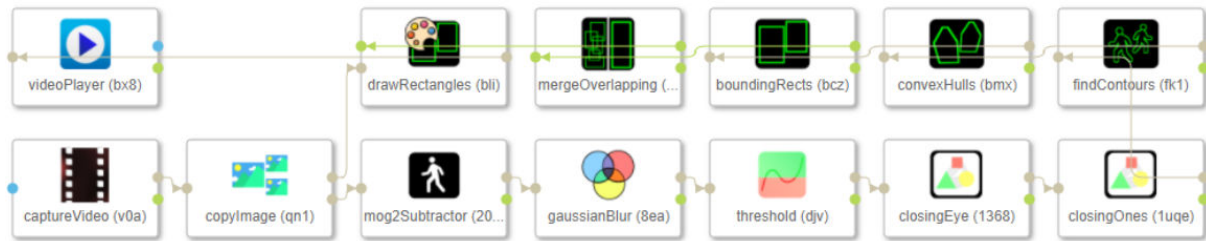
You can model data processing pipelines as a computation graph, which can help to achieve the required data ingestion and transformation capabilities. In this graph, nodes represent operations on the data, while edges represent the data flow.

The Modeler application helps you to model graphs using graphical capabilities and execute them. It provides a runtime component to execute graphs in a containerized environment that runs on Kubernetes.

The Modeler also provides predefined operators, which you can use for many productive business use cases. These operators can help you define graphs, including non terminating, non connected, or cyclic graphs. The following example shows a simple interaction with Apache Kafka. The graph consists of two subgraphs. The first subgraph generates some data and writes the data into a Kafka message queue. The second subgraph reads the data from Kafka, converts it to string and prints the data to a terminal.



You can also create generic data processing pipelines. The following example shows a graph that detects objects in a video stream.



Related Information

[Log in to SAP Data Hub Modeler \[page 8\]](#)

[Navigating the SAP Data Hub Modeler \[page 9\]](#)

2.1 Log in to SAP Data Hub Modeler

You can access the SAP Data Hub Modeler from the SAP Data Hub Launchpad or directly launch the application with a stable URL.

Procedure

1. Launch the SAP Data Hub Launchpad user interface in a browser using one of the following URLs:

- For on-premise installations (if TLS is enabled):

```
https://<node-ip>:<port>
```

where

- **<node-ip>**: The IP address of any node in the cluster.
- **<port>**: The SAP Data Hub Launchpad port number.
- For cloud installations:

```
https://<domain>:<ingress vsystem port>
```

where

- **<domain>**: The domain is specified when you expose the application externally. For more information, see [Configuring SAP Data Hub Foundation on Cloud Platforms](#)
- **<ingress vsystem port>**: To obtain the `ingress vsystem port`, execute the following command:

```
kubectl get ingress -n $NAMESPACE
```

The welcome screen appears where you can enter the login credentials.

2. Log into the SAP Data Hub Launchpad application using the following information:

- Tenant ID
- Your user name
- Your password

The SAP Data Hub Launchpad opens and displays the initial home page. User details and the tenant ID are displayed in the upper-right area of the screen. The home page displays all applications available in the tenant.

3. On the home page, choose *Modeler*.

The application UI opens displaying the initial screen.

2.2 Navigating the SAP Data Hub Modeler



In the SAP Data Hub Modeler, you can choose from different types of panes and toolbars to perform your actions.

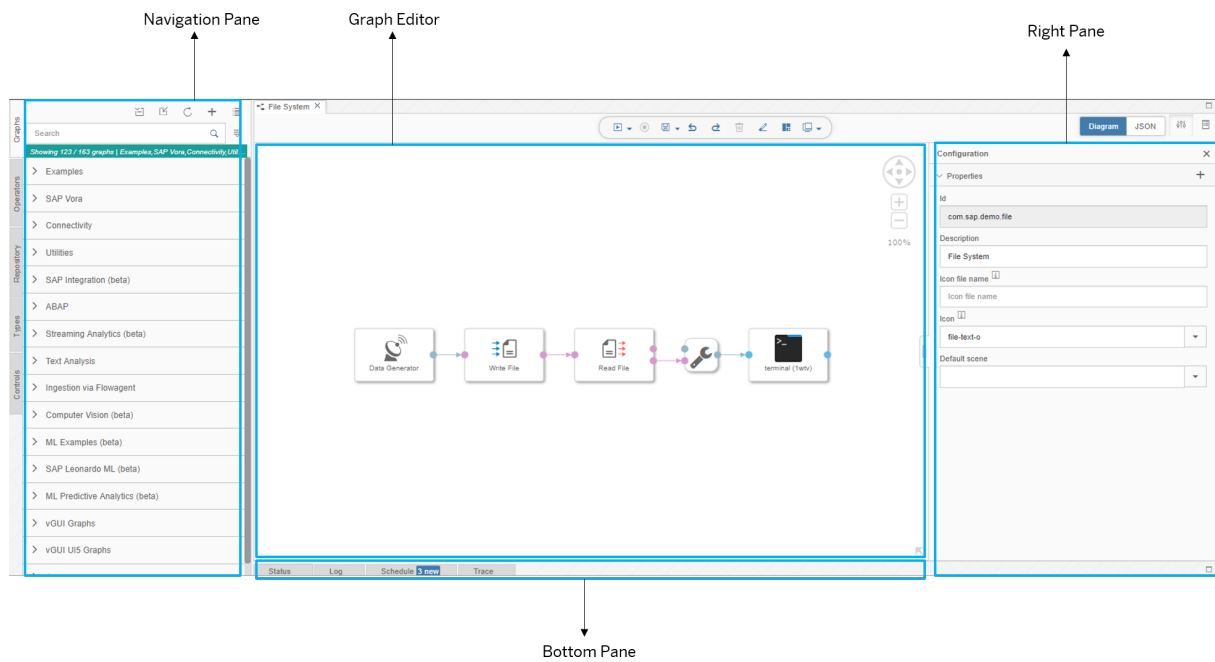
The Modeler user interface layout has the following panes or toolbar:

- **Graph editor:** Use this editor to create the graph with one or more operators.
- **Navigation pane:** Use this pane to access operators, graphs, repository, and the types.
 - In the *Graphs* tab, you can access the built-in graphs and other user created graphs organized under various categories. You can also create new graphs from here.
 - In the *Operators* tab, you can access the built-in operators and other user created operators organized under various categories. You can also create new operators from here.
 - In the *Repository* tab, you can create and work with different Modeler objects such as graphs, operators, types, and so on. You can also create new folders, import auxiliary files or solutions, or export folders as .tgz files or as vSolution files.

i Note

The SAP Data Hub Modeler provides individual Dockerfiles to create containerized environments for the operator groups. The Dockerfiles are selected using a tag-matching mechanism.

- In the *Types* tab, you can access all type definitions or create new types.
- In the *Controls* tab, you can access different UI elements to create user interfaces for graphs using the scene editor.
- **Bottom pane:** The bottom pane consists of a *Status* pane, *Log* pane, *Schedule*, and a *Trace* pane for the modeler. You can use these panes to monitor the status of the graph execution, trace messages based on severity levels, monitor graph schedules, and view various logs that the application creates for graph execution, respectively.
- **Editor toolbar:** The graph editor includes a toolbar, which you can use to perform operations on the graph, for example, to save and execute a graph, to perform an auto-layout of operators, and more. Use this pane to define the configuration parameters (select ) for the graph, groups, and operators. The  icon in this pane provides detailed documentation (help information) for various operators and example graphs that the application provides.



Related Information

- [Creating Graphs \[page 11\]](#)
- [Creating Operators \[page 78\]](#)
- [Create Dockerfiles \[page 218\]](#)
- [Monitor the Graph Execution Status \[page 28\]](#)
- [Create Types \[page 221\]](#)

3 Creating Graphs

A graph is a network of operators connected to each other using typed input ports and output ports for data transfer. Users can define and configure the operators in a graph.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane bar, choose **+** (Create Graph).


The application opens an empty graph editor in the same window, where you can define your graph.

4. Select operators.

A graph can contain a single operator, or a network of operators based on the business requirement.

- a. In the navigation pane, choose the *Operators* tab.

→ Tip

In the application UI, the operators are grouped under specific categories. If you want to customize the list of operators that the application displays, in the navigation bar, choose  (Customize Visible Categories).

- b. Select the required operator.

You can also use the search bar to search and select the required operator. Double-click the operator (or drag and drop it to the graph editor) to include it as a process for the graph execution.

5. Configure the operators.

Operators are defined with default configuration parameters. You can customize the operator by providing own values to the parameters, or you can also define new additional configuration parameters.

- a. In the graph editor, select the required operator.

- b. In the editor bar, choose  (Open Configuration).

The application displays the default configuration parameters defined for the selected operator. You can customize the operator by providing new values to the default parameters. Depending on the operator type, you can also specify values to the following configuration parameters.

Configuration Parameter	Description
subengines	If multiple implementations exist for the operator, you can select the required subengine in which you want to execute the operator. The default value is <i>main</i> (Pipeline Engine). To select the required engine, in the <i>subengines</i> section, choose + and in the dropdown list, select the required subengine.

6. (Optional) Define new configuration parameters.

You can add and define new configuration parameters for an operator:

- a. In the *Configuration* pane, choose **+** (Add Parameter).
- b. In the *Add Property* dialog box, provide a name for the new configuration parameter.
- c. Select the property type.

Select *Text*, *JSON*, or *Boolean*, based on whether the parameter value is a string value, JSON value, or a Boolean value.

- d. In the *Value* text field, provide a value.
- e. Choose *OK*.

7. (Optional) Validate configurations.

If the operator has a type scheme associated with it, then the application enables you validate the configuration parameters values against the conditions defined in the schema. For example, you can validate mandatory fields, minimum length or maximum length, value formats, regular expression, and so on, which are defined in the type schema.

i Note

The validation is based on the constraints defined in the scheme and does not validate all configuration parameter values.

- a. In the *Configuration* pane, choose the *Validate* button.

The application runs validations on the configuration parameter values and displays validation errors, if any.

8. (Optional) Add new ports.

For JavaScript operators, Python operators, multiplexer operators, and other extensible operators, you can define more input and output ports.

- a. In the graph editor, right-click the operator (multiplexer and JavaScript) and choose *Add Port*.
- b. Provide a port name and a type.
- c. Select whether the port is an input or an output port.
- d. Choose *OK*.

For more information on valid port types, see the topic **Port Types**.

9. Connect operators.

Select an output port of an operator and drag the cursor to an input port of another operator.

The application highlights all input ports, based on the output port type, to which you can connect the operator.

10. (Optional) Creating groups.

You can partition a graph into many groups. Each group's subgraph will run in a different Docker container that can be assigned to different cluster nodes. On the other hand, operators inside the same group always run in the same node. You can configure each group with a different restart policy, tags or multiplicity. For more information on the use cases of groups, see [Groups, Tags, and Dockerfiles \[page 23\]](#).

- a. Using the **SHIFT** key, select the multiple operators that you want to group together.

i Note

You can also create a group with just one operator.

- b. In the context menu, choose *Group*.

- c. If you want to ungroup the operators, right-click the group and choose *Ungroup*.

i Note

By ungrouping, you can still retain the operators on the editor, but you can't retain the configurations defined for the group. If you want to remove the group along with the operators, then right-click the group and choose *Remove*.

- d. If you want to expand an existing group to include more operators, click the mouse cursor and drag the group boundary over the required operators.

Alternatively, you can also drag and drop the operators inside the existing group region.

11. (Optional) Configure groups.

Like operators, each group (subgraph) is associated with certain configuration parameters. You can provide your own values to the configuration parameters, or you can also define new additional configuration parameters for the group.

- a. In the editor, select a group (subgraph).

- b. In the editor bar, choose .

The application displays the predefined configuration parameters available for the group. You can override the default values for the predefined configuration parameters.

Configuration Parameter

Description

Configuration Parameter	Description
description	Provide a description for the group.
restartPolicy	<p>The restart policy describes the behavior of the cluster scheduler when a group execution results in a crash. In the dropdown list, select a value for this property. If you do not specify any value, the application uses the default value <i>never</i>.</p> <p>If you set the restart policy to <i>never</i>, the cluster scheduler does not restart the crashed group. The crash results in the final state of the graph as being 'dead'. If the restart policy is set to <i>restart</i>, then the cluster scheduler restarts the group execution. The restart changes the state of the graph from <i>dead</i> to <i>pending</i> and then <i>running</i>.</p>
tags	<p>Tags describe the runtime requirements of groups.</p> <p>Choose + (Add Tag) to define tags for the group. In the dropdown list, select the required tag and version. You can associate the group with more than one tag. For more information on tags, see Groups, Tags, and Dockerfiles [page 23]</p>
multiplicity	Specify the multiplicity as an integer value. For example, a multiplicity of 3 implies that the application executes 3 instances of this group at runtime. The application uses the round-robin fashion to send the data arriving at a group with multiplicity larger than one.

⚠ Caution

When you restart a group, a single message per inbound connection can be lost.

- c. If you want to define new configuration parameters for the group, in the *Configuration* pane, choose + (Add Parameter).

- d. In the *Add Property* dialog box, provide a name for the new property.
- e. Select the property type.
Select *Text*, *JSON*, or *Boolean*, based on whether the property value is a string value, JSON value, or a Boolean value.
- f. In the *Value* text field, provide a value.
- g. Choose *OK*.

12. (Optional) Configure the graph.

You can provide a description and select a display icon that the application must use for the graph.

- a. In the editor bar, choose  .

→ Remember

The focus of your cursor must be on the empty area in the graph editor canvas. If you select any operator or a group before selecting the *Configuration* pane, then the displays the configuration parameters associated with the operator or group and not the graph.

- b. Provide values to the configuration parameters.

Configuration Parameter	Description
description	Description of the graph
iconsrc	Path to the graph display icon that the application must use.
icon	In the <i>icon</i> dropdown list, you can select the required icon for the graph. The application uses this icon for display only if you have not provided any value to <i>iconsrc</i> .

i Note


In the *Configuration* pane, choose + (Add Parameter) to define new configuration parameters for the graph.

13. (Optional) Export the JSON definition.


- a. After creating the graph, if you want to export the JSON definition for the graph, in the navigation pane, select the *Graphs* tab.
- b. Right-click the required graph and choose the *Export* menu option.


14. (Optional) Refer to operator documentation.

SAP Data Hub provides documentation for its built-in operators and example graphs. The documentation provides more information on how to use the operator or a graph.

Documentation	Steps
Graphs	<p>Open the graph in the graph editor and in the editor bar, choose  (Show Documentation).</p> <div style="border: 1px solid #ccc; padding: 5px; background-color: #f9f9f9;"> <p>i Note</p> <p>You can also right-click the graph in the navigation pane and choose the <i>Open Documentation</i> menu option.</p> </div>
Operator	<p>Right-click an operator in the graph editor and choose the <i>Open Documentation</i> menu option.</p> <div style="border: 1px solid #ccc; padding: 5px; background-color: #f9f9f9;"> <p>→ Remember</p> <p>For operators that you've created (user-defined operators), you can access the documentation only if you've uploaded the documentation when creating the operator. For more information, see topic <i>Creating Operators</i>.</p> </div>

15. Save the graph.

- a. After creating a graph, in the editor bar, choose  (Save) to save your graph.
- b. Choose the *Save* menu option.
- c. Provide a name along with the fully qualified path to the graph.
For example, `com.sap.others.graphname`.
- d. Provide a description for the graph.
- e. Choose *OK*.

The graphs and operators are stored in a folder structure within the modeler repository. For example, `com.sap.others.graphname`. If you want to save another instance of the graph, in the editor bar, choose  (Save As) and provide a name along with the fully qualified path to the graph.

16. (Optional) Validate the graph

The Modeler allows you to validate the configurations of operators in a graph before executing the graph.

i Note

The application performs validation run only on operators in the graph, which have type schemas associated with them.

- a. In the bottom pane, choose the *Validation* tab.
- b. Select the *Enable validation* toggle button.
The application validates the graph only if this toggle button is enabled. Once enabled for a selected graph, the changes are saved for the logged in user and the application runs validations for all open graphs.

i Note

User must explicitly disable the toggle button to disallow the application to run graph validations.

- c. In the editor toolbar, choose  (Save) to save the changes.

In the *Validation* tab, the application displays the validation results. The validation errors are grouped for each open graph. You can expand the results to view operator-specific validation results.

Related Information

[Execute Graphs \[page 16\]](#)

[Schedule Graph Executions \[page 18\]](#)

[Maintain Resource Requirements for Graphs \[page 21\]](#)

[Groups, Tags, and Dockerfiles \[page 23\]](#)

[Execution Model \[page 25\]](#)

[Tutorials \[page 48\]](#)

[SAP Data Hub Operators](#)


[Creating Operators \[page 78\]](#)

[Monitor the Graph Execution Status \[page 28\]](#)

3.1 Execute Graphs

After creating a graph, you can execute the graph based on the configuration defined for the graph. The operators in the graph are executed as individual processes.

Procedure

1. Open the graph that you want to execute in a graph editor.
2. In the editor toolbar, choose  (Run).
3. Choose a menu option.

Menu Option	Description
Run	Direct and immediate graph execution.
Run As	Execute the graph with an alias name. In the <i>Name</i> text field, provide a name for the graph execution.

If you have defined configuration substitution parameters for the operators in the graph, execute the graph with *Run As* to provide values to the substitution parameters. In the *Run As* dialog box, the application displays all the configuration substitution parameters. Provide the required values.

→ Tip

In the *Run As* dialog box, you can select the *Save Run Configuration* checkbox to save the configuration (values to the configuration substitution parameters) for subsequent executions of the graph. If you

have saved the configuration, for the next graph execution, you can select the configuration that the application must use to execute the graph or update the existing run configuration. If you have defined and saved multiple configurations, then you can also define a default run configuration. In the *Run As* dialog box, select the *Make Default Run Configuration*.

Related Information

[Parameterize the Process Execution \[page 17\]](#)

3.1.1 Parameterize the Process Execution

A graph is a network of operators connected to each other using typed input ports and output ports for data transfer. You can define and configure the operators in a graph.

You can parameterize the process execution using configuration substitution parameters. There are two ways to provide values to a configuration substitution parameter. You can provide values at runtime (when you execute the graph) or reference the value of another configuration parameter from the same operator.

To define a configuration substitution parameter, provide the value to a configuration parameter in the format, `${parameter_name}`.

Here, `parameter_name` is the name of the configuration substitution parameter.

For providing values to the configuration substitution parameter:

- If the name of the configuration substitution parameter matches the name of an operator's configuration parameter, the modeler executes the process with the value provided to that configuration parameter, which has the same name as that of the configuration substitution parameter.
- If the name of the configuration substitution parameter does not match the name of any configuration parameter of the operator, then you can provide a value to it at runtime.
For example, the operator configuration parameter `Path` can be defined with the value `URL: //${name}`. In this example, the `name` is the configuration substitution parameter and you can define the `Path` value by providing a value to the `name` at runtime.

→ Remember

You can also use the same configuration substitution parameter for more than one operator configuration parameter.

3.2 Schedule Graph Executions

The SAP Data Hub Modeler provides capabilities to schedule graph executions.


Context

Scheduling graph execution is useful, for example, to schedule graph executions with recurring conditions.

i Note

We recommend scheduling executions only for those graphs, which when executed run for a limited period and finish with the status either as *completed* or *dead*. For example, Data Workflows (graphs with data workflow operators).

Procedure

1. Open the graph that you want to execute in a graph editor.
2. In the editor toolbar, choose  (Run).
3. Choose the *Schedule* menu option.
4. Schedule a graph execution.

In the *Schedule Graph* dialog box, define the schedule.

- a. In the *Schedule Description* text field, provide a name for the schedule.
- b. Select a schedule property.

The application supports a form-based approach or a cron expression to define the schedule.

Table 1: Schedule Property


Property	Description
Form	<p>Provide a form-based UI to define a condition that specifies the frequency (or the number of occurrences) for executing the graph.</p> <p>The system uses the UTC equivalent of the frequency pattern that you specify to schedule the executions.</p> <p>For example, you can define a recurring condition that executes the graph every day at 9:00 AM</p>

Property	Description
Expression	<p>Use this mode to define a cron expression that provides the condition for scheduling a recurring graph execution. The cron expression is a string comprising of five fields separated by white spaces.</p> <p>The syntax for the cron expression is Minute Hour DayOfMonth Month DayOfWeek.</p>

5. Choose *Schedule*.

Next Steps

After creating a schedule, you can monitor the schedule within the Modeler or use the SAP Data Hub Monitoring application to monitor and manage all schedules.

In the bottom pane of the Modeler application, under the *Schedule* tab, you can monitor and manage the schedule. In this tab, you can view the description of the schedule and the graph execution state. If the graph is in the *active* state, the schedule trigger runs, otherwise, it is suspended. To stop a schedule, select the required graph and choose  (Stop Process).

Note

A special scheduling graph running in the SAP Data Hub Modeler performs the scheduled operation.

Related Information

[Cron Expression Format \[page 19\]](#)

[Working with the Data Workflow Operators \[page 97\]](#)

[Using SAP Data Hub Monitoring \[page 33\]](#)

3.2.1 Cron Expression Format

Use cron expressions to define the recurrence condition that the tool must use to schedule the graph execution.

Cron Format

Cron expression to execute graphs in the Modeler is a string comprising of five fields. The table lists the order of fields (from left to right) in a cron expression and the permitted values for each field.

Cron Field (from left to right)	Description
Minute	Representation for a minute. Permitted: 0 to 59
Hour	Representation for an hour. Permitted: 0 to 23
DayOfMonth	Day of the month. Permitted: 1 to 31
Month	Numeric representation for month. Permitted: 1 to 12
DayOfWeek	3-letter representation for a day of week. Permitted: mon, tue, wed, thu, fri, sat, sun

Note

The system uses the UTC (offset 0) equivalent of the condition specified with the cron expression.

Cron Syntax

The table provides the syntax that the application supports to define a cron expression.

Expression	Where Used	Value
.	Anywhere	Any value
*/a	Anywhere	Any a-th value
a:b	Anywhere	Values in range a to b
a:b/c	Anywhere	Every c-th value between a and b
a.y	DayOfWeek	On the a-th occurrence of the weekday y (a = 0 to 6)
a,b,c	Anywhere	a or b or c

Cron Expression: Examples

The table lists some examples of cron expressions that you can use.

Expression	Description
* * * * *	Run the schedule every minute. For this cron expression, the pipeline engine sets the graph to Run Always mode. In this scenario, the pipeline engine always checks if a graph is in running state, if it is not running, it attempts to run the graph.

Expression	Description
<code>*/5 13 * 12 0</code>	Run the schedule every 5 th minute between 1:00 PM (UTC) and 1:59 PM (UTC) on Sundays in December.
<code>** -1.sun 9 0</code>	Run the schedule on the last Sunday of every month at 09:00.

3.3 Maintain Resource Requirements for Graphs

You can specify compute resource requirements (memory or CPU) for graph groups.

For each resource, you can specify the `request` and `limit` properties. The `request` property specifies the initial resource quantity that the application requires to start the group execution. If there is not enough resource available, then graph execution fails to start. The `limit` property specifies limit for a resource usage.

If `memory` limit is violated, then group execution is terminated. The `cpu` limit, in turn, helps to limit CPU consumption in case of excessive CPU usage by a group execution. Resource requirements use the same notation as Kubernetes to specify resource quantity. For more information on `memory` limit and `cpu` limit, see the Kubernetes documentation:

[memory](#) ↗

[cpu](#) ↗

Specifying Resource Requirements for a Graph

Resource requirements are specified in graph description (`graph.json`). The `groupResources` property provides default resource requirements for all groups. For a group (except `default`), you can overwrite requirements in `resources` property in the group definition.

Here is an example of modified data generator demo graph (unnecessary details are omitted):

Sample Code

```
{
  "groupResources": {
    "memory": {
      "request": "4M",
      "limit": "16M"
    },
    "cpu": {
      "request": "0.5",
      "limit": "1.5"
    }
  },
  "description": "Data Generator",
  "processes": {
    "datagenerator1": {
      "component": "com.sap.util.dataGenerator",
      ...
    }
  }
}
```

```

    },
    "lmnu": {
      "component": "com.sap.util.terminal",
      ...
    }
  },
  "groups": [
    {
      "name": "group1",
      "nodes": [
        "datagenerator1"
      ],
      "resources": {
        "memory": {
          "limit": "32M"
        },
        "cpu": {
          "request": "1",
          "limit": "2"
        }
      }
    }
  ],
  "connections": [
    {
      "src": {
        "port": "output",
        "process": "datagenerator1"
      },
      "tgt": {
        "port": "in1",
        "process": "lmnu"
      }
    }
  ]
}

```

In this example graph, there are two groups: `default` (contains `terminal` operator) and `group1` (contains `dataGenerator` operator). Resource requirements from `groupResources` are applied to `default` group:

Sample Code

```

{
  "groupResources": {
    "memory": {
      "request": "4M",
      "limit": "16M"
    },
    "cpu": {
      "request": "0.5",
      "limit": "1.5"
    }
  },
  ...
}

```

First, the resource requirements from `groupResources` property is applied to `group1`. Then, all resource requirements are overwritten for `group1` but not `memory` request (since it's not specified for `group1`):

Sample Code

```

{

```

```

...
"groups": [
  {
    "name": "group1",
    "resources": {
      "memory": {
        "limit": "32M"
      },
      "cpu": {
        "request": "1",
        "limit": "2"
      }
    }
  }
],
...
}

```

3.4 Groups, Tags, and Dockerfiles

This section describes working with Groups, Tags, and Dockerfiles in SAP Data Hub.

Groups

When you execute a graph with groups, each group's subgraph will run in a different Docker container with a possibly different Docker image. The Docker image used by a group is automatically selected based on the tags associated with it. On the other hand, operators inside the same group are assured to run in the same node. Each group can also be configured with a different restart policy, tags or multiplicity.

The most common use case for using groups is to distribute work among many compute nodes either through partitioning the graph into many groups, adding multiplicity larger than 1 for a group, or both. This can lead to better graph throughput and cluster utilization. A user may also want to have different restart policies for different groups. For example, in a group one may want the container to be redeployed when it fails, while in a second group the user may want the graph to terminate if this group fails. Finally, one can also create a group if currently there is no dockerfile satisfying the requirements of a graph. In this case, the user needs to partition the graph into groups in such a way that for each of them exists at least one dockerfile which satisfies its requirements.

A graph with no explicitly defined group will have only one group (called the default group) which contains all graph's operators. The user can further partition the graph by assigning subset of operators to an explicit group. For example, suppose a graph has the following topology: A->B->C->D->E. If no explicit groups are defined then all those operator will run in the default group. Now suppose we create two explicit groups: group-1 containing A and B; and group-2 containing only E. The topology would look like this: (A->B)->C->D->(E). Now the graph has three groups: group-1 (A, B); default group (C, D); and group-2 (E).

Tags and Dockerfiles

When executing a graph, a Docker image will be selected for each group based on its tags. If more than one Dockerfile satisfying those tags are found, the application will select the one with the fewest tags (if multiple satisfying dockerfiles have the fewest number of tags then ties are broken arbitrarily). The selected Dockerfile will be built upon graph execution if the corresponding image has not already been cached.

Each tag represents a runtime requirement for the group (for example, packages and libraries) and is specified by a pair (`<resource_id>`: `<resource_version>`). Some examples are: (`"python36": ""`), (`"opencv": ""`) and (`"tornado": "5.0.2"`). An empty `<resource_version>` implies that a Docker image containing any version of The tags associated with a group are the union of each operator tags and the ones specified in the group configuration. Once the resulting tags are calculated, a Dockerfile satisfying all the group's tags is searched in the repository directory, **resource_id** is enough. If two operators in the same group have a tag with the same `<resource_id>`, but different `<resource_version>`, then when calculating the resulting tags for the group, those two tags will be merged into one and the final `<resource_version>` will assume the value of the more specific one if both versions are compatible. If they are incompatible, then an error is thrown. For example, if one operator has the tag (`"foo": "1.1"`) and another has (`"foo": "1.1.2"`), then the result of the merge will be (`"foo": "1.1.2"`) because the "1.1.2" is more specific than "1.1". On the other hand, if the last operator required version "2.1.1" then an error would happen because the versions would not be compatible since they don't share a common prefix.

When searching for Dockerfiles that satisfy the resulting tags of a group, it is a must to know whether a particular group's tag is satisfied by some tag of a Dockerfile. A group's tag G (`<resourceG_id>`: `<resourceG_version>`) is satisfied by a Dockerfile tag D (`<resourceD_id>`: `<resourceD_version>`) if and only if `<resourceG_id>` equals `<resourceD_id>`, `<resourceG_version>` shares a common prefix with `<resourceD_version>`, and the former is not more specific than the latter. For example, the group tag (`"foo": "1.1"`) is satisfied by the Dockerfile tag (`"foo": "1.1"`) or (`"foo": "1.1.2"`), but not by (`"foo": "1"`), (`"foo": ""`), or (`"bar": "1.1"`). If more than one Dockerfile satisfies the tags of a group, the specific `<resource_versions>` defined in each Dockerfile will not be used as a tie breaking criterion.

If no Dockerfile can satisfy one of the groups requirement in your graph then an error message will be shown when trying to run it. In this case, there are two options. You can either split the problematic group into smaller groups so that each one of them matches some existing Dockerfile, or you can create a new Dockerfile which attends all the group requirements.

Example

Suppose we have 3 Dockerfiles defined in our repository with the following associated tags:

- `com.sap.d1: {"foo": "", "bar": "1.2.3", "baz": "", "qux": "1.1.1", "abc": ""}`
- `com.sap.d2: {"foo": "", "corge": "2.2.2"}`
- `com.sap.d3: {"xyz": ""}`

The tags associated with a group are the union of each operator tags and the ones specified inAlso, suppose we have a graph with the following topology: $(A \rightarrow B \rightarrow C) \rightarrow D \rightarrow E$. So, it has one explicit group (*group-1*) containing (A, B, C) and the *default* group containing (D, E). Assume the groups' configuration and operators have the following tags:

- *group-1* configuration: `{"foo": "", "bar": ""}`

- *default* group configuration: {}
- operator A: {"baz": ""}
- operator B: {"bar": "1.2", "qux": "1.1.1"}
- operator C: {}
- operator D: {}
- operator E: {"foo": ""}

By making the union of the tags associated with each group we get the following result:

- *group-1*: {"foo": "", "bar": "1.2", "baz": "", "qux": "1.1.1"}
- *default* group: {"foo": ""}

Now, we need to find a Dockerfile to be used by each group. Clearly, *group-1*'s aggregated tags can only be satisfied by the *com.sap.d1* dockerfile. This Dockerfile has one more tag ("abc": "") than *group-1* requires. Also, the "1.2.3" version for "bar" in *com.sap.d1* satisfies *group-1*, which requires version "1.2" which is more generic than "1.2.3". On the other hand, the *default* group has 2 Dockerfiles which satisfy its aggregated tags: *com.sap.d1* and *com.sap.d2*. In such scenarios, we choose the one with fewer tags, which in this case is *com.sap.d2*.

Operator Tags

You can view or edit the operators tags in the operator editor. Right-click the operator and choose *Edit*. The operators that run on subengines may have implicit tags. Those tags will not appear on the operator editor screen, but they will be included in the requirements of a group using this operator. You can check what are the implicit tags for each subengine in the subengine documentation.

3.5 Execution Model

This section provides technical details of Modeler's execution behavior.

Operators in a graph execute concurrently. They communicate with each other by sending data to their outputs and by receiving data from their inports. *Back-pressure* in a connection between two operators is handled by blocking the operator that tries to send data until the receiving operator reads it. This prevents data to accumulate on a region of the pipeline that produces data faster than other parts can consume.

i Note

The back-pressure may cause some graph with cycles to deadlock.

Deadlock on Graphs with Cycles

Consider the graph below which has a cycle containing operators B and C:

```
A---->B---->C
```

Suppose A generates just one message and sends it to B. Each time B or C receives a message through their inport, they will process it and then send a new message through their outport. This implies that there will always be a single message circulating around the cycle. This graph does not deadlock. Now suppose that instead of A feeding just one message into the graph, it produces two messages. In this case, the graph will deadlock because at some point B will be blocked trying to send to C, which doesn't read from its inport because it is trying to send to B, which also does not read from its inport because it is trying to send to C, and so on. In general, a deadlock happens in those types of graphs when there are at least as many messages within the cycle as there are nodes in it. Another way in which the above example graph could fit into this deadlock rule is if A generated just one message, but B outputted two messages for each one that it received at its input port.

References

When an operator sends data to another operator the data sent is not copied, but rather only the reference to the data is sent. This behavior decreases the communication cost and is also important because when programming a script operator, if you change a mutable object received as an input then this may reflect in other parts of the pipeline. To be safe, we recommend making a copy of a mutable object before changing it. The message data type is a common example of mutable object that should be copied when changed.

4 Monitoring the SAP Data Hub Modeler

You can monitor the working of the SAP Data Hub Modeler using several different monitoring options.

After creating and executing the graph, you can choose to perform any of the following options to monitor the graph status of the SAP Data Hub Modeler.

Monitor Status of Graph Execution	After creating and executing a graph, monitor the status of the graph execution within the modeler. You can also use the standalone monitoring application that SAP Data Hub provides to monitor the status of all graphs executed in the modeler.
Trace Messages	Tracing messages within the modeler help monitor the system and to isolate the problems or errors that may occur.
SAP Vora Diagnostics	SAP Vora Diagnostics deploys one of the most widely used stacks of open-source monitoring and diagnostic tools for Kubernetes. For health and performance monitoring, SAP Vora Diagnostics provides cluster administrators access to a cluster-wide system and application metrics. For more information, see Using SAP Data Hub Diagnostics .
Using SAP Data Hub Monitoring	SAP Data Hub provides a stand-alone monitoring application to monitor the status of graphs executed in the SAP Data Hub.
Downloading Diagnostic Information for Graphs	You can download diagnostic information about all graphs or a specific graph and its subgraphs.

i Note

By default, the Modeler also performs logging. The log messages are intended for a broader audience with different skills. If you want to view the log messages, start the Modeler, and in the bottom pane, select the *Logs* tab.

Related Information

[Monitor the Graph Execution Status \[page 28\]](#)

[Trace Messages \[page 31\]](#)

[Using SAP Data Hub Monitoring \[page 33\]](#)

[Downloading Diagnostic Information for Graphs \[page 42\]](#)




4.1 Monitor the Graph Execution Status


After creating and executing a graph, you can monitor the status of the graph execution within the SAP Data Hub application.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the bottom pane, select the *Status* tab.

The *Status* tab provides information on the status of all graphs that were executed.

Action	Description
Monitor status of subgraphs	<p>When a graph execution triggers or spawns the execution of another graph, the latter is called a subgraph. You can monitor the status of various subgraphs that the Modeler has executed:</p> <ol style="list-style-type: none">1. In the <i>Status</i> tab, enable the <i>Show Subgraphs</i> toggle button.
Download diagnostic information	<p>You can download the diagnostic information that the application generators for a graph as a zipped archive.</p> <ol style="list-style-type: none">1. In the <i>Status</i> tab, next to the required graph, choose  (Download Diagnostic Information).2. Open or save the zipped archive file.
	<div style="background-color: #f0f0f0; padding: 10px;"><p>i Note</p><p>If a graph has subgraphs or it is a subgraph, the archive contains information about all of the graphs in the hierarchy.</p></div>
Execute another graph instance	<p>For each graph, you can create and execute another instance of the same graph.</p> <ol style="list-style-type: none">1. In the <i>Status</i> tab, next to the required graph, choose  (Start New Process).
Edit a graph	<ol style="list-style-type: none">1. In the <i>Status</i> tab, next to the required graph, choose  (Open Graph Editor).
View execution details	<p>You can view more details for a graph execution</p> <ol style="list-style-type: none">1. In the <i>Status</i> tab, select the required graph. The application opens a new pane with detailed information on the status of the graph execution process.2. If the selected graph contains groups, then in the bottom pane, choose the <i>Group</i> tab. The application provides more information on different groups executed as part of the graph execution.

Action	Description
	<div data-bbox="831 331 1394 548" style="background-color: #f0f0f0; padding: 5px;"> <p>i Note</p> <p>If you have defined a multiplicity of greater than 1 for a group, for example, a multiplicity of 3 for a group, then the application displays three instances for the same group.</p> </div> <ol style="list-style-type: none"> <li data-bbox="794 555 1394 667">3. If you want to view the details of various processes that the application executed as part of the graph execution, then in the bottom pane, choose <i>Process</i> tab. Each process has a state that may change over time. <li data-bbox="794 674 1394 763">4. If you want to view details of various metrics that the application provides as part of the process, then in the bottom pane, choose the <i>Metrics</i> tab.
<p>Stop process execution.</p>	<p>To stop the execution of any graph:</p> <ol style="list-style-type: none"> <li data-bbox="794 801 1394 891">1. In the <i>Status</i> pane, select the required graph and choose  (Stop Process). <div data-bbox="799 898 1394 1055" style="background-color: #f0f0f0; padding: 5px;"> <p>i Note</p> <p>The status of the graph changes to <i>completed</i> or <i>dead</i> depending on the state of the graph when the execution is stopped.</p> </div>

Related Information

[Graph Execution \[page 29\]](#)

[Graph States \[page 30\]](#)

[Process States \[page 31\]](#)

4.1.1 Graph Execution

The Modeler application supports executing a graph and monitoring its status from within the application.

When you schedule a graph for execution, the application translates the graphical representation (internally represented as a JSON document) into a set of running processes. These processes are responsible for the graph execution.

During the graph execution, the application translates each operator in the graph into (server) processes and translates the input and output ports of the operators into message queues. The process runs and waits for an input message from the message queue (input port). Once it receives the input message, it starts processing to produce an output message and delivers it to the outgoing message queue (output port). If a message reaches a termination operator, the application stops executing all processes in the graph and the data flow stops.

i Note

If a graph does not have a termination operator, it continues to execute all processes until you manually stop the graph execution.

4.1.2 Graph States

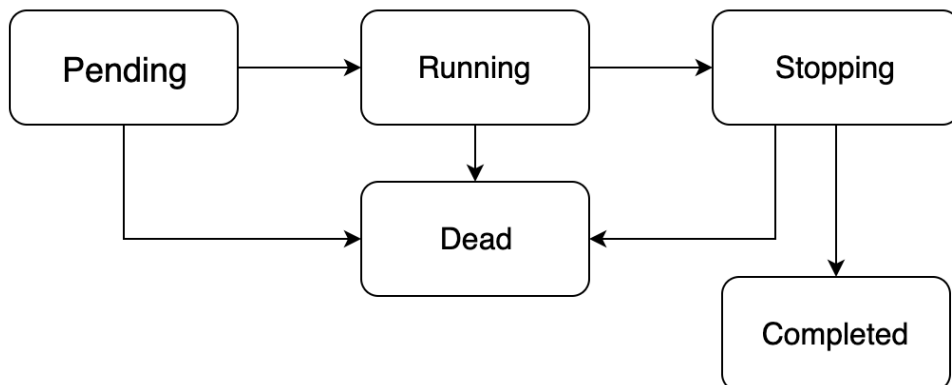
When you create a graph, each graph is associated with a graph state, which may vary with time and can be manipulated with operations on the graph.

The table lists the possible graph states. You can view the status of the graph in the *Status* tab.

Status	Description
pending	Graph is being prepared for execution. Initial state.
running	Graph is currently running
stopping	Graph execution is stopping
completed	Graph terminated successfully
dead	Graph terminated abnormally because one or more operators in the graph failed.
unknown	State of graph is unknown. Indicates internal problems.

Initially, the graph is in the *pending* state. In this state, the graph is being prepared for execution. It remains in this state until either an error occurs, or state of all subgraphs in graph is *running*.

The following image depicts the potential graph transition states.



If an error occurs during the *running* state, the graph allocation (and hence the overall graph) changes to *dead*. If all subgraphs terminate successfully, the graph state changes to *completed*.

Related Information

[Monitor the Graph Execution Status \[page 28\]](#)

4.1.3 Process States

When you execute a graph, the tool executes each operator in the graph as processes. Each process execution is associated with a state, which may vary with time.

The following table lists the possible states for process execution. Select the *Process* tab in status details to view the status of process execution.

Status	Description
initializing	process is initializing
running	process is running
stopping	user called STOP, the process is still running
stopped	process has stopped
dead	process crashed and is unrecoverable

Related Information

[Monitor the Graph Execution Status \[page 28\]](#)

4.2 Trace Messages

Tracing is a complete set of information (messages), which enables you to monitor the performance of the tool. Tracing messages also help isolate the problems or errors that may occur based on different severity threshold levels.

Context

You can activate application logging to find errors that occur sporadically. After you activate tracing, the tool logs the trace messages and categorizes them based on the severity levels. You can also limit the number of trace files that are written at the same time. Tracing messages are intended for various user personas. For

example, user personas with development skills and a deep understanding of the tool can use the trace messages to debug errors.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the bottom pane, select the *Trace* tab.
The *Trace* tab helps to configure the trace publisher and to monitor the trace messages for different severity levels.

3. Configure the trace publisher.

By default, trace publishing is disabled.

- a. In the tab menu, click the *INFO | disabled* button.
- b. In the *Trace Level* dropdown list, select a value.
- c. Use the *Trace Streaming* toggle button to activate the streaming.
- d. In the *Filter* text field, enter a value to filter the messages based on certain keywords.

The filter configuration constraints the trace messages streamed by the publishers. You define the filter as a comma-separated string of filter keywords.

→ Remember

A trace message is streamed if the filter string is empty (default) or if it contains any of the filter keywords.

- e. In the *Buffer Capacity* text field, enter a number value.

The buffer capacity helps the publisher identify the number of trace message that it must stream or publish in the server.

i Note

The trace server begins to store the trace messages only after you have enabled tracing. If the trace buffer reached its capacity due to network congestions the publisher drops additionally streamed traced messages. In such cases, all trace messages are still written to standard tracing. An additional trace warning is generated documenting the number of lost streaming traces. This warning trace is likely to be dropped but can be retrieved from the standard trace.

- f. Choose *OK*.
4. (Optional) Filter and view messages based on severity levels.
 - a. In the trace pane, select the severity level to view messages associated with that level only.

→ Tip

Maximize the *Trace* tab to view all messages in larger pane. You can use a keyboard shortcut Ctrl + C and Ctrl + V to copy and paste the trace messages.

5. If you want to export the messages to a *.csv* file, in the trace pane, choose  (Download as CSV).

Related Information

[Trace Severity Levels \[page 33\]](#)

4.2.1 Trace Severity Levels

When streaming the traces for the SAP Data Hub Modeler application, you can set a trace severity threshold. Depending on the threshold level that you define, the application streams messages accordingly into the trace server.

The application supports the following trace severity threshold levels.

Trace Severity Threshold	Description
INFO	The publisher streams trace messages that contain informational text, mostly for echoing what has been performed in the application. The trace messages streamed also includes messages associated with warnings, errors, and fatal errors.
DEBUG	The publisher streams trace messages that contain useful information for developers mostly to debug and analyze the application. The trace messages streamed also include messages associated with info, error, warning, and fatal error threshold levels.
ERROR	The publisher streams trace messages that contain information describing the error conditions that may occur when working with the application. The trace messages streamed also includes messages associated with the fatal error threshold level.
FATAL	The publisher streams trace messages associated with fatal errors that may occur when working with the application.
WARNING	The publisher streams messages associated with warnings and errors that may occur when working with the application. The trace messages streamed also includes messages associated with error and fatal error threshold levels.

4.3 Using SAP Data Hub Monitoring

SAP Data Hub provides a stand-alone monitoring application to monitor the status of graphs executed in the SAP Data Hub Modeler.

The Monitoring application provides capabilities to visualize the summary of graphs executed in the Modeler with relevant charts. Additionally, the application also allows you to schedule graph executions. For each graph instance, the application also provides details such as graph execution status, time of execution, graph type, graph source, and more.

The application also allows you to open a graph in the SAP Data Hub Modeler, view graph configurations, or stop process executions.

This section describes accessing and using the SAP Data Hub Monitoring application.

Related Information

[Log in to SAP Data Hub Monitoring \[page 34\]](#)

[Monitoring with the SAP Data Hub Monitoring Application \[page 35\]](#)

[Schedule Graph Executions with SAP Data Hub Monitoring \[page 39\]](#)

4.3.1 Log in to SAP Data Hub Monitoring

You can access the SAP Data Hub Monitoring application from the SAP Data Hub Launchpad or directly launch the application with a stable URL.

Procedure

1. Launch the SAP Data Hub Launchpad user interface in a browser using one of the following URLs:
 - For on-premise installations (if TLS is enabled):

```
https://<node-ip>:<port>
```

where

- `<node-ip>`: The IP address of any node in the cluster.
- `<port>`: The SAP Data Hub Launchpad port number.
- For cloud installations:

```
https://<domain>:<ingress vsystem port>
```

where

- `<domain>`: The domain is specified when you expose the SAP Data Hub Monitoring externally. For more information, see [Configuring SAP Data Hub Foundation on Cloud Platforms](#)
- `<ingress vsystem port>`: To obtain the `ingress vsystem port`, execute the following command:

```
kubectl get ingress -n $NAMESPACE
```

The welcome screen appears where you can enter the login credentials.

2. Log into the SAP Data Hub Launchpad application using the following information:
 - Tenant ID
 - Your user name

- Your password

The SAP Data Hub Launchpad opens and displays the initial home page. User details and the tenant ID are displayed in the upper-right area of the screen. The home page displays all applications available in the tenant.

3. On the home page, choose [Monitoring](#).

The application UI opens displaying the initial screen.

4.3.2 Monitoring with the SAP Data Hub Monitoring Application

Login to the SAP Data Hub Monitoring application to perform various monitoring actions.

This table lists the different capabilities of the SAP Data Hub Monitoring application and the various actions that users can perform in the monitoring application.

Actions	Description
Visualize status of graph executions with charts	<p>The home page of the monitoring application includes a pie chart (donut chart) and a scatter chart.</p> <p>Pie Chart</p> <p>The Pie chart provides information on:</p> <ul style="list-style-type: none"> • the number of graph instances executed in the Modeler • the status of graph instances executed. <p>In the Status pane, you can view the pie chart. Each sector in the pie chart represents a graph state.</p> <p>Scatter Chart</p> <p>The scatter chart includes a time period axis and duration of graph execution (in seconds) axis. This chart provides information on each graph execution and plots them in the scatter chart against the time of execution and duration of execution.</p> <p>Each point in the chart represents a graph instance. Place the mouse cursor on one such point for more information on that particular graph instance.</p> <p>In the Runtime Analysis pane, you can view the scatter chart. You can also configure the chart and view results for a selected time period. In the chart header, select the required time period (Hour, Day, Week).</p>
View graph execution summary	The bottom of the home page provides information on currently running graph instances, recently executed graph instances, and successfully executed graph instances.

Actions	Description
View execution details of individual graph instances	<p>In the home page, choose the Instances tab. The application displays a list view of the execution details of all graph instances.</p> <p>For each graph instance, the application provides information on the status of the graph execution, the graph execution name, graph name (the source of the graph in the repository), the time of execution, and more.</p> <p>Click a graph instance to open the execution details pane. This pane displays more execution details and helps monitor the graph execution.</p>
View subgraph execution details	<p>When a graph execution triggers or spawns the execution of another graph, the latter is called a subgraph. You can configure the monitoring application to view execution details of all subgraph instances. In the menu bar, under the Instances tab, select Show Subgraphs. The application refreshes the list view and displays all subgraph instances along with all other graph instances.</p> <div data-bbox="807 1010 1394 1128" style="background-color: #f0f0f0; padding: 10px;"> <p>i Note</p> <p>All subgraph instances are named as subgraph.</p> </div> <p>Click a subgraph instance to open the execution details pane. This pane displays more execution details and helps monitor the subgraph execution. In the Overview tab, under the Parent field, the application displays the Handle Id of the subgraph's parent. Click the Handle Id to open its parent graph execution details.</p>
View subgraph hierarchy details	<p>For a selected subgraph, you can view its parent graph or the complete hierarchy of graph instances associated with the subgraph.</p> <ol style="list-style-type: none"> In the Instances tab, select the required subgraph instance. In the Actions column, click the overflow icon. Choose the Show Hierarchy menu option. <p>In the Hierarchy dialog box, the application displays all graph instances associated with the subgraph and groups them as a hierarchical representation. Select a graph and choose View details for more details on its execution.</p>

Actions	Description
Filter graph instances	<p>You can filter the graph execution that the application displays based on the source name, execution status, time of execution and more. In the <i>Instances</i> tab, choose <i>Filters</i> and define the required filter conditions.</p> <p>You can also filter and view graph instances executed on the same day, hour, or week. In the menu bar, select the required time period.</p>
Open source graph	<p>For any selected graph instance, the application enables you to launch the source graph in the Modeler application.</p> <ol style="list-style-type: none"> In the <i>Instances</i> tab, select a graph instance that you want to open in a graph editor. In the <i>Actions</i> column, click the overflow icon. Choose the <i>Open Source Graph</i> menu option. <p>The application launches the SAP Data Hub Modeler in a new browser tab with the source graph of the selected instance opened in the graph editor. You can view or edit graph configurations.</p>
View graph configurations	<p>For any selected graph instance, you can view the configurations defined for its source graph.</p> <ol style="list-style-type: none"> In the <i>Instances</i> tab, select the required graph instance. The application opens a <i>Graph Overview</i> pane and displays the source graph of the selected graph instance. Choose <i>Show Configuration</i>. The application opens a <i>Configuration</i> pane, where you can view the graph configurations (read-only). If you want to view configurations for any of the operators in the source graph, right-click the operator and choose <i>Open Configuration</i>. The <i>Configuration</i> pane displays the operator configurations.
Stop a graph instance execution	<p>If you want to stop the execution of a running graph instance,</p> <ol style="list-style-type: none"> In the <i>Instances</i> tab, select the required graph instance. In the <i>Actions</i> column, click the overflow icon. Choose the <i>Stop Execution</i> menu option. <p>You can stop the execution only if the graph status is running or pending.</p>
Search graph instances	<p>In the <i>Instances</i> tab, use the search bar to search for any graph instance based on the instance name or its source graph name.</p>

Actions	Description
Remove graph instance	<p>If you want to remove a graph instance from the Pipeline engine,</p> <ol style="list-style-type: none"> 1. In the <i>Instances</i> tab, select the required graph instance. 2. In the <i>Actions</i> column, click the overflow icon. 3. Choose <i>Remove</i>. <p>You can remove the process only if the graph status is not running or pending.</p>
Cleanup graph instances	<p>If you want to cleanup the instances from the Pipeline engine:</p> <ol style="list-style-type: none"> 1. In the <i>Instances</i> tab, select <i>Cleanup</i>. 2. In the <i>Cleanup</i> dialog box, choose <i>Delete only completed instances</i> or <i>Delete completed and dead instances</i>.
Download diagnostics information and view logs	<p>You can download diagnostic information for graphs</p> <ol style="list-style-type: none"> 1. In the <i>Instances</i> tab, select the required graph instance. 2. In the <i>Actions</i> column, click the overflow icon. 3. Choose the <i>Download Diagnostic Info</i> menu options <p>The application automatically downloads a zipped archive of information for the graph that can help diagnose issues, if any.</p> <p>For certain operators, such as the data workflow operators, you can view logs generated for the operator execution. To view these logs,</p> <ol style="list-style-type: none"> 1. In the <i>Instances</i> tab, click the required graph instance. 2. In the bottom pane, select the <i>Logs</i> tab.

Actions	Description
Managing schedules	<p>In the Monitoring application, you can also view graphs that are scheduled for execution or edit a schedule.</p> <p>View Schedules</p> <ol style="list-style-type: none"> At the top of the page, choose the Schedules tab. The application displays a list view of all that are scheduled for execution and its status. <p>Edit a Schedule</p> <ol style="list-style-type: none"> If you want to edit a schedule, select the required schedule. In the Actions columns, click the overflow icon. Choose the Edit menu option. Define the new schedule and choose Save. <p>Suspend all Schedules</p> <ol style="list-style-type: none"> If you want to suspend all schedules, select the Suspend option. A warning dialog appears along with the option to resume. If you want to resume the suspended schedules, click the Resume option in the warning dialog.

4.3.3 Schedule Graph Executions with SAP Data Hub Monitoring

The SAP Data Hub Monitoring provides capabilities to schedule graph executions.

Context

Scheduling graph execution is useful, for example, to schedule graph executions with recurring conditions.

Procedure

- Start the SAP Data Hub Monitoring application.
- In the home page, choose the [Schedules](#) tab. The application displays the graphs that are already executed and the status of the execution.
- Create a schedule.
 - In the top right corner of the [Schedules](#) page, choose [Create](#).

- b. Provide a description for the schedule.
- c. In the *Graph Name* dropdown list, select a graph that you want to schedule for execution.

The application populates all graphs that are available for scheduling executions.

i Note

We recommend scheduling executions only for those graphs, which when executed runs for a limited period and finishes with the status either as *completed* or *dead*. For example, Data Workflows (graphs with data workflow operators).

- 4. Define a recurrence.

You can define recurring schedules to execute the graph. The application supports a form-based approach or a cron expression to define the recurrence.

Type	Description
Form	<p>Provide a form-based UI to define a condition that specifies the frequency (or the number of occurrences) for executing the graph.</p> <p>The system uses the UTC equivalent of the frequency pattern that you specify to schedule the executions.</p> <p>For example, you can define a recurring condition that executes the graph every day at 9:00 AM</p>
Expression	<p>Use this mode to define a cron expression that defined the recurring condition to schedule a graph execution. The cron expression is a string comprising of five fields separated by white spaces.</p> <p>The cron expression syntax is Minute Hour DayOfMonth Month DayOfWeek</p>

- 5. Use the *Star Time* and *End Time* timepickers to define the boundary values.

For example, you can schedule a graph every day at 9:00 AM starting from 01 JAN, 2019 to 31 JAN, 2019.

Next Steps

After creating a schedule, you can monitor and manage the schedule within the SAP Data Hub Monitoring application.

i Note

A special scheduling graph running in the SAP Data Hub Modeler performs all the scheduled operations.

Related Information

[Cron Expression Format \[page 19\]](#)

4.3.3.1 Cron Expression Format

Use cron expressions to define the recurrence condition that the tool must use to schedule the graph execution.

Cron Format

Cron expression to execute graphs in the Modeler is a string comprising of five fields. The table lists the order of fields (from left to right) in a cron expression and the permitted values for each field.

Cron Field (from left to right)	Description
Minute	Representation for a minute. Permitted: 0 to 59
Hour	Representation for an hour. Permitted: 0 to 23
DayOfMonth	Day of the month. Permitted: 1 to 31
Month	Numeric representation for month. Permitted: 1 to 12
DayOfWeek	3-letter representation for a day of week. Permitted: mon, tue, wed, thu, fri, sat, sun

Note

The system uses the UTC (offset 0) equivalent of the condition specified with the cron expression.

Cron Syntax

The table provides the syntax that the application supports to define a cron expression.

Expression	Where Used	Value
.	Anywhere	Any value
*/a	Anywhere	Any a-th value
a:b	Anywhere	Values in range a to b
a:b/c	Anywhere	Every c-th value between a and b
a.y	DayOfWeek	On the a-th occurrence of the weekday y (a = 0 to 6)

Expression	Where Used	Value
a,b,c	Anywhere	a or b or c

Cron Expression: Examples

The table lists some examples of cron expressions that you can use.

Expression	Description
* * * * *	Run the schedule every minute. For this cron expression, the pipeline engine sets the graph to Run Always mode. In this scenario, the pipeline engine always checks if a graph is in running state, if it is not running, it attempts to run the graph.
* / 5 13 * 12 0	Run the schedule every 5 th minute between 1:00 PM (UTC) and 1:59 PM (UTC) on Sundays in December.
* * -1.sun 9 0	Run the schedule on the last Sunday of every month at 09:00.

4.4 Downloading Diagnostic Information for Graphs

Download a zipped archive of information for your graphs to help diagnose issues.

The SAP Data Hub Modeler lets you download diagnostic information for graphs in two ways:

- All graphs: In the [?](#) (*Help*) menu, select *Download Diagnostic Information* to generate a zip archive of information for all graphs.
- Specific graph: In the *Status* tab, select [↓](#) (*Download Diagnostic Information*) next to a graph to generate a zip archive of information about the graph and its subgraphs.

Related Information

[Diagnostic Information Archive Structure and Contents \[page 43\]](#)

4.4.1 Diagnostic Information Archive Structure and Contents

The directory structure and contents of the diagnostic information archive.

Zip Archive Structure

```
vflow-diagnostic-<timestamp>.zip
├── version.json
├── graphs.json
├── errors.txt
├── api-pods
│   ├── pods.json
│   ├── goroutine.txt
│   └── <podname>
│       ├── goroutine.txt
│       ├── logs-<podname>.txt
│       └── pod-<podname>.json
├── <graph-source>-<handle>
│   ├── graph.json
│   ├── execution.json
│   └── <group-instance-id>
│       ├── goroutine.txt
│       ├── heap.txt
│       ├── logs-<podname>.txt
│       └── pod-<podname>.json
├── ...
│   └── <group-instance-id>
│       ├── goroutine.txt
│       ├── heap.txt
│       ├── logs-<podname>.txt
│       └── pod-<podname>.json
└── <graph-source>-<handle>
    ...
```

Related Information

[version.json File \[page 44\]](#)

[graphs.json File \[page 44\]](#)

[graph-source-handle Folders \[page 44\]](#)

[graph.json File \[page 45\]](#)

[execution.json File \[page 45\]](#)

[events.json File \[page 46\]](#)

[group-instance-id Folders \[page 46\]](#)

[logs-pod-name.txt File \[page 46\]](#)

[pod-pod-name.json File \[page 46\]](#)

[goroutine.txt File \[page 47\]](#)

[heap.txt File \[page 47\]](#)

[api-pod Folder \[page 47\]](#)

4.4.1.1 `version.json` File

The `version.json` file contains version information.

For example:

```
{
  "version": "2.3.30-dev-0828",
  "buildTime": "2018-08-28T17:10:27",
  "gitCommit": "a1897fcf38788b55be1ce177e32bb2ebc733b28c",
  "platform": "linux"
}
```

4.4.1.2 `graphs.json` File

The `graphs.json` file contains brief information about executed graphs, including files that are completed and not deleted.

The file can be used, for example, to identify graphs with a status of `dead`, the last messages of a graph, and so on.

```
[
  {
    "src": "com.sap.demo.datagenerator",
    "name": "com.sap.demo.datagenerator",
    "executionType": "",
    "handle": "9a6eeb59abb242baabb110dba50ae178",
    "status": "running",
    "terminationRequested": false,
    "message": "Graph is currently running",
    "started": "2018-08-29T10:26:41Z",
    "updated": "2018-08-29T10:26:45Z",
    "stopped": "",
    "submitted": "2018-08-29T10:26:40Z"
  },
  ...
]
```

4.4.1.3 `<graph-source>-<handle>` Folders

The `<graph-source>-<handle>` folder contains detailed information about graph execution. For example: `com.sap.demo.datagenerator-9a6eeb59abb242baabb110dba50ae178`.

To identify the files that you want to open, see the `graphs.json` file.

```
[
  {
    "src": "com.sap.demo.datagenerator",
    ...
    "handle": "9a6eeb59abb242baabb110dba50ae178",
    ...
  },
  ...
]
```

4.4.1.4 graph.json File

The `graph.json` file contains the graph definition.

The content in the `graph.json` file is the same that you get when you export the graph from the Modeler application.

4.4.1.5 execution.json File

The `execution.json` file contains information about the execution for a graph, such as groups, pods, processes (operator instances in a graph), and so on.

You can use the `execution.json` file to identify which pod or pods failed.

```
{
  "src": "com.sap.demo.datagenerator",
  "name": "com.sap.demo.datagenerator",
  "executionType": "stream",
  "handle": "9a6eeb59abb242baabb110dba50ae178",
  "status": "running",
  "terminationRequested": false,
  "message": "Graph is currently running",
  "remoteExecution": {
    "parent": ""
  },
  "started": 1535538401,
  "updated": 1535538405,
  "stopped": 0,
  "submitted": 1535538400,
  "allocations": [
    {
      "groupName": "default",
      "groupDescription": "",
      "subgraph": "default",
      "container": "vflow-graph-9a6eeb59abb242baabb110dba50ae178-com-sap-
demo-ngq4m",
      "containerIp": "172.17.0.8",
      "host": "minikube",
      "status": "running",
      "message": "Container is currently running",
      "restartCount": 0,
      "image": "a-docker-registry:5000/vora/vflow-node:2.3.30-dev-0829-
com.sap.debian",
      "destination": "",
      "updated": "2018-08-29T10:26:40Z",
      "processes": [
        {
          "id": "16d1",
          "componentId": "com.sap.system.jsengine",
          "status": "running",
          "processName": "datagenerator (16d1)",
          "engine": "main",
          "timestampPublished": "2018-08-29T10:26:41Z",
          "timestampReceived": "2018-08-29T10:26:41.031657526Z",
          "published": 1535538401,
          "received": 1535538401,
          "message": "Process is running",
          "metrics": null
        }
      ]
    }
  ]
}
```

```

        "id": "1mnu",
        "componentId": "com.sap.system.terminal",
        "status": "running",
        "processName": "terminal (1mnu)",
        "engine": "main",
        "timestampPublished": "2018-08-29T10:26:41Z",
        "timestampReceived": "2018-08-29T10:26:41.033268298Z",
        "published": 1535538401,
        "received": 1535538401,
        "message": "Process is running",
        "metrics": null
      }
    ]
  }
}

```

4.4.1.6 `events.json` File

The `events.json` file contains information about graph events.

For example, when a group is initialized, a group execution is started, or a graph has received a shutdown command, and so on.

4.4.1.7 `<group-instance-id>` Folders

For each instance of a group, a folder is created.

To determine which `<group-instance-id>` to open, see the `execution.json` file.

```

{
  "allocations": [
    {
      "subgraph": "<group-instance-id>",
      ...
    }
  ]
}

```

4.4.1.8 `logs-<pod-name>.txt` File

The `logs-<pod-name>.txt` file contains logs for a specific pod.

4.4.1.9 `pod-<pod-name>.json` File

The `pod-<pod-name>.json` file contains a pod description.

The file content is similar to the output of a `kubectl describe pod`.

4.4.1.10 `goroutine.txt` File

The `goroutine.txt` file contains `pprof` output for a `goroutine` profile.

4.4.1.11 `<heap.txt>` File

The `<heap.txt>` file contains `pprof` output for a `heap` profile.

4.4.1.12 `api-pod` Folder

The `api-pod` folder contains information about `vflow` API nodes.

The folder structure is similar to the `<group-instance-id>` folder, except that it does not contain a `heap.txt` file (a `heap.txt` file can become quite large for long-running `vflow` API nodes, and it is rarely used).

5 Tutorials

Tutorials help users understand and work with graphs and operators in the SAP Data Hub Modeler.

The tool provides example graphs and predefined operators, which users can use to create productive applications. These graphs and operators come along with necessary documentation, which helps you to understand the graph or the operator.

You can access the graphs and operators in the navigation pane. For documentation of the graphs or operators, right-click the graph or operator and choose *Open Documentation*.

This section includes tutorials for creating a simple TensorFlow application and a graph for text analysis in the modeler. It also provides information on how to use the example graphs available in the tool for TensorFlow applications and text analysis.

! Restriction

Tutorials for the SAP Data Hub, developer edition are available at [Tutorial Navigator](#). The tutorials described here are not applicable to and are not supported for SAP Data Hub, developer edition.

Related Information

[Create a Graph to Execute a TensorFlow Application \[page 48\]](#)

[Creating a Graph to Perform Text Analysis \[page 52\]](#)

5.1 Create a Graph to Execute a TensorFlow Application

The SAP Data Hub Modeler provides a dockerized execution environment for TensorFlow (TF) programs with Python 2.7 and TensorFlow 1.7.0.

Context

i Note

Python 2.7 will no longer be supported in a future SAP Data Hub release; Python 3.6 will continue to be supported.

This tutorial explains how to use the SAP Data Hub Modeler to execute a TensorFlow Python application.

! Restriction

Tutorials for the SAP Data Hub, developer edition are available at [Tutorial Navigator](#). The tutorials described here are not applicable to and are not supported for SAP Data Hub, developer edition.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, choose the *Repository* tab.
The tool displays all graphs, operators, and Dockerfiles available in your repository and is grouped under the tabs, *Graphs*, *Operators*, *Docker Files* respectively.
3. Create an operator.

If you want to execute a TensorFlow application using the modeler, first create an operator that includes the TensorFlow application code.

- a. Right-click the *Operators* section and choose *Create Folder* to create a new folder in which you want to define the operator.

i Note

If you want to create a folder structure (sub directories) within the root folder, right-click the root folder and choose *Create Folder*.

- b. Right-click the folder in which you want to create the operator and choose the *Create Operator* menu option.
- c. In the *Name* text field, provide a name as the identifier name for the operator as **trainOP**.
- d. In *Display Name* text field, a display name for the operator as **Train MNIST**.
You can use the display name to search and add the operator when creating graphs.
- e. In the *Base Operator* dropdown list, select *Python2Operator*.
- f. Choose *OK*.

The operator that you create is derived from the base operators that SAP Data Hub provides.

- f. Choose *OK*.

The tool opens an operator editor window. The operator editor is a form-based editor, where you can define the operator.


4. Define the operator.

Use the operator editor view to define your operator.



- a. In the *Tags* section, choose the + (Add tag) icon.
Tags describe the runtime requirements of operators and are the annotations of Dockerfiles that the tool provides.
- b. In the dropdown list, select the *python27* tag and choose no version.
- c. In the *Tags* section, choose the + (Add tag) icon to create an additional tag.
- d. In the dropdown list, select the *tensorflow* tag and choose the version *1.7.0*.

i Note

The current version of the SAP Data Hub Modeler supports TF version 1.7.0 with Python 2.7.

- e. Define input and output ports of the type. For example, `message.python.image` for incoming images and `string` for the output result.
 - f. In the inline editor add the tensorflow application code.
5. Save the operator.
 - a. In the editor toolbar, choose  (Save) to save the operator.
You can now use the TensorFlow training operator in a graph.
 6. Create a graph with TensorFlow operator.

After creating a TensorFlow operator, you can use the operator in a graph.

 - a. Switch to the *Graphs* tab.
 - b. In the navigation pane toolbar, choose + (Create Graph).
The tool opens an empty graph editor in the same window, where you can define your graph.
 - c. In the navigation pane, choose the *Operators* tab.
 - d. In the list of operators, select the *Train MNIST* operator that you created.
Your graph now consists of only one operator, which contains the TensorFlow application.
 7. Save the graph.
 - a. After creating a graph, in the editor toolbar, choose  (Save) to save your graph.
 - b. Choose the *Save* menu option.
 - c. Provide a name along with the fully qualified path to the graph.
For example, `com.orgname.others.graphname`.
 - d. Provide a description to the graph.
 - e. Choose *OK*.
 8. Execute the graph.
 - a. In the editor toolbar, choose  (Run).
 - b. Select the *Run* menu option to execute the graph.

Related Information

[Example Graphs for TensorFlow \[page 50\]](#)

5.1.1 Example Graphs for TensorFlow



Use the TensorFlow-based example graphs in the SAP Data Hub Modeler to build rich machine learning applications.

The SAP Data Hub Modeler provides the following example graphs, which you can use to build productive applications. These graphs are based on the TensorFlow graph described in the tutorial.

- `com.sap.ml.tensorflow.trainMnistRepo`
- `com.sap.ml.tensorflow.evaluateMnistRepo`
- `com.sap.ml.tensorflow.classifyVideoStream`

- `com.sap.ml.tensorflow.classifyImages`



Executing the `com.sap.ml.tensorflow.trainMnistRepo` graph

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the *ML Tensorflow (beta)* section, select the *Train MNIST Model* graph.
4. In the editor toolbar, choose  (Save) to save the graph.
5. In the editor toolbar, choose  (Run) to execute the graph.

The *Status* tab in the bottom pane shows the status for the graph execution as *running* to indicate that the graph is being executed. In the editor, right-click the *Terminal* operator and choose *Open UI*. The tool provides information on the accuracy and the model save progress.



Executing the `com.sap.ml.tensorflow.evaluateMnistRepo` graph

While the training graph is running (or after the graph is stopped), execute the inference graph.

1. In the navigation pane, select the *Graphs* tab.
2. In the *ML Tensorflow (beta)* section, select the *Infer MNIST Str. Repo* graph.
3. In the editor toolbar, choose  (Save) to save the graph.
4. In the editor toolbar, choose  (Run) to execute the graph.



The *Status* tab in the bottom pane shows the status for graph execution as *running* to indicate that the graph is being executed. You can now see inference values (deduced number in the drawn image) of the images and the image paths that are sent to the *Terminal* operator.

Executing the `com.sap.ml.tensorflow.classifyImages` graph.

1. In the navigation pane, select the *Graphs* tab.
2. In the *ML Tensorflow (beta)* section, select the *Classify Images with Inception* graph.
3. In the editor toolbar, choose  (Save) to save the graph.
4. In the editor toolbar, choose  (Run) to execute the graph.

The *Status* tab in the bottom pane shows the status for graph execution as *running* to indicate that the graph is being executed.

Executing the `com.sap.ml.tensorflow.classifyVideoStream` graph.

1. In the navigation pane, select the *Graphs* tab.
2. In the *ML Tensorflow (beta)* section, select the *Classify Video* graph.
3. In the editor toolbar, choose  (Save) to save the graph.
4. In the editor toolbar, choose  (Run) to execute the graph.

The *Status* tab in the bottom pane shows the status for graph execution as *running* to indicate that the graph is being executed.

5.2 Creating a Graph to Perform Text Analysis

The SAP Data Hub Modeler provides a dockerized execution environment to perform text analysis.

This section describes two example graphs available within the SAP Data Hub Modeler to perform text analysis and to build applications with natural language processing capabilities.

! Restriction

Tutorials for the SAP Data Hub, developer edition are available at [Tutorial Navigator](#). The tutorials described here are not applicable to and are not supported for SAP Data Hub, developer edition.

Related Information

[Example Graph for Simple Text Analysis \[page 52\]](#)

[Example Graph for Text Analysis of Files \[page 54\]](#)

5.2.1 Example Graph for Simple Text Analysis

Use the text analysis example graph in the SAP Data Hub Modeler to build applications with natural language processing capabilities.

The example graph `com.sap.textanalysis.example` is a simple Text Analysis application. You can use the graph to analyze text documents given in-place or as files in HDFS. This graph uses the Text Analysis Connector operator, which sends JSON-formatted requests to a text analysis server.

If the request is received through the `inFileData` port (a 'data' type of request), then the graph analyzes each text document provided as input to the TA Request Creator operator in a separate request and the Text Analysis Connector operator outputs the result as a string to the terminal.

If the request is received through the `inFolderPath` port (a 'folder' type of request), then the request must include a folder name indicating the location of the documents the graph must analyze. The graph writes the result of the analysis directly in the specified location in two files `[folder_name]_TA.csv` and `[folder_name]_TADOC.csv` and the graph terminates.

i Note

Please refer to the graph `com.sap.textanalysis.files` and the section *Example Graph for Text Analysis of Files* for an example of how to analyze a collection of files residing in other remote storage locations.

Prerequisites to execute the text analysis example graph

- You have installed the `vora-textanalysis` service and have the port number.
- For folder type of requests:

- You have an HDFS server available, which is reachable from a network firewall, if any.
- You have the hostname for the HDFS server and its port number.
- If Kerberos is not enabled, the folder and subfolders in HDFS, where your files are located allow write permissions to the 'root' user.
For more information, see

Executing the `com.sap.textanalysis.example` graph

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the search box, enter `com.sap.textanalysis.example`.
The application loads the selected graph in the graph editor.
4. Select the *TA Request Creator* operator and in the right pane, select the *Configuration* tab to set the configuration parameter values.



Operator	Configuration Parameter	Value
TA Request Creator (Operator id: javascryptoperator1)	serverendpoints	Host name and port number of the vora-textanalysis service
	taconfig	Use either LINGANALYSIS_BASIC, LINGANALYSIS_STEMS, LINGANALYSIS_FULL, EXTRACTION_CORE, EXTRACTION_CORE_ENTERPRISE, EXTRACTION_CORE_PUBLIC_SECTOR, or EXTRACTION_CORE_VOICEOFCUSTOMER. For more information on the description of each configuration, see the <i>Text Analysis</i> section in the <i>Developer Guide for SAP Vora</i> .
	languages (Optional)	A list of languages used for language detection specified in ISO 639-1 codes. For example: 'EN,DE,ES'. If no language is specified, then automatic detection is attempted.
	mimetype (Optional)	The type of input documents. Allowed values are 'text/plain', 'text/html', 'text/xml', and 'text'. The value 'text' indicates that the input is one of plain text, HTML or XML. If not set, or if value is 'text', document identification and conversion are performed.
	encoding (Optional)	If the document contains text, this parameter indicates the encoding. For example: 'UTF-8'. If not set and the MIME type indicates text, encoding detection and conversion are performed.
folderpath	Folder path to analyze, without trailing '/'. This value is required only if connected to <code>inFolderPath</code> port.	

i Note
Analysis of documents in binary formats is also supported, but automatic format detection is used in this case.

i Note
Relevant for 'folder' type request.

Operator	Configuration Parameter	Value
	recursive	If true, the analysis is done recursively in the subfolders in the specified location. Output files are written locally in each subfolder.

i Note
Relevant for 'folder' type request.

- In the editor toolbar, choose  (Save) to save the graph.
- In the editor toolbar, choose  (Run) to execute the graph.
The *Status* tab in the bottom pane shows the status for the graph execution as *running* to indicate that the graph is being executed.

→ Remember

For 'data' type of requests, you can use the terminal to input text documents. Right-click the *Terminal* operator and choose *Open UI*. Enter a text and press the enter key. The graph prints the result in the same terminal.

For more information on the results of text analysis, see the section *Text Analysis* in the [Developer Guide for SAP Vora](#).

Related Information

[Text Analysis Connector](#)

5.2.2 Example Graph for Text Analysis of Files

Use the text analysis example graph in the SAP Data Hub Modeler to build applications with natural language processing capabilities.

The SAP Data Hub Modeler provides a text analysis example graph:

`com.sap.textanalysis.files`. This graph helps perform text analysis on files stored on a HDFS file system. The graph listens to changes to files under a given root folder and sends requests to a text analysis server. The server analyzes the contents of the files in that folder and stores the results of the analysis on the same location of the files.

The graph also loads the results of the tokenization into SAP Vora. After loading the results, it optionally creates a full-text index on the file collection using the results of the tokenization for fast query execution.

For more information on SAP Vora, see [Developer Guide for SAP Vora](#).

i Note

The graph sends requests to the text analysis server at the level of the modified subfolder, and only when modifications occur.

Prerequisites to execute the text analysis example graph

- Your file collection resides in a storage location which is reachable from a network firewall, if any.
- You have the connection parameters and credentials for the file storage service.
- You have the connection parameters and credentials to connect to Vora's transaction coordinator.

Executing the `com.sap.textanalysis.files` graph

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the search box, enter `com.sap.textanalysis.files`.
The application loads the selected graph in the graph editor.
4. Select an operator in the graph and in the right pane, select the *Configuration* tab to set the configuration parameter values.



Operator	Configuration Parameter	Value
Read File (Operator id: read-file1)	Service	Storage location service
	Connection Properties	Connection ID or parameters and credentials to connect the storage server
	Path	Path to the folder to be analyzed
	Delete after Send	false
	Recursive	true
	Only Read On Change	true
	Poll Period	Interval between two content change detection events, must be >= 1000
Read File (Operator id: read-file2)	Service	Storage location service
	Connection Properties	Connection ID or parameters and credentials to connect the storage server (must be the same as in readfile1)
	Path	Path to the folder on HDFS to be analyzed (must be the same as in readfile1)
	Delete after Send	false
	Recursive	true
	Only Read On Change	false

Operator	Configuration Parameter	Value
	Poll Period	Interval between two content change detection events, must be >= 1000
Modification Checker (Operator id: javas-criptoperator5)	folderpath	Path to the folder to be analyzed, without trailing '/' or '\` (must be the same as in readfile1)
	hadoopNameNode	Host name and port number of the HDFS server (must be the same as in readfile1).
	schemaname	Name of the schema in SAP Vora under which the analysis results tables are to be created.
	tablenamesuffix	Suffix to the names of the analysis results tables.
		<div style="border-left: 2px solid orange; padding-left: 10px;"> <p>⚠ Caution</p> <p>Tables with names <code>TA_tablenamesuffix</code> and <code>TADOC_tablenamesuffix</code> must not already exist in SAP Vora. If they exist, the graph will immediately terminate.</p> </div>
	duration	(Recommended): At least 2 seconds larger than the Poll Period value set in readfile2.
		<div style="border-left: 2px solid blue; padding-left: 10px;"> <p>i Note</p> <p>The value of the duration parameter must be large enough, so that the Modification Checker operator can receive the complete set of modifications on the file collection reported by the Read File operators. A small value for a slow network may lead to incorrect text analysis results.</p> </div>
Create TA Request (Operator id: tarequestcreator1)	serverendpoints	Host name and port number of the vora-textanalysis service
	Storage Service	Storage location service
	Connection Properties	Parameters and credentials to connect the storage server (must be the same as in readfile1)
	Text Analysis Configuration	Use either <code>LINGANALYSIS_BASIC</code> , <code>LINGANALYSIS_STEMS</code> , <code>LINGANALYSIS_FULL</code> , <code>EXTRACTION_CORE</code> , <code>EXTRACTION_CORE_ENTERPRISE</code> , <code>EXTRACTION_CORE_PUBLIC_SECTOR</code> , or <code>EXTRACTION_CORE_VOICEOFCUSTOMER</code> . For more information on the description of each configuration, see the <i>Text Analysis</i> section in the <i>Developer Guide for SAP Vora</i> .

Operator	Configuration Parameter	Value
	languages (Optional)	A list of languages used for language detection specified in ISO 639-1 codes. For example: 'EN,DE,ES'. If no language is specified, then automatic detection is attempted.
	mime_type (Optional)	The type of input documents. Allowed values are 'text/plain', 'text/html', 'text/xml', and 'text'. The value 'text' indicates that the input is one of plain text, HTML or XML. If not set, or if value is 'text', document identification and conversion are performed.
	text_encoding (Optional)	If the document contains text, this parameter indicates the encoding. For example: 'UTF-8'. If not set and the MIME type indicates text, encoding detection and conversion are performed.
SQL Creator (Operator id: javas-criptoperator6)	createIndex	If true, the engine builds a full-text index on the file collection
SAP Vora Client (Operator id: sap-voraclient1)	Connection Properties	Connection ID or host name of the vora-tx-coordinator service, tc port number, and authentication credentials. Right-click the operator and choose Open Documentation for more information on the format of these values.
SAP Vora Client (Operator id: sap-voraclient2)	Connection Properties	Host name of the vora-tx-coordinator service, tc port number, and authentication credentials. Right-click the operator and choose Open Documentation for more information on the format of these values.

i Note

Analysis of documents in binary formats is also supported, but automatic format detection is used in this case.

- In the editor toolbar, choose  (Save) to save the graph.
- In the editor toolbar, choose  (Run) to execute the graph.
The *Status* tab in the bottom pane shows the status for the graph execution as *running* to indicate that the graph is being executed.

Querying text analysis results in SAP Vora

The results of the text analysis are loaded into two tables in SAP Vora: `TA_tablenamesuffix` and `TADOC_tablenamesuffix`, where `tablenamesuffix` is defined in the configuration of the [Modification Checker](#) operator. These two tables are in the SAP Vora relational disk engine.

If the `createIndex` parameter in the configuration of the [SQL Creator](#) operator of the graph is set to true, a full-text index is created using the results of text analysis. The index is associated with the `filename` column of a table `TADOC_tablenamesuffix_mem`. If you want to query using this index, use the `file_contains` predicate:

```
select * from TADOC_tablenamesuffix_mem where file_contains(filename,
'search_key');
```

i Note

For more information on the results tables for text analysis, see the *Text Analysis* section in the *Developer Guide for SAP Vora*.

6 Using Scenario Templates

This section describes a selection of common usage scenarios and how they can be implemented using operators and graphs provided by SAP Data Hub.

The templates that are shipped via an example graph can be found in the *Modeler* under the graph section *Scenario Templates*. You can also search for the package `com.sap.scenarioTemplates`.

To learn how to set up and run each scenario template, see *Scenario Templates* in the *Repository Objects Reference for SAP Data Hub*.

Related Information

[Load Data from Data Lake to Database \(HANA/VORA\) \[page 59\]](#)

[E\(T\)L from Database \[page 60\]](#)

[ABAP with Data Lakes \[page 61\]](#)

[BW with Data Lakes \[page 62\]](#)

[Data Processing with Scripting Languages \[page 75\]](#)

6.1 Load Data from Data Lake to Database (HANA/VORA)

Batch Processing and Stream Processing.

Batch Processing

These graphs show how to load data in batch fashion from different (cloud) storages into a SAP HANA database or a SAP Data Hub built-in Vora database. Batch fashion in this context means that all data available at runtime is loaded and the execution stops once all files have been processed.

Available Templates

Table 2:

Template	Path	Description
Load Files into HANA	<code>com.sap.scenarioTemplates.datalake-ToDatabase.loadToHana</code>	Load product data from CSV files into a HANA table, offering at-least-once guarantee between multiple graph runs.

Template	Path	Description
Load Files into Vora	com.sap.scenarioTemplates.datalake-ToDatabase.loadToVora	Load product data from CSV files into a Vora table supporting SQL on Files, offering at-least-once guarantee between multiple graph runs.

Stream Processing

These graphs show how to load data in near real-time fashion from different (cloud) storages into a SAP HANA database or a SAP Data Hub built-in Vora database. Real-time fashion in this context means that the graphs run continuously and read new files once available by polling repeatedly for changes on the connected file storage.

Available Templates

Table 3:

Template	Path	Description
Ingest Files into HANA	com.sap.scenarioTemplates.datalake-ToDatabase.ingestToHana	Ingest product data in parallel from CSV files into a HANA table with a long-running graph, offering at-least-once guarantee between multiple graph runs.
Ingest Files into Vora	com.sap.scenarioTemplates.datalake-ToDatabase.ingestToVora	Ingest product data in parallel from CSV files into a Vora table with a long-running graph, offering at-least-once guarantee between multiple graph runs.

For more information about setting up and running each template, please see [Scenario Templates](#) in the [Repository Objects Reference for SAP Data Hub](#).

6.2 E(T)L from Database

These graphs show how to replicate data from different databases into file storages or other databases.

Available Templates

Table 4:

Template	Path	Description
Initial Load from AnyDB (parallel)	com.sap.scenarioTemplates.ETL-FromDB.cdclInitialLoad	Read contents from an Oracle table and load it into files stored on a connected (cloud) storage.

Template	Path	Description
Initial Load + Delta Extraction from AnyDB	com.sap.scenarioTemplates.ETL-FromDB.cdcGraphGenerator	Use the CDC Graph Generator operator for replication of relational databases. The CDC Graph Generator operator generates the required SQL Scripts so users can capture changes from a database source. The graphs generated from this graph capture the changes.

To learn how to set up and run each scenario template, see [Scenario Templates](#) in the [Repository Objects Reference for SAP Data Hub](#).

6.3 ABAP with Data Lakes

These graphs show how to ingest ABAP Table or CDS View data from S/4 HANA and Business Suite Systems into a (cloud) storage.

For both data source types there are example graphs that showcase how to supply data lakes using a full and/or delta load mechanism.

A typical template scenario consists of a reader operator (SLT Connector or CDS Reader) that runs on a connected ABAP system, which streams the data into a pipeline with a file writer or a Kafka producer operator.

i Note

The ABAP system needs to fulfill the prerequisites documented in [2835207](#) before it can be connected to SAP Data Hub.

Available Templates

Table 5:

Template	Path	Description
Data Extraction using SLT to a File Store	com.sap.scenarioTemplates.ABAP.SLTtoFile	Connect to an SAP Landscape Replication Server (SLT) configuration to read table data and write the data to a (cloud) storage or data lake.
Data Extraction using SLT to KAFKA	com.sap.scenarioTemplates.ABAP.SLTtoKafka	Connect to an SAP Landscape Replication Server (SLT) configuration to read table data and feed the data into a Kafka pipeline.

Template	Path	Description
Data Extraction from SAP S/4HANA CDS View to a File Store	com.sap.scenarioTemplates.ABAP.CDSstoFile	Connect to an S/4 HANA system to read CDS View Data and write the data to a cloud storage or data lake.
Data Extraction from SAP S/4HANA CDS View to KAFKA	com.sap.scenarioTemplates.ABAP.SLTtoKafka	Connect to an S/4 HANA system to read CDS View Data and feed the data into a Kafka pipeline.

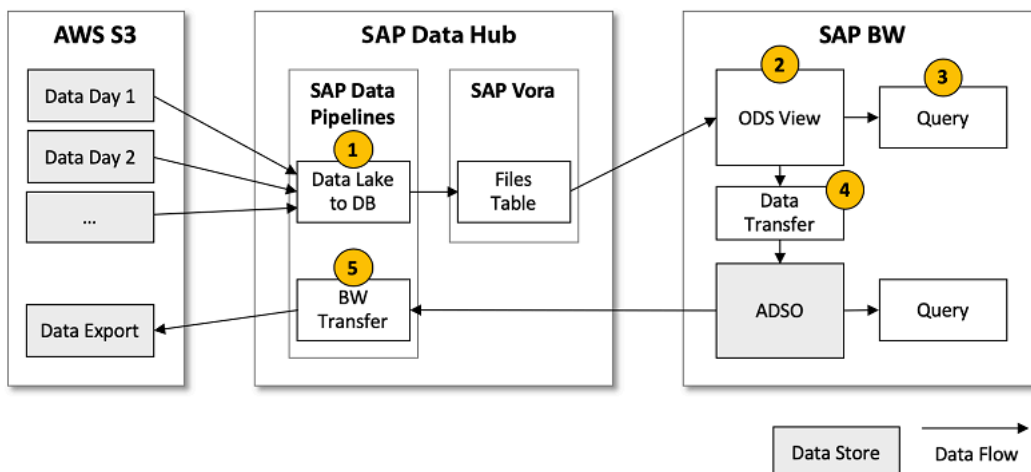
To learn how to set up and run each scenario template, see [Scenario Templates](#) in the [Repository Objects Reference for SAP Data Hub](#).

6.4 BW with Data Lakes

SAP Data Hub allows connecting SAP BW Systems to Data Lakes (i.e., to AWS S3 or Azure ADL).

In the following scenario we detail use cases to make data on a Data Lake available to SAP BW by using SAP Data Hub. We also provide an example pipeline to make data in SAP BW (i.e., in DataStore Objects) available for processing and storage in Data Lakes.

The scenario is comprised of the following main use case as shown in the figure below:



- Register new data from a Data Lake to a `Vora` file engine table;
- Make the registered data in a `Vora` table available to SAP BW by using an ODS view;
- Query the `Vora` data from BW by using a query object on the Open ODS view;
- Insert new data registered to the `Vora` table into an ADSO (delta load) for complex BW queries;
- Access data of an ADSO from a data pipeline for further processing or exporting to a Data Lake.

Setup and examples of the such cases are presented below.

Related Information

[Register New Data from Data Lake to Vora Table \[page 63\]](#)

[Setup BW to Access Vora Table \[page 65\]](#)

[Create an Example Scenario \[page 66\]](#)

[Query Vora Table Using Query Objects \[page 71\]](#)

[Insert New Data from ODS View to a Data Store \(Advanced\) Object \[page 72\]](#)

[Retrieve Data from BW \[page 75\]](#)

6.4.1 Register New Data from Data Lake to Vora Table

Data is often arriving regularly on a specific Data Lake inbox location. For example, new IoT records are put in daily or hourly organized folders and files. This data should be made available to a SAP BW system for reporting.

We assume to receive a set of files with new device records in a daily fashion:

```
s3://iot-projects/inbox/2019-01-01/file_0001.csv
s3://iot-projects/inbox/2019-01-01/file_0002.csv
s3://iot-projects/inbox/2019-01-01/file_0003.csv
s3://iot-projects/inbox/2019-01-02/file_0001.csv
s3://iot-projects/inbox/2019-01-02/file_0002.csv
s3://iot-projects/inbox/2019-01-02/file_0003.csv
...
```

In our example, the device records in the files are comprised of time information, device and action IDs, and sensor values:

```
1556661600000448,2019-05-01,dev-4295366171,sys.signal,75.03,5CDB3A4508-
dev-4295366171,54.0,app.maps,,75.03,,,,,2019-05-01
00:00:00.000448,2019,201905,20190501,000000,1
1556661600000679,2019-05-01,dev-4294974315,app.clock,1,5CDB3A6008-
dev-4294974315,49.0,app.clock,,,,,2019-05-01
00:00:00.000679,2019,201905,20190501,000000,1
1556661600000735,2019-05-01,dev-4295361882,app.call,1,5CDB3A6008-
dev-4295361882,50.0,app.call,,,,,2019-05-01
00:00:00.000735,2019,201905,20190501,000000,1
1556661600000940,2019-05-01,dev-4295110634,sys.gps,
51.797441;11.063932,5CDB3A5808-dev-4295110634,58.0,app.call,,,,,
51.797441,11.063932,2019-05-01 00:00:00.000940,2019,201905,20190501,000000,1
1556661601000523,2019-05-01,dev-4295360021,sys.gps,51.797441;0.063932,5CDB3A5208-
dev-4295360021,56.0,app.clock,,,,,51.797441,0.063932,2019-05-01
00:00:01.000523,2019,201905,20190501,000001,1
...
```

File Registration using Data Pipelines

In order to make the new files available to SAP BW we will register the files in a Vora `file` engine table.

i Note

By using a file engine table, the data is not moved to a Vora table but only the files are registered to belong to the table. A query against the table will then access the data on the underlying file system.

To register the files in a regular fashion we use the following graph, which can be found in the [Scenario Templates](#) category in the SAP Data Hub Modeler:

Table 6:

Template	Path	Description
Load Files into Vora	com.sap.scenarioTemplates.datalake-ToDatabase.loadToVora	Load product data from CSV files into a Vora table supporting SQL on Files, offering at-least-once guarantee between multiple graph runs.

It checks for new files in a configured inbox location and registers the new files to the configured Vora table. Since this graph is aware of the already registered files, the graph can be executed in a daily schedule to add the new files to the Vora table regularly.

To configure the Vora Loader in the graph for our sample data, you need to make the following changes:

1. In this example, you would add the following SQL statement to the **Init Statements** property of the Vora Loader. This defines the Vora table FT_RECORDS_RAW in the diuser schema:

```
CREATE TABLE IF NOT EXISTS "default\diuser".FT_RECORDS_RAW(
  TS BIGINT,
  DS DATE,
  DID varchar(16),
  EVT_KEY varchar(32),
  EVT_VAL varchar(32),
  SID varchar(32),
  S_LENGTH float,
  APP varchar(16),
  S_PULSE float,
  S_PHONE float,
  S_BATT float,
  S_LAT float,
  S_LNG float,
  TIMEST timestamp,
  CALYEAR varchar(4),
  CALMONTH varchar(6),
  CALDAY varchar(8),
  TIMS varchar(6),
  CNT BIGINT
) WITH TYPE DATASOURCE ENGINE 'FILES';
```

2. Set the Tablename property to FT_RECORDS_RAW.
The statement creates a file engine table named FT_RECORDS_RAW.

i Note

The "default\diuser" schema indicates that the table is created for the user diuser in the default tenant.

The table definition can either be put in the **Init Statements** property of the Vora Loader operator as explained above, or be executed in Vora Tools before running the graph. Then, the correct table is selected by the **Tablename** property.

3. Click the Vora Loader operator **Column Options** edit button and, within the **Edit property 'Column Options'** edit box, click the **JSON** button. Delete all of the lines within the **Column Options** JSON array.
4. Click **Save**.

Data Views

In our example data, we have a huge number of values which may not be needed for the BW reporting. In this case you can create a view in `Vora Tools` which is used as an access point for BW. For this example, we want to make all values available to BW but we still want to create a view to organize the exposed information in a different schema, such as the one below:

```
CREATE VIEW "default".FT_RECORDS_TEST_FILES AS
SELECT * FROM "default\diuser".FT_RECORDS_RAW;
```

Continue to [Setup BW to Access Vora Table \[page 65\]](#) to learn more.

6.4.2 Setup BW to Access Vora Table

SAP BW allows you to access Vora tables using the Open ODS View feature. This technology provides access to remote sources inside of BW.

Under the hood, the Open ODS View uses SAP HANA Smart Data Access functionality (SDA).

In the following sections, we will give a brief step-by-step overview on how to configure the SAP BW system to access the Vora table using SDA. Details to configure the connection and to create ODS views can be found in the official documentation.

Setting up the Remote Source

In order to connect an ODS view to a Vora table, you need to create a remote source in the HANA database connected to SAP BW. Setting up a remote source can be done using HANA Tools or directly as a SQL statement. [Please refer to the detailed documentation here](#).

The following statement creates a remote source pointing to the Vora database. It needs to be executed as a privileged user in the HANA database connected to the BW system.

```
CREATE REMOTE SOURCE BW_Q_SCENARIO adapter voraodbc
CONFIGURATION 'ServerNode=<URL-TO-VORA-ENDPOINT>:<HANA-
PORT>;Driver=libodbcHDB;sslValidateCertificate=true;encrypt=true;PACKETSIZE=32000
000'
WITH CREDENTIAL TYPE 'PASSWORD' USING 'user=default\diuser;password=XXXXXX';
```

Please refer to [Accessing SAP Vora from SAP HANA](#) for details about the connection settings and how to open the <VORA-ENDPOINT> in SAP Data Hub.

i Note

We are using a huge packetsize (`PACKETSIZE=32000000`, 32 Megabyte) since we assume a reasonable number of queries with huge data size against the Vora table. The setting above will allow to transfer Gigabytes of data in a few seconds on a standard SAP Data Hub setup.

BW might generate join queries between data on Vora and data residing on HANA. HANA SDA will then try to optimize a join query by re-allocating the smaller table to the system with larger data (normally the Vora system). To check for join reallocation capabilities the user of the remote source connection needs to have privileges to access the `SYS.VORA_CAPABILITIES` table.

For this, log into SAP Data Hub as an admin user (or to the `system` tenant) and execute the following statement:

```
GRANT SELECT ON SYS.VORA_CAPABILITIES TO "default\diuser";
```

Create ODS View

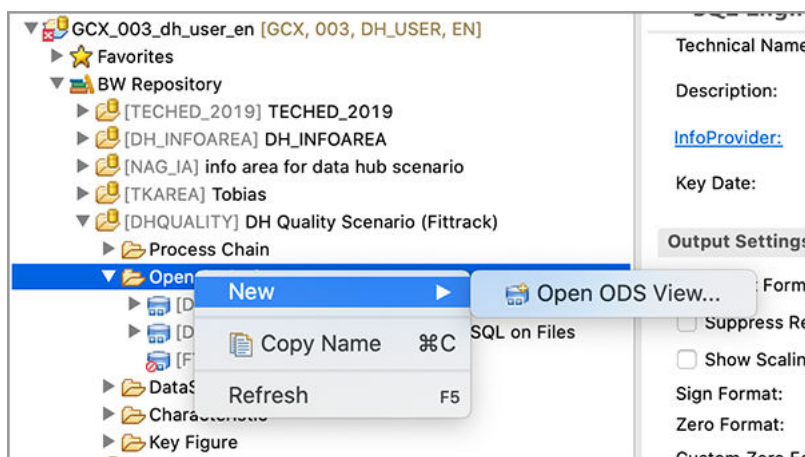
Once the remote source has been set up, you can create an ODS View using the HANA Studio BW Modeling perspective. Please refer to the SAP Note [2198480](#) for details.

Continue to [Create an Example Scenario \[page 66\]](#) to learn more.

6.4.3 Create an Example Scenario

Follow these steps to create an example scenario.

1. Go to your *Info Area* and create a new *Open ODS View*:



2. Complete the **InfoArea** (if not filled already) and provide a name for the ODS view.
3. Choose the semantics of the data (in our case, this is **Facts** data):

Open ODS View

Create Open ODS View

BW Project:* GCX_003_dh_user_en Browse...

InfoArea:* DHQUALITY Browse...

Add to Favorites

Name:* DH_FT_REC_VT

Description:

Create with Proposal

Copy From: Browse...

Semantics:*

- Facts
- Master Data
- Texts

? < Back Next > Cancel Finish

4. Choose **Big Data** and **Remote Source** as the source type:

Select Source Type

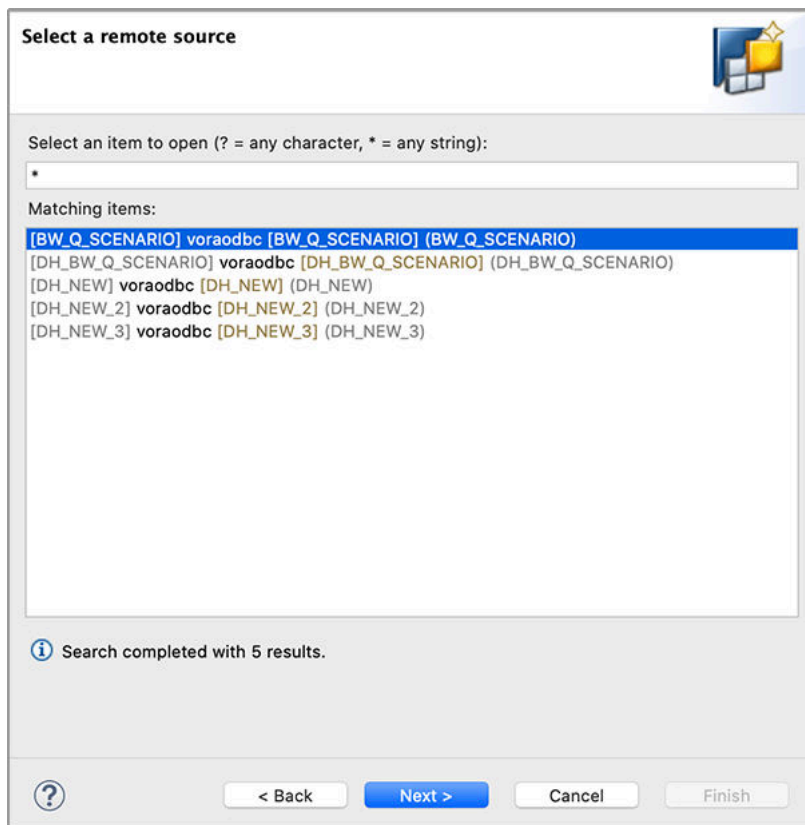
Source Type:*

- DataSource (BW)
- Database Table or View
- Virtual table using SAP HANA Smart Data Access
- Big Data
- DataStore Object (advanced)
- Transformation

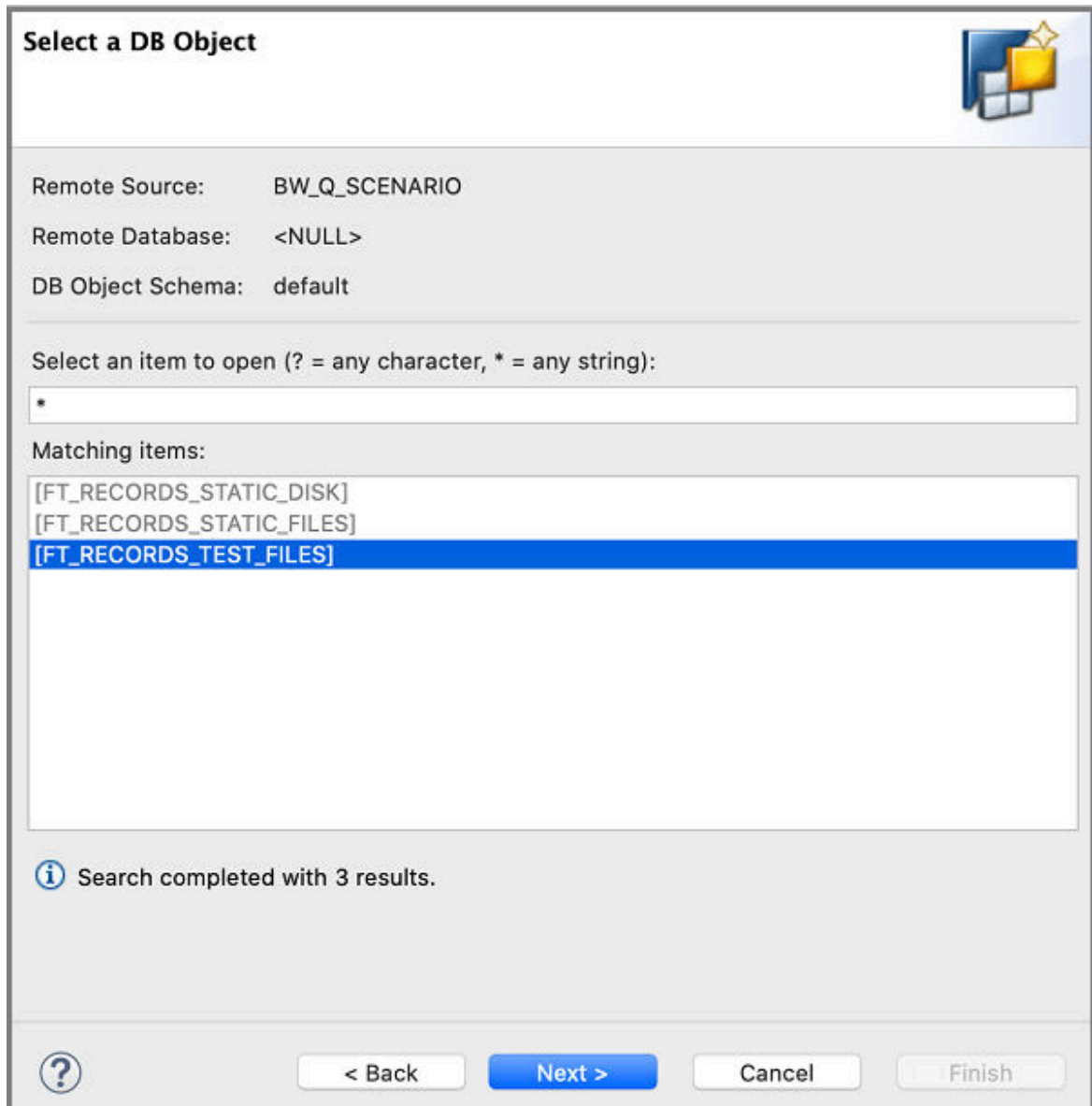
- Existing Source System
- Remote Source

? < Back Next > Cancel Finish

5. Select the remote source system created before. To find the existing sources, you can search using a * wildcard.
6. Select the **BW_Q_SCENARIO** remote source:

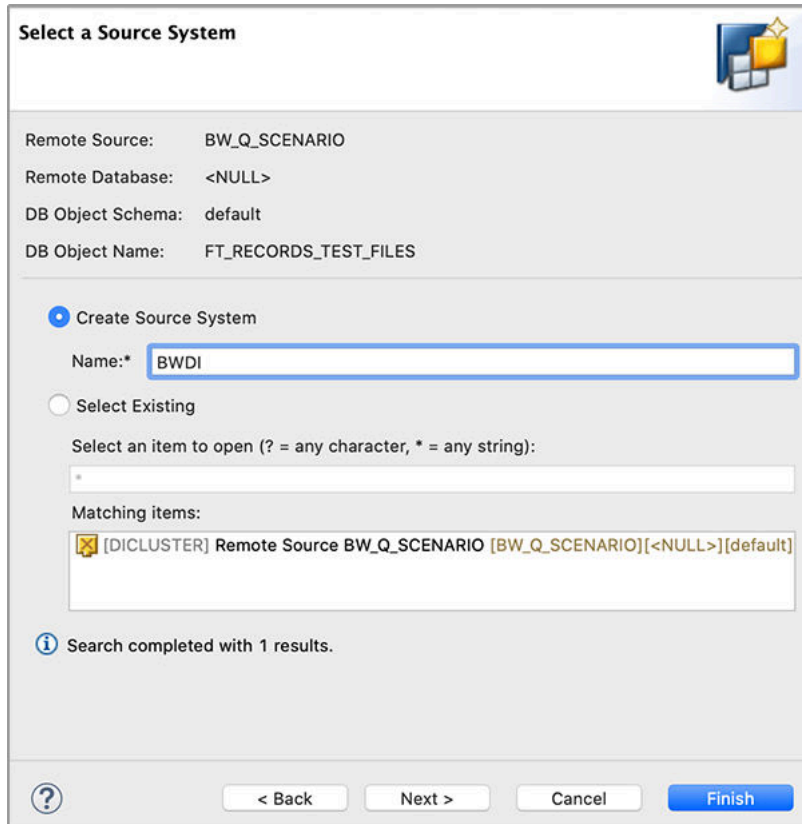


7. Select the Vora table you want to connect to the *Open ODS* view by selecting it from the list. To find the existing table, you can search using a * wildcard:



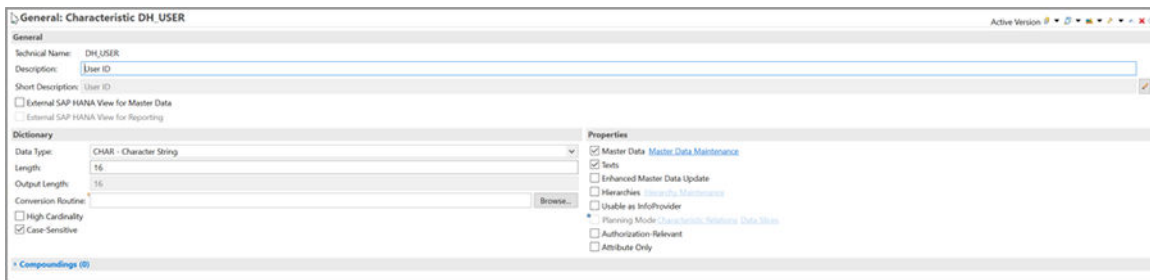
8. If this is the first ODS view using the remote source, choose a name for the source system.

Otherwise, you can select the already created source system (*BW_Q_SCENARIO* in this example):

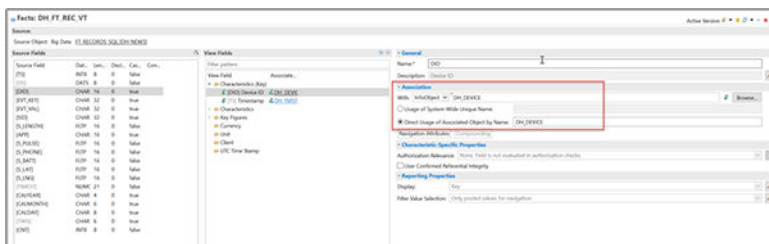


The ODS View could now be used for reporting. However, for better mapping of the ODS View fields to other BW objects (i.e., Data Store Objects), we can map the fields to dedicated Info Objects.

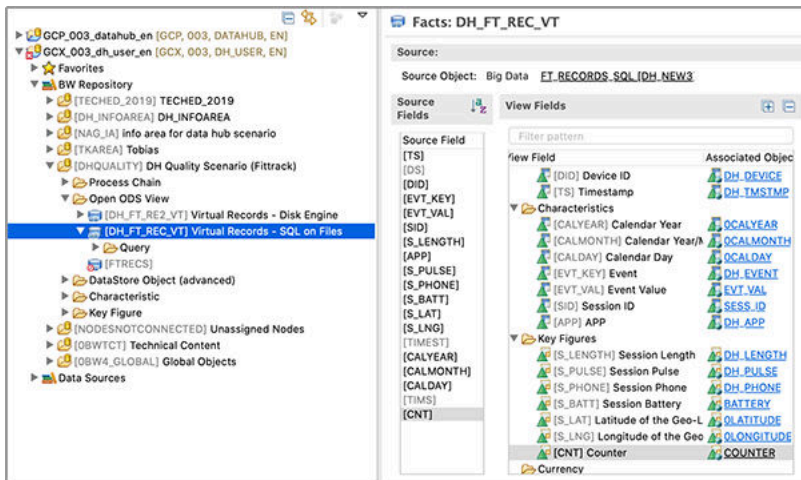
The Info Object for the *DevideID* field would be used in several Data Store Objects (including the *ODS View*) and looks as follows:



It can now be mapped to the field of the ODS View:



The final *ODS View* would look similar to the example below:



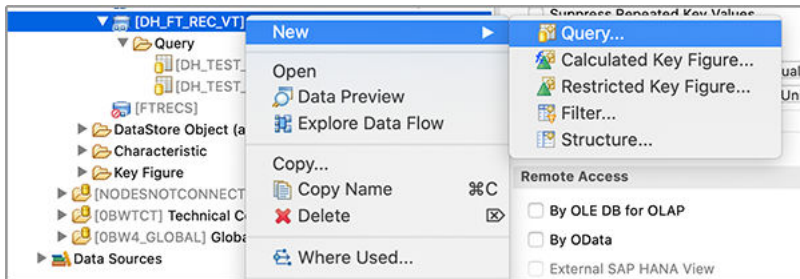
Continue to [Query Vora Table Using Query Objects \[page 71\]](#) to learn more.

6.4.4 Query Vora Table Using Query Objects

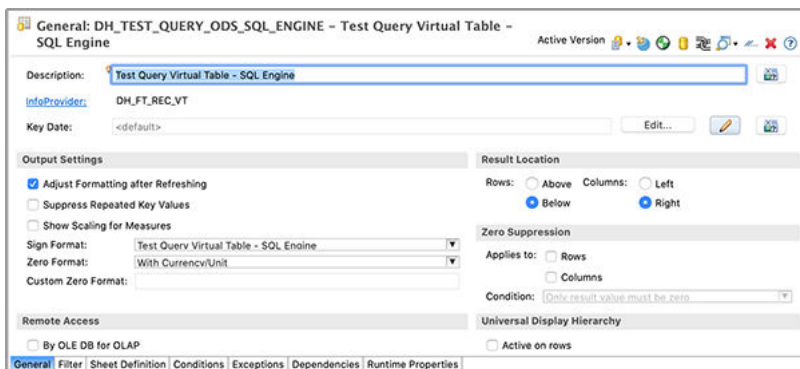
Given the ODS view, we can now create queries to allow for reporting on the files registered to the Vora table.

Creating Query

1. Right click on the *ODS view* in the BW Modeling perspective and choose *New* → *Query*;

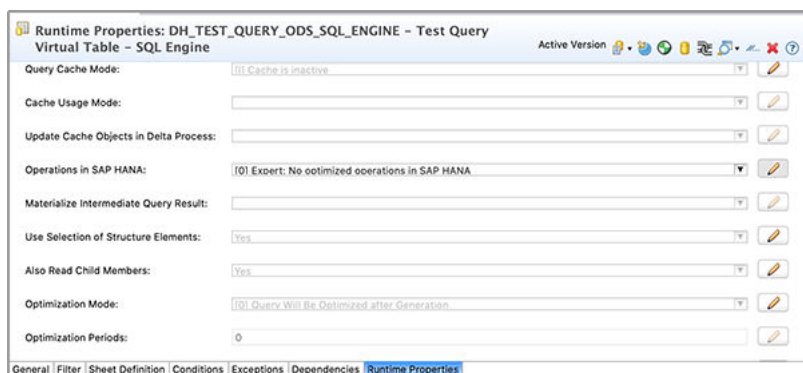


2. Choose a description of the query, set filters, and define the available characteristics:



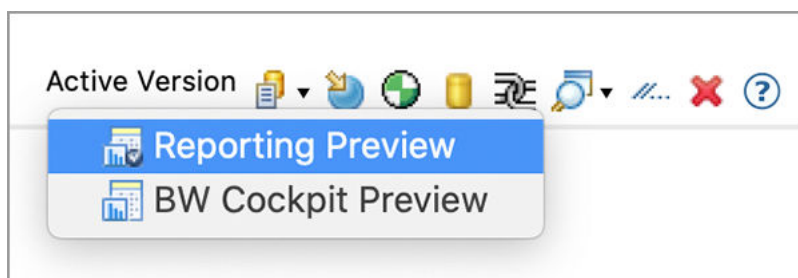
3. SAP BW generates a SQL query against the underlying HANA SDA table. For better performance, the BW SQL generation needs to be changed:

1. Click the *Runtime Properties* tab on the bottom of the query page
2. Go to the *Operations* in SAP HANA property and set value to **[0] Expert: No optimized operations** in SAP HANA



Query Execution

There are several ways to execute a query in SAP BW. From the BW Modeling perspective, the easiest way to test your query is to use the built in *preview* feature. For this, click the *preview icon* on the top right of the query page and choose *Reporting Preview*.



Continue to [Insert New Data from ODS View to a Data Store \(Advanced\) Object \[page 72\]](#) to learn more.

6.4.5 Insert New Data from ODS View to a Data Store (Advanced) Object

Since our ODS view points to a Vora table using SDA, there might be queries that have very slow performance because they need to be pushed down to Vora by BW.

For example, if we model a Key Figure that utilizes Exception Aggregation (i.e., count distinct) the generated BW SQL query might fetch almost all records from the Vora table before computing the result. This can slow down the query if the number of records in Vora gets very high (several GBs).

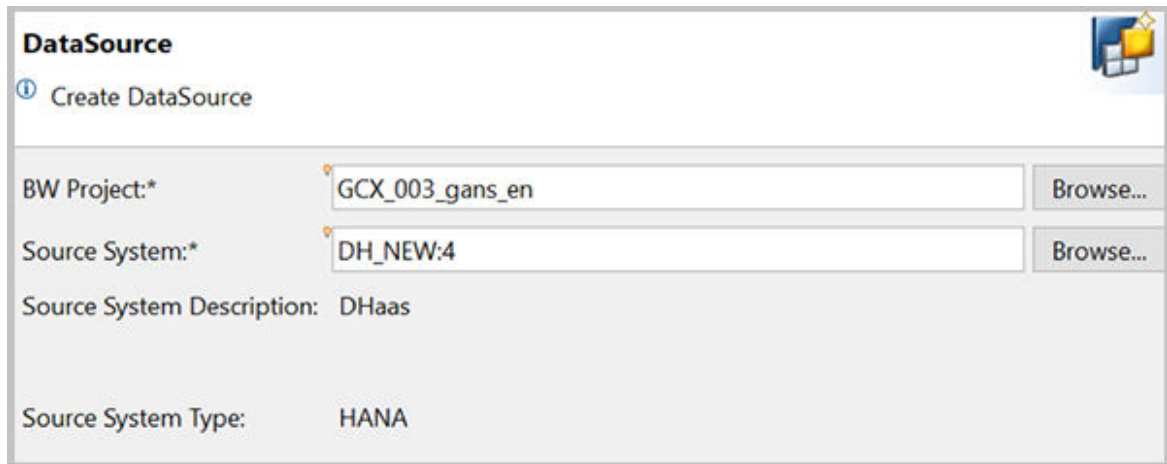
In case report generation should have very low latency (sub-seconds) and the query cannot be pushed down to Vora by BW (Exception Aggregation) a usual performance optimization is to insert the ODS data into an Data Store (Advanced) object (ADSO). In the following we will make use of the standard SAP BW functionality to insert ODS View data into an ADSO.

Create ADSO from ODS View

For details, please refer to the official documentation about ADSOs .

Follow the steps below for an overview according to our example:

1. Create a Data Source inside of the existing Source System (see ODS steps above)



DataSource

Create DataSource

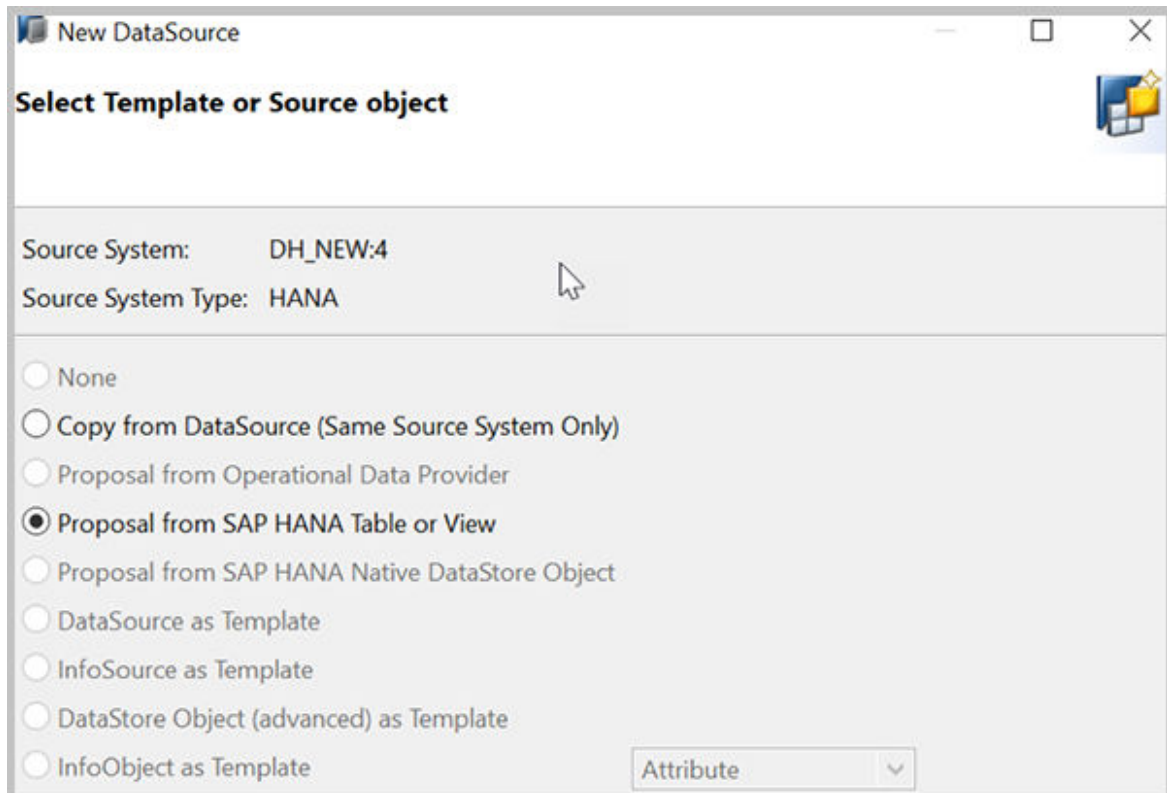
BW Project:* GCX_003_gans_en Browse...

Source System:* DH_NEW:4 Browse...

Source System Description: DHaas

Source System Type: HANA

2. Select the Vora Table



New DataSource

Select Template or Source object

Source System: DH_NEW:4

Source System Type: HANA

None

Copy from DataSource (Same Source System Only)

Proposal from Operational Data Provider

Proposal from SAP HANA Table or View

Proposal from SAP HANA Native DataStore Object

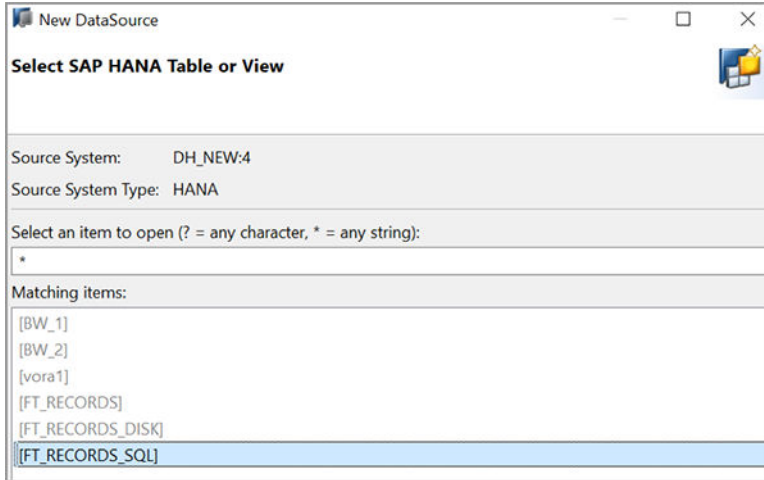
DataSource as Template

InfoSource as Template

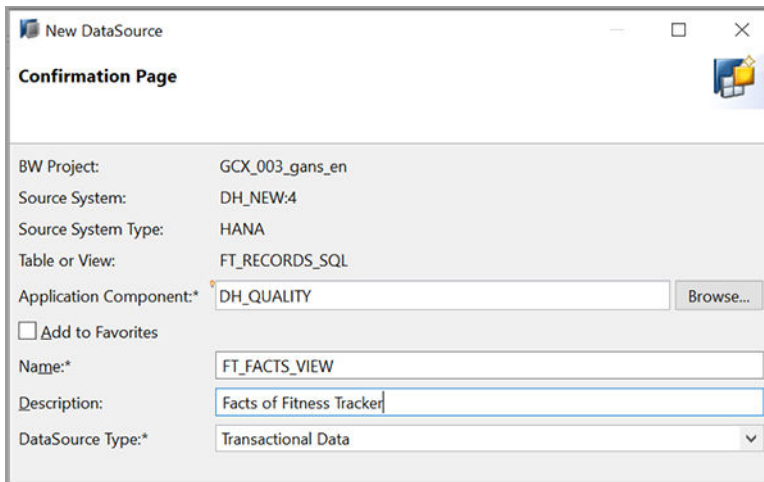
DataStore Object (advanced) as Template

InfoObject as Template

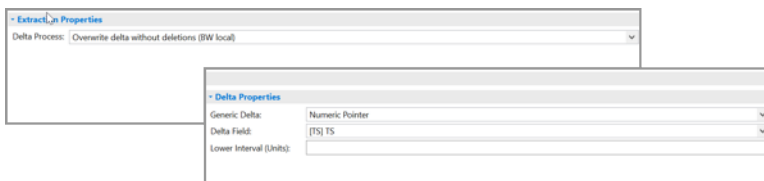
Attribute



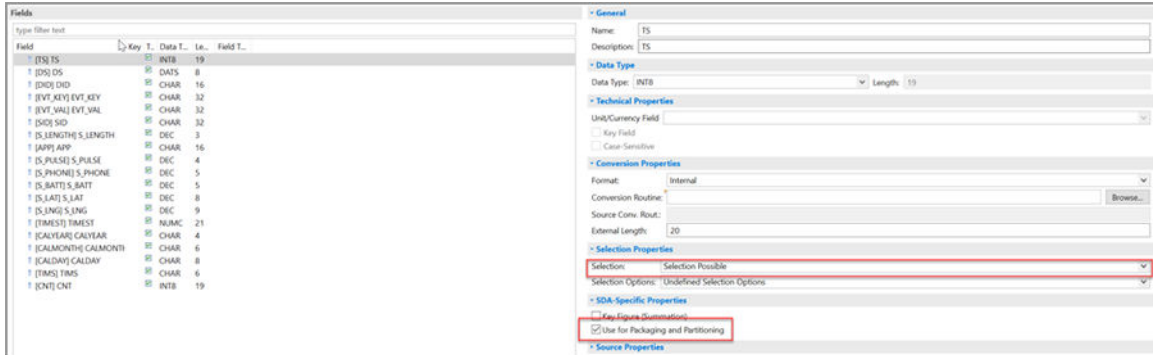
3. Enter **Name**, **Description** and **DataSource Type**



4. Confirm and activate the Data Source
5. To enable delta capability of your Data Source, go to tab *Extraction*, activate the *Delta Process*, and choose the *Delta Field*



6. Go to the *Fields* tab and activate the following delta fields: **Selection possible** and **Packaging and Partitioning**



Continue to [Retrieve Data from BW \[page 75\]](#) to learn more.

6.4.6 Retrieve Data from BW

We now want to use data in an Info Object (includes ADSOs) inside of a data pipeline, i.e., to export data to a Data Lake.

The following graph shows how data is fetched from a BW source and written into a target (cloud file storage).

A typical template scenario consists of a workflow trigger operator, a data transfer operator and a workflow terminator operator that runs on a connected ABAP system.

Table 7:

Template	Path	Description
BW HANA View to File	com.sap.scenarioTemplates.BW.bwDataTransferHANAView	Transfer data from SAP BW Info Objects and/or Data Store Objects and write into CSV files on a cloud file storage system.

6.5 Data Processing with Scripting Languages

These graphs show how to manipulate data with scripting languages.

A typical scenario consists of:

- Reading some input data from a storage, such as a file or a database table;
- Applying a processing algorithm to the data that is implemented in a scripting language such as Javascript, Node, Python, or R;
- Writing the results of the data processing to another storage which may or may not be the same as the one providing the input data.

Available Templates

Table 8: Javascript

Template	Path	Description
Simple Javascript File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2FJavascript	File data manipulation using Javascript and writing the manipulated data back to another file.
Simple Javascript File-to-DB Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2DBJavascript	File data manipulation using Javascript and writing the manipulated data to an SAP HANA database table.
Simple Javascript DB-to-File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleDB2FJavascript	Manipulation of data from an SAP HANA database table using Javascript and writing the manipulated data to a file.
Javascript File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.F2FJavascript	File data manipulation with a custom Javascript operator and writing the manipulated data back to another file.

Table 9: Node

Template	Path	Description
Simple Node.js File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2FNode	File data manipulation using Node.js and writing the manipulated data back to another file.

Table 10: Python

Template	Path	Description
Simple Python File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2FPython	File data manipulation using Python and writing the manipulated data back to another file.
Simple Python File-to-DB Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2DBPython	File data manipulation using Python and writing the manipulated data back to the SAP Data Hub built-in Vora database.
Python File Data Manipulation with Pandas	com.sap.scenarioTemplates.customDataProcessing.F2FPython	File data manipulation with a custom Python operator using the Pandas module and writing the manipulated data back to another file.

Table 11: R

Template	Path	Description
Simple R File Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2FR	File data manipulation using R and writing the manipulated data back to another file.
Simple R File-to-DB Data Manipulation	com.sap.scenarioTemplates.customDataProcessing.simpleF2DBR	File data manipulation using R and writing the manipulated data to an SAP HANA database table.

To learn how to set up and run each scenario template, see [Scenario Templates](#) in the [Repository Objects Reference for SAP Data Hub](#).

7 Creating Operators

You can use the SAP Data Hub Modeler to create your own operators and execute them in the graphs. The Modeler provides a form-based editor to create operators.

Context

An operator represents a vertex of a graph. An operator is a reactive component, hence isn't intended to terminate, and reacts only to events from the environment. The operators that you create in the modeler are derived from the base operators that the application provides.

Procedure

1. Start the SAP Data Hub Modeler.
2. Select a folder.

You create operators within a folder structure in the Modeler repository.

- a. In the navigation pane, choose the *Repository* tab.

The application displays all graphs, operators, and Dockerfiles, and more that are available in your repository. These objects are grouped under specific folders.

- b. Right-click the *Operators* section and choose *Create Folder*.

→ Tip

Optionally, you can also create an operator by clicking + (Create Operator) in the *Operators* tab. If you're creating the operator from outside of the folder, then enter the name of the operator along with the fully qualified path.

- c. Provide a name for the folder and choose *OK*.

This operation creates a folder within which you can create and define the operator.

i Note

If you want to create a folder structure (sub-directories) within the above folder, right-click the root folder and choose *Create Folder*.

3. Create an operator.
 - a. Right-click the folder in which you want to create the operator and choose the *Create Operator* menu option.
 - b. In the *Name* text field, provide a name for the new operator.
 - c. In the *Display Name* text field, provide a display name for the operator.

The application uses the display name for the operator in the UI. You can use this name to search for the operator in the application.

- d. In the *Base Operator* dropdown list, select the required base operator.

The operators that you create are derived from the built-in base operators that the application provides.

- e. Choose *OK*.

The application opens the operator editor. Use this form-based editor to define the operator and its configurations.

4. Define the operator.


Use the operator editor to define your operator.


Action	Steps
Define ports	<ol style="list-style-type: none"> 1. In the operator editor, choose the <i>Ports</i> tab to define the input and output ports. 2. In the <i>Input Ports</i> section, choose + (Add input port) to define the input port name and input port type. 3. In the <i>Output Ports</i> section, choose + (Add output port) to define the output port name and output port type.
Define tags	<p>Tags describe the runtime requirements of operators and are the annotations of Dockerfiles that the application provides. If multiple implementations exist for the operator, the application displays the different subengines in which you can execute the operator. For each subengine, you can further associate them with tags from the database. To define tags,</p> <ol style="list-style-type: none"> 1. In the operator editor, choose the <i>Tags</i> tab. 2. In the <i>Tags</i> section, choose + (Add tag) if you want to select a tag available in the tag database. 3. For each subengine, choose + (Add tag) to select a tag available in the tag database.

Action

Steps

Define configurations

1. In the operator editor, choose the *Configuration* tab.
2. If you want the application to auto propose a type definition based on the existing operator configurations, choose *Generate Config Schema*.
3. You can edit the auto proposed type and modify the type properties. In the *Parameters* section, choose  (Open Type Schema) to edit the auto proposed type properties.
4. (Optional) If you want to import an existing type definition and reuse it to define the operator configurations, choose *Import*, and select the required type.

Types allow you to enable semantic type validations on top of the configuration parameters and to define UI conditions. You can create new types or edit an existing type definition available in the repository and reuse it to define the operator configuration. In the navigation pane, choose the *Types* tab to create new types. After importing a type, if you want to edit its properties, in the *Parameters* section, choose  (Open Type Schema).

Note


By default, the type definition of script-based operators such as JavaScript, Python, Go operators, and so on, include a *Script* property. This property enables users to view or modify the operator script when creating a graph. If you want to hide the script to users working with this operator or if you don't want them to modify the script, you can do so by editing this *Script* property in the type schema. Select the property and set the *Visibility* value to *Hidden*.

5. (Optional) If you want to add and define new configuration parameters, in the *Operator Configuration* section, choose + (Add Parameter).

Provide the configuration parameter name, select the parameter type, and provide a parameter value.


Remember

For some base operator, the application doesn't allow you to define new configuration parameters. In such cases, the + (Add Parameter) is disabled, and you can only edit the existing parameter values.

Action	Steps
Add scripts	<p>In graphs, you can use operators that help execute script snippets when the graph is executed. If you want to include such scripts in the operator definition,</p> <ol style="list-style-type: none"> 1. In the operator editor, choose the <i>Script</i> tab. 2. In the inline editor, enter the required script. You can also change the default programming language of the editor. In the top-right corner of the editor, select the required language. 3. If you want to upload a script file from your system, in the top left corner of the editor, choose <i>Upload File</i> and click the  icon to browse and select the required file.
Add operator documentation	<p>You can maintain documentation for operators, or if documentation already exists for an operator, then you can modify the documentation. The documentation can contain information, for example, the operator's configuration parameters, input ports, output ports, and more. Adding documentation helps other users when working with this operator.</p> <ol style="list-style-type: none"> 1. In the operator editor toolbar, choose the <i>Documentation</i> tab. 2. Provide the required documentation. 3. In the editor toolbar, choose <i>Save</i>. <div data-bbox="783 1025 1402 1173" style="background-color: #f0f0f0; padding: 10px; border: 1px solid #ccc;"> <p>→ Tip We recommend you to use the Markdown format to maintain the operator documentation.</p> </div>

5. (Optional) Select a display icon for the operator.


When you use the operator in the graph editor, if you have not uploaded the required display icon, the application uses a default operator display icon. You can replace the display icon with any of other icons that the application provides, or you can upload your own icon in Scalable Vector Graphics (SVG) format and use it for the operator display.

- In the operator editor, select the operator default icon.
- If you want to use a default that the application provides, in the *Icon* dropdown list, select the required icon.
- If you want to upload an SVG file, choose  and browse to the required `.svg` file.
- Choose *OK*.


The application uses the new icon for the operator on the graph editor.

6. (Optional) Upload auxiliary files.

You can upload such auxiliary files to the repository that the operator may access or execute it at runtime. It can be any kind of file, for example, binary executables such as a JAR file that the application must execute when executing the operator.

- In the operator editor toolbar, choose  (Upload Auxiliary File).

These files are stored along with the operators in the operator directory. You can access these files from the SAP Data Hub System Management application. Alternatively, you can also upload files from the *Repository* tab.

7. (Optional) Edit an existing operator.
 - a. If you want to edit an existing operator, in the navigation pane, choose the *Operators* tab.
 - b. Right-click the operator, which you want to edit and choose *Edit* menu option.
8. View JSON definition.
 - a. After defining the operator, if you want to view the JSON definition for the operator, in the editor toolbar, switch to *JSON*.
9. Save changes.
 - a. In the editor toolbar, choose  (Save) to save the operator.
You can now access this operator from the *Operators* tab (under the *Others* section) in the navigation pane, and use it for creating graphs.

Related Information

[Operators \[page 82\]](#)

[Port Types \[page 89\]](#)

[Create Types \[page 221\]](#)

7.1 Operators

An operator represents a vertex of a graph. An operator is a reactive component that reacts only to events from the environment.

An event from the environment is a message delivered to the operator through its input ports. The operator can interact with the environment through its output ports. The operators are unaware of the graph in which it is defined and the source and target of its incoming and outgoing connections.

i Note

The events can also be internal to the operator. For example, clock ticks.

The following image is a sample operator along with the input ports and output ports. Each port is associated with a port type, and the application uses color codes to identify compatible port types.



The application provides certain base operators. You can use these base operators in a graph for productive use cases, or customize and use them to derive other operators (for other use cases).

The predefined operators within the application can be classified as,

- Connectors to messaging systems (Kafka, MQTT, NATS, and WAMP)
- Connectors to store and read data (File, HDFS, S3)
- Connectors to databases (SAP HANA, SAP Vora)
- Javascript engine for manipulating arbitrary data
- Process operators to execute any program (stateful and stateless) for manipulating data in the graph.
- Operators for type conversion
- Operators for digital signal processing (beta)
- Operators for machine learning (beta)
- Operators for image processing (beta)

Operator behavior at runtime

Operators require a certain runtime environment for their execution. For example, if an operator executes some JavaScript code, then the operator requires an environment with a JavaScript engine. The Modeler application provides certain predefined environments for operators, and these environments are made available to users as a library of Dockerfiles.

When you execute a graph, the application translates each operator in the graph into processes. It then searches the Dockerfiles for an environment suitable for the operator execution and instantiates a docker image.

i Note

The docker image with the environment and the operator process is executed on a Kubernetes cluster.

Related Information

[Operator Details \[page 83\]](#)

[Port Types \[page 89\]](#)

[Graph Execution \[page 29\]](#)

[Creating Operators \[page 78\]](#)

7.1.1 Operator Details

Every operator has an ID (also known as name) and a title (also known as the description). The ID is its unique identifier, with a strict format. The title is what the graphical interface displays.

All the operators available when creating a graph are known as extensions (referred to as simply operators). They extend base operators, which are visible when creating a new operator.


The extension is expressed by a file in the Modeler file system. This file must be named `operator.json` and its folder hierarchy must match its ID. This is automatically done when an operator is created following Create Operators.

E.g.:

```
ID: 'com.sap.foo.bar'  
Filepath: './operators/com/sap/foo/bar/operator.json'
```

Operator JSON

The operator.json file is responsible for the definition of the operator, including the graphical interface information. It has the following structure (mandatory fields are denoted by *):

- `description`: the operator title;
- `*icon`: the operator icon as a Font Awesome icon name available at <https://fontawesome.com/icons/>  ;
- `*iconsrc`: the path to the SVG icon file - path is relative to the operator.json;
- `*component`: the base operator ID to be extended;
- `inports`: an array of input ports;
- `outports`: an array of output ports;
- `config`: a map of configuration parameters, mapping a configuration parameter ID to its default value;
- `*config.$type: $type` field pointing to its schema. More...
- `tags`: a map of tags, mapping each tag ID to its default value;
- `enableportextension`: a boolean value that, if set to true, allows adding additional ports and configurations to the operator through the UI;
- `extensible`: a boolean value that, if set to true, allows a base operator to be extended;

Note

`subenginestags` do not exist in the filesystem, they are included on the operator JSON for UI purposes;
`icon` and `iconsrc` are mutually exclusive; any field may be derived from the base operator (`component`).

The operator.json should result in the following structure:

```
{  
  "description": "<operator-title>",  
  "icon": "<fontawesome-icon>",  
  "iconsrc": "<icon-file>",  
  "component": "base-operator-id",  
  "inports": [  
    {  
      "name": "<inport1-id>",  
      "type": "<inport1-type>"  
    },  
    ...  
  ],  
  "outports": [  
    {  
      "name": "<outport1-id>",  
      "type": "<outport1-type>"  
    },  
    ...  
  ],  
  "config": {  
    "<config-id>": "<config-value>",  
    ...  
  },  
}
```

```

    "tags": {
      "<tag-id>": "<tag-value>",
      ...
    },
    "enableportextension": <true/false>,
    "extensible": <true/false>,
  }

```

Example:

```

{
  "iconsrc": "read.svg",
  "component": "com.sap.storage.read",
  "config": {
    "$type": "http://sap.com/vflow/com.sap.storage.read.schema.json#"
  },
  "tags": {},
}

```

Naming

The documentation is presented in a README file in Markdown format.

The file must be named README.md and placed at the same operator.json folder if the documentation only makes sense for the extension - for multiple subengine implementations, where we have multiple operator.json, each of them should have a README in the same folder.

Documentation

The documentation is presented in a README file in Markdown format.

The file must be named README.md and placed at the same operator.json folder if the documentation only makes sense for the extension - for multiple subengine implementations, where we have multiple operator.json, each of them should have a README in the same folder.

README structure

It should follow the structure:

```

<operator-title>
===
<introduction>
<links-to-examples>
Configuration parameters
---
- <configuration-parameter-1>
- <configuration-parameter-2>
- ...
Input
---
- <input-port-1>
- <input-port-2>
- ...
Output
---
```

```
- <output-port-1>
- <output-port-2>
- ...
```

If an item list (parameters or ports) is empty, a single item None must be listed. E.g.:

```
Configuration parameters
---
- None
```

Introduction

Text with the content as follows:

```
- <configuration-id>
  (type <configuration-type>, default: <configuration-default>)
  <!-- mandatory: only if applicable --> mandatory:
  <!-- brief description --> Lorem ipsum dolor sit amet, consectetur
  adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
  aliqua.
  <!-- if needed, link to document with further description -->
  (Details are described here)[<link-to-config-docs>].
  - ID: `<configuration-id>`
  - Type: `<configuration-type>`
  <!-- Default: a value must be expressed according to its type formatting,
  e.g.:
  string -> `"value"`,
  int -> `42`,
  object -> `{ "k": ["v1, "v2"] }`,
  ...
  -->
  - Default: `<default-configuration-value>`
  <!-- Possible values: only valid for "enum" type -->
  - Possible values:
    - `<value-1>`
    - `<value-2>`
  <!-- Expected input: only valid when the "pattern" is set -->
  - Expected input: `<pattern-regex>`
  <!-- Additional specification fields may be provided -->
```

This should result in something like the following:

- Connection Protocol [mandatory]
Protocol used in the request to the service.
 - ID: connProtocol
 - Type: string
 - Default: "HTTP"
 - Possible values:
 - "HTTP"
 - "HTTPS"

Ports

Ports are identified by a unique ID (aka name).

They're formatted in the following way:

```
- <port-id> (type <port-type>): Express the parameter.
  If further is needed, document it in a separate file, and [link]()
  to it in this sentence. External documentation may be linked.
```

Configuration

Parameter naming follows the same standard as operator naming.

Configuration schema

A schema for the configuration must be given. It contains each parameter and further constraints for the UI.

It must be linked in the `operator.json`, like:

```
{
  ...
  "config": {
    "$type": "<$id-from-schema>"
  },
}
```

Each parameter in the schema must:

- point to its ID with the object's key;
- have a set title;
- use the most strict type;
- have a validation regex in pattern, if applicable;
- be listed as required if applicable.

The schema may be written manually or with the help of the Types panel of the graphical interface. It should either be saved as:

1. `configSchema.json` in the same `operator.json` folder;
2. `schema.json` in `/types/<operator-id>/schema.json`.

Approach 1. should be considered first (and is the one taken by the graphical interface tool).

A brief example of the schema is:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "$id": "http://sap.com/vflow/<operator.id>.schema.json",
  "title": "<schema-title>",
  "description": "<schema-description>",
  "type": "object",
  "properties": {
    "user": {
      "title": "User",
      "type": "string",
      "secure": true
    },
    "password": {
      "title": "Password",
      "type": "string",
      "secure": true,
      "format": "password"
    },
    "fooConnectionID": {
      "title": "Foo Connection ID",
      "description": "Connection ID used to connected to Foo",
      "type": "string",
```

```

        "format": "com.sap.dh.connection.id"
    },
    "isFoo": {
        "title": "Is Foo",
        "type": "boolean",
        "description": "Determines if operator is Foo.",
    },
    "reqMode": {
        "title": "Request Mode",
        "type": "string",
        "enum": [
            "Foo",
            "Bar"
        ]
    },
    "noOfReqs": {
        "title": "Number of Requests",
        "type": "integer",
        "description": "Number of Bar requests to be done.",
        "sap_vflow_constraints": {
            "ui_visibility": [
                {
                    "name": "reqMode",
                    "value": "Bar"
                }
            ]
        }
    },
    "filepath": {
        "title": "File path",
        "type": "string",
        "description": "File path to save request. Must start with '/'.",
        "pattern": "^\\/.*$"
    }
}

```

Schema Types

The available types for a configuration parameter are:

```
"array", "boolean", "integer", "number", "object", "string"
```

Find out more at <http://json-schema.org/draft-06/schema> ➔

Category

An operator is recommended to belong to at least one category, and may belong to more than one. A new operator can be added to the relevant categories, and new categories can also be created. This can be done manually, or through the graphical interface. To do it manually (this is what the graphical interface does):

- on the Repository tab, open `./general/ui/*:settings.json`;
- in the object "operatorCategories" ("graphCategories" for a graph), add your operator ID to the array of an existing category or create a new one. The category object has the following structure:

```
{
```



```

    "name": "Category Name",
    "entities": [
      "com.sap.foo.bar",
      "com.sap.operator.test"
    ]
  }

```

The category name follows the same operator title naming constraints, already mentioned here.

7.2 Port Types

The operator uses ports as an interface to communicate between operators in a graph.

A port definition includes the following:

- Input or output port; there are no specific error ports, use output ports to communicate error messages.
- Name of the port: A unique port name that consists of alphanumeric characters only.
- Port type.

A port type is a string with a defined structure having a mandatory base type and an optional semantic type. The latter can have a hierarchical substructure, separated with periods, and an optional wildcard at the end. It has the following form: `<base type>.<semantic type>`.

❖ Example

```

string.com.sap.base64.*
com.mycompany

```

Types with a wildcard are called incomplete types. A general port type specification may look like.

❖ Example

```

int64.com.sap.base64.*
[]blob.com.mycompany

```

You can use the semantics of the type specification to enrich types with additional information, which the owner of the types can use. However, the engine does not evaluate beyond the compatibility checks described below.

Base Types

All types that you can use to type a port are technically reduced to one of the following built-in base types:

Name	Description	Is compatible with type "any"	Array
any	generic type	yes	no
string	character sequence	yes	yes
blob	binary large object	yes	yes
int64	8 byte signed integer	yes	yes
float64	8 byte decimal number	yes	yes
byte	single character	yes	yes
message	structure with header and body	no	no
stream	unstructured data stream	no	no

The last column in the table indicates whether it is possible to use arrays of the type. For example, you can use `[]float64` but not `[]message`.

i Note

Not all subengines may support all array types.

Some use cases for pipeline-specific types are:

- The type `any` can be used if an operator is agnostic of the type and helps to avoid the redefinitions of the operator for each type. Typical examples are the multiplexer operators.
- The type `message` consists of a message header and the payload stored in the body. Messages have a size limit of currently 10 MB. This means that larger payloads have to be split up into chunks. For example, the *Read File* operator, where the header of the response messages contains the information to interpret the content of the body. In other scenarios, the input message triggers an operator to transfer data specified by the message and the output message transfers the result of this operation. For example, the *Copy File* operator. The header information can then be used to match the requests with the results. Therefore, arrays of messages themselves do not make much sense. However, arrays in the body are possible.

i Note

`any` is not compatible with `message`.

- The type `stream` is special in the sense that the other types (including `any` or `message`) have at least at execution time a fixed structure (elementary type and length). Streams, in general, are unstructured. Typical examples are the IO streams `stdin`, `stdout`, `stderr` of the operating system or data streams generated by sensors.

! Restriction

- `stream` is not compatible with other types including `any`
- cannot use arrays of streams
- `stream` can only be used for connections that do not cross subengine or group borders.

Conversions

In general, there are no implicit type conversions or propagations. If the types are incompatible according to the rules above, you cannot run a graph. However, there are two exceptions to the rule.

They are both concerned with the type `message`. The first one for an input port of type `message` and the second one for output ports of type `message`.

Input ports of type `message`

If the output port of the other operator is a non-stream base type, there is an automated handling to transform the output into a message. For message types, this behavior is obvious and for all other types, the engine generates a minimal message storing the output result in its body.

Output ports of type `message`

The engine automatically handles the incoming message, but the outgoing message triggers an action in the UI when you try to connect the ports.

Example:

- output port is of type `message`, incomplete, or generic type that would allow for a message to be passed.
- input port of the receiving operator is of type `string`.

In such cases, there are two ways to transform the outgoing message to a string:

- either concatenate the string from the serialized header and body of the message or
- use only the body of the message and output it as a string (if this body itself is a message then it is handled as in the first case).

The choice may well depend on the semantics of the receiving operator. Thus, there is no one recommended approach.

Related Information

[Compatible Port Types \[page 91\]](#)

[Table Messages \[page 93\]](#)

7.2.1 Compatible Port Types

You can connect two operators only if the output port of the first operator and the input port of the second operator are compatible.

The engine performs compatibility checks only when you run the graph and when the graph is loaded. If the port types are not compatible, the graph fails with a corresponding message in the trace.

Compatibility is checked in two steps as follows:

1. For the base types one of the following rules is satisfied:
 - the base types of the operators are identical or

- one of them is of type `any` and the other is one of the compatible types listed in the preceding table.
2. The semantic type of one port type is a specialization of the semantic type of another port type. The semantic type, including the empty one, is a specialization of `*`.

Note

Omitting the semantic type (or empty semantic type) yields a complete type. This type is different from `<base type>.*` which is an incomplete type.

Output Port Types	Input Port Types	Compatible	Reason
<code>any</code>	<code>any</code>	yes	Identical base type. No semantic type
<code>any</code>	<code>any.*</code>	yes	Identical base type. Here, <code>*</code> can be substituted by the empty semantic type. So, <code>any</code> is a specialization of <code>any.*</code>
<code>any</code>	<code>string</code>	yes	Compatible base types and no semantic type
<code>stream</code>	<code>any</code>	no	Incompatible base types
<code>float64.*</code>	<code>int64.*</code>	no	Incompatible base types
<code>any.*</code>	<code>string.*</code>	yes	Compatible base types and identical semantic type
<code>any.*</code>	<code>string.com.sap</code>	yes	Compatible base types. <code>com.sap</code> is a specialization of <code>*</code>
<code>any.*</code>	<code>string.com.sap.*</code>	yes	Compatible base types. <code>com.sap.*</code> is a specialization of <code>*</code>
<code>any.com.sap</code>	<code>any.com.sap</code>	yes	Identical base and semantic types
<code>string.com.*</code>	<code>string.com.sap.*</code>	yes	Identical base type. <code>com.*</code> is a specialization of <code>com.sap.*</code>
<code>any</code>	<code>any.com</code>	no	Identical base type. But <code>com</code> is not a specialization of the empty semantic type. Both sides are specializations of <code>.*</code> though.
<code>any</code>	<code>any.com.*</code>	no	Identical base type. But, <code>com.*</code> is not a specialization of the empty semantic type

Related Information

[Port Types \[page 89\]](#)

7.2.2 Table Messages

A table message is a specific kind of Modeler message used to represent tabular data. Its port type is named `message.table`.

Attributes

All table messages have an attribute named `table` whose value is an object with the following properties:

- **version (mandatory):** an integer reporting the version of the table message type, as indicated in the title of this document.
- **name:** a string holding the name of the table as reported, for example, by the database it came from. If the name is case-insensitive, it must be spelled in uppercase letters. Otherwise, the actual casing must be used.
- **columns:** an array containing objects describing each column in the table. Each object holds the following properties:
 - **name:** a string containing the name of the column. Can be the empty string if the name is unknown. If the name is case-insensitive, it must be spelled in uppercase letters. Otherwise, the actual casing must be used.
 - **class:** a string containing the type class of the column. Must be either the empty string (if the type is unknown) or one of the type classes specified in "Supported types" below. Class names are always lowercase.
 - **type:** an object containing database-specific type information. The keys in this object must be the lower-case name of the DBMS, and each value a string with the type name.
 - **size:** an integer that can be used to specify the column size limit, where applicable.
 - **precision and scale:** integers specifying the precision and the scale of a decimal column.
 - **nullable:** a boolean indicating whether the column accepts NULL values.
- **primaryKey:** a string containing the name of the column used as primary key, or an array of column names in the case of a compound key.

Body

The message body must be an array of arrays of generic elements (`[] []interface{}` in Golang). The data in the body must always be row-based.

All rows must contain the same number of values.

Supported Types

We use the term "class" to refer to a group of similar column types commonly found in DBMSs and file formats. For example, the string class may be found in the form of SQL types such as VARCHAR and ALPHANUM. The following correspondence is established between classes and the concrete data types used to hold their values:

Table 12:

Type Class	Data Type (1)	String Format
timestamp	string	RFC 3339 (2)
integer	int	integer
decimal	int (3)	Decimal number with dot as separator (4)
float	float64	Decimal number with dot as separator (4)
string	string	
binary	[]byte	Base 64 (RFC 4648, padded)
bool	bool	1, t, T, TRUE, true, True, 0, f, F, FALSE, false or False

Notes:

- (1) In the "Data type" column, int refers to the following types: int8, uint8, int16, uint16, int32, uint32, int64, uint64, int and uint.
- (2) Including a numeric time zone (or z for UTC) and an optional value of nanoseconds:
2006-01-02T15:04:05.999999999Z07:00.
 - Columns that store only part of a timestamp should leave the unused portion at the zero value (midnight for the time, 0000-01-01 for the date).
- (3) The value for a decimal in an integer representation is given by dividing the integer by 10^s , where s is the column's scale.
- (4) May optionally be suffixed with the letter e followed by an exponent. For example: "1.43e-1" for the value 0.143.

When a column's class is unknown, operators are allowed to leave its values as strings. Later on in the pipeline, an operator that expects a particular class for that column may choose to convert its values. In this case, the operator should assume that the string is in the format specified under "String format". For example, this can occur when reading from a CSV file to insert its data into an existing database table: column classes are unknown when parsing CSV, but the database operator may be capable of asking the server for the table schema and treat each column accordingly.

Encoding

Must be set to table so that operators can detect a table message even if their input is of the more general types message.* Or any.*.

Examples

Parsing CSV

Take the following CSV data whose first line is a header (the column names):

```
ID,NAME,BIRTH,INTERNAL
0,John Doe,15-04-1982,no
1,Nancy Milburn,24-11-1991,yes
```

A CSV parser might output the following table message (here represented in JSON):

```
{
  "Attributes": {
    "table": {
      "version": 1,
      "columns": [
        {"name": "ID"},
        {"name": "NAME"},
        {"name": "BIRTH"},
        {"name": "INTERNAL"}
      ]
    }
  },
  "Encoding": "table",
  "Body": [
    ["0", "John Doe", "15-04-1982", "no"],
    ["1", "Nancy Milburn", "24-11-1991", "yes"]
  ]
}
```

Since the formats for timestamp and boolean don't comply with the ones expected for a table message, an intermediate operator could be used to adapt it into:

```
{
  "Attributes": {
    "table": {
      "version": 1,
      "columns": [
        {"name": "ID"},
        {"name": "NAME"},
        {"name": "BIRTH", "class": "timestamp"},
        {"name": "INTERNAL", "class": "bool"}
      ]
    }
  },
  "Encoding": "table",
  "Body": [
    ["0", "John Doe", "1982-04-15T00:00:00Z", false],
    ["1", "Nancy Milburn", "1991-11-24T00:00:00Z", true]
  ]
}
```

Querying Database

Suppose the following table exists on an SAP HANA database:

Table 13:

ID (BIGINT)	SALARY (DECIMAL (10,2))	HIRED (DATE)
0	4367.20	'2003-01-14'
1	7823.14	'2001-07-28'

If an operator were to run a `SELECT` statement on this table and output the values in a table message, the following format would be expected:

```
{
  "Attributes": {
    "table": {
      "version": 1,
      "columns": [
        {"name": "ID", "class": "integer", "type": {"hana": "BIGINT"}},
        {"name": "SALARY", "class": "decimal", "precision": 10, "scale":
2, "type": {"hana": "DECIMAL"}},
        {"name": "HIRED", "class": "timestamp", "type": {"hana":
"DATE"}},
      ],
      "primaryKey": "ID"
    }
  },
  "Encoding": "table",
  "Body": [
    [0, 436720, "2003-01-14T00:00:00Z"],
    [1, 782314, "2001-07-28T00:00:00Z"]
  ]
}
```


8 Working with the Data Workflow Operators

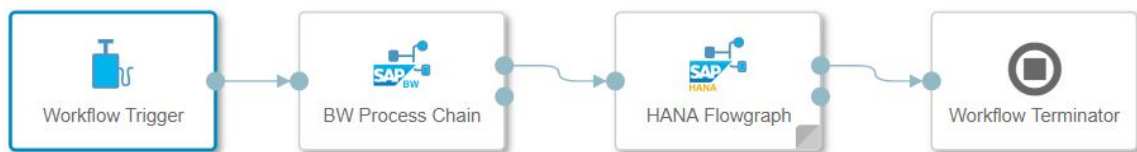
SAP Data Hub Modeler categorizes certain operators as Data Workflow operators, which when executed runs for a limited period of time and finishes with the status either as *completed* or *dead*.

The graphs modeled with the data workflow operators are referred to as Data Workflows. The operators in a data workflow communicate with signals transferred at its input and output ports. All data workflow operators have an input, an output, and an error port. Thus, the operator begins execution only when it receives such a signal at its input port. The execution of other connected operators in the data workflow begins only after the previous operator has finished its execution.

i Note

We recommend not to model graphs that have both data workflow operators and non-data workflow operators.

For example, the following graph includes the data workflow operators, Workflow Trigger, BW Process Chain, Flowgraph, and Workflow Terminator.



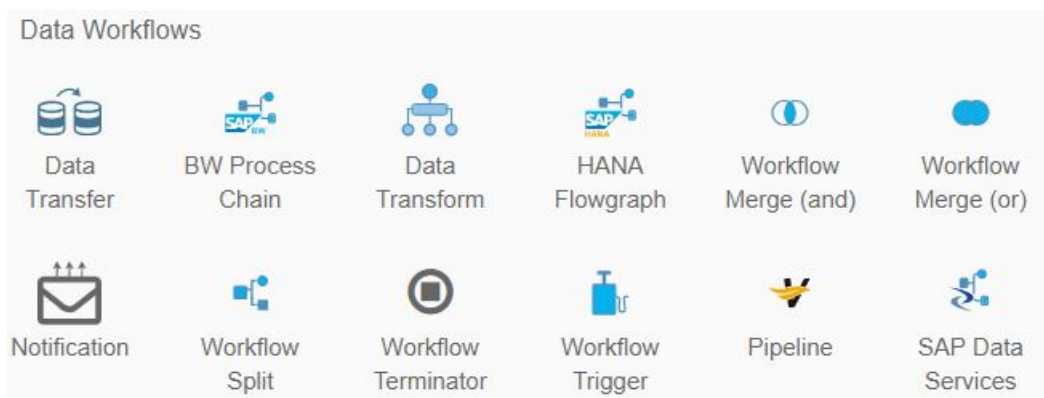
In this example, the Workflow Trigger operator sends a start execution signal to the SAP BW Process Chain operator. Only after the SAP BW Process Chain operator has completed successfully, the SAP HANA Flowgraph operator will then start to run.

i Note

If there is an outgoing signal sent to the output port and if the output port is unconnected, it results in a graph execution failure.

Data Workflow Operators

In the SAP Data Hub Modeler, you can access the data workflow operators from the navigation pane and in the [Operators](#) tab. These operators are categorized under the [Data Workflows](#) category. The following data workflow operators are supported in the Modeler.




- [Transfer Data from SAP BW to SAP Vora or Cloud Storage \[page 123\]](#)
- [Execute an SAP BW Process Chain \[page 101\]](#)
- [Transform Data \[page 104\]](#)
- [Execute an SAP HANA Flowgraph Operator \[page 102\]](#)
- [Control Flow of Execution \[page 133\]](#)
- [Control Flow of Execution \[page 133\]](#)
- [Send E-mail Notifications \[page 135\]](#)
- [Control Flow of Execution \[page 133\]](#)
- [Control Start and Shut Down of Data Workflows \[page 134\]](#)
- [Control Start and Shut Down of Data Workflows \[page 134\]](#)
- [Execute an SAP Data Hub Pipeline \[page 118\]](#)
- [Execute an SAP Data Services Job \[page 120\]](#)

Table 14: Supported Data Workflow Operators

Operator	Description
BW Process Chain	Executes an SAP BW process chain in a remote SAP BW system.
HANA Flowgraph	Executes an SAP HANA flowgraph in a remote SAP HANA system.
Workflow Merge (and)	Combines the output from two data workflow operators in a "logical AND".
Workflow Merge (or)	Combines the output from two data workflow operators in a "logical OR".
Data Transform	Provides a variety of options for data transformation.
Notification	Sends email notifications.
Workflow Split	Duplicates the incoming signal from a data workflow operator.
Workflow Trigger	Sends a start signal to trigger the execution of a data workflow.
Workflow Terminator	Terminates the current data workflow.

Operator	Description
Pipeline	Executes an SAP Data Hub graph on either a remote or the local system.
Data Transfer	Transfers data from an SAP BW system or an SAP HANA system to an SAP Vora table or to supported cloud storages.
SAP Data Services Job	Executes an SAP Data Services Job in a remote SAP Data Services system.

→ Tip

The modeler groups operators under specific categories. You can customize the list of operators that the application displays in the navigation pane. In the navigation pane, choose  (Customize Visible Categories) and select the required categories (or deselect the unwanted) that must appear (or not) in the list. To search for an operator, use the operator's name to look for it in the search bar.

Modeling Data Workflows

Model data workflows with a *Workflow Trigger* operator to send the start execution signal to the connected operator. You can use the *Workflow Terminator* operator to shut down the data workflow. This means that for a data workflow operator, you must connect it to the Workflow Trigger or Workflow Terminator operator if it is the first or the last operator that you want to be executed in a data workflow.

If you are using operators to perform actions in remote systems, then it is necessary to first create a connection to the remote system using the SAP Data Hub Connection Management application. For more information on how to create a connection, see *Create a Connection* in the *Related Information* section.

i Note

Use the data workflows operators in a graph with other data workflow operators only.

Execution Logic and Data Workflow Status

The operators within a data workflow start their execution once it receives such a signal at its input port. After executing, each data workflow operator sends a signal to the connected operator via its output port if it successfully finished its execution (or via the error port if the execution results in an error).

If the port to which the message is being sent is connected to another data workflow operator, the operator that receives the signal begins the execution. If the port to which the signal is being sent is not connected to another operator, the overall execution of the data workflow stops with the status of the execution as **dead**.

→ Tip

If you want the data workflow execution to fail when any operator execution fails, then model the workflow in such a way that the respective error ports are unconnected.

Importing Certificates for Remote Systems

For operators that leverage HTTPS as the underlying transport protocol \(\using TLS transport encryption\), the certificate of the upstream system needs to be trusted. To import a certificate into the trust chain, obtain the certificates from the target system and import them using the SAP Data Hub Connection Management application. The next execution of the graph involving the HTTPS connection will automatically pick up any certificate that is present in the trust chain. For more information on how to create a connection, see *Manage Certificates* in the *Related Information* section.

→ Remember

SAP Data Hub considers the imported certificates are for all SAP Data Hub applications and does not restrict it only for the data workflow operators.

i Note

Adding any certificate overwrites the default chain of trust. Thus, the engine may require you to add further certificates for the existing graphs to continue working.

Related Information

[Execute an SAP BW Process Chain \[page 101\]](#)

[Execute an SAP HANA Flowgraph Operator \[page 102\]](#)

[Transform Data \[page 104\]](#)

[Execute an SAP Data Hub Pipeline \[page 118\]](#)

[Execute an SAP Data Services Job \[page 120\]](#)

[Transfer Data \[page 122\]](#)

[Control Flow of Execution \[page 133\]](#)

[Control Start and Shut Down of Data Workflows \[page 134\]](#)

[Send E-mail Notifications \[page 135\]](#)

[Using SAP Data Hub System Management](#)

[Create a Connection](#)

[Manage Certificates](#)

8.1 Execute an SAP BW Process Chain

Use the BW Process Chain operator in the SAP Data Hub Modeler application to execute an SAP BW process chain in an SAP BW system.

Prerequisites



- You have created a connection to an SAP BW system using the SAP Data Hub Connection Management application.

Context

SAP BW process chain is a sequence of processes that are scheduled to wait in the background for an event. Some of these processes can trigger a separate event that can, in turn, start other processes.

In the SAP BW process chain, you can define the start condition of a process chain with the start process type. All other processes in the chain are scheduled to wait for an event. These processes are connected using events that are triggered by a predecessor process to start a subsequent process.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *BW Process Chain* operator.
 - c. In the search results, double-click the *BW Process Chain* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
5. Configure the operator.
 - a. In the graph editor, select the *BW Process Chain* operator and choose  (Open Configuration).
 - b. In the *SAP BW Connection* text field, enter a connection ID that references a connection to a remote SAP BW system.
You can also use the form-based editor that the application provides to select or enter the connection details. In the configuration pane, choose  (Open editor). If you have already created connections in the SAP Data Hub Connection Management application, in the *Connection ID* dropdown list, you can

browse and select the required connection. If you want to manually enter the connection details to the remote system, in the *Configuration Type* dropdown list, select *Manual* and enter the required connection details.

- c. In the *SAP BW ProcessChain ID* dropdown list, select the SAP BW process chain that you want to execute in the remote system.

The application populates the dropdown list based on the connection ID.

- d. (Optional) In the *Retry Interval* text field, enter the time interval in seconds for the engine to wait until the next status update.

The default value is 20 seconds.

- e. (Optional) In the *Retry Attempts* text field, enter the maximum number of attempts to query for the status update.

The default value is 1080 attempts.

i Note

The operator execution fails and a message is sent to the error output port after **<Retry_Interval> * <Retry_Attempts>** seconds. This means that a status update has been requested for the number of times set in *Retry Attempts* and the process chain has not been executed (or executed with error).

6. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.2 Execute an SAP HANA Flowgraph Operator

Use the HANA flowgraph operator in the SAP Data Hub Modeler application to execute an SAP HANA flowgraph in an SAP HANA system.

Prerequisites

- You have created a connection to an SAP HANA system using the SAP Data Hub Connection Management application.

Context

Executing a HANA flowgraph operator within a graph helps you to transform data from a remote source into SAP HANA either in batch or real-time mode. The flowgraphs are predefined in the server and represent a complete data flow. You must select the required SAP HANA flowgraph, and all the operations that the flowgraph must perform are defined in the server.

i Note

The current version of the application supports only executing XSC-based HANA flowgraphs.

Procedure


1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose + (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *HANA Flowgraph* operator.
 - c. In the search results, double-click the *HANA Flowgraph* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
5. Configure the operator.
 - a. In the graph editor, double-click the *HANA Flowgraph* operator.
 - b. In the *Connection* text field, enter the required connection ID.
You can also browse and select the required connection.
 - c. In the *SAP HANA Flowgraph* field, browse and select the required SAP HANA flowgraph.
 - d. (Optional) In the *Retry Interval* text field, enter the time interval in seconds for the engine to wait until the next status update.
The default value is 20 seconds.
 - e. (Optional) In the *Retry Attempts* text field, enter the maximum number of attempts to query for the status update.
The default value is ten attempts.

i Note

The operator execution fails and a message is sent to the error output port after $\langle \text{Retry_Interval} \rangle * \langle \text{Retry_Attempts} \rangle$ seconds. This means that a status update has been requested for the number of times set in *Retry Attempts* and the SAP HANA flowgraph has not been executed (or executed with error).


6. (Optional) Edit variables.

After you select an SAP HANA flowgraph, in the *Variable* section, the application displays the variables associated with the SAP HANA flowgraph, its data type, and default value. You can edit the default value of a variable.

- a. If you want to edit the default variable value, in the *Variables* section, choose , and provide the required variable value.

7. (Optional) Edit table variables.

In the *Table Variables* section, the application displays the tables variables associated with the SAP HANA flowgraph, its data type, and default value. You can edit the default value of a table variable or define new table variables.

- a. In the *Table Variables* section, under the *Name* column, enter (or select) the name of the table variable.
- b. In the *Value* text field, choose  and provide the required variable value.
- c. If you want to define a new table variable for the selected SAP HANA flowgraph, in the *Table Variables* section, choose + (Add New).

8. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.3 Transform Data

The Data Transform operator in the SAP Data Hub Modeler provides wide variety of options to meet your data transformation needs.


Context

There are nodes available in the operator that provides capabilities to meet your data transformation requirement. For example, you can use these nodes to create aggregations, projections, joins, unions, and more. Configure each node to meet your individual data specifications.

The nodes in the operators can consume:

- data sets
- relational operators such as filter (projection), join, and union

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *Data Transform* operator.
 - c. In the search results, double-click the *Data Transform* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
5. Configure the operator.
Operators are defined with default configuration parameters values. For example, subengine. You can provide new values.
 - a. Select the operator and choose  (Open Configuration).
 - b. Define the required configurations.
6. Configure the nodes.
The *Data Transform* operator provides different nodes that you can use to define your data transformation requirements. To add and configure the node:
 - a. In the graph editor, double-click the *Data Transform* operator.
 - b. In the *Nodes* tab, drag and drop the required node to the operator editor.

Node	Description
Data Source	Data Source nodes provide connections to the input data.
Data Target	Data Target nodes provide connections to the output data.
Projection	The Projection node represents a relational selection (filter) combined with a projection operation. It also allows calculated columns to be added to the output.
Aggregation	The Aggregation node represents a relational group-by and aggregation operation.
Join	The Join node represents a relational multiway join operation. It supports multiple input ports.
Union	The Union node represents a relational union operation. It supports multiple input ports.

Node	Description
Case	The Case node specifies multiple paths so that the rows are separated and processed in different ways. It supports multiple output ports.

- c. Double-click the node and define the required node configurations.

For more information on configuring the various Data Transform nodes, see the Related Information section.

7. Connect nodes.

If you have configured the Data Transform operator with more than one node, you can connect the nodes like how you connect the operators.

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. Add new nodes.
- c. If you want to connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

i Note

One input port only supports one link. Join and Union nodes can have multiple input ports. Case node supports multiple output ports. Each output port supports multiple output links to other nodes.

- d. If you want to create a input or new output ports (for join, union, or case), right-click the node and choose *Add Input* or *Add Output*.

8. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Configure the Data Source Node \[page 107\]](#)

[Configure the Data Target Node \[page 108\]](#)

[Configure the Projection Node \[page 110\]](#)

[Configure the Aggregation Node \[page 112\]](#)

[Configure the Join Node \[page 114\]](#)

[Configure the Union Node \[page 116\]](#)

[Configure the Case Node \[page 117\]](#)

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.3.1 Configure the Data Source Node

Data Source nodes provide connections to the input data.

Prerequisites

- You have modeled the Data Transform operator with the Data Source node.
- You have created connections in the SAP Data Hub Connection management application.
- The connection type used in the connection definition has the STRUCTURED_TRANSFORM capability.

Procedure

1. Double-click the *Data Source* node.
2. Select the source data set.
 - a. In the *Connection ID* text field, enter the required connection ID.
You can also browse and select the required connection.
 - b. In the *Source* field, browse and select the required source data set.In the *Columns* section, the application displays all columns from the selected data set.
3. (Optional) Import data set from Metadata Catalog.
You can browse the folders in the Metadata Catalog and select the required data set.
 - a. At the top-right corner of the editor, choose *Import Dataset*.
 - b. Browse and select the required data set.
 - c. Choose *OK*.
The application automatically populates the connection details based on the selected data set. For more information on Metadata Catalog, see the topic *Managing the Catalog* in the *Data Governance User Guide*.
4. Define the output columns.
Depending on the connection type of the selected connection ID, in the *Columns* section, define the data set.

Connection Type	Next Steps
HDFS, Amazon S3, ADL, Google Cloud Storage (GCS), Alibaba Object Storage Service (OSS), and WASB	<ol style="list-style-type: none">1. In the <i>Source</i> field, browse to the required file.2. In the <i>Format</i> dropdown list, select the file format. The application supports Parquet, CSV, and ORC file formats. For CSV files, you can define other CSV-specific properties such as the character set, column delimiter, and the text delimiter. The application parses the CSV files based on the values that you provide.3. For CSV files, select a value for <i>Includes Header</i>.

Connection Type	Next Steps
	This value helps the application identify whether the selected file contains a header row. If the file already has a header row, enable the <i>Includes Header</i> toggle button.
VORA	<ol style="list-style-type: none"> 1. In the <i>Schema Name</i> field, browse to the required SAP Vora table. The application automatically populates the <i>Table Name</i> field. <ol style="list-style-type: none"> a. (Optional) Choose + (Add Column) to define more columns. b. Select a data type for each column. Depending on the selected data type, you can define the length, scale, precision, or format. c. Under the <i>Primary Key</i> column, select a column that serves as the primary key. Typically the data in this column is unique. d. Under the <i>Nullable</i> column, choose whether the column value can be empty (nullable). e. (Optional) To reorder the columns, click the up and down arrow icon. Select the column that you want to move and click the up or down arrows. f. (Optional) If you want the application to automatically propose a schema based on the values in the dataset, at the top-right corner of the editor, choose <i>Fetch Metadata</i>.

5. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. Add new nodes.
- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.2 Configure the Data Target Node

Data Target nodes provide connections to the output data.

Prerequisites

- You have modeled the Data Transform operator with the Data Target node and connected the previous node to the Data Target node.
- You have created connections in the SAP Data Hub Connection management application.
- The connection type used in the connection definition has the STRUCTURED_TRANSFORM capability.

Procedure

1. Double-click the *Data Target* node.

2. Select the target data set.
 - a. In the *Connection ID* text field, enter the required connection ID.

You can also browse and select the required connection.

If you have connected the input of the Data Target node to other nodes, then in the *Columns* section, the application displays all columns from the connected node.

3. (Optional) Import target data set from Metadata Catalog.

You can browse the folders in the Metadata Catalog and select the required data set.

- a. In the editor, choose *Import Dataset*.
- b. Browse and select the required data set.
- c. Choose *OK*.

The application automatically populates the connection details based on the selected data set. For more information on Metadata Catalog, see topic *Managing the Catalog* in the *Data Governance User Guide*.

4. Define the output columns.


Select a target data set depending on the selected connection ID. After selecting the target data set, in the *Columns* section, define the data set.

Connection Type	Next Steps
HDFS, Amazon S3, ADL, Google Cloud Storage (GCS), Alibaba Object Storage Service (OSS), and WASB	<ol style="list-style-type: none"> 1. In the <i>Target</i> field, browse to the required file. <div style="border: 1px solid #ccc; background-color: #f9f9f9; padding: 10px; margin: 10px 0;"> <p>i Note</p> <p>If you want to use a new file and not any of the existing files, enter the name of the new file along with the fully qualified path.</p> </div> <ol style="list-style-type: none"> 2. In the <i>Format</i> dropdown list, select the file format type. <p>The application supports Parquet, CSV, and ORC file formats. For CSV files, you can define other CSV-specific properties such as the character set, column delimiter, and the text delimiter. The application parses the CSV files based on the values that you provide.</p> <ol style="list-style-type: none"> 3. For CSV files, select a value for <i>Includes Header</i>. <p>This value helps the application identify whether the selected file contains a header row. If the file already has a header row, enable the <i>Includes Header</i> toggle button.</p>
VORA	<ol style="list-style-type: none"> 1. In the <i>Schema Name</i> field, browse to the required SAP Vora table. <p>The application automatically populates the <i>Table Name</i> field.</p>

- a. (Optional) Choose *+* (Add Column) to define extra columns.
- b. Select a data type for each column.

Depending on the selected data type, you can define the length, scale, precision, or format.
- c. Under the *Primary Key* column, select a column that serves as the primary key.

Typically the data in this column is unique.
- d. Under the *Nullable* column, choose whether the column value can be empty (nullable).

- e. (Optional) To reorder the columns, click the up and down arrow icon. Select the column that you want to move and click the up or down arrows.
 - f. (Optional) If you want the application to automatically propose a schema based on the values in the dataset, at the top-right corner of the editor, choose *Fetch Metadata*.
5. If you want to overwrite the changes in the target data set, in the *Columns* section toolbar, choose *Overwrite*.
- You can also append the changes in the target data set without overwriting them. In the toolbar, choose *Append* instead of *Overwrite*.
6. Map input columns and output columns.
- a. In the *Columns* section, under *Mapping Column*, select an appropriate column to map the input column names to the output columns.
 - b. If you want to automap the columns based on column names, in the *Columns* pane, choose  (Auto Map by Name).
7. Connect nodes.
- If you want to configure the Data Transform operator with another node:
- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
 - b. Add new nodes.
 - c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.3 Configure the Projection Node

A Projection node represents a relational selection (filter) combined with a projection operation. It also allows calculated attributes to be added to the output.

Prerequisites

You have defined the operator with a Projection node and connected the previous node to this node.

Procedure


1. Double-click the *Projection* node.
2. Define the output columns.

If you have connected the previous node to the Projection node, you can map any columns from the input as output columns of the Projection node. You can also add, delete, and rename the columns, as needed.

- a. In the *Mapping* pane, select a *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.


i Note

You can also duplicate an output column (map the same source column more than once). Select the required *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.

- b. (Optional) If you want to automap the columns based on column names, in the *Source* section, choose  (Auto Map by Name).
- c. If you want to add new output columns, in the *Target* section, choose + (Add Column) to add new rows and define its data type.


i Note

For all new output columns, define the output column with a column expression or with a mapping to any of the existing source columns.

- d. If you want to use an expression to define an output column, click the required *Target* column. In the bottom pane, the modeler opens an expression editor. Use the expression editor to define the output column expression.
 - e. If you want to remap an output column with a different source column, right-click the mapping and choose *Remap*.
Select a new source column and choose *OK*.
 - f. If you want to edit a column name or its data type, select a *Target* column and choose  (Edit).
 - g. If you want to delete a mapping without deleting the output column, right-click the mapping and choose *Delete*.
3. (Optional) Define additional configurations for output columns.
- In the *Columns* tab toolbar, switch to the *Form* pane to define additional configurations for output columns.
- a. If you have identical records from the previous node, in the toolbar, select the *Distinct* checkbox to output unique records only.

→ Remember

The duplicate records must match exactly; similar records are included to the output. For example, if the only difference between record 1 and record 2 is that the first name is spelled "Jane" and "Jayne" respectively, then both records are output.

- b. If you want to add new output columns by duplicating an existing output column definition, select an output column and in the toolbar, choose .
 - c. Under the *Primary Key* column, select an output column that serves as the primary key. Typically the data in this column is unique.
 - d. Under the *Nullable* column, choose whether the column value can be empty (nullable).
 - e. To reorder the columns, select the column that you want to move and in the toolbar click the up or down arrows.
4. (Optional) Define filters.
- For example, if you want to move all the records that are in Canada for the year 2017, your filter might look like the following: "Filter1_input"."COUNTRY" = "Canada" AND "Filter1_input"."DATE"="2017".
- a. Select the *Filters* tab to compare the column name against a constant value.

- b. In the expression editor, enter the required expression.

→ Tip

Use the *SQL Helper* column, choose whether to select the required columns, functions, or operators to the expression editor. Double-click the columns, operators, or functions to include them in the expression editor. For more information on each function, see the "SQL Functions" topic in the *SAP HANA SQL and System Views Reference* guide.

5. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. (Optional) Add new nodes.
- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.4 Configure the Aggregation Node

An Aggregation node represents a relational group-by and aggregation operation.

Prerequisites

You have defined the operator with an Aggregation node and connected the previous node to this node.

Procedure


1. Double-click the *Aggregation* node.
2. Define the output columns.


If you have connected the previous node to the Aggregation node, you can map any columns from the input as output columns of the Aggregation node. You can add, delete, and rename the columns, as needed.

- a. In the *Mapping* pane, select a *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.

i Note


You can also duplicate an output column (map the same source column more than once). Select the required *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.

- b. (Optional) If you want to automap the columns based on column names, in the *Source* section, choose  (Auto Map by Name).

- c. If you want to edit a column name, select a *Target* column and choose  (Edit).
- d. If you want to remap an output column with a different source column, right-click the mapping and choose *Remap*.
Select a new source column and choose *OK*.

3. Define the aggregation type.

You can specify the columns that you want to have the aggregate or group-by actions taken upon.

- a. If you want to define an aggregation type, select a *Target* column and choose  (Edit).
- b. In the *Aggregation Type* dropdown list, select a value.

Aggregation Type	Description
<empty>	This type is the default aggregation type. You this aggregation type to specify a list of columns for which you want to combine output. For example, group sales orders by date to find the total sales ordered on a particular date.
Avg	Calculates the average of a given set of column values.
Count	Returns the number of values in a table column.
Max	Returns the maximum value from a list.
Min	Returns the minimum value from a list.
Sum	Calculates the sum of a given set of values.

i Note

You cannot edit the data type of output columns of an Aggregation node.

4. (Optional) Reorder output columns.

In the *Columns* tab toolbar, switch to the *Form* pane to reorder the output columns.

- a. To reorder the columns, select the column that you want to move and in the toolbar click the up or down arrows.

5. (Optional) Define filters.

For example, if you want to view the number of sales that are greater than 10000, your expression might look like the following: "Aggregation1_input"."SALES" > 10000.

- a. Select the *Filters* tab to compare the column name against a constant value.
- b. In the expression editor, enter the required expression.

→ Tip

Use the *SQL Helper* to select the required columns, functions, or operators to the expression editor. Double-click the columns, operators, or functions to include them in the expression editor. For more information on each function, see the "SQL Functions" topic in the *SAP HANA SQL and System Views Reference* guide.

6. (Optional) Define SQL HAVING clause.

If you want to retrieve records from the output of a source only when the aggregate values satisfy a defined condition, then:

- a. Select the *Having* tab.
 - b. In the expression editor, define the required HAVING condition.
7. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. Add new nodes.
- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.5 Configure the Join Node

A Join node represents a relational multiway join operation.

Prerequisites

You have configured the operator with the Join node and connected output of two or more previous nodes to the Join node.

Context

The Join node can perform multiple step joins on two or more inputs. Configure the Join node to define the join condition and the output columns of the Join node.


i Note

The Join node is not available for real-time processing.

Procedure

1. Double-click the *Join* node.
2. Create a join.

If there are more than two sources at the input of the join node, first create the join between the sources.

- a. In the *Definition* tab, select a source.
- b. Choose  (Create Join) and drag the cursor to another source on the canvas with which you want to create a join.

You can also create a join by selecting a column in the source and dragging the cursor to the required column in a different source. In this case, the application automatically creates a join condition.

3. Define the join.

The *Join Definition* pane, define the join type and the join condition.

- a. In the *Join Type* dropdown list, select a value.

Join Type	Description
Inner Join	Use when each record in the two tables has matching records.
Left Outer	Output all records in the left table, even when the join condition does not match any records in the right table.
Right Outer	Output all records in the right table, even when the join condition does not match any records in the left table.
Full Outer	Output all matching records from both tables. The records are output regardless of whether they are contained in both tables. In other words, if a record is in the left table and matches the criteria, it is output even though the same record is not in the right table.
Cross	Output all possible combinations of rows from the two tables.

- b. In the expression editor, enter a join condition.
- c. (Optional) To use the join condition that the application proposes, select *Propose Condition*.
The application analyzes the sources participating in the join and proposes a condition.



4. Define the output columns.

In the *Columns* tab, define the output columns of the join node.

- a. In the toolbar, choose the *Columns* tab.
- b. In the *Mapping* pane, select a *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.

i Note

You can also duplicate an output column (map the same source column more than once). Select the required *Source* column and hold and drag the cursor to the *Drop row here* zone in the *Target* section.

- c. (Optional) If you want to automap the columns based on column names, in the *Source* section, choose  (Auto Map by Name).
- d. If you want to edit a column name, select a *Target* column and choose  (Edit).

i Note

You cannot edit the data type of output columns of a join node.

- e. If you want to remap an output column with a different source column, right-click the mapping and choose *Remap*.
Select a new source column and choose *OK*.

5. (Optional) Reorder output columns.

In the *Columns* tab toolbar, switch to the *Form* pane to reorder the output columns.

- a. To reorder the columns, select the column that you want to move and in the toolbar click the up or down arrows.

6. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. (Optional) Add new nodes.
- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.6 Configure the Union Node

A Union node represents a relational union operation.

Prerequisites

You have configured the operator with the Union node and connected the previous nodes to Union node.

Context

The union operator forms the union from two or more inputs with the same signature. This operator can either select all values including duplicates (UNION ALL) or only distinct values (UNION).

Procedure

1. In the canvas, select the *Union* node.

2. Choose  (Open Configuration).

In the *Configuration* pane, under the *Columns* section, you can see the column information from the previous nodes.

3. If you want to merge all of the input data (including duplicate entries) into one output, enable the *Union All* toggle button.
4. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. Add new nodes.

- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.

8.3.7 Configure the Case Node

The Case node specifies multiple paths so that the rows are separated and processed in different ways.

Prerequisites

You have defined the operator with a Case node and connected the previous node to the Case node.

Context

Route input records from a single source to one or more output paths. You can simplify branch logic in data flows by consolidating case or decision making logic in one node. Paths are defined in an expression table. By default, there are two output ports defined. The one ending in *Default* contains any records that do not meet the expression definition in one of the other output ports.

Procedure

1. Double-click the *Case* node.
2. (Optional) Select *Output Row Once* to specify whether a row can be included in only one or many output targets.

For example, you might have a partial address that does not include a country name such as 455 Rue de la Marine. It is possible that this row could be output to the tables named *Canada_Customer*, *France_Customer*, and *Other_Customer*. Select this option to output the record into the first output table whose expression returns TRUE. Not selecting this option would put the record in all three tables.
3. (Optional) Choose + (Add Port) to add more Case output ports and expressions.
4. Click the name under the *Output Port Name* heading to rename the port to be more meaningful. For example, *Canada_Customer*.
5. Click the link under the *Expression* heading to open the expression editor.
6. In the expression editor, define an expression for the records that you want to include in this output.
7. Define the default port.

The default port contains any records that do not meet the expression definition in one of the other output ports.

 - a. Select the required output port and choose *Default*.
8. Connect nodes.

If you want to configure the Data Transform operator with another node:

- a. In the menu bar, use the breadcrumb navigation to navigate back to the operator configuration editor.
- b. Add new nodes.
- c. To connect the nodes, select the output port of a node and drag the cursor to an input port of another node.


8.4 Execute an SAP Data Hub Pipeline


Use the Pipeline operator in the SAP Data Hub Modeler application to execute an SAP Data Hub Pipeline in an SAP Data Hub system.

Context

You can use the Pipeline operator to execute a data pipeline in a remote SAP Data Hub system or in a local system. A data pipeline represents a concrete and complex data flow and helps transform data between elements connected in a series. When you execute a data pipeline, it can help process the raw data from multiple sources and make it available for different use cases.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *Pipeline* operator.
 - c. In the search results, double-click the *Pipeline* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
5. Configure the operator.
 - a. In the graph editor, select the *Pipeline* operator and choose  (Open Configuration).
 - b. In the *VFlow Connection* text field, enter a connection ID that references a connection to a remote SAP Data Hub system.

You can also use the form-based editor that the application provides to select or enter the connection details. In the configuration pane, choose  (Open editor). If you have already created connections in the SAP Data Hub Connection Management application, in the *Connection ID* dropdown list, you can browse and select the required connection. If you want to manually enter the connection details to the

remote system, in the *Configuration Type* dropdown list, select *Manual* and enter the required connection details.

i Note

Providing connection details for the Pipeline operator is optional and is necessary only if you want to execute a graph in a remote SAP Data Hub system. If you do not provide any value, you can only execute a pipeline from the local system (a pipeline in the SAP Data Hub Modeler that you are currently working in).

- c. In the *Graph Name* dropdown list, select the SAP Data Hub pipeline (graph) that you want to execute. The application populates the dropdown list with all graphs in the remote system (based on the connection ID) or all graphs available in the local system.
- d. (Optional) In the *Retry Interval* text field, enter the time interval in seconds for the engine to wait until the next status update.
The default value is 20 seconds.
- e. (Optional) In the *Retry Attempts* text field, enter the maximum number of attempts to query for the status update.
The default value is ten attempts.

i Note

The operator execution fails and a message is sent to the error output port after $\langle \text{Retry_Interval} \rangle * \langle \text{Retry_Attempts} \rangle$ seconds. This means that a status update has been requested for the number of times set in *Retry Attempts* and the data pipeline has not been executed (or executed with error).

- f. In the *Running Permanently* dropdown list, select a value to indicate whether the selected SAP Data Hub pipeline is a permanently running data pipeline.

Value	Description
true	If set to <i>true</i> , the operator execution checks whether the data pipeline is in a running state. If it is not running, it starts the data pipeline execution and terminates immediately (status: completed) while the data pipeline remains in the running state. But, if the task was already running with a different task version, then that instance is stopped, and the new task version is started and immediately terminated (status: completed).
false	This is the default value. If set to <i>false</i> , the operator executes the data pipeline once, and the operator execution terminates after the pipeline execution terminates.

6. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.5 Execute an SAP Data Services Job

Use the SAP Data Services Job operator in the SAP Data Hub Modeler application to execute an SAP Data Services Job in a remote system.

Prerequisites

- You have created a connection to an SAP Data Services system using the SAP Data Hub Connection Management application.

Context

Executing an SAP Data Services job helps you to integrate, transform, and improve the data quality. In SAP Data Services, the unit of execution is called jobs. Executing an SAP Data Services job in SAP Data Hub helps in:

- Data ingestion into a Hadoop cluster for further processing (natively in a Hadoop cluster).
- Moving data out of SAP Data Hub after processing.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *SAP Data Services Job* operator.
 - c. In the search results, double-click the *SAP Data Services Job* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
5. Configure the operator.

- a. In the graph editor, double-click the *SAP Data Services Job* operator.
- b. In the *Connection Id* text field, enter the required connection ID.

You can also browse and select the required connection.

- c. In the *Description* text field, provide a description for the operator.
- d. In the *Job* field, browse and select the required SAP data services job.

In the *Repository* field, the application automatically populates the repository that contains the data services job. The application also displays the global variables and the substitution parameters (if any) that are associated with the job.

- e. In the *Job Server* dropdown list, select the required job server that the modeler must use to execute the job.

i Note

You can also use a job server group to execute the job. If you have selected a job server group, specify the distribution level (job level, data flow level, or sub data flow level).


- f. In the *System Configuration* field, enter the required system configuration details that the modeler must use for running the job.

6. (Optional) Edit global variables.

If the selected job is associated with global variables, in the *Global Variables* section the application displays the global variables, its data type, and default value. You can edit the default value of a global variable associated with the job.

- a. In the *Global Variables* section, select the required global variable.

The application populates values the *Global Variables* section depending on the version of the SAP Data Service system in which you have created the selected job.

- b. In the *Value* text field, choose  and provide the required value.
- c. If you want to define a new global variable for the SAP Data Services job, in the *Global Variables* section, choose + and provide the variable and value.

i Note

Adding new global variables is not applicable for all data service jobs. It depends on the version of the SAP Data Service system in which you have created the selected job.


7. (Optional) Edit substitution parameters.

In the *Substitution Parameters* section, define the substitution parameters associated with the job, its data type, and default value. You can edit the default value of a substitution parameter or define new substitution parameters.

- a. In the *Substitution Parameter* section, choose + (Add) to add a new substitution parameter.

i Note

Substitution parameters that the application displays in the dropdown list are those parameters that are associated with the selected system configuration.

- b. Under the *Name* column, select (or enter) the name of the substitution parameter.
- c. In the *Value* text field, choose  and provide the required parameter value.

i Note

The modeler always does not auto-populate the global variables or substitution parameters. It depends on the version on the SAP Data Services system in which you have created the selected job. In such cases, manually enter the required details.

8. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.6 Transfer Data

SAP Data Hub provides capabilities that enable you to transfer data from a source system to a target system.

The *Data Transfer* operator in SAP Data Hub Modeler supports:

- SAP BW or SAP HANA as source systems from which you can transfer the data
- SAP Vora or cloud storages such as Amazon S3, Google Cloud Platform, Hadoop Distributed File System, and more as targets to which you can transfer the data.

You can use the SAP Data Hub Connection Management application to create connections to the source and target systems. After creating these connections, you can browse and select them in the *Data Transfer* operator.

Related Information

[Transfer Data from SAP BW to SAP Vora or Cloud Storage \[page 123\]](#)

[Transfer Data from SAP HANA to SAP Vora or Cloud Storage \[page 130\]](#)

8.6.1 Transfer Data from SAP BW to SAP Vora or Cloud Storage

SAP Data Hub provides capabilities that enable you to transfer data from an SAP BW system to an SAP Vora system or cloud storage.

Prerequisites

- You have created a connection to an SAP BW system using the SAP Data Hub Connection Management application.
- You have created a connection to an SAP Vora system or a cloud storage using the SAP Data Hub Connection Management application.

i Note

The supported cloud storages are S3, HDFS, ADL, GCS, WASB, and OSS.

Context


In the Modeler, configure and execute the *Data Transfer* operator in a graph to transfer data from SAP BW to SAP Vora or to cloud storage. You can execute the data transfer using three different access layers

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose + (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.
 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *Data Transfer* operator.
 - c. In the search results, double-click the *Data Transfer* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
 - d. In the graph editor, double-click the *Data Transfer* operator.
The application opens a form-based editor where you can define the source and target for data transfer
5. Define the source for data transfer.
In the *Source* tab, provide details of the source dataset for the data transfer operation.

- a. In the *Connection ID* text field, enter a connection ID that provides a connection to an SAP BW system. You can also click the browse icon to browse and select the required connection ID.
- b. In the *Source* text field, enter the required source.
You can also browse and select the required source by clicking the browse icon. In the *Browse File* dialog box, the application displays all queries or InfoProviders or DataStores from the BW system used in the connection ID definition.

Source Type	Description
Query	Queries are a combination of characteristics and key figures (InfoObjects) that allow you to analyze the data in an InfoProvider. A query corresponds to one InfoProvider, although you can define any number of queries for each InfoProvider.
InfoProvider	InfoProviders are typically made up of InfoObjects. InfoObjects are the smallest (meta-data) units in SAP BW. You can create logical views on the physical data stores in the form of InfoProviders to provide data from different data stores for a common evaluation.
DataStore	A DataStore Object (Classical DataStore Object for classic BW and advanced DataStore Object for BW/4 HANA) is a special InfoProvider that enables delta handling. It is used to keep cleansed and consolidated transactions or master data at a rather fine granularity level. An (advanced) DataStore Object contains figures and characteristics as fields. As DataStore Objects are also InfoProviders they appear as well under the InfoProvider folder. But to use the delta capabilities in the transfer task you have to choose the DataStore Objects under the DataStore Folder.

- c. Select the required source that you want to transfer to the target.
6. (Optional) Import source dataset from the Metadata Catalog.
You can browse the folders in the Metadata Catalog and select the required data set (Query or InfoProvider or DataStore).
 - a. In the editor, choose *Import Dataset*.
 - b. Browse and select the required data set.
 - c. Choose *OK*.
The application automatically populates the connection details based on the selected data set. For more information on Metadata Catalog, see *Manage Metadata* in the *Data Governance User Guide for SAP Data Hub*.
 7. (
 8. Provide values to parameters.
If you have selected a query as the source dataset, and if the query is defined with parameters, then it is necessary to provide values to those parameters. The application automatically populates the default values, if any, that are already defined for the parameters.
 - a. In the *Variables* text field, click the edit icon to provide values to the parameters.
 - b. In the *Provide Parameter Values* dialog box, select the required parameter, operator type, and provide values.
 - c. If you want to view only the mandatory parameters, choose  and select *Show Mandatory Only*.
 - d. Choose *OK*.
 9. (Optional) Specify the transfer mode.

SAP Data Hub supports three types of transfer modes. The preferred way is to utilize SAP HANA.

- a. If the engine uses the INA protocol to retrieve data from the source, in the *Timeout* text field, enter a value in milliseconds.

The modeler waits for the time that you specify before executing a timeout on the data retrieval. This means that after the timeout period, the graph execution fails. The default timeout value is 60 seconds.

- b. If an external HANA view exists, the Modeler displays the name of the SAP HANA view that it is using to retrieve data from the source.

To use a HANA view, you can do so by marking a specific query or InfoProvider in the query designer to generate an underlying calculation view. SAP Data Hub can query this view to transfer the data to other target systems.

→ Remember

The engine uses the non-optimized BW INA provider to retrieve data from the source only if no external SAP HANA view exists.

- c. If you have selected a DataStore, then the application by default uses the BW ODP as the data access configuration. In this case, you cannot modify the access configuration.

10. (Optional) Using DataStore as the source.

If you are using BW DataStore for the data transfer operation, then provide the following additional values.

- a. Select the extraction mode.

You can select the extraction mode as *Full* or *Delta*. Select *Full* to extract all data at once. Select *Delta* extraction mode to extract only what has changed with each run of the graph.

i Note

The full data will be extracted when the graph executes for the first time and thereafter only delta changes for all subsequent executions.

- b. Provide a subscription ID.

Mention the desired subscription ID to look up your subscriptions later. The subscription ID must be unique for the same InfoProvider in the same BW system for all clients accessing it. In the case of full extraction mode, the subscription ID is generated automatically. The subscription ID serves as the session for the delta extraction and stores the delta pointer. It ensures just the changed data is transferred.

i Note

In case of a failure of the writing to VORA, the data has been read from the source system (and the delta pointer there has been moved) but was not yet written to VORA. This means that the next run will already fetch the new delta, but you are missing the current delta (for which the writing failed).

In such cases, we recommend to reset the delta handling by choosing a new (so far unused) subscription ID and to truncate the VORA table. Then when running the graph a new full load is triggered followed then by deltas.

11. Select columns.

Select the required measures and dimensions from the source dataset that you want to project to the target.

- a. Choose the *Target* tab.

In the *Column Mapping* section, the application displays all measures and dimensions from the selected source.

- b. Select the columns that you want to project to the target and drag the cursor to the *Target* pane.

12. Filter dimensions.

You can apply filter conditions on dimensions in the source dataset and project only the filtered values.

- a. At the top of the Operator editor, choose the *Source* tab.
- b. In the *Filters* text field, click the edit icon.
- c. In the *Provide Filter Values* dialog box, select the dimension and define the filter condition.
- d. Choose *OK*.

13. (Optional) Define partitions.

For HANA views, the Modeler provides capabilities to define a maximum of two partition conditions on columns in the source dataset to optimize the data transfer operation. It supports the partition types List and Range to define the partition conditions.

- a. In the *Partition Conditions* section, choose *Add Condition*.
- b. Select the required partition column and its data type.
- c. In the *Type* dropdown list, select the required partition type.
- d. In the *Partition Values* text field, define one or more partition values.

For range partition type, define only the low boundary value.

i Note

If data is retrieved using the BW INA provider as the transfer mode, then the partitions are ignored.

14. Define the target (SAP Vora).

If you want to use an SAP Vora table as the target dataset for data transfer, in the *Target* tab, provide details of the required SAP Vora table.

- a. In the *Connection ID* text field, enter a connection ID that provides a connection to an SAP Vora system.

You can also click the browse icon to browse and select the required connection ID.

- b. In the *HANA Wire Port* text field, enter the SAP HANA Wire port of the SAP Vora Transaction Coordinator, which is determined as 3<XX>15, where <XX> is the instance number of the SAP Vora cluster as configured in the SAP Vora Manager.

i Note

As the Data Transfer is done via the HANA Wire protocol, it is necessary to connect to the external SAP Vora Transaction Coordinator, which can speak this protocol. The cluster-internal default hostname for this is vora-tx-coordinator-ext.

- c. In the *Table* text field, enter the SAP Vora table name to which you want to transfer the data.
- d. In the *Schema* text field, enter the schema name of the SAP Vora table selected in the previous step. Optionally, in the *Table* text field, you can click the browse icon to browse and select the required SAP Vora schema and table.

i Note

If you manually entered the table name, and if the table does not exist in the selected SAP Vora system, the application creates them in the SAP Vora system. The columns and its data types in the SAP Vora table depend on the columns you have projected from the source dataset.

15. Define the target (cloud storage).

If you want to use any of the supported cloud storage as the target dataset for data transfer, in the [Target](#) pane, provide details of the required cloud storage.

- a. In the [Connection ID](#) text field, enter a connection ID that provides a connection to the required cloud storage.

You can also click the browse icon to browse and select the required connection ID.

- b. In the [Target](#) text field, enter the path of the file to which you want to transfer the data.

You can also click the browse icon to browse and select the required file location. If the selected connection has a root path specified in the connection definition, then the content of this field is relative to this path.

i Note

The application supports CSV, ORC, or Parquet file formats.

- c. (Optional) If you have manually entered the file path, at the top right corner of the editor, choose [Fetch Metadata](#).

The [Fetch Metadata](#) functionality helps to fetch the metadata (schema) from the selected source or target and populates the column details accordingly in the UI.

16. Map source and target columns.

In the [Target](#) tab, under the [Column Mapping](#) section, use the mapping editor to map the columns from the source dataset that you want to project to the target.

- a. Map a source column with a target column by selecting the source column and dragging the cursor to the target column.

i Note

When you execute the data transfer, only the mapped columns are projected to the target dataset.

17. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

Related Information

[Transfer Modes \[page 128\]](#)

[Data Type Mapping and Miscellaneous \[page 129\]](#)

[Working with the Data Workflow Operators \[page 97\]](#)

[Create a Connection](#)

8.6.1.1 Transfer Modes

The data transfer in SAP Data Hub supports three different modes.

You can retrieve data using any of the following transfer modes:

- retrieve data via the BW Online Analytical Processor (OLAP) using the INA protocol as the access method
- retrieve data via generated SAP HANA Views
- retrieve data via BW ODP for full or delta data extraction

Using the Online Analytical Processor

The default method uses access via the OLAP processor. This method ensures that the data you write matches the data the user knows from the common BW UIs (e.g. Transaction RSRT2). It supports a wide variety of BW functionality.


Due to the "Online" nature of this access method, it is not recommended to use this method for large-scale data transfer. This mode transfers the complete data in a single result set.

i Note

There is a maximum number of cells which can be exported via BW INA.

Using a generated HANA View

In SAP BW on HANA and BW/4HANA, it is possible to generate designated calculation views on the underlying SAP HANA Database for each Query or InfoProvider. SAP Data Hub leverages these views in case the following conditions are met:

- You have created a connection to an SAP BW system using the SAP Data Hub Connection.
- The BW on HANA approach is available with BW version 4.2.0 onwards.
- You specified a working connection to the SAP HANA database and referenced it in the SAP BW connection that you are using.
- For SSL access you need to upload two certificates in case the two connections can't be covered by the same root certificate.
- The corresponding Query or InfoProvider has a generated calculation View.
- The projection of Dimensions and Measures you choose does not contain a restricted attribute (for example, as specified in [2145502](#) .
- No further errors occur (for example, SAP HANA authorization errors).

If a data is retrieved with generated SAP HANA views, then you can specify partitions of data, which will be transferred separately, enabling large result sets.

i Note

The engine processes the partition result sets one after the other and not in parallel.

Using BW ODP

You can use the BW ODP mode only with DataStores. For all other cases the preferred way is to utilize SAP HANA. This is done by marking a specific query or infoprovider or datastore in the query designer to generate an underlying calculation view. This view will then be queried by SAP Data Hub to transfer the data to other target systems.

i Note

You must manually change the setting in the data transfer operator to activate querying via SAP HANA. If SAP HANA is not available or the necessary calculation view has not been generated or the INA option has been explicitly selected, then the non-optimised BW INA Provider will be used for data extraction. In this case the user is also asked to provide an appropriate timeout for the BW INA call, depending on the amount of data expected to be extracted.

With the BW INA or ODP modes the partitions are ignored. In the current version, SAP Data Hub supports only Vora tables as the target for data transfer using the BW OPD mode.

8.6.1.2 Data Type Mapping and Miscellaneous

When using the OLAP-based access, the SAP Vora Table is created based on the data types used for displaying the content. This behavior creates a certain inexactness (especially for time-related values which are stored as VARCHAR).

When using the view-based access, the SAP Vora Table is created based on the data types specified in the SAP HANA View. This behavior enables precise mapping of the data types. However, it is not possible to rename columns or change the order of columns in the target.

The table that follows lists the data type mapping between SAP BW and SAP Data Hub.

SAP BW Type	SAP Data Hub Type
STRING, NUMC, LINE_STRING, MULTI_LINE_STRING, LOWER_CASE_STRING, UPPER_CASE_STRING, URI	STRING
DOUBLE	DOUBLE
PERCENT, QUANTITY, AMOUNT, PRICE, DECOMAL_FLOAT	DECIMAL
INTEGER	INTEGER
BOOLEAN	BOOLEAN
CALENDAR_DAY, DATE	DATE
DATE_TIME	DATETIME
TIME, TIMESPAN	TIME
LONG	INTEGER

By default, the data transfer operation generates two columns per dimension. The internal key is used for the data transfer. The text is written into a separate column (as can be seen in the UI). It is currently not possible to change the type of text or key.

i Note

Filter keys, if manually provided, must be in the internal key format. This is done automatically if the value help is used. Excluding filters are currently not supported.

For more information on SAP BW INA, see [2415249](#).

Due to the table format in the target, the result will be flattened. Meaning all structures and hierarchies are disabled.

i Note

The data transfer operation uses the default logon client when authenticating against the BW system. The description and other language-dependent attributes of a query or infoprovider are taken with the standard logon language of the BW user used in the connection.

8.6.2 Transfer Data from SAP HANA to SAP Vora or Cloud Storage

SAP Data Hub provides capabilities that enable you to transfer data from an SAP HANA system to an SAP Vora system or to cloud storage.

Prerequisites

- You have created a connection to an SAP HANA system using the SAP Data Hub Connection Management application.
- You have created a connection to an SAP Vora system or to the required cloud storage using the SAP Data Hub Connection Management application.

- **i Note**

The supported cloud storages are S3, HDFS, ADL, GCS, WASB, and OSS.

Context

In the Modeler, configure and execute the *Data Transfer* operator in a graph to transfer data from SAP HANA to SAP Vora or to cloud storage.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).

The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.

A graph can contain a single operator or a network of operators based on the business requirement.

 - a. In the navigation pane, choose the *Operators* tab.
 - b. In the search bar, search for the *Data Transfer* operator.
 - c. In the search results, double-click the *Data Transfer* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.
 - d. In the graph editor, double-click the *Data Transfer* operator.

The application opens a form-based editor where you can define the source and target for data transfer.
5. Define the source for data transfer.

In the *Source* tab, provide details of the source dataset for the data transfer operation.

 - a. In the *Connection ID* field, enter a connection ID that provides a connection to an SAP HANA system.


You can also click the browse icon to browse and select the required connection ID.
 - b. In the *Table Name* field, click the browse icon to browse and select the required SAP HANA object.
6. (Optional) Import source dataset from the Metadata Catalog.

You can browse the folders in the Metadata Catalog and select the required data set.

 - a. In the editor, choose *Import Dataset*.
 - b. Browse and select the required data set.
 - c. Choose *OK*.

The application automatically populates the connection details based on the selected data set. For more information on Metadata Catalog, see *Manage Metadata* in the *Data Governance User Guide for SAP Data Hub*.
7. Provide values to parameters.

If the selected SAP HANA objects are defined with parameters, then it is necessary to provide values to those parameters. The application automatically populates the default values, if any, that are already defined for the parameters.

 - a. In the *Variables* text field, click the edit icon to provide values to the variables.
 - b. In the *Provide Parameter Values* dialog box, select the variable, operator type, and provide the required values.
 - c. If you want to view only the mandatory parameters, choose  and select *Show Mandatory Only*.
 - d. Choose *OK*.
8. Select columns.

Select the required columns from the source dataset that you want to project to the target.

 - a. Choose the *Target* tab.

In the *Column Mapping* section, the application displays all columns from the selected source.
 - b. Select the columns that you want to project to the target and drag the cursor to the *Target* pane.

9. Filter columns.

You can filter conditions on columns in the source dataset and project only filtered values.

- a. At the top of the Operator editor, choose the *Source* tab.
- b. In the *Filters* text field, click the edit icon.
- c. In the *Provide Filter Values* dialog box, select the column and define the filter condition.
- d. Choose *OK*.

10. Define the target (SAP Vora).

If you want to use an SAP Vora table as the target dataset for data transfer, in the *Target* tab, provide details of the required SAP Vora table.

- a. In the *Connection ID* text field, enter a connection ID that provides a connection to an SAP Vora system.

You can also click the browse icon to browse and select the required connection ID.

- b. In the *HANA Wire Port* text field, enter the SAP HANA Wire port of the SAP Vora Transaction Coordinator, which is determined as 3<XX>15, where <XX> is the instance number of the SAP Vora cluster as configured in the SAP Vora Manager.

i Note

As the Data Transfer is done via the HANA Wire protocol, it is necessary to connect to the external SAP Vora Transaction Coordinator, which can speak this protocol. The cluster-internal default hostname for this is vora-tx-coordinator-ext.

- c. In the *Table* text field, enter the SAP Vora table name to which you want to transfer the data.
- d. In the *Schema* text field, enter the schema name of the SAP Vora table selected in the previous step. Optionally, in the *Table* text field, you can click the browse icon to browse and select the required SAP Vora schema and table.

i Note

If you manually entered the table name, and if the table does not exist in the selected SAP Vora system, the application creates them in the SAP Vora system. The columns and its data types in the SAP Vora table depend on the columns you have projected from the source dataset.

11. Define the target (cloud storages)

If you want to use any of the supported cloud storages as the target dataset for data transfer, in the *Target* pane, provide details of the required cloud storage.

- a. In the *Connection ID* text field, enter a connection ID that provides a connection to the required cloud storage.

You can also click the browse icon to browse and select the required connection ID.

- b. In the *Target* text field, enter the path of the file to which you want to transfer the data.

You can also click the browse icon to browse and select the required file location. If the selected connection has a root path specified in the connection definition, then the content of this field is relative to this path.

12. Map source and target columns.

In the *Target* tab, under the *Column Mapping* section, use the mapping editor to map the columns from the source dataset that you want to project to the target.

- a. Map a source column with a target column by selecting the source column and dragging the cursor to the target column.

i Note

When you execute the data transfer, only the mapped columns are projected to the target dataset.

13. Save and execute the graph.

You can control the start and stop of the graph execution using the Workflow Trigger and Workflow Terminator operators respectively.

→ Tip

You can also schedule the graph execution. For more information, see [Schedule Graph Executions \[page 18\]](#).

8.7 Control Flow of Execution

SAP Data Hub provides the data workflow operators Workflow Merge (or), Workflow Merge (and), and Workflow Split to control the flow of execution in a data workflow.

A typical use case for these data workflow operators is parallel executions within a data workflow.

i Note

You can connect the input ports of the operators to other data workflow operators. But, if the operator receives a message at any of its input ports, then leaving the output ports unconnected results in a data workflow execution to fail or stop.

Operator	Description
Workflow Merge (or)	<p>A message is sent to the output port once the operator receives a message at any of its input ports. All further inputs are ignored.</p> <p>The operator has two input ports and one output port.</p>
Workflow Merge (and)	<p>This operator sends a message to the output port once it receives a message on both the input ports. The process shuts down after sending the message.</p> <p>The Workflow Merge (and) operator is intended to control the flow of execution when there are parallel executions within the data workflow.</p> <p>The operator has two input ports and one output port. Leaving the output port unconnected results in a data workflow execution failure once the operator receives an input message.</p>

Operator	Description
Workflow Split	<p>This operator helps duplicate an input message into two output messages.</p> <p>The Workflow Split operator is intended to control the flow of execution when there are parallel executions within the graph.</p> <p>The operator has one input port and two output ports. The Operator sends the message at the input port to both the output ports.</p>

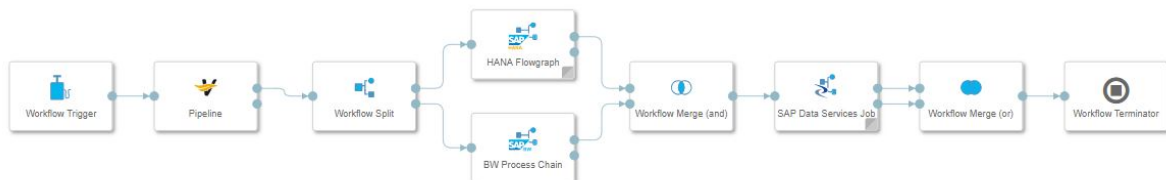
These operators are listed in the SAP Data Hub Modeler under the Data Workflow category.

i Note

Use the operators Workflow Split, Workflow Merge (and), and Workflow Merge (or) operators in a graph with other data workflow operators only.

Example

Consider this data workflow example that uses the Workflow Split, Workflow Merge (and), and Workflow Merge (or) operators.



In this example, after the engine successfully executes the Pipeline operator, the output message is duplicated to both the HANA Flowgraph operator and the BW Process Chain operator. This means that both of the latter operators run in parallel. The Workflow Merge (and) operator ensures that a message is sent to the SAP Data Services operator only if both the operators have finished successfully. The Workflow Merge (or) operator takes the first incoming message and forwards it to the Workflow Terminator operator that shuts down the data workflow execution.

8.8 Control Start and Shut Down of Data Workflows

SAP Data Hub provides the operators Workflow Trigger and Workflow Terminator to control the start of a data workflow execution and to shut down the execution respectively.

Use these operators with other data workflow operators only.

Operator	Description
Workflow Trigger	The Workflow Trigger operator sends a start message, which you can use to start a data workflow. This operator has one output port. Once started, the operator sends a message on its output port. If the output port is connected, the next data workflow operator starts its execution. But, if the output port is unconnected, the data workflow execution fails.
Workflow Terminator	The Workflow Terminator operator helps shut down the data workflow execution. It shuts down the data workflow. The data workflow terminates with the state, <i>Completed</i> .

8.9 Send E-mail Notifications

Use the Notification operator in the SAP Data Hub Modeler to send e-mail notifications to users at certain points during the data workflow execution.

Prerequisites

You have created a connection to an SMTP server using the SAP Data Hub Connection Management application.

Context


As an example, a typical use case for using the notification operator is to send an e-mail when an operator writes onto its `error` or `success` port. The message body contains technical information that the notification operators receive at its input port from the previous operator in the data workflow.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, select the *Graphs* tab.
3. In the navigation pane toolbar, choose **+** (Create Graph).
The application opens an empty graph editor in the same window, where you can define your graph.
4. Select the operator.
A graph can contain a single operator or a network of operators based on the business requirement.

- a. In the navigation pane, choose the *Operators* tab.
- b. In the search bar, search for the *Notification* operator.
- c. In the search results, double-click the *Notification* operator (or drag and drop it to the graph editor) to add it as a process in the graph execution.

5. Configure the operator.

- a. In the graph editor, select the *Notification* operator and choose  (Open Configuration).
- b. In the *Connection* text field, enter the required connection ID.
You can also browse and select the required connection.
- c. If you want to delete the attachments after sending the e-mail, in the *Delete attachments after sending* dropdown list, select *true*.
- d. In the *Default From Value* text field, enter the value of `email.from` that the modeler must use when no such value is received through the incoming message.
- e. In the *Default To Value* text field, enter the value of `email.to` that the modeler must use when no such value is received through the incoming message.
- f. In the *Default Subject Value* text field, enter the value of `email.subject` that the modeler must use when no such value is received through the incoming message.

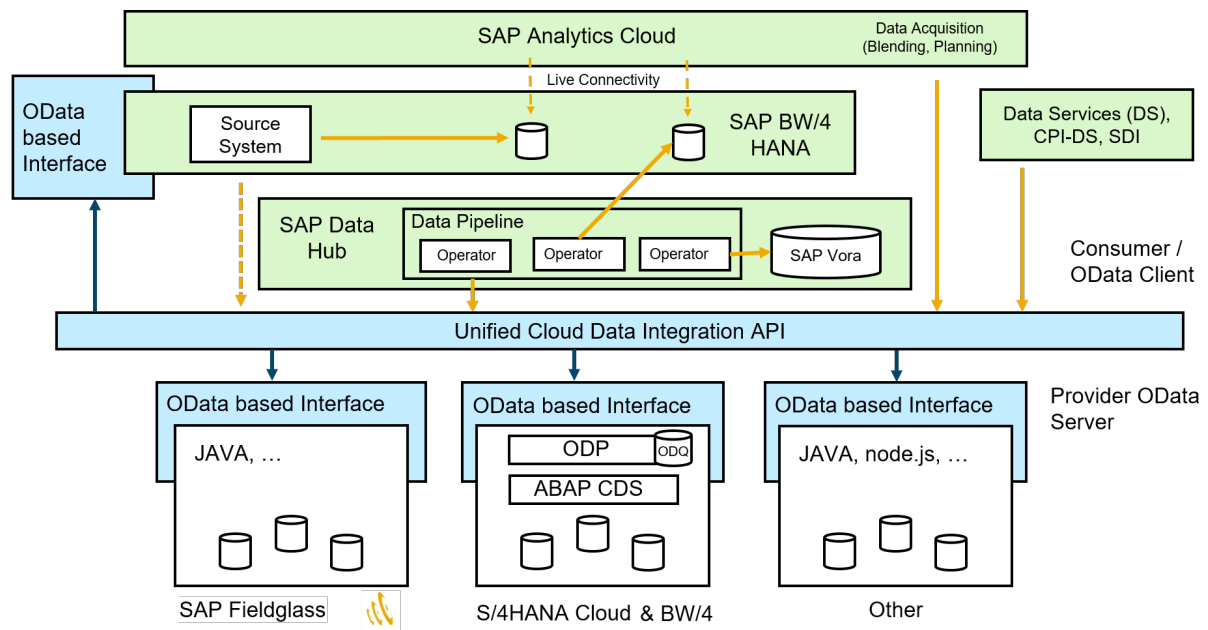
i Note

The following characters are not supported in message header names: `<`, `>`, `$`, and `{ }`.

9 Integrating SAP Cloud Applications with SAP Data Hub

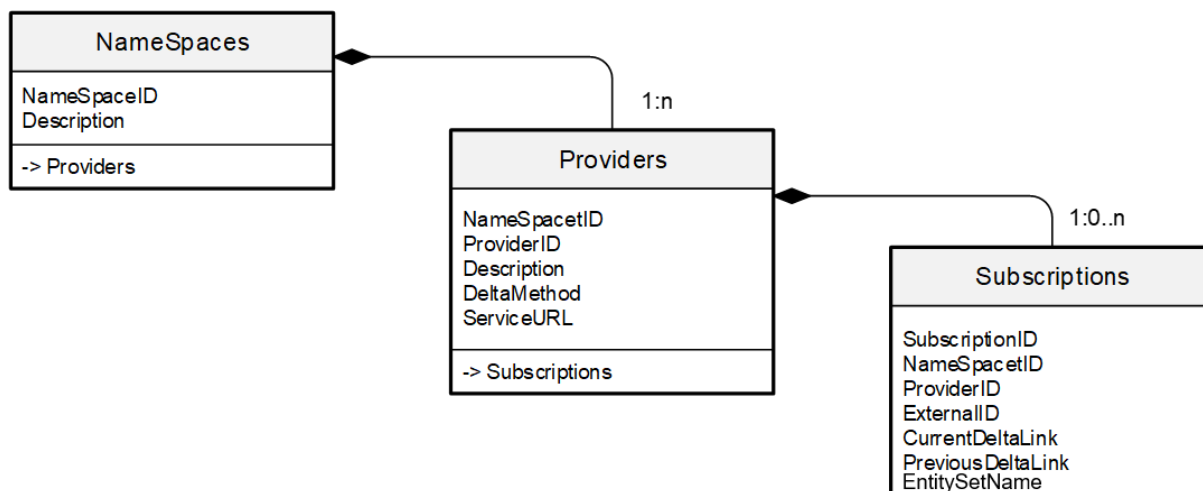
SAP Data Hub enables you to integrate SAP cloud applications with SAP Data Hub for holistic data management. The integration is supported using the Cloud Data Integration API, which is based on the OData V4 specifications.

SAP Cloud Application Data Integration – Providers and Consumers



Cloud Data Integration API

SAP Cloud applications implement the Cloud Data Integration API, which consists of two types of OData services. The first type is the Administrative service that exists once per system. The structure of the administrative service is as below.



The administrative service provides a catalog of providers that are organized along namespaces. Each provider represents a business object and consists of one more entity sets that semantically belong together.

i Note

Every implementation must produce the same \$metadata.

The second type of service exists once per data provider, and the structure of these providers depend on the entity sets of the provider. This service provides access to the metadata and the data of the entity sets.

In the current version, you can integrate the following SAP Cloud applications with SAP Data Hub.

- SAP Fieldglass
- SAP Sales Cloud
- SAP Service Cloud

Creating a Connection

In the SAP Data Hub Connection Management application, you can use the connection type, `CLOUD_DATA_INTEGRATION` to create connections to SAP Cloud applications. In the connection definition, configure the service endpoints. For more information see, [CLOUD_DATA_INTEGRATION](#)

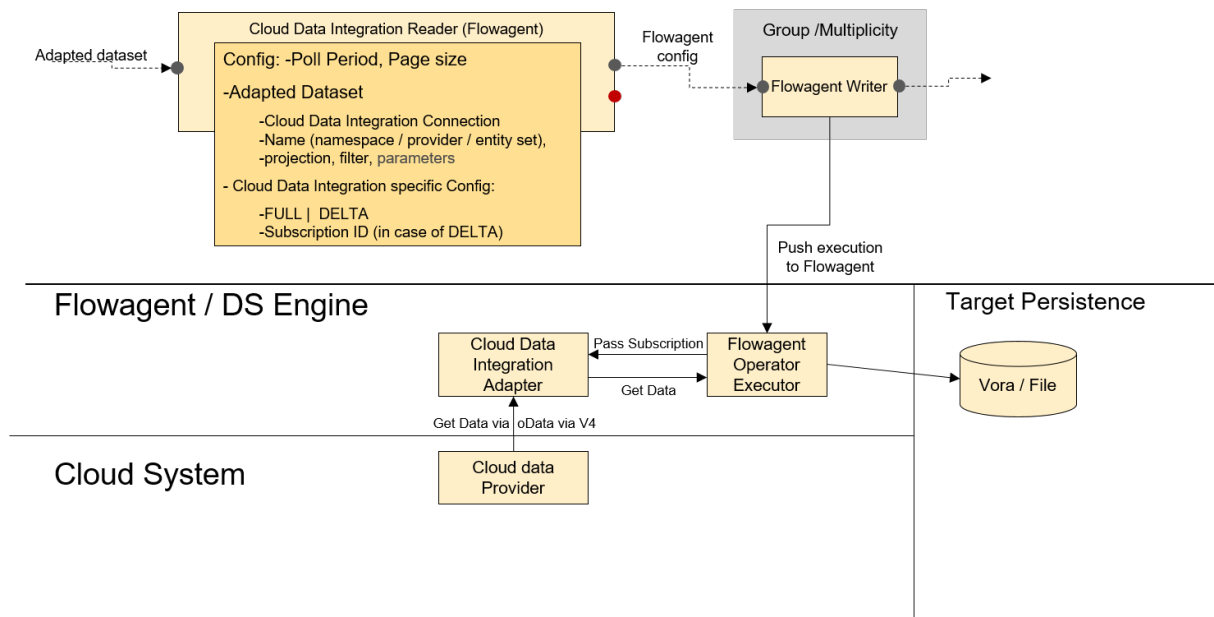
Capabilities

Creating a connection to an SAP Cloud application enables you to use entity sets of the service providers as Datasets in the Metadata explorer. Additionally, the connection also maps the namespaces and providers to folders in the Metadata explorer. As a result, you can view the metadata of entity sets and preview them in the Metadata Explorer application.

You can use the Cloud Data Integration Operator in the SAP Data Hub Modeler for data ingestion. For more information see, [Cloud Data Integration Consumer \(Beta\)](#)

The following graphical representation provides you a snapshot of streaming replications using the flowagent.

Streaming Pipeline



10 Service-Specific Information

Related Information

[Alibaba Cloud Object Storage Service \(Alibaba Cloud OSS\) \[page 140\]](#)

[Amazon S3 \[page 144\]](#)

[Azure Data Lake \(ADL\) \[page 147\]](#)

[Google Cloud Storage \(GCS\) \[page 149\]](#)

[Hadoop File System \(HDFS\) \[page 152\]](#)

[Local File System \(File\) \[page 155\]](#)

[Windows Azure Blob Storage \(WASB\) \[page 155\]](#)

[WebHDFS \[page 158\]](#)

10.1 Alibaba Cloud Object Storage Service (Alibaba Cloud OSS)

Alibaba Cloud Object Storage Service (Alibaba Cloud OSS). Additional information, including the documentation, can be found at the official Alibaba Cloud OSS's official website.

Many of the Storage operators offer support for Alibaba Cloud OSS. This documentation regards the common characteristics that this service has across operators.

This document may refer to an object as a "file", and to an object's prefix as a "directory", if it fits the context of the operator.

Connection

In order to use any operator that connects to Alibaba Cloud OSS, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- **Endpoint [Mandatory]**
 - Allows using an endpoint to access the Alibaba Cloud OSS service.
 - ID: endpoint
 - Type: string
 - Default: "oss-cn-hangzhou.aliyuncs.com"

- **Protocol [Mandatory]**
Sets which protocol to be used. The set value overwrites the protocol prefix in the Endpoint configuration, if any given.
 - ID: protocol
 - Type: string
 - Default: "HTTPS"
 - Possible values:
 - "HTTP"
 - "HTTPS"
- **Region**
The Alibaba cloud region the configured bucket (found in Root Path) belongs to.
 - ID: region
 - Type: string
 - Default: "oss-cn-hangzhou"
- **Access Key [Mandatory]**
The Access Key ID used to authenticate to the service. It pairs with the Secret Key in order to authenticate.
 - ID: accessKey
 - Type: string
 - Default: "OSSAccessKeyID"
- **Secret Key [Mandatory]**
The Secret Access Key used to authenticate to the service. It pairs with the Access Key in order to authenticate.
 - ID: secretKey
 - Type: string
 - Default: ""
- **Root Path**
The bucket and an optional root path name for browsing. Starts with a slash and the bucket name (e.g. /MyBucket/My Folder), followed by another slash and the optional root path. Dataset names for this connection don't contain segments of the `rootPath`; instead, their first segment is a subdirectory of the root path.
 - ID: rootPath
 - Type: string
 - Default: "/MyBucket/MyFolder"

Permissions

Permissions in Alibaba Cloud are required to operate over OSS objects. Refer to [Alibaba Cloud's documentation on Access and Control](#) .

OSS provides an Access Control List (ACL) for bucket-level access control. Currently, [three ACLs are available for a bucket](#) :

- public-read-write
- public-read

- private

If you do not set an ACL for a bucket when you create it, its ACL is set to private automatically.

If the ACL rule of the bucket is set to private, only authorized users can access and operate on objects in the bucket.

To authorize other users to access your Alibaba Cloud OSS resources, refer to [Alibaba Cloud OSS Configure Bucket Policy documentation](#) .

Read File Permissions

To read a single object ("file"), you need the permissions:

- `oss:GetObject` for the given object
See [Alibaba Cloud OSS Object Operations documentation](#) for more information.

To read multiple objects in a prefix ("directory"), you need the permission:

- `oss:GetBucket` for the bucket where the prefix is to be listed. Note that the permission may be narrowed to a directory inside the bucket, and the prefix is subject to this restriction.
See [Alibaba Cloud OSS Object Operations documentation](#) for more information.

Write File Permissions

To write an object ("file"), you need the permission:

- `oss:PutObject` for the bucket to receive the object.

If using mode "Append", you also need:

- `oss:GetObject` for the given object.

Remove File Permissions

To remove an object ("file"), you need the permission:

- `oss:DeleteObject` for the given object.

Move File Permissions

As moving consists of copying and removing in Alibaba Cloud OSS, you will need the permissions documented in Remove File Permissions and Copy File Permissions.

Copy File Permissions

To copy an object ("file"), you need the permissions:

- `oss:GetObject` for the source object.
- `oss:PutObject` for the bucket to receive the copied object.
See [Alibaba Cloud OSS Multipart Upload Operations documentation](#) for more information.

If copying by prefix (i.e. a "directory"), the operation is bound to the same permissions documented in Read File Permissions.

Restrictions

Any OSS specific restriction in the operators is documented here. Some may apply broadly to every operator:

- Directories: in order for a path to be interpreted as a directory, it should end with /. For example: /tmp/ is a directory, while /tmp is a file named tmp.
- Working directory: since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (/) as working directory.

Write File Restrictions

If using "Append" mode, as the Alibaba Cloud OSS API does not support it, the whole file is retrieved from the service in order to append the data and write back to OSS; thus, compromising the operation's efficiency.

Move File Restrictions

As the Alibaba Cloud OSS API does not support the move operation, the operation consists of a copy followed by removing the source file. Thus, in cases of failure, the file may be copied and not removed.

Further restrictions are documented in Copy File Restrictions.

Copy File Restrictions

Taking that the operation has a "source" and a "destination" path:

- If the destination is a file, source must also be a file.
- If the destination is a directory, it must be empty.

For instance, in the given file structure:

```
.
|
+-- a
|   +-- file1.txt
|   +-- file2.txt
+-- b
    +-- f1.txt
    +-- f2.txt
```

- Copying source a/file1.txt to destination newfile.txt would succeed, since the destination does not exist.
- Copying source a/file1.txt to destination b/f1.txt would succeed and overwrite b/f1.txt, since the destination is an existing file.
- Copying source a/file1.txt to destination: b/ would fail, since b/ already exists and is not empty.
- Copying source a/ to destination b/ would fail, since b/ already exists and is not empty.
- Copying source a/ to destination b/dir/ would succeed, since b/dir/ does not exist.

Related Information

[Alibaba Cloud's Official Website](#) 

10.2 Amazon S3

AWS S3 is an Object Store service, further documented in the owner's page. Other services may also support the S3 API, such as Rook, Minio and Swift, which have already been tested. Any other service supporting the S3 API is not guaranteed to be compatible.

This aims to document only the operators' relation with the S3 API.

This document may refer to an object as a "file", and to an object's prefix as a "directory", if it fits the context of the operator.

Connection

In order to use any operator that connects to S3, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- Custom endpoint
Allows using a custom endpoint to access the S3 service. If not set, the default AWS endpoint is used.
 - ID: `endpoint`
 - Type: `string`
 - Default: `""`
- Protocol [Mandatory]
Sets which protocol to be used. The set value overwrites the protocol prefix in the Custom endpoint configuration, if any given.
 - ID: `protocol`
 - Type: `string`
 - Default: `"HTTP"`
 - Possible values:
 - `"HTTP"`
 - `"HTTPS"`
- Region [Mandatory]
The AWS region the configured bucket (found in Root Path) belongs to.
 - ID: `region`
 - Type: `string`
 - Default: `"eu-central-1"`
- Access Key [Mandatory]
The Access Key ID used to authenticate to the service. It pairs with the Secret Key in order to authenticate.
 - ID: `accessKey`
 - Type: `string`
 - Default: `"AWSAccessKeyID"`
- Secret Key [Mandatory]
The Secret Access Key used to authenticate to the service. It pairs with the Access Key in order to authenticate.
 - ID: `secretKey`

- Type: `string`
- Default: `"AWSecretAccessKey"`
- Root Path

The bucket and an optional root path name for browsing. Starts with a slash and the bucket name (e.g. `/MyBucket/My Folder`), followed by another slash and the optional root path. Dataset names for this connection don't contain segments of the `rootPath`; instead their first segment is a subdirectory of the root path.

 - ID: `rootPath`
 - Type: `string`
 - Default: `"/MyBucket/MyFolder"`

Further connection configurations may be set, which are not in the Connection Management. Such are:

- Bucket

Optional bucket name to be accessed. It works as a "fallback" of the Connection's Root Path configuration. For instance, if no bucket is given in the Root Path, the value from Bucket is used.

 - ID: `awsBucket`
 - Type: `string`
 - Default: `"com.sap.datahub.test"`
- Proxy

An option proxy to be used in the connection to the service.

 - ID: `awsProxy`
 - Type: `string`
 - Default: `""`
- Use SSL

Whether to use SSL/TLS when connecting to the service.



 - ID: `useSSL`
 - Type: `boolean`
 - Default: `true`

Permissions


Permissions in AWS are required to operate over S3 objects. Each operator may require a determined set to successfully operate.

Read File Permissions

To read a single object ("file"), you need the permissions:

- `s3:GetObject` for the given object. See also [AWS S3 GET Object](#)  .
- `s3:GetObjectVersion` for the given object. See also, [AWS S3 Object Permissions](#)  .

To read multiple objects in a prefix ("directory"), you need the permission:

- `s3:ListBucket` for the bucket where the prefix is to be listed. Note that the permission may be narrowed to a directory inside the bucket, and the prefix is subject to this restriction. See also, [AWS S3 GET Bucket](#)  .

If Delete After Send is being used, you also need the permission:

- `s3:DeleteObject` for the given object. See also, [DELETE Object](#) .

Write File Permissions

To write an object ("file"), you need the permission:

- `s3:PutObject` for the bucket to receive the object. See also, [AWS S3 Object Permissions](#) .

If using mode "Append", you also need:

- `s3:GetObject` for the given object. See also, [AWS S3 GET Object](#) . This is due to the restrictions documented further.

Remove File Permissions

To remove an object ("file"), you need the permission:

- `s3:DeleteObject` for the given object. See also, [AWS S3 Object Permissions](#) .

Move File Permissions

As moving consists of copying and removing in S3, you will need the permissions documented in Remove File Permissions and Copy File Permissions.

Copy File Permissions

To copy an object ("file"), you need the permissions:

- `s3:GetObject` for the source object.
- `s3:PutObject` for the bucket to receive the copied object. See also, [AWS S3 Multipart Upload API and Permissions](#) .

If copying by prefix (i.e. a "directory"), the operation is bound to the same permissions documented in Read File Permissions.

Restrictions

Any S3 specific restriction in the operators is documented here. Some may apply broadly to every operator:

- Directories:
In order for a path to be interpreted as a directory, it should end with `/`. For example: `/tmp/` is a directory, while `/tmp` is a file named `tmp`.
- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (`/`) as working directory.

Write File Restrictions

If using "Append" mode, as the S3 API does not support it, the whole file is retrieved from the service in order to append the data and write back to S3; thus, compromising the operation's efficiency.

Move File Restrictions

As the S3 API does not support the move operation, the operation consists of a copy followed by removing the source file. Thus, in cases of failure, the file may be copied and not removed.

Further restrictions are documented in Copy File Restrictions.

Copy File Restrictions

Taking that the operation has a "source" and a "destination" path:

- If the destination is a file, source must also be a file.
- If the destination is a directory, it must be empty.

For instance, in the given file structure:

```
.
|
+-- a
|   +-- file1.txt
|   +-- file2.txt
+-- b
    +-- f1.txt
    +-- f2.txt
```

- Copying source: `a/file1.txt` to destination: `newfile.txt`, would succeed, since the destination does not exist.
- Copying source: `a/file1.txt` to destination: `b/f1.txt`, would succeed and overwrite `b/f1.txt`, since the destination is an existing file.
- Copying source: `a/file1.txt` to destination: `b/`, would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/` would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/dir/` would succeed, since `b/dir/` does not exist.

Related Information

[Amazon S3 Owner's Page](#) ➔

10.3 Azure Data Lake (ADL)

Azure Data Lake (ADL) is Microsoft's Data Lake cloud storage service. Additional information, including the documentation, can be found at the official ADL Homepage.

Many of the SAP Data Hub storage operators offer support for ADL. This documentation covers the common characteristics that this service has across operators.

Connection

In order to use any operator that connects to ADL, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- Account Name [Mandatory]

The ADL account name.

- ID: `accountName`
- Type: `string`
- Default: `""`
- Tenant ID [Mandatory]
The tenant ID from ADL.
 - ID: `tenantID`
 - Type: `string`
 - Default: `""`
- Client ID [Mandatory]
The client ID from ADL.
 - ID: `clientID`
 - Type: `string`
 - Default: `""`
- Client Key [Mandatory]
The ADL client key (secret).
 - ID: `clientKey`
 - Type: `string`
 - Default: `""`
- Root Path
The optional root path name for browsing. Starts with a slash (e.g. `/MyFolder/MySubfolder`).
 - ID: `rootPath`
 - Type: `string`
 - Default: `""`

Permissions

The ADL interface is based on the WebHDFS service, thus the [ADL Permissions](#) for files and directories are based on the POSIX model, that is, for each file or directory there are W, R and X permissions that may be attributed separately to the owner, the group associated with the file/directory and the group of remaining users.

For a finer control, it is possible to define an [Access Control List](#), which allows the definition of specific rules for each user or each group of users.

Read File Permissions

To read a file, you need `w` and `R` permissions on the file.

Write File Permissions

To write a new file, you need `w` permission on the directory where the file will be created.

To append or overwrite an existing file, you need `w` permission on the file.

Remove File Permissions

To remove a file or directory, you need `w` and `R` permissions on the corresponding file/directory.

Move File Permissions

- Moving a File:
To move a file you need `w` permission on the original file and `R` permission on the original directory.
If the destination file already exists and is being overwritten, you need `w` permission on the destination file.
On the other hand, if the file does not exist on the destination, you need `w` permission on the destination directory.
- Moving a Directory:
To move a directory, you need `w` and `R` permissions on the original directory and `w` permission on every file within it.
On the destination folder, you need `w` permission.
If any of the files already exist at the destination and needs to be overwritten, you need the `w` permission on the destination file as well.

Restrictions

Any ADL specific restriction in the operators is documented here. Some may apply broadly to every storage operator:

- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (`/`) as working directory.

Copy File Restrictions

Since the ADL API does not support the copy operation, this behavior can be achieved through `Read + Write`.

Related Information

[Official ADL Homepage](#) 

10.4 Google Cloud Storage (GCS)


GCS is Google's Object Storage cloud service. Additional information, including the documentation, can be found at the official GCS Homepage.

Many of the Storage operators offer support for GCS. This documentation regards the common characteristics that this service has across operators.

This document may refer to an object as a "file", and to an object's prefix as a "directory", if it fits the context of the operator.

Connection

In order to use any operator that connects to GCS, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:


- **Project ID [Mandatory]**
The ID of project that will be used.
 - ID: `projectID`
 - Type: `string`
 - Default: `"project-id"`
- **Key File [Mandatory]**
Path to or content of the service account JSON key. Further information about the JSON key is given [here](#) .
 - ID: `keyFile`
 - Type: `string`
 - Default: `""`
- **Root Path**
The optional root path name for browsing. Starts with a slash and the bucket name (e.g. `/MyBucket/MyFolder`).

i Note

Limitation: currently it is not possible to browse at the root of the connection; i.e. the root path cannot be empty or `/`, unless a fallback bucket is given.

- ID: `rootPath`
- Type: `string`
- Default: `""`

Further connection configurations may be set, which are not in the Connection Management. Such are:

- **Bucket**
Optional bucket name to be accessed. It works as a "fallback" of the Connection's Root Path configuration. For instance, if no bucket is given in the Root Path, the value from Bucket is used.
 - ID: `gcsBucket`
 - Type: `string`
 - Default: `"bucket"`
- **Content Type**
Informs the type of data being sent, allowing the correct rendering of objects. For instance, if you send a JSON file, this should be set to `application/json`. Additional information here: [Working With Object Metadata](#) .
 - ID: `gcsContentType`
 - Type: `string`
 - Default: `""`

Permissions

[GCS Permissions](#) for manipulating objects are described in the [Access Control List](#) documentation as `WRITER`, `READER` and `OWNER`. Each operator may require a determined set to successfully operate.

Read File Permissions

To read a single object ("file"), you need `READER` and `WRITER` permissions on the bucket.

Write File Permissions

To write an object ("file"), you need `WRITER` permission on the bucket.

Remove File Permissions

To remove an object ("file"), you need the `READER` and `WRITER` permissions on the bucket.

Move File Permissions

To move an object ("file"), you need the `READER` and `WRITER` permissions on the origin bucket plus the `READER` and `WRITER` permissions on the destination bucket.

Copy File Permissions

To copy an object ("file"), you need the `READER` permission on the origin bucket, plus the `READER` and `WRITER` permission on the destination bucket.

Restrictions

Any GCS specific restriction in the operators is documented here. Some may apply broadly to every operator:

- Directories:
In order for a path to be interpreted as a directory, it should end with `/`. For example: `/tmp/` is a directory, while `/tmp` is a file named `tmp`.
- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (`/`) as working directory.

Move File Restrictions

As the GCS API does not support the move operation, the operation consists of a copy followed by removing the source file. Thus, in cases of failure, the file may be copied and not removed.

Further restrictions are documented in Copy File Restrictions.

Copy File Restrictions

Taking that the operation has a "source" and a "destination" path:

- If the destination is a file, source must also be a file.
- If the destination is a directory, it must be empty.

For instance, in the given file structure:

```
.
|
+-- a
|   +-- file1.txt
|   +-- file2.txt
+-- b
    +-- f1.txt
    +-- f2.txt
```

- Copying source: `a/file1.txt` to destination: `newfile.txt`, would succeed, since the destination does not exist.
- Copying source: `a/file1.txt` to destination: `b/f1.txt`, would succeed and overwrite `b/f1.txt`, since the destination is an existing file.
- Copying source: `a/file1.txt` to destination: `b/`, would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/` would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/dir/` would succeed, since `b/dir/` does not exist.

Related Information

[Official GCS Documentation](#) ➔

10.5 Hadoop File System (HDFS)

Hadoop Distributed File System is Apache's distributed storage solution. For more information, see the official HDFS documentation.

i Note

Some configurations are only supported with connections defined via Connection Management.

Many of the SAP Data Hub storage operators offer support for HDFS. This documentation covers the common characteristics that this service has across operators.

Connection

In order to use any operator that connects to HDFS, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- Host [Mandatory]
The IP address to the Hadoop name node.
 - ID: `host`
 - Type: `string`

- Default: "127.0.0.1"
- Port

The port to the Hadoop name node. If not informed, will use the protocol's default port.

 - ID: `port`
 - Type: `string`
 - Default: "8020"
- Adicional Hosts

Allow setting additional hosts for high availability. Only supported in connections defined via Connection Management.

 - ID: `additionalHosts`
 - Type: `array`
 - Default: `[{}]`
- Protocol

The protocol to be used. It will not work properly if set differently than `rpc` when the `HDFS` service is selected. In order to use the `webhdfs` or `swebhdfs` protocols, select the `WebHDFS` service in the configurations and it will allow the use of these protocols.

 - ID: `protocol`
 - Type: `string`
 - Default: "rpc"
- Authentication Type

The authentication type to be used. Types "kerberos" and "simple" are only supported in connections defined via Connection Management.

 - ID: `authenticationType`
 - Type: `string`
 - Default: "simple"
 - Possible values:
 - "simple"
 - "kerberos"
 - "basic"
- User

The Hadoop username.

 - ID: `user`
 - Type: `string`
 - Default: "hdfs"
- Root Path

The optional root path name for browsing. Starts with a slash (e.g. `/MyFolder/MySubfolder`).

 - ID: `rootPath`
 - Type: `string`
 - Default: ""

Permissions

The [HDFS Permissions](#) for files and directories are based on the POSIX model, that is, for each file or directory there are W, R and X permissions that may be attributed separately to the owner, the group associated with the file/directory and the group of remaining users.

For a finer control, it is possible to define an [Access Control List](#), which allows the definition of specific rules for each user or each group of users.

Read File Permissions

To read a file, you need `w` and `R` permissions on the file.

Write File Permissions

To write a new file, you need `w` permission on the directory where the file will be created.

To append or overwrite an existing file, you need `w` permission on the file.

Remove File Permissions

To remove a file or directory, you need `w` and `R` permissions on the corresponding file/directory.

Move File Permissions

- Moving a File:
To move a file you need `w` permission on the original file and `R` permission on the original directory.
If the destination file already exists and is being overwritten, you need `w` permission on the destination file.
On the other hand, if the file does not exist on the destination, you need `w` permission on the destination directory.
- Moving a Directory:
To move a directory, you need `w` and `R` permissions on the original directory and `w` permission on every file within it.
On the destination folder, you need `w` permission.
If any of the files already exist at the destination and needs to be overwritten, you need the `w` permission on the destination file as well.

Restrictions

Any HDFS specific restriction in the operators is documented here. Some may apply broadly to every storage operator:

- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (/) as working directory.

Copy File Restrictions

Since the HDFS API does not support the copy operation, this behavior can be achieved through `Read + Write`.

Related Information

[Official HDFS Documentation](#) ↗

10.6 Local File System (File)

The local file system is subject to the cluster's file system.

Many of the SAP Data Hub storage operators offer support for the local file system. This documentation covers the common characteristics that this service has across operators.

Restrictions

Any File-specific restrictions in the operators are documented here. Some may apply broadly to every storage operator:

- Working directory:
For any path that is given to/by this service, the current working directory (.) will be the repo root (the value passed to the command-line argument `-reporoot`), which is shared among all graphs and operators of this Data Pipelines instance. In a cluster, this is usually `/vrep/vflow`. Other directories can be accessed using absolute paths or `..` to access the parent directory.

Copy File Restrictions

Since the local file system API does not support the copy operation, this behavior can be achieved through `Read + Write`.

10.7 Windows Azure Blob Storage (WASB)

Windows Azure Storage Blob (WASB) is one of Microsoft Azure's Storage cloud service. Additional information, including the documentation, can be found at the official WASB homepage.

Many of the SAP Data Hub storage operators offer support for WASB. This documentation covers the common characteristics that this service has across operators.

This document may refer to an object as a "file", and to an object's prefix as a "directory", if it fits the context of the operator.

Connection

In order to use any operator that connects to WASB, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- Account Name [Mandatory]
The account name from WASB.
 - ID: `accountName`
 - Type: `string`
 - Default: `"myaccount"`
- Root Path
- The optional root path name for browsing. Starts with a slash and the container name (e.g. `/MyContainer/MyFolder`).

i Note

Limitation: currently it is not possible to browse at the root of the connection; i.e. the root path cannot be empty or `/`, unless a fallback bucket is given.

- ID: `rootPath`
- Type: `string`
- Default: `""`
- Endpoint Suffix
Optional endpoint suffix.
 - ID: `endpointSuffix`
 - Type: `string`
 - Default: `"core.windows.net"`
- Protocol [Mandatory]
The protocol schema to be used (WASBS/HTTPS or WASB/HTTP).
 - ID: `protocol`
 - Type: `string` (May be `"wasb"` or `"wasbs"`)
 - Default: `"wasbs"`
- Account Key [Mandatory]
The account key from WASB.
 - ID: `accountKey`
 - Type: `string` (Password format)
 - Default: `""`

Further connection configurations may be set, which are not in the Connection Management. Such are:

- Container
Optional container name to be accessed. It works as a "fallback" of the Connection's Root Path configuration, i.e. if no bucket is given in the Root Path, the value from Container is used.
 - ID: `containerName`
 - Type: `string`
 - Default: `"mycontainer"`
- Blob Type
Only used in Write File operator. It sets the blob type of the destination blob (`"file"`).

- ID: `wasbBlobType`
Type: `string`
Default: "BlockBlob"
- Values:
 - "BlockBlob"
 - "PageBlob"
 - "AppendBlob"

Permissions

Permissions in Azure Blob Storage are required to operate over blobs. WASB currently restricts access to blobs through the container's policy:

- Full public read access
- Public read access for blobs only
- No public read access

Find more information [here](#) and [here](#).

Operators will need full access to the data, thus the container should have "Full public read access" if the given credentials are not from the owner of the container; otherwise, any permission should be enough.

Restrictions

Any WASB specific restriction in the operators is documented here. Some may apply broadly to every operator:

- Directories:
In order for a path to be interpreted as a directory, it should end with `/`. For example: `/tmp/` is a directory, while `/tmp` is a file named `tmp`.
- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (`/`) as working directory.

Move File Restrictions

As the WASB API does not support the move operation, the operation consists of a copy followed by removing the source file. Thus, in cases of failure, the file may be copied and not removed.

Further restrictions are documented in Copy File Restrictions.

Copy File Restrictions

Taking that the operation has a "source" and a "destination" path:

- If the destination is a file, source must also be a file.
- If the destination is a directory, it must be empty.

For instance, in the given file structure:

```
.
```

```
|
+-- a
|   +-- file1.txt
|   +-- file2.txt
+-- b
    +-- f1.txt
    +-- f2.txt
```

- Copying source: `a/file1.txt` to destination: `newfile.txt`, would succeed, since the destination does not exist.
- Copying source: `a/file1.txt` to destination: `b/f1.txt`, would succeed and overwrite `b/f1.txt`, since the destination is an existing file.
- Copying source: `a/file1.txt` to destination: `b/`, would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/` would fail, since `b/` already exists and is not empty.
- Copying source: `a/` to destination: `b/dir/` would succeed, since `b/dir/` does not exist.

Related Information

[Official WASB Homepage](#) ➔

10.8 WebHDFS

WebHDFS supports Hadoop Distributed File System through the REST API. It is one of the protocols of Apache's distributed storage solution. For more information, see the official WebHDFS home page.

i Note

Some configurations are only supported with connections defined via Connection Management.

Many of the SAP Data Hub storage operators offer support for WebHDFS. This documentation covers the common characteristics that this service has across operators.

Connection

In order to use any operator that connects to WebHDFS, you may use a Connection ID from the Connection Management, or set a Manual connection with the following values:

- Host [Mandatory]
The IP address to the Hadoop name node.
 - ID: `host`
 - Type: `string`
 - Default: `"localhost"`
- Port
The port to the Hadoop name node. If not informed, will use the protocol's default port.

- ID: `port`
- Type: `string`
- Default: `"50070"`
- **Additional Hosts**
Allow setting additional hosts for high availability. Only supported in connections defined via Connection Management.
 - ID: `additionalHosts`
 - Type: `array`
 - Default: `[{}]`
- **Protocol**
The protocol to be used. The WebHDFS service supports the `webhdfs` and `webhdfs` protocols. To use the `rpc` protocol, the `HDFS` service must be chosen in the configurations.
 - ID: `protocol`
 - Type: `string`
 - Default: `"rpc"`
- **Authentication Type**
The authentication type to be used. Types `"kerberos"` and `"simple"` are only supported in connections defined via Connection Management.
 - ID: `authenticationType`
 - Type: `string`
 - Default: `"simple"`
 - Possible values:
 - `"simple"`
 - `"kerberos"`
 - `"basic"`
- **User**
The Hadoop user name.
 - ID: `user`
 - Type: `string`
 - Default: `"hdfs"`
- **Root Path**
The optional root path name for browsing. Starts with a slash (e.g. `/MyFolder/MySubfolder`).
 - ID: `rootPath`
 - Type: `string`
 - Default: `""`

Further connection configurations may be set, which are not in the Connection Management. Such are:

- **Token**
The Token to authenticate to WebHDFS with.
 - ID: `webhdfsToken`
 - Type: `string`
 - Default: `""`
- **OAuth Token**
The OAuth Token to authenticate to WebHDFS with.
 - ID: `webhdfsOAuthToken`

- Type: `string`
- Default: `""`
- Do As
 - The user to impersonate. Has to be used together with `User`.
 - ID: `webhdfsDoAs`
 - Type: `string`
 - Default: `""`

Permissions

The [WebHDFS Permissions](#) for files and directories are based on the POSIX model, that is, for each file or directory there are W, R and X permissions that may be attributed separately to the owner, the group associated with the file/directory and the group of remaining users.

For a finer control, it is possible to define an [Access Control List](#), which allows the definition of specific rules for each user or each group of users.

Read File Permissions

To read a file, you need `w` and `R` permissions on the file.

Write File Permissions

To write a new file, you need `w` permission on the directory where the file will be created.

To append or overwrite an existing file, you need `w` permission on the file.

Remove File Permissions

To remove a file or directory, you need `w` and `R` permissions on the corresponding file/directory.

Move File Permissions

- Moving a File:
 - To move a file you need `w` permission on the original file and `R` permission on the original directory.
 - If the destination file already exists and is being overwritten, you need `w` permission on the destination file.
 - On the other hand, if the file does not exist on the destination, you need `w` permission on the destination directory.
- Moving a Directory:
 - To move a directory, you need `w` and `R` permissions on the original directory and `w` permission on every file within it.
 - On the destination folder, you need `w` permission.
 - If any of the files already exist at the destination and needs to be overwritten, you need the `w` permission on the destination file as well.

Restrictions

Any WebHDFS specific restriction in the operators is documented here. Some may apply broadly to every storage operator:

- Working directory:
Since there is no concept of a "working directory", any relative directory given to/by this service will have the root directory (/) as working directory.

Copy File Restrictions

Since the WebHDFS API does not support the copy operation, this behavior can be achieved through `Read + Write`.

Related Information

[Official WebHDFS Homepage](#) ➔

11 Change Data Capture (CDC)

SAP Data Hub supports CDC technology for some connection types.

Databases

The following databases support CDC capability by using a trigger-based approach for capturing changes:

- DB2
- HANA
- MSSQL
- MySQL
- Oracle

Because creating the graph can become complex, SAP Data Hub provides a [CDC Graph Generator \(Experimental\)](#) operator that helps you create the appropriate graphs. For more information about setting up trigger-based delta graphs, see [CDC Graph Generator Sample Graph](#).

Cloud Data Integration (Beta)

Cloud Data Integration (CDI) supports polling-based CDC technology.

12 Subengines

Subengines in the SAP Data Hub Modeler allow you to implement operators for different runtimes apart from the main engine. The main engine is a process that coordinates the executions of graphs and native operators.

Subengines allow one graph to contain operators that run in the main engine, other operators that are implemented in Python and executed in the Python subengine, and yet other operators that are implemented in C++ and run in the C++ subengine. Some operators can have implementations for more than one engine (the term engine here means either the main engine or a subengine). In this case, you can select a subset of the available engines in the operator's configuration panel from which the optimizer can choose from. The optimizer assigns an engine for each operator in a way that tries to minimize the number of edges crossing different engines. A cluster of connected operators scheduled to be executed in the same engine all run in the same operating system process.

Current available subengines are: ABAP, C++, Node.js, Python 2.7, and Python 3.6.

⚠ Caution

Future releases of SAP Data Hub will no longer support Python 2.7.

i Note

When you build a graph, communication between different engines can incur different communication costs. For example, if there is a *File Consumer* (an operator that is implemented only in the main engine) that sends data to the *Python3 Operator*, the data must be serialized, sent through a pipe to another operating system process, and then deserialized.

Some advantages of subengines are:

- Connected operators that belong to the same subengine can be run in a single process. This behavior is better than using the *ProcessExecutor Operator* to execute an external script that launches a new process for each operator.
- Scriptable operators in different languages (for example, *Python3 Operator* and *Node.js Base Operator*).
 - The scripts for these operators can be edited in the user interface, and you don't need to handle serializing and deserializing outgoing and incoming data.
- It makes it possible for SAP to develop and deliver operators implemented in programming languages other than the one used in the main engine.
- You can create and add new operators to the SAP Data Hub Modeler in different programming languages.
 - For example, even though the C++ subengine has no script operator, you can still develop new operators in C++ in your local machine, compile it, and then upload a package with the generated `.so` files to the cluster through the SAP Data Hub System Management. More information can be found in the C++ section.
 - For the Python subengines, although you can extend the script operator *Python2 Operator* and *Python3 Operator* with new behavior, you can also develop new operators in your own machine, and then upload the new operators to the cluster via SAP Data Hub System Management. For more information, see [Working with Python 2.7 and Python 3.6 Subengines to Create Operators \[page 186\]](#).

Related Information

[Working with the C++ Subengine to Create Operators \[page 164\]](#)

[Working with Python 2.7 and Python 3.6 Subengines to Create Operators \[page 186\]](#)

[Working with the Node.js Subengine to Create Operators \[page 203\]](#)

[Working with Flowagent Subengine to Connect to Databases \[page 214\]](#)

[SAP Data Hub Operators](#)

12.1 Working with the C++ Subengine to Create Operators

The SAP Data Hub Modeler C++ subengine let you code your own operators in C++ and make them available for use from the user interface.

Introduction

The C++ subengine can detect and execute operators that were compiled into shared objects (*.so). When a user runs a graph on the modeler, the main engine (which also serves the UI over HTTP) breaks it into large subgraphs such that every operator in the same subgraph can be run by the same subengine. The main engine then executes a subengine process for each subgraph.

When launched, the C++ subengine does the following:

1. Looks for and registers operators.
2. Initializes the mechanisms for communicating with the main engine.
3. Receives its subgraph from the main engine.
4. Instantiates the processes in the subgraph and initializes them.
5. Sets up the connections among processes.
6. Starts the graph, handles its input, invokes user-supplied handlers, and writes the output.
7. When instructed by the main engine to stop, or when a fatal error occurs, it cleans up and terminates.

Quick Start

```
#include <v2/subengine.h>
// Port handler
const char* echo_input( v2_process_t proc, v2_datum_t datum )
{
    // Write the input unchanged
    v2_process_output( proc, "output", datum );
    return NULL; // No error
}
// Shutdown handler (optional)
const char* echo_shutdown( v2_process_t proc )
{
```

```

// Clean up echo's resources here.
return NULL;
}
// Init handler
const char* echo_init( v2_process_t proc )
{
    // Call echo_input whenever port "input" receives data
    v2_process_set_port_handler( proc, "input", echo_input );
    // Call echo_shutdown when the process is stopped
    v2_process_set_shutdown_handler( proc, echo_shutdown );
    return NULL;
}
// Init function
// Remove `extern "C"` when compiling as pure C
extern "C" V2_EXPORT void init( v2_context_t ctx )
{
    // Create an operator called "Echo" with ID "demo.echo" and call
    // echo_init when initializing its processes.
    auto op = v2_operator_create( ctx, "demo.echo", "Echo", echo_init );
    // Add an input port called "input" of type string.
    v2_operator_add_input( op, "input", "string" );
    // Add an output port called "output" of type string.
    v2_operator_add_output( op, "output", "string" );
}

```

Related Information

[Getting Started \[page 165\]](#)

[Creating an Operator \[page 166\]](#)

[Logging and Error Handling \[page 168\]](#)

[Port Data \[page 169\]](#)

[Setting Values for Configuration Properties \[page 171\]](#)

[Process Handlers \[page 172\]](#)

[API Reference \[page 174\]](#)

12.1.1 Getting Started

At initialization, the subengine recursively iterates over the contents of its `lib/` directory (in the root folder of the subengine) and loads each `.so` file.

You can use the fact that the iteration is recursive to organize your libraries into subdirectories as necessary. The subengine ignores files that do not end in `.so`.

The engine also expects to find a symbol called `init` in each of those shared objects. In pure C, this must be a function with signature:

```
V2_EXPORT void init( v2_context_t ctx ) { ... }
```

i Note

If you compile your library from C++, mandatorily prepend the function declaration with `extern "C"` to prevent name mangling.

The signatures of the `init` function use the `V2_EXPORT` macro to ensure that the symbol is visible from the engine executable.

The main role of the `init` function is to tell the engine about the operators implemented by this library. Thus, this function is called once when the engine starts running, before its subgraph is even received.

12.1.2 Creating an Operator

To register an operator, call `v2_operator_create` from `init`:

```
extern "C" V2_EXPORT void init( v2_context_t ctx )
{
    v2_operator_t op = v2_operator_create(
        ctx,           // the context passed by the engine to init
        "demo.strlen", // operator ID; must be globally unique
        "Strlen Demo", // user-friendly operator name
        init_strlen    // initialization handler (see below)
    );
    // Add an input port so the operator can receive and process data:
    v2_operator_add_input(
        op,           // pointer returned by v2_operator_create
        "inString",  // port name; must be unique within this operator
        "string"     // port type
    );
    // Add an output port so the operator can send data:
    v2_operator_add_output( op, "outLength", "int64" );
}
```

The subengine does not impose any special format for port names, but we recommended prefixing them with `in` or `out`. The prefixing makes it easier to identify them apart when their names appear in the logs.

i Note

We recommend that you namespace your operator IDs with a meaningful and unique prefix to avoid name clashes. For example, you can prefix with the operator IDs with the organization name.

`op` has an input port of type `string` and an output port of type `int64`. Now, define its initialization handler (`init_strlen`) that will be called every time a process is created to instantiate the `"demo.strlen"` operator:

```
const char* init_strlen( v2_process_t proc )
{
    v2_process_set_port_handler(
        proc,           // the process, created by the engine
        "inString",    // which port this handler is associated with
        strlen_on_input // the actual handler (below)
    );
    return NULL; // no error
}
const char* strlen_on_input( v2_process_t proc, v2_datum_t datum )
{
    // Extract the string contained in `datum`
    const char* str = v2_datum_get_string( datum );
    size_t len = strlen( str );
    // Create the output datum containing the length
    // of the input string
    v2_datum_t out = v2_datum_from_int64( (int64_t)len );
    // Write it to the output port
    v2_process_output( proc, "outLength", out );
    // Release the `out` handle (see explanation below)
```

```
v2_datum_release( out );
return NULL; // no error
}
```

Result

You have created a basic operator. The initialization and termination, configuration properties, and timed handlers (timers) are described in the subsequent chapters of this guide.

Compiling an Operator Library

Compile the code to create an operator into a dynamic library (shared object), so that it can be consumed by the C++ subengine executable.

The minimal commands to compile are:

```
# Compile C/C++ sources into position-independent object files (.o)
$ gcc -c -fPIC <sources>
# Link the object files together into a shared object
$ gcc -shared -o <output-name>.so <object-files>
```

After compiling, you can place the `<output-name>.so` file in the `lib/` directory of the subengine and call `run.sh -j`. This call helps generate the JSON descriptions for your new operator.

Standard and External Libraries

The C++ subengine runs in a Docker container with Debian 9.2. This operating system provides `libstdc++.so.6` that the C++ subengine executable requires to consume the operators.

If your dynamic libraries use a different version or vendor, then you must either:

- Compile with `-static-libstdc++`; or
- Write a custom Dockerfile based on Debian 9.2 in which your desired Standard Library version or vendor is installed and available.

⚠ Caution

Do not replace `libstdc++.so.6` provided by the operating system.

Next, associate your operator with the docker image by adding a tag using the `v2_operator_add_tag` function in your operator library.

The same concept applies to all external libraries on which the operators may depend. You can either link to them statically or include them in a custom Dockerfile.

12.1.3 Logging and Error Handling

The C++ subengine defines the following logging levels:

- INFO
- DEBUG
- WARNING
- ERROR
- FATAL

The engine is in the debug mode when the debug tracing is enabled for the main engine. The debug messages are printed only when the engine is in the debug mode.

→ Remember

Fatal messages stop the graph execution. Error messages do not cause the graph execution to stop.

To log a single string, use the `v2_log_<level>_string` set of function that are declared in `v2/subengine.h`.

For convenience, the variadic functions (such as, printf-like) are declared and implemented in the optional `v2/log.h` header. If you want to include their implementation in your operator library, then before including `v2/log.h` in a source file, define the `V2_LOG_IMPLEMENTATION` macro.

→ Tip

Include in only one of your source files to avoid multiple definitions of those symbols.

For example:

```
#define V2_LOG_IMPLEMENTATION
#include <v2/log.h>
#undef V2_LOG_IMPLEMENTATION
```

This helps to include the header from multiple files at wherever those function declarations are required, and still continue to keep their implementation in a single place.

Error Handling

All types of handlers, which users can provide to the subengine have the return type, `const char*`. This return type is a null-terminated string containing the error message or a `NULL` value, if no error occurred. Errors returned by handlers are always fatal (otherwise, use `v2_log_error_string` or `v2_log_error` instead).

12.1.4 Port Data

Port types are not known at compile time. The subengine uses the handle, `v2_datum_t` to carry the generic inputs and outputs between the operators. The `v2_datum_t` handle is a reference-counted handle to an underlying piece of data.

Also, because this is a pure C API, you cannot use constructors and destructors to acquire and release these handles. Therefore, we recommend that you follow these guidelines:

- If you get a datum as a parameter in the port handler, do not release it. The engine does that.
- If you create a datum using the `v2_datum_create` set of functions, release it before it gets out of scope. Not releasing it may result in memory leaks. It is also important to release it even if you output the datum because this datum is going to be given to the next operator downstream, and the call to `v2_process_output` increases its reference count by one. This is similar to making a copy of a `std::shared_ptr`.

→ Tip

You can explicitly increase reference count of datum by calling `v2_datum_acquire`. This increase can be useful if you want to store a datum given to a port handler. But, do not forget to release it later.

Data Types

A datum may contain any one of the following types:

Modeler type	C type	ID (<code>v2_type_t</code>)	Size (bytes)
string	char*	V2_TYPE_STRING	Length of the string
int64	int64_t	V2_TYPE_INT64	8
float64	double	V2_TYPE_DOUBLE	8
blob	blob	V2_TYPE_BLOB	Size of the blob
uint64	uint64_t	V2_TYPE_UINT64	8
message	v2_message_t	V2_TYPE_MESSAGE	0 (Size of a message is only known when it is serialized, so it cannot be queried).
byte	char	V2_TYPE_BYTE	1
User-defined data types	void*	V2_TYPE_CUSTOM	0 (Engine does not know the size of user-defined data types.)

For each data type, the ID column in the table shows the return value of `v2_datum_get_type`. This allows you to have generic operators (for example, ports with type `any`) that can check their input type at runtime.

The Size column in the table shows the return value of `v2_datum_get_size`.

! Restriction

Array types are not supported in the current version.

Ownership

Whenever the reference count of datum reaches zero, the engine deallocates the memory that belongs to the datum. For `string`, `blob`, `message`, and custom types this behavior can have one further consequence: whether or not to free the memory associated with the data itself. So, when creating a datum from one of those types, you can choose whether or not you want it to own the data:

- A nonowning datum is created using the `v2_datum_from_<type>` functions, which do not deallocate the given data pointer. Thus, ensure that the functions remain available throughout the lifetime of the datum. This behavior is typical of static lifetime variables. For example:

```
const char* YES = "yes";
const char* NO = "no";
// A port handler
const char* is_even_input( v2_process_t proc, v2_datum_t datum )
{
    int64_t i = v2_datum_get_int64( datum );
    v2_datum_t out;
    if ( i % 2 ) // not even
        out = v2_datum_from_string( NO, 2 );
    else
        out = v2_datum_from_string( YES, 3 );
    v2_process_output( proc, "outEven", out );
    // Release local handle
    v2_datum_release( out );
    return NULL;
}
```

- An owning datum is created using the `v2_datum_own_<type>` functions. These functions take not only the pointer to the data, but also a "destructor" function pointer, which will be invoked with the data pointer as the argument. For example, an operator that concatenates the strings it receives two by two:

```
// Port handler
const char* concat_on_input( v2_process_t proc, v2_datum_t datum )
{
    v2_datum_t previous = (v2_datum_t)v2_process_get_user_data( proc );
    if ( previous == NULL ) {
        // Store this datum for later use
        v2_process_set_user_data( proc, datum );
        // Let the engine know we've just stored an additional
        // reference to this datum
        v2_datum_acquire( datum );
    } else {
        char* prev_str = v2_datum_get_string( previous );
        uint64_t prev_size = v2_datum_get_size( previous );
        char* curr_str = v2_datum_get_string( datum );
        uint64_t curr_size = v2_datum_get_size( datum );
        // Allocate enough memory for both strings together. Null-terminating it
        // is optional, since we provide its length to v2_datum_own_string.
        uint64_t out_size = prev_size + curr_size;
        char* out_string = (char*)malloc( out_size );
        // Copy them to out_string
        strncpy( out_string, prev_str, prev_size );
    }
}
```

```

strncpy( out_string + prev_size, curr_str, curr_size );
// Since out_string is on the heap, we need an owning datum
// so it will call `free` on the string once its done.
v2_datum_t out = v2_datum_own_string( out_string, out_size, free );
v2_process_output( proc, "output", out );
// Cleanup
v2_datum_release( out );
v2_datum_release( previous );
v2_process_set_user_data( proc, NULL );
}
return NULL;
}

```

i Note

The datum never makes a copy of the data you provide regardless of the ownership. It only stores the given pointer.

i Note

Both datum constructors for type string require the length of the string as a parameter. This requirement not only ensures safety (if the string is not null-terminated, buffer overruns might occur), but also enables different datum objects to point to different substrings of the same string without making any copies.

User-defined Types

Declare the Modeler name for a user-defined types in the `meta.json` file that is under the root directory of the subengine. Typically, it is `<repo-root>/subengines/<sub-engine-id>/meta.json`.

For example, if you are creating image-processing operators, you may want to define your own "image" type. This user-defined type helps the data to flow among such operators without having to conform to the standard types in the modeler. In this example, your `meta.json` might look like this:

```

{
  "versions": ["1.0"],
  "types": ["image"]
}

```

Adding a type to `meta.json` enables the main engine to recognize its existence and accept graphs that use these types as valid.

→ Remember

The underlying structure of this type is completely unknown to both the main engine and the subengine. Thus, you cannot convey user-defined types between two operators pertaining to different subengines.

12.1.5 Setting Values for Configuration Properties

You set the configuration properties at the operator level and each property is associated with a default value.

If you have provided new values to the properties through the user interface of the Modeler application, then the main engine sends the value to the subengine, when you execute the graph. To set an operator

configuration property along with its default value, use the `v2_operator_config_add_<type>` set of functions.

Using the `v2_process_t` handle, which is first obtained in the `init` handler of that operator, you can query the actual values for each property by calling the `v2_process_config_get_<type>` functions.

If the defined value for a property does not correspond to its declared value, the engine throws an error for the type mismatch. The only exception is for `double` configurations that can receive an integer value by implicitly converting it to `double`.

12.1.6 Process Handlers

You can use the handler types to have more control over the behavior of the operators.

The following are the handler types that you can use.

Initialization Handler

The initialization handler is set for an operator when it is created (`v2_operator_create`) and is invoked for every process that is instantiated for that operator.

The role of this handler is to initialize per-process data, resources, and connections, as required. Typically, the configuration values are also queried here.

Shutdown Handlers

In some cases, you use the `init` handler to only set a port handler. But, if it was a more complex case, you may have acquired some resources or allocated memory for that process individually. The place to clean up such resources is the shutdown handler:

```
// shutdown handler
const char* proc_shutdown( v2_process_t proc )
{
    // free proc's memory and resources here
}
// init handler
const char* proc_init( v2_process_t proc )
{
    ...
    v2_process_set_shutdown_handler( proc, proc_shutdown );
    ...
}
```

Shutdown handlers are called when the graph stops if (and only if) the `init` handler for that process was called (regardless of its returned status). This means that, the only case in which the shutdown handler will not be called is when the `init` handler is not even called because a previous process failed to initialize.

Input Handlers

Operator input is given to the handler bound to the port that received the data. Port handlers are set per process instead of per operator. The handlers allow you to change the behavior of a process depending on its configuration properties. This means that, your operator can have different "modes" without having to check which one was set at every input.

An input port will be blocked while its handler is still running on the last piece of data it received. Therefore, the handler of a given port will never be called multiple times simultaneously, but rather sequentially for every consecutive piece of data. On the other hand, handlers for different ports may be called simultaneously depending on when each port received the input.

One port may not have multiple handlers. Calling `v2_process_set_port_handler` on an already bound port will replace the existing handler. This call can be useful to change the behavior of the process at runtime.

Timers

Timed handlers are a way of executing logic repeatedly without having to wait for the input on a port. They are completely independent from input ports and may, if desired, produce output. Use `v2_process_add_timed_handler` to register a timer.

Timed handlers are the only ones that may be registered multiple times, as they behave independently from one another, calling the provided callback function at every period.

For example, you can use a timer in an operator whose purpose is to generate random numbers at a specific interval:

```
// Timed handler
const char* gen_tick( v2_process_t proc )
{
    v2_datum_t datum = v2_datum_from_int64( rand() );
    v2_process_output( proc, "output", datum );
    v2_datum_release( datum );
    return NULL;
}
// Init handler
const char* gen_init( v2_process_t proc )
{
    ...
    v2_process_add_timed_handler(
        proc,
        "gen",      // Name to be used in the logs to tell handlers apart
        gen_tick,  // Handler function
        0,         // Repeat count (zero for unlimited)
        1000      // Interval (ms)
    );
    ...
}
```

12.1.7 API Reference

Use the C++ API to create operators and to work with the SAP Data Hub subengine.

Required header files to work with the C++ subengine:

- `log.h`
- `subengine.h`

Related Information

[log.h \[page 174\]](#)

[subengine.h \[page 175\]](#)

12.1.7.1 log.h

Declares and implements convenience wrappers for logging.

In one of your source files, add

Source Code

```
#define V2_LOG_IMPLEMENTATION
#include "v2/log.h"
#undef V2_LOG_IMPLEMENTATION
```

→ Remember

Always add the code block in only one of your source files.

This helps to compile the code with not only their declarations, but with also the definition of these functions. Everywhere else in your code, when `log.h` is included without the macro, it brings over the declarations only.

We recommend adding the code in only one of your source file, because it includes variadic functions that cannot be implemented within the engine for consumption from another binary. If you add them in more than one file, the executable would make assumptions about the implementation (for plugin) of variadic arguments that may not hold for different compiler vendors and versions.

→ Tip

If you do not prefer to use the convenience functions, and if you want to call only the `_string`-suffixed ones declared in the `subengine.h` header file, then it is not required to define the `V2_LOG_IMPLEMENTATION` macro or to include `log.h`.

Formatted Logging

The following are convenience functions to send `sprintf`-formatted strings to the engine log.

- `void v2_log_info (const char *fmt, ...)`
- `void v2_log_debug (const char *fmt, ...)`
- `void v2_log_warning (const char *fmt, ...)`
- `void v2_log_error (const char *fmt, ...)`
- `void v2_log_fatal (const char *fmt, ...)`

12.1.7.2 subengine.h

Establishes the core set of functions for the subengine interface.

Macros

```
#define V2_TYPE_ARRAY 0x0B
#define V2_TYPE_BLOB 0x03
#define V2_TYPE_BOOL 0x08
#define V2_TYPE_CUSTOM 0x07
#define V2_TYPE_DOUBLE 0x02
#define V2_TYPE_INT64 0x01
#define V2_TYPE_MAP 0x0A
#define V2_TYPE_MESSAGE 0x05
#define V2_TYPE_NULL 0x09
#define V2_TYPE_STRING 0x00
#define V2_TYPE_UINT64 0x04
```

Typedef

Typedef	Description
<code>typedef void* v2_context_t</code>	Handle to the internal data that the engine may need to relay from the <code>init()</code> function to some other with the help from the plugin.

Typedef	Description
<pre>typedef struct v2_datum* v2_datum_t</pre>	<p>Handle to a piece of data exchanged between two ports.</p> <div style="background-color: #f0f0f0; padding: 10px; border: 1px solid #ccc;"> <p>i Note</p> <p>A datum can hold any of the types known to the engine or a pointer to a user-defined type. In the latter, the datum should not cross the subengine boundaries, as there is no guarantee that other subengines or the main engine will recognize this type.</p> </div>
<pre>typedef void* v2_map_t</pre>	<p>Handle to a JSON-like object, which is a mapping of <code>const char*</code> keys to <code>v2_value_t</code>.</p>
<pre>typedef void* v2_message_t</pre>	<p>Handle to a message, when the data type transferred between ports of type "message".</p>
<pre>typedef void* v2_process_t</pre>	<p>Handle to a process (a running instance of an operator), which is created internally by the engine and supplied to the plugin in a call to the init handler of the operator.</p>
<pre>typedef void* v2_value_t</pre>	<p>Handle to a JSON-like value.</p> <p>This can contain one of the following:</p> <ul style="list-style-type: none"> • <code>string(const char*)</code> • <code>integer(int64_t)</code> • <code>decimal(double)</code> • <code>byte(char)</code> • <code>null</code> (see <code>v2_value_is_null()</code>) • <code>object(v2_map_t)</code> • <code>array(v2_array_t)</code>

Functions

- `V2_EXPORT v2_datum_t v2_datum_acquire (v2_datum_t datum)`

Use:

Notifies that a new handle to `datum` is being held, incrementing its (handle) internal reference counter.

Description:

If `datum` were an `std::shared_ptr`, this function would be analogous to the copy constructor.

Returns

`datum`

i Note

Every call to `acquire` must be paired with a call to `release`. If it is not paired, the memory for the datum will be leaked.

- `V2_EXPORT v2_datum_t v2_datum_copy (v2_datum_t datum)`

Use:

Creates a new datum that contains a copy of the value of the datum.

Description:

If datum contains a string, blob, or message, a deep copy will be made and the resulting datum will own this copy.

If datum contains custom data, this function will raise an error. To make a copy of a custom type:

```
void* ptr = v2_datum_get_custom( my_datum );
void* ptr_copy = copy_my_type( (my_type*)ptr );
v2_datum_t my_datum_copy = v2_datum_own_custom( ptr_copy, free_my_type );
```

Here, it is assumed that `copy_my_type` allocates a new `my_type` value that can later be deallocated with `free_my_type`.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_blob (void * buf, uint64_t size)`

Use:

Creates a nonowning datum that refers to an existing blob.

Description:

The blob will not be copied and will not be deallocated when the datum is destroyed.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_byte (char b)`

Use:

Creates a datum containing a byte.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_custom (void * ptr)`

Use:

Creates a nonowning datum that refers to a user-defined area in the memory.

Description:

The contents of this area will not be copied and the area will not be deallocated when the datum is destroyed.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_double (double d)`

Use:

Creates a datum containing a double.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_int64 (int64_t i)`

Use:

Creates a datum containing an `int64_t`.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_message (v2_message_t message)`

Use:

Creates a nonowning datum that refers to an existing message.

Description:

The message will not be copied and will not be deallocated when the datum is destroyed.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_string (char * s, uint64_t size)`

Use:

Creates a nonowning datum that refers to an existing string.

Description:

The string will not be copied and it will not be deallocated when the datum is destroyed.

Parameters

[in] `s`: pointer to the start of the string

[in] `size`: the size of the string in bytes, excluding the terminating null byte

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT v2_datum_t v2_datum_from_uint64 (uint64_t u)`

Use:

Creates a datum containing a `uint64_t`.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT void* v2_datum_get_blob (v2_datum_t datum)`

Use:

Returns `datum` as a blob.

Description:

This function will not check if the contained data is a blob (see, `v2_datum_get_type()`). Also, you can use `v2_datum_get_size()` to get the size of the blob.

- `V2_EXPORT char* v2_datum_get_byte (v2_datum_t datum)`
Use:
 Returns datum as a byte.
Description:
 This function will not check if the contained data is a byte (see, `v2_datum_get_type()`).
- `V2_EXPORT void* v2_datum_get_custom (v2_datum_t datum)`
Use:
 Returns datum as custom data.
Description:
 This function will not check if the contained data is a custom pointer (see, `v2_datum_get_type()`).
- `V2_EXPORT double* v2_datum_get_double (v2_datum_t datum)`
Use:
 Returns datum as a double.
Description:
 This function will not check if the contained data is a double (see, `v2_datum_get_type()`).
- `V2_EXPORT int64_t* v2_datum_get_int64 (v2_datum_t datum)`
Use:
 Returns datum as an `int64_t`.
Description:
 This function will not check if the contained data is an `int64_t` (see, `v2_datum_get_type()`).
- `V2_EXPORT v2_message_t v2_datum_get_message (v2_datum_t datum)`
Use:
 Returns datum as a message.
Description:
 This function will not check if the contained data is a message (see, `v2_datum_get_type()`).
- `V2_EXPORT uint64_t v2_datum_get_size (v2_datum_t datum)`
Use:
 Returns the size in bytes of the data held by the datum.
Description:
 The return value for each contained type is:
 - `V2_TYPE_STRING`: the length of the string in bytes, excluding the terminating null byte.
 - `V2_TYPE_INT64`: 8
 - `V2_TYPE_DOUBLE`: 8
 - `V2_TYPE_BLOB`: the size of the blob in bytes.
 - `V2_TYPE_UINT64`: 8
 - `V2_TYPE_MESSAGE`: 0 (only known when the message is serialized)
 - `V2_TYPE_BYTE`: 1
 - `V2_TYPE_CUSTOM`: 0 (not known by the engine)
- `V2_EXPORT char* v2_datum_get_string (v2_datum_t datum)`
Use:
 Returns datum as a null-terminated string.
Description:
 This function will not check if the contained data is a string (see, `v2_datum_get_type()`). Also, you can use `v2_datum_get_size()` to get the size of the string (not including the terminating null byte).
- `V2_EXPORT v2_type_t v2_datum_get_type (v2_datum_t datum)`
Use:
 Returns a `v2_type_t` constant that describes the type held by datum.

Description:

The possible values are:

- V2_TYPE_STRING
- V2_TYPE_INT64
- V2_TYPE_DOUBLE
- V2_TYPE_BLOB
- V2_TYPE_UINT64
- V2_TYPE_MESSAGE
- V2_TYPE_BYTE
- V2_TYPE_CUSTOM

- V2_EXPORT uint64_t* v2_datum_get_uint64 (v2_datum_t datum)

Use:

Returns datum as a uint64_t.

Description:

This function will not check if the contained data is a uint64_t (see, v2_datum_get_type()).

- V2_EXPORT v2_datum_t v2_datum_own_blob (void * buf, uint64_t size, v2_destructor_t destructor)

Use:

Creates a datum that takes ownership of an existing blob.

Description:

The blob is not copied, and it will be deallocated when the data is destroyed by calling destructor(s).

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, v2_datum_release().

- V2_EXPORT v2_datum_t v2_datum_own_custom (void * ptr, v2_destructor_t destructor)

Use:

Creates a datum that takes ownership of a user-defined area in the memory.

Description:

The contents of this area will not be copied, and the area will be deallocated when the data is destroyed by calling destructor(s).

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, v2_datum_release().

- V2_EXPORT v2_datum_t v2_datum_own_message (v2_message_t message)

Use:

Creates a datum that takes ownership of an existing message.

Description:

The message will not be copied, and the message will be deallocated when the data is destroyed by calling destructor(s).

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, v2_datum_release().

- `V2_EXPORT v2_datum_t v2_datum_own_string (char * s, uint64_t size, v2_destructor_t destructor)`

Use:

Creates a datum that takes ownership of an existing string.

Description:

The string will not be copied, and the string will be deallocated when the data is destroyed by calling `destructor(s)`.

→ Remember

Release the returned datum (even if it was outputted) once you are done with it. See, `v2_datum_release()`.

- `V2_EXPORT void v2_datum_release (v2_datum_t datum)`

Use:

Notifies that the handle `datum` is being released.

Description:

After this call, it cannot be guaranteed that the actual data underlying the handle will be available.

Therefore, you should no longer use `datum`. If `datum` were an `std::shared_ptr`, this function would be analogous to the destructor.

i Note

Every call to `v2_datum_acquire`, `v2_datum_from_*`, `v2_datum_own_*`, or `v2_datum_copy` must be paired with a call to `release`, if not the memory for the datum will be leaked. If you output such a datum before releasing, the engine will ensure that it survives the call to `release` and that the downstream operators have access to it.

- `V2_EXPORT void v2_log_debug_string (const char * str)`

Use:

Logs the debug information.

Description:

If either the main engine was run in debug mode, or if the subengine was forced into debug mode by the `run.sh` script, then this debug information will show up in the output of the main engine,

- `V2_EXPORT void v2_log_fatal_string (const char * str)`

Use:

Logs a string as a fatal error that will cause the main engine to terminate the subengine, its graph, and processes.

- `V2_EXPORT void v2_message_destroy (v2_message_t msg)`

Use:

Frees the memory associated with a message.

Description:

Call this function only if you obtained the message using `v2_message_create()`.

- `V2_EXPORT v2_value_t v2_message_get_attribute (v2_message_t msg, const char * key)`

Use:

Returns the value associated to a message attribute.

Description:

This is a convenience for `v2_map_find(v2_message_get_attributes(msg), key)`.

- `V2_EXPORT void v2_message_set_body (v2_message_t msg, v2_datum_t body)`

Use:

Sets the message body by releasing the previous one (if any) and by calling `v2_datum_acquire(body)`.

- `V2_EXPORT void v2_operator_add_input (v2_operator_t op, const char * port_id, const char * port_type)`

Use:

Adds an input port called `port_id` to `op`.

Description:

`port_type` is a null-terminated string describing the type accepted by the port. The possible port types are:

- "string"
 - "blob"
 - "int64"
 - "uint64"
 - "float64"
 - "message"
 - "byte"
 - anything else will be regarded as a custom (user-defined) type, which can be conveyed between any two operators, provided they are both implemented by the same subengine.
- `V2_EXPORT void v2_operator_add_output (v2_operator_t op, const char * port_id, const char * port_type)`

Use:

Adds an output port called `port_id`.

Description:

`port_type` will be treated as described in `v2_operator_add_input()`.

- `V2_EXPORT v2_array_t v2_operator_config_add_array (v2_operator_t op, const char * key)`

Use:

Creates and returns a JSON array configuration value named `key` to `op`.

Description:

You can modify the returned array to customize the initial value for this property.

⚠ Caution

The array may be relocated in memory in a subsequent call to `v2_operator_config_add_*`. Thus, ensure you are modifying it immediately after obtaining it. Do not store the array for later use.

- `V2_EXPORT v2_map_t v2_operator_config_add_map (v2_operator_t op, const char * key)`

Use:

Creates and returns a JSON object configuration value named `key` to `op`.

Description:

You can modify the returned map to customize the initial value for this property.

i Note

The map may be relocated in memory in a subsequent call to `v2_operator_config_add_*`. Thus, ensure you are modifying it immediately after obtaining it. Do not store the array for later use.

- `V2_EXPORT void v2_operator_config_add_string (v2_operator_t op, const char * key, const char * default_value)`

Use:

Adds a string configuration value named `key` to `op`.

Description:

`default_value` must be a null-terminated string containing the initial value for this property.

- `V2_EXPORT v2_operator_t v2_operator_create (v2_context_t ctx, const char * id, const char * name, v2_init_handler_t init)`

Use:

Creates an operator.

Parameters

[in] `ctx`: the context handle supplied to the `init()` function

[in] `id`: the operator ID

[in] `name`: the operator name, which will be visible on the GUI

[in] `init`: the callback function that the engine will call when a process is instantiated from this operator.

Description:

The init handler can (not necessarily in this order):

- Access configuration values: use the `v2_process_get_config_*` set of functions to get the actual configuration values for your process. These values remain available throughout the lifetime of the processes, but the init handler is usually the first (rarely) place where they are read.
- Set input handlers: if your operator has input ports, the process is expected to set a handler for each of them (see, `v2_process_set_port_handler()`). It is not an error to leave a port unhandled, provided it is never connected. It is also possible to conditionally set input handlers depending on the configuration values.

i Note

Connected ports that are unhandled results in an error and cause the engine to stop.

- Set the shutdown handler: if your operator needs to clean up after running. This callback will be invoked by the engine when the graph is stopped. This is also usually the place to release resources acquired in the init handler.

- `V2_EXPORT void v2_operator_set_visibility (v2_operator_t op, int visible)`

Use:

Sets whether or not `op` should be visible.

Description:

Visible components have their JSON descriptions automatically generated by the subengine so that they show up on the user interface.

- `V2_EXPORT void v2_process_add_timed_handler (v2_process_t proc, const char * name, v2_timed_handler_t handler, uint64_t repeat, uint64_t interval_ms)`

Use:

Adds a timed handler.

Parameters

[in] `proc`

[in] `name`: an optional name for the handler (for clearer logging). It can be NULL or empty too. It does not have to be unique.

[in] `handler`: the callback function

[in] `repeat`: the number of times the handler is to be called, or zero for unlimited (as long as the graph is running).

[in] `interval_ms` the period in milliseconds for the timer

- `V2_EXPORT double v2_process_config_get_double (v2_process_t proc, const char * key)`

Use:

Returns the final value for a `double` configuration property.

i Note

Properties originally declared with `v2_operator_config_add_int64()` can also be retrieved using this function. It automatically converts the integer to double.

- `V2_EXPORT v2_bool_t v2_process_config_is_null (v2_process_t proc, const char * key)`

Use:

Returns whether the final value for a configuration property is the JSON keyword `null`.

- `V2_EXPORT const char* v2_process_get_id (v2_process_t proc)`

Use:

Returns the process ID, which uniquely identifies it in the graph.

→ Remember

This is not the same as the operator ID.

- `V2_EXPORT void v2_process_output (v2_process_t proc, const char * port_id, v2_datum_t data)`

Use:

Sends output to a port.

Description:

This call will block until the destination (the process downstream) is ready to consume.

- `V2_EXPORT void v2_process_set_port_handler (v2_process_t proc, const char * port_id, v2_port_handler_t handler)`

Use:

Sets a callback function to be invoked when the input port named `port_id` receives data.

Description:

Calls to `handler` will be made sequentially, never concurrently. Successive calls to this function replace the currently set handler.

- `V2_EXPORT void v2_process_set_user_data (v2_process_t proc, void * data)`

Use:

Stores a user-supplied pointer.

Description:

This function is useful if you need each instance of an operator to carry some individual information (for example, a file descriptor, a connection object, and so on). The initial value is `NULL`.

- `V2_EXPORT v2_array_t v2_value_get_array (v2_value_t v)`

Use:

Returns the array contained in `v`.

Description:

If `v` does not contain an array, it will raise an error.

- `V2_EXPORT v2_bool_t v2_value_get_bool (v2_value_t v)`

Use:

Returns the boolean contained in `v`.

Description:

If `v` does not contain a boolean, it will raise an error.

- `V2_EXPORT double v2_value_get_double (v2_value_t v)`

Use:

Returns the decimal contained in `v`.

Description:

If `v` does not contain a decimal, it will raise an error.

- `V2_EXPORT int64_t v2_value_get_int64 (v2_value_t v)`

Use:

Returns the integer contained in `v`.

Description:

If `v` does not contain an integer, it will raise an error.

- `V2_EXPORT v2_map_t v2_value_get_map (v2_value_t v)`

Use:

Returns the object contained in `v`.

Description:

If `v` does not contain an object, it will raise an error.

- `V2_EXPORT const char* v2_value_get_string (v2_value_t v)`

Use:

Returns the string contained in `v`.

Description:

If `v` does not contain a string, it will raise an error.

- `V2_EXPORT v2_type_t v2_value_get_type (v2_value_t v)`

Use:

Returns the type contained in a JSON value.

Description:

The possible values are:

- `V2_TYPE_STRING`
- `V2_TYPE_INT64`
- `V2_TYPE_DOUBLE`
- `V2_TYPE_BOOL`
- `V2_TYPE_NULL`
- `V2_TYPE_MAP`
- `V2_TYPE_ARRAY`

12.2 Working with Python 2.7 and Python 3.6 Subengines to Create Operators

The SAP Data Hub Modeler Python subengine lets you code your own operators in Python and make them available for use from the user interface.

Introduction

i Note

Python 2.7 will no longer be supported in a future SAP Data Hub release; Python 3.6 will continue to be supported.

The Modeler subengines are a way to allow you to code your own operators in a particular programming language and make them available for use in pipelines. The Python subengines can execute operators that are written and stored in the way specified by this guide.

When you run a graph (pipeline) in SAP Data Hub Modeler, the main engine, which coordinates all subengines, breaks it into large subgraphs so that every operator in the same subgraph can be run by the same subengine. The main engine then fires a subengine process for each subgraph.

i Note

When you create a new dockerfile that you want to use in graphs containing Python operators, at minimum the following tags must be associated with the dockerfile: `'tornado': '5.0.2'`, `'sles': ''` and either `'python27': ''` or `'python36': ''`. Make sure that the resources are installed on the dockerfile.

⚠ Caution

Future releases of SAP Data Hub will no longer support Python 2.7.

This guide provides information so that you can create new operators for both the Python 2.7 and Python 3.6 subengines. Both subengines are similar, except that one accepts only Python 2.7 and the other only Python 3.6 compatible code. Another important difference is the mapping between Python types and SAP Data Hub Modeler types. The following table shows the type mapping for both subengines and SAP Data Hub Modeler types:

Modeler	Python 2.7	Python 3.6
string	unicode	str
blob	str	bytes
int64	int, long	int
uint64	int, long	int

Modeler	Python 2.7	Python 3.6
float64	float	float
byte	int	int
message	Message	Message
[]x	list	list

Replace the letter `x` in the last row of the table with any of the allowed types (except for `message`); for example, `[]string`, `[]uint64`, `[]blob`, and so on.

If you create operators that receive or send data types that have no mapping to SAP Data Hub Modeler types, you must create ports with type `python27` or `python36`. For example, you can have connect many [Python3 Operators](#), all of which have inports and outports of type `python36`. Those operators can communicate with any Python object among themselves (including those with no corresponding SAP Data Hub Modeler type; for example, `set`, `numpy` arrays, and so on). There is one caveat: Subengine-specific types cannot cross the boundary between two SAP Data Hub Modeler [groups](#) (see step 9). However, if you place your Python-specific object inside the body of a message, then the body is correctly serialized and deserialized with `pickle` (a Python module) when crossing the boundary between two groups. The behavior of serialization of the body with `pickle` is specific to the Python subengine, and should not be expected in other subengines.

Create the message type using the format `Message(body, attributes)`, where `body` can be any object, and `attributes` is a dictionary mapping string to any object. If `m` is an object of type `message`, then you can use the commands `m.body` and `m.attributes` to access the fields initialized in the class constructor. The `attributes` argument is optional in the message constructor. If you want an empty `attributes` string, you can construct your message using the format `Message(body)`.

We recommend that you do not use a message inside the body of another message. Instead, use a dictionary inside the body of a message. The dictionary must have the keys "Body" and "Attributes". If you use a message object as the body of another message, you may get unexpected behavior when transferring the message across the boundary of different subengines. For example, the inner message may be automatically converted to a dictionary when crossing the boundaries of different subengines, but it is not converted back to message when coming back to your operator's subengine. Thus, we recommend to always prefer dictionaries over messages inside the body of a message, because it will not change type during the communication.

```
# DO NOT DO THE FOLLOWING:
inner_msg = Message("body_of_inner_msg")
outer_msg = Message(body=inner_msg, attributes={})
```

```
# If you want to have a message as the body of another message, do this instead:
inner_msg = {"body": "body_of_inner_msg"}
outer_msg = Message(inner_msg)
```

You can use one of two methods to create your own Python operators:

- Normal Python subengines usage involves using only the SAP Data Hub Modeler user interface.
- The advanced method shows you how to create new Python operators on your own machine and then upload them to SAP Data Hub. The advanced method of creating operators is useful when you want to code in your own IDE and create unit tests for your operators.

Related Information

[Normal Usage \[page 188\]](#)

[Advanced Usage \[page 190\]](#)

12.2.1 Normal Usage

If you want to create simple scripts quickly, we recommend that you customize the behavior of operators using the SAP Data Hub Modeler user interface.

There are two ways to customize the behavior of a *PythonX Operator* (where X is either 2 or 3):

- Drag the *PythonX Operator* to the graph canvas and edit the script of the operator by right-clicking on it and clicking the open script option.
- Create a new operator in the *Repository* tab and select *PythonX Operator* as the base operator to be extended.

The advantage of the second approach is that you can reuse your new operator across many graphs, while in the first approach, the edited script is specific to the given graph and operator instance.

To create a new operator in the *Repository* tab using the *PythonX Operator*:

1. Open the *Repository* tab and expand the *Operators* section.
2. To create an operator, right-click the appropriate *Operators* subfolder, and select *Create Operator*.
3. In the dialog, enter the operator name, display name, and base operator and choose *PythonX Operator* as the base operator.
4. In the operator editor view, you can create input ports, output ports, tags, and configuration parameters, and write your script.

Related Information

[Using Python Libraries \[page 188\]](#)

12.2.1.1 Using Python Libraries

To make Python libraries available to your operator, you can add tags to it so that upon graph execution, the appropriate docker image can be chosen.

i Note

Python 2.7 will no longer be supported in a future SAP Data Hub release; Python 3.6 will continue to be supported.

You can create or extend dockerfiles through the user interface and add tags to associate it with an operator. The following example details the necessary steps.

Suppose that you want to use the numpy library (version 1.16.1) on your custom Python operator.

First, create a new dockerfile:

1. Open the [Repository](#) tab and select a subfolder.
2. Right-click the subfolder and select [Create Docker File](#).
3. Provide a name and choose [OK](#).

There are many ways to create a dockerfile that contains Python and the numpy library. For example, you can enter `FROM <public_numpy_docker_image>` in the first line. If you use a public image, make sure that it also contains the requirements to run the Python subengine, which are: `tornado==5.0.2` and either `python2.7` or `python3.6`. The tag key-value pairs for the requirements are: `'tornado': '5.0.2'` and `'python27': ''` or `'python36': ''`.

Alternatively, you can inherit from the default Modeler python3 dockerfile and add numpy to it. In this way, all of the requirements for the standard [Python3 Operator](#) are already satisfied. The following dockerfile shows one way that you can accomplish this:

```
FROM $com.sap.sles.base
RUN pip3.6 pip install numpy=="1.16.1"
```

4. After you write the dockerfile content, add tags on the configuration panel for every relevant resource that this dockerfile offers, in addition to the tags from its parent dockerfile. That is, you don't need to repeat the tags that the parent dockerfile already has. For example, the numpy for python3 dockerfile needs only the following tag `numpy36: 1.16.1`.

Note

Make sure that you install the library for the right version of Python. If you use [Python2 Operator](#), then you should install the library for Python 2.7 and you should target Python 3.6 when using [Python3 Operator](#). Notice that we added the suffix "27" and "36" to the numpy tag to reflect the Python version where they were installed.

Caution

The future releases of SAP Data Hub will no longer support Python 2.7.

5. The final step is to add the new tag to your operator in the operator editor view. Add the tag `"numpy36": "1.16.1"` to your Python operator. Alternatively, you can also add tags to a group defined on the graph.

For more information about the [Python2 Operator](#) and [Python3 Operator](#), read their respective documentation in the SAP Data Hub Modeler user interface.

Related Information

[Create Dockerfiles \[page 218\]](#)

12.2.2 Advanced Usage

You can create operators without using the Modeler user interface.

i Note

This method of creating operators requires that you restart the Modeler instance for the changes to take effect.

The following example assumes that you use a UNIX-like system to develop your operators. To run tests, you must download the `pysix_subengine` package on your local machine.

To download the `pysix_subengine` package, follow these steps:

1. Log in to SAP Data Hub System Management as an administrator.
2. Click the *File* button.
3. Click the *Union View* button.
4. Navigate to `files/vflow/subdevkits` on the file explorer.
5. Right-click the `pysix_subengine` folder and select *Export File*.

To create Python 3.6 operators, you must structure your solution (a package that can be imported by SAP Data Hub System Management) as follows:

```
my_solution/
  vrep/vflow/
    subengines/com/sap/python3.6/operators/
      myOperator1/
        operator.py
        operator.json
      com/
        mydomain/
          myOperator2/
            operator.py
            operator.json
  vsolution.json
```

The `vsolution.json` file should look like the following:

```
{
  "name": "vsolution_vflow_my_solution",
  "description": "My Solution",
  "license": "my license"
}
```

⚠ Caution

Future releases of SAP Data Hub will no longer support Python 2.7.

Define the Operator Attributes and Behavior

The `operator.py` script defines both the operator attributes and the operator behavior.

i Note

The script must be called `operator.py`.

i Note

In these instructions, we refer to the Python 3.6 subengine, but the explanations apply equally to the Python 2.7 subengine. Simply replace 36 with 27 in the directory paths shown in this topic.

The following is example code in the `operator.py` file:

```
from pysix_subengine.base_operator import PortInfo, OperatorInfo
from pysix_subengine.base_operator import BaseOperator
# The operator class name (which inherits BaseOperator) should have the same
name as
# the operator's folder, except that its first letter should be uppercase.
class AppendString(BaseOperator):
    def __init__(self, inst_id, op_id):
        super(AppendString, self).__init__(inst_id=inst_id, op_id=op_id)
        # Default configuration fields. They will be shown in the UI.
        self.config.stringToAppend = ""
        self.config.method = "NORMAL"
        # Adds a callback '_data_in' that is called every time the
        # operator receives data in the inport 'inString'.
        self._set_port_callback('inString', self._data_in)
        self.__transform_data = None
    # This method is mandatory.
    # The operator.json will be generated based mostly on the OperatorInfo
    returned by this method.
    def _get_operator_info(self):
        inports = [PortInfo("inString", required=True, type_="string")]
        outports = [PortInfo("outString", required=True, type_="string")]
        return OperatorInfo("Append String",
                            inports=inports,
                            outports=outports,
                            icon="puzzle-piece",
                            tags={"numpy36": "1.16.1"},
                            dollar_type="http://sap.com/vflow/
com.mydomain.appendString.schema.json#")
    def _set_websocket(self, handler):
        self.web_handler = handler
        self.web_handler.set_message_callback(self._on_message)
    def _on_message(self, message):
        self.web_handler.write_message(str(message))
        self._send_message("outString", message)
    def _data_in(self, data):
        self.metric.include_value(1)
        self._send_message('outString', self.__transform_data(data +
self.config.stringToAppend))
    # Configs set in the UI are already available when this method is called.
    # _init is called before any operator main loop has started.
    def _init(self):
        metric_infos = []
        metric_infos.append(self._registry.create_metric_info(u'count', 1,
u'name', u'display_name', u'unit'))
        self.metric = self.register_metric(u'int", u"TEST", metric_infos)
        self.register_websocket_route('/socket', 'test', self._set_websocket)
        self.register_static_route('/ui', 'static')
        self.register_rproxy_route('/rproxy/*path', 'http://localhost:' +
str(self.config.port) + '/')

    if self.config.method == "NORMAL":
        self.__transform_data = lambda x: x
    elif self.config.method == "UPPERCASE":
        self.__transform_data = lambda x: x.upper()
```

```

        else:
            raise ValueError("Unknown config set in configuration: '%s'." %
self.config.method)
        # Called before the operator's main loop execution.
        # Other operators may already have started execution.
        def _prestart(self):
            pass
        # Called when the graph is being terminated.
        def shutdown(self):
            pass

```

Let's examine the script step-by-step. First, look at the class definition:

```
class AppendString(BaseOperator):
```

As you can see, the `AppendString` class extends the built-in `BaseOperator` class.

Note

The class name must have the same name as the folder, except the first letter must be uppercase, while the folder's first letter is lowercase. For example, if the folder is named `appendString`, the class should be named `AppendString`.

Now let's examine each of the methods:

```

def __init__(self, inst_id, op_id):
    super(AppendString, self).__init__(inst_id=inst_id, op_id=op_id)
    # Default configuration fields. They will be shown in the UI.
    self.config.stringToAppend = ""
    self.config.method = "NORMAL"
    # Adds a callback '_data_in' that is called every time the
    # operator receives data in the inport 'inString'.
    self._set_port_callback('inString', self._data_in)
    self.__transform_data = None

```

This method is the class constructor. First it calls `super(AppendString, self).__init__(inst_id=inst_id, op_id=op_id)`.

In the next line, the method creates two new configuration fields called `stringToAppend` and `method`, and the default values `""` and `"NORMAL"` are assigned to it, respectively. All configuration fields that you create appear in the user interface and are configurable by the user. You can always access these values in the script, as we will do soon.

Next, we set a callback called `_data_in`, which is called every time the operator receives data in the inport `inString`. To set the callback, use the method `_set_port_callback` defined by the `BaseOperator` class.

```

def _get_operator_info(self):
    inports = [PortInfo("inString", required=True, type_="string")]
    outports = [PortInfo("outString", required=True, type_="string")]
    return OperatorInfo("Append String",
                        inports=inports,
                        outports=outports,
                        icon="puzzle-piece",
                        tags={"numpy36": "1.16.1"},
                        dollar_type="http://sap.com/vflow/
com.mydomain.appendString.schema.json#")

```

All of the operators that you create in this way must always implement the `_get_operator_info` method. The `_get_operator_info` method is used to generate the operator json automatically, so you must specify

the operator attributes here. To specify the operator attributes, the method must return an `OperatorInfo` object. The `OperatorInfo` object has the following attributes:

```
class OperatorInfo(object):
    """
    Attributes:
        description (str): Human readable name of the operator.
        icon (str): Icon name in font-awesome.
        iconsrc (str): Path to a icon image. Alternative to the option to a icon
from font-awesome.
        inports (list[PortInfo]): List of input ports.
        outports (list[PortInfo]): List of output ports.
        tags (dict[str,str]): Tags for dependencies. dict[library_name,
lib_version].
        component (str): This field will be set automatically.
        config (dict[string,any]): This field will be set automatically.
        dollar_type (str): Url to $type schema.json for this operator.
    """
```

You can specify all of the attributes using the `get_operator_info` method. In the example, we specify an input port called `inString` that receives a string, an output port called `outString` that also receives a string, the built-in icon `puzzle-piece` as the operator's icon, and `Append String` as operator description. The `inString` port name is used in the constructor to set the correct callback to the port.

```
def _init(self):
    metric_infos = []
    metric_infos.append(self._registry.create_metric_info(u'count', 1, u'name',
u'display_name', u'unit'))
    self.metric = self.register_metric(u"int", u"TEST", metric_infos)
    self.register_websocket_route('/socket', 'test', self._set_websocket)
    self.register_static_route('/ui', 'static')
    self.register_rproxy_route('/rproxy/*path', 'http://localhost:' +
str(self.config.port) + '/')

if self.config.method == "NORMAL":
    self.__transform_data = lambda x: x
elif self.config.method == "UPPERCASE":
    self.__transform_data = lambda x: x.upper()
else:
    raise ValueError("Unknown config set in configuration: '%s'." %
self.config.method)
```

The method is overridden by the `BaseOperator` superclass. The method is called after the user-defined configurations have already been set. The method is called for all graph operators before any operator has been started. Here, we are deciding which `transform_data` function to use based on the `self.config.method` parameter set by the user in the interface. In this method, the user can register metrics (`register_metric`) and routes (`register_websocket_route`, `register_static_route` and `register_rproxy_route`). It is only possible to register these functionalities using the following function:

```
def _data_in(self, data):
    self.metric.include_value(1)
    self._send_message('outString', self.__transform_data(data +
self.config.stringToAppend))
```

The function is the callback that we create to handle inputs to the `inString` input port. Here, we append the input data with our `stringToAppend` configuration field, apply the function to it, and use the `BaseOperator_send_message` method to send the result to our output port `outString`. In addition, we set a new value in the

`self.metric` value, because the only `MetricInfo` is of type `count`, we update the counter of processed strings.

```
def _set_websocket(self, handler):
    self.web_handler = handler
    self.web_handler.set_message_callback(self._on_message)
def _on_message(self, message):
    self.web_handler.write_message(str(message))
    self._send_message("outString", message)
```

The method is overridden by the `BaseOperator` superclass. It contains code that is executed before the operator main loop is started:

```
def _prestart(self):
    pass
```

The method is overridden by the `BaseOperator` superclass. It contains code that is executed after the operator main loop is finished:

```
def shutdown(self):
    pass
```

A list of all methods is available at [List of BaseOperator Methods \[page 197\]](#).

Create an Operator

After you implement the `operator.py` file to define the operator, run a bash script to create the operator.

To create an operator, create a folder (or a series of folders) inside the `<ROOT>/subengines/com/sap/python36/operators/` directory, where `<ROOT>` is `my_solution/vrep/vflow` in the example structure. For example, to create an operator with the ID `com.mydomain.util.appendString`, create the folders for the path `<ROOT>/subengines/com/sap/python36/operators/com/mydomain/util/appendString` and place two files inside the last directory: `operator.py` and `operator.json`. If you are targeting the Python 2.7 subengine, you must create empty `__init__.py` files inside every subfolder starting at `operators` onward. To automatically generate the `operator.json` file, run the script `gen_operator_jsons.py`, which is located inside the `pysix_subengine` package.

The following bash script shows how you can use `gen_operator_jsons.py` in your solution:

```
SCRIPT_PATH=<PYSIX_SUBENGINE_PATH>/scripts
SUBENGINE_ROOT=<ROOT>/subengines/com/sap/python36
START_DIR=operators
python $SCRIPT_PATH/gen_operator_jsons.py --subengine-root $SUBENGINE_ROOT --
start-dir $START_DIR "$@"
```

After you create an operator, restart the Modeler instance for the changes to take effect.

Related Information

[Using Python Library \[page 195\]](#)

[Adding Documentation \[page 195\]](#)
[Creating Tests \[page 196\]](#)
[List of BaseOperator Methods \[page 197\]](#)
[List of BaseOperator Attributes \[page 199\]](#)
[List of Metric Methods \[page 200\]](#)
[Logging \[page 200\]](#)
[Uploading to SAP Data Hub System Management \[page 201\]](#)

12.2.2.1 Using Python Library

Create a dockerfile as specified in the *Using Python Library* sub-section in the *Normal Usage* section and add the relevant tags to the `_get_operator_info` method of `BaseOperator`.


Related Information

[Using Python Libraries \[page 188\]](#)

12.2.2.2 Adding Documentation

Create the operator documentation in a file named `README.md` in the operator's folder. For example, to create the documentation for the dummy operator `AppendString`, create a new `README.md` file in the following path:

```
{ROOT}/subengines/com/sap/python36/operators/com/mydomain/util/appendString/  
README.md
```

The documentation is written using Markdown syntax (more information [here](#) ). You can check the result by right-clicking your operator and choosing *Open Documentation*, in the SAP Data Hub Modeler UI.

To add a custom icon for your operator, copy the icon to the operator's folder; for example:

```
{ROOT}/subengines/com/sap/python36/operators/com/mydomain/util/appendString/  
icon.png
```

Then, make sure that your `_get_operator_info` method sets the `icon.png` file (or another file) as the `iconsrc` field, as shown in the following example:

```
def _get_operator_info(self):  
    ...  
    return OperatorInfo(...,  
                        iconsrc="icon.png",  
                        ...)
```

Last, you must regenerate the operator's json.

12.2.2.3 Creating Tests

When you create a new operator extending *BaseOperator*, you can also create unit tests using Python's unit testing framework.

We recommend that you create a folder called `test` in your project root (parallel with the folder "my_solution" in our sample hierarchy) and in it, use the same folder structure that you used when creating the operator. For example, for the operator `appendString`, we create the following folders:

```
test/operators/com/mydomain/util/appendString/
```

Create a file named `test_append_string.py` inside the `appendString` folder:

```
import unittest
from Queue import Queue
from operators.com.mydomain.appendString.operator import AppendString
class TestAppendString(unittest.TestCase):
    def testUpper(self):
        op = AppendString.new(inst_id="1", op_id="anything")
        qin1 = Queue(maxsize=1)
        op.inqs["inString"] = qin1
        qout1 = Queue(maxsize=1)
        op.outqs["outString"] = qout1
        input_config = "config_test"
        op.config.stringToAppend = input_config
        op.config.method = "UPPERCASE"
        op.base_init() # This will execute the _init method and also other
things.
        op.start() # Start the operator's thread.
        try:
            input_str = "input_test|"
            qin1.put(input_str) # Send data in the 'inString' input port.
            ret = qout1.get() # Gets data from the 'outString' output port.
            self.assertEqual((input_str + input_config).upper(), ret) # Verify
that the output is equal to the expected result.
        finally:
            op.stop() # stop the operator no matter what happens. Otherwise the
test may hang forever.
```

To run unit tests for all Python files that start with the prefix "test", create a script such as the following:

```
PYSIX_PATH=<PYSIX_SUBENGINE_PATH> # replace <PYSIX_SUBENGINE_PATH> by the path
to your pysix_subengine folder.
SCRIPT_DIR=$(dirname "$0")
cd $SCRIPT_DIR # changes working dir to the folder where the script is located
(which should be the 'test' folder).
SUBENGINE_ROOT=$(realpath ../my_pysolution/vrep/vflow/subengines/com/sap/
python36) # Gets absolute path of python27 folder.
export PYTHONPATH=$PYTHONPATH:$PYSIX_PATH:$SUBENGINE_ROOT # Append paths to
pythonpath.
# Or call with python2.7 if your operators are written in Python 2.7
python3.6 -m unittest discover -s . -p "test*.py" -v
cd -
```

Place the script in the `test` directory that you created.

To check the official documentation of the Python unit test module, see <https://docs.python.org/2/library/unittest.html> .

12.2.2.4 List of BaseOperator Methods

When subclassing *BaseOperator*, you will need to use a set of methods to set callbacks, send messages, etc. The list below summarizes the *BaseOperator* methods you might need.

`_get_operator_info(self)`

This method should be overridden by a subclass. It should return the *OperatorInfo* of this operator. *OperatorInfo* defines some key features of the operator such as port names and other things.

Returns:

OperatorInfo

`_init(self)`

This method can be overridden by a subclass. This method will be called after the user defined configurations have already been set. This method will be called for all graph's operators before any operator has been started.

`_prestart(self)`

This method can be overridden by a subclass. It should contain code to be executed before the operator's callbacks start being called. The code here should be non-blocking. If it is blocking, the callbacks you have registered will never be called. You are allowed to use the method `self._send_message` in `self._prestart`. However, bare in mind that other operators will only receive this data sent when their `_prestart` have already finished. This is because their callback will only be active after their `_prestart` have finished

`shutdown(self)`

This method can be overridden by a subclass. It should contain code to be executed after the operator's main loop is finished.

`_send_message(self, port, message)`

Puts an item into an output queue.

Args:

- `port (str)`: name of output port
- `message (...)`: item to be put

`_set_port_callback(self, ports, callback)`

This method associate input 'ports' to the 'callback'. The 'callback' will only be called when there are messages available in all ports in 'ports'. If this method is called multiple times for the same group of ports then the old 'callback' will be replaced by a new one. Different ports group cannot overlap.

Args:

- `ports (str|list[str])`: input ports to be associated with the callback. 'ports' can be a list of strings with the name of each port to be associated or a string if you want to associate the callback just to a single port.
- `callback (func[...])`: a callback function with the same number of arguments as elements in 'ports'. Also the arguments will passed to 'callback' in the same order of their corresponding ports in the 'ports' list.

`_remove_port_callback(self, callback)`

Remove the 'callback' function. If it doesn't exist, the method will exit quietly.

Args:

- `callback`: Callback function to be removed.

`_add_periodic_callback(self, callback, milliseconds=0)`

Multiple distinct periodic callbacks can be added. If an already added callback is added again, the period 'milliseconds' will be replaced by the new one. If you want two callbacks with identical behavior to be run simultaneously then you will need to create two different functions with identical body.

Args:

- `callback (func)`: Callback function to be called every 'milliseconds'.
- `milliseconds (int|float)`: Period in milliseconds between calls of 'callback'. When not specified it is assumed that the period is zero.

`_change_period(self, callback, milliseconds)`

Args:

- `callback (func)`: Callback for which the period will be changed. If callback is not present on the registry, nothing will happen.
- `milliseconds (int|float)`: New period in milliseconds.

`_remove_periodic_callback(self, callback)`

Args:

- `callback (func)`: Callback function to be removed.

`register_metric(self, metric_type, name, metric_infos=[])`

Register a new metric and its respective 'MetricInfo' (to create it check next section).

Args:

- `name (str)`: It is the metric name defined in the operator (this should be a string without spaces).
- `metric_type (str)`: It defines the type of the updated value. They can be 'consumer', 'producer', 'float' or 'int'.
Consumer and producer aggregators are 'int' and already have the 'metric_infos' field pre-configured, so these aggregators ignore it.
- `metric_infos (list[MetricInfo])`: It is the list of information to be presented about a specific metric.

Returns:

- `Metric`: a metric instance which allows the operator to update its value.

`create_metric_info(self, metric_type, scale, name, display_name, unit)`

This method is a helper function to create 'MetricInfo' in order to register metrics for the operator.

Args:

- `metric_type (str)`: It is the metric type and it can be: 'value', 'sum', 'max', 'min', 'count', 'avg', 'rate'.
- `scale (int)`: It is an integer factor that multiplies all 'MetricInfo' output values.

- name (str): It is a metric name following the previous 'Name' pattern.
- display_name (str): The name displayed to the user (it should be friendlier).
- unit (str): It is a string with the metric unit. The following units are pre-configured to scale up when necessary: 'Bytes', 'KB', 'MB', 'GB', 'TB', 'PB', 'Bytes/s', 'KB/s', 'MB/s', 'GB/s', 'TB/s' and 'PB/s'. For example, when a value of unit 'Bytes' reaches '1024' it automatically becomes 'KB'.

Returns:

- MetricInfo

register_websocket_route(self, path, target, handler)

Args:

- path (str): operator complete url.
- target (str): target path to possible websocket connection to be used.
- handler (func[websocket_handler]): function that will be called with the websocket handler as argument.

register_static_route(self, path, target)

Registers a new static route for the operator. This method only works as expected when called during the operator initialization.

Args:

- path (str): path for the operator complete url.
- Registers a new websocket route for the operator. This method only workstarget (str): target path to directory whose elements will be served.

register_rproxy_route(self, path, target)

Registers a new rproxy route for the operator. This method only works as expected when called during the operator initialization.

Args:

- path (str): path from the operator complete url.
- target (str): target url.

12.2.2.5 List of BaseOperator Attributes

self_inst_id

Operator instance identifier in the graph.

self_op_id

Operator identifier. For example: com.sap.dataGenerator.

self_repo_root

Path to the modeler root directory.

self_subengine_root

Path to the dub-engine root directory.

self_graph_name

Graph identifier. For example: com.sap.dataGenerator.

self._graph_handle

Unique graph instance identifier.

self._group_id

Group at which the subgraph is being executed. If no graph is specified the default is: default.

12.2.2.6 List of Metric Methods

include_value(self, value)

This method allows the user to update a metric value. Depending on the 'metric_info' of that metric the update will be different.

Args:

- value (int\float): New metric value, the value type should follow 'metric_type'.

12.2.2.7 Logging

There are a set of built-in methods to enable logging in the Python subengine. The table below summarizes the set of methods you can use to log information.

Subclassing BaseOperator	Description
self.logger.info(str)	Log with level <i>INFO</i>
self.logger.debug(str)	Log with level <i>DEBUG</i>
self.logger.error(str)	Log with level <i>ERROR</i>
self.logger.warning(str)	Log with level <i>WARNING</i>
self.logger.fatal(str)	Log with level <i>FATAL</i> (stops the graph)
self.logger.critical(str)	Log with level <i>CRITICAL</i> (stops the graph)

i Note

If you want to report an error and stop the graph from inside the operator code, raise an exception instead of logging with the `logger.fatal("")` or `logger.critical("")` command. Using those will have unintended consequences.

i Note

If you start a new thread inside your operator then you should capture your exceptions inside this thread and use `self._propagate_exception(e)` function so that the main thread can handle it.

You can check all logs using SAP Data Hub Modeler's UI. To do that, you must click the [Trace](#) tab. You can filter the logging messages by level or you can even search for a specific log. You can also download the logging history as a CSV file.

You can also send the logging messages to SAP Data Hub Modeler's external log. To achieve that, open [Trace Publisher Settings](#), next to the search bar. Turn on [Trace Streaming](#) and configure the set of logging messages you want to publish. For example, if you wanted to publish all logging messages which have level *fatal* or *error*, you would have to change [Trace Level](#) to *ERROR*.

12.2.2.8 Uploading to SAP Data Hub System Management

You can either upload your solution to the SAP Data Hub cluster as a new solution layer (need to be logged as an admin) or you can upload your solution into your tenant or user workspace.

Related Information

[Uploading Solution as a Layer in System Management \[page 201\]](#)

[Uploading Solution in System Management to Tenant or User Workspace \[page 202\]](#)

12.2.2.8.1 Uploading Solution as a Layer in System Management

Procedure

1. First you need to compress your solution to a .tar.gz format: `tar -czf my_pysolution.tar.gz -C my_pysolution/ .`
2. Log into SAP Data Hub System Management as Cluster Administrator.
3. Choose the [Tenant](#) button on the horizontal bar at the top of the screen.
4. Click [Layers](#) and then the + icon to create a new layer. Fill in the name field and find the `my_solution.tar.gz` file in your system by clicking on the [Browse](#) button.
5. Add this new layer to an existing strategy or create a new one by clicking the [Strategies](#) button.

If you create a new strategy you will need to associate it with a tenant in the [Tenants](#) tab.

Results

If you log into a user from the tenant that you just associated the strategy containing the new layer, you will be able to see the operators from your solution when you launch a new SAP Data Hub Modeler instance or when you restart an existing one.

12.2.2.8.2 Uploading Solution in System Management to Tenant or User Workspace

Context

In this option you will upload your solution to the workspace of a given tenant or user in the SAP Data Hub cluster. The downside of this option is that it cannot be reversed easily. In the first option you can always remove a layer from a given strategy. On the other hand, in this option if some of the files in your solution overwrote some existing files in the tenant or user workspace, this will not be able to be reversed.

Procedure

1. First you need to compress your solution to a `.tar.gz` format: `tar -czf my_pysolution.tar.gz -C my_pysolution/ .`
2. Log into your user in SAP Data Hub System Management and click the *File* tab at the top of the screen.
3. Make sure no folder is selected in the *File* window.
 - a. For the *current user* - Click on *Import File > Import Solution File* button on the *My Workspace* section.
 - b. For *all users* in your current tenant (only possible for the tenant admin) - Click on *Import File > Import Solution File* button on the *Tenant Workspace* section
4. In the popup window, select your `tar.gz` file and your solution will then be uploaded to the selected workspace.

Results

You will be able to see the operators from your solution when you launch a new SAP Data Hub Modeler instance or restart the application.

12.3 Working with the Node.js Subengine to Create Operators

The SAP Data Hub Node.js subengine let you code your own operators in a particular programming language and make them available from the SAP Data Hub user interface.

The Node.js subengine can run Node.js operators side by side with operators written for different platforms like Go, Python, C++ or others. It can execute individual operators or entire subgraphs. The more Node.js operators are directly interconnected, the bigger the pure Node.js subgraphs are.

The Node.js subengine enables you to:

- Develop operators in modern JavaScript (ECMAScript 6 and beyond).
- Use the Node.js JavaScript runtime built on Chrome V8.
- Use your preferred language that can be compiled into JavaScript; for example, TypeScript.
- Use third-party libraries.

Related Information

[Node.js Operators and OS Processes \[page 203\]](#)

[Use Cases for the Node.js Subengine \[page 204\]](#)

[The Node.js Subengine SDK \[page 205\]](#)

[Data Types \[page 207\]](#)

[Additional Type Specific Constraints \[page 208\]](#)

[Develop a Node.js Operator \[page 209\]](#)

[Project Structure \[page 210\]](#)

[Project Files and Resources \[page 211\]](#)

[Logging \[page 213\]](#)

12.3.1 Node.js Operators and OS Processes

Every Node.js operator runs in its own operating system process. This makes it possible to utilize multi-core CPUs and parallel system architectures from Node.js, which would otherwise not be possible, because Node.js works single-threaded. In addition, this provides a strict isolation between Node.js operators for enhanced security.

The Node.js processes communicate by TCP socket based inter-process communication (IPC), which is coordinated by the Node.js subengine. In turn, the Node.js subengine runs in its own process. There is one coordinating Node.js subengine process per subgraph consisting of pure Node.js operators.

i Note

All multiplexers run directly in the Node.js subengine process. This reduces communication overhead. Please make sure that the configuration of the multiplexers within the setting *Subengine* within a Node.js

subgraph is set to `com.sap.node`. Otherwise the subgraph divides into two Node.js subgraphs with a multiplexer in between running in Go or another subengine.

Example

A subgraph may consist of 10 Node.js operators. 3 of them are multiplexer. The multiplexer are configured to run in subengine 'com.sap.node'. The number of operating system process for this scenario computes as follows:

```
1 (Node.js Subengine) + 10 (Node.js operators) - 3 (Node.js Multiplexer) = 8
operating system processes.
```

i Note

A pipeline can result in multiple Node.js subgraphs, depending on what operators are interconnected.

12.3.2 Use Cases for the Node.js Subengine

There are two main use cases for the Node.js subengine:

- SAP Data Hub Modeler
Using the SAP Data Hub Modeler is the easiest way to quickly change or add functionality of pipelines. You can simply add a Node.js Script operator to existing pipelines or modify the JavaScript code of already used Node.js script operators.

i Note

The only external node-module that can be required inside the SAP Data Hub Modeler is `@sap/vflow-sub-node-sdk`.

- Use a dedicated local Node.js project to develop your Node.js operator
This is the most flexible method. It allows to use your own tools and frameworks and code Node.js operators like any other Node.js program.
Among other advantages, you can:
 - Re-use own JavaScript libraries you already created
 - Use your own required set of third-party node-modules
 - Leverage test driven development
 - Use other languages that compiles to JavaScript, e.g. TypeScript

The Node.js subengine consists of two major building blocks: the Node.js subengine core and the Node.js Subengine SDK. While you probably never work directly with the core engine, you will work with the Node.js Subengine SDK to develop your operators.

12.3.3 The Node.js Subengine SDK

The Node.js subengine consists of two major building blocks: the Node.js subengine core and the Node.js Subengine SDK. While you probably never work directly with the core engine, you will work with the Node.js Subengine SDK all the time to develop your operators.

The Node.js subengine SDK is a Node.js module named '@sap/vflow-sub-node-sdk'. The purpose of it is to simplify the development of Node.js operators. It allows easy access to:

- the in and out ports defined in the operator.json
- static operator configuration defined in the operator.json
- additional ports added directly in the SAP Data Hub Modeler
- dynamic configuration added directly in the SAP Data Hub Modeler
- a system-logging API to send status information about the operator

The SDK also supports a shutdown hook for the operator. This allows to cleanup resources if necessary like closing files or logging out from remote systems before terminating the operator.

Related Information

[Availability of the Node.js Subengine SDK \[page 205\]](#)

[SDK API Reference \[page 205\]](#)

12.3.3.1 Availability of the Node.js Subengine SDK

The SDK is already included into the Node.js subengine. All Node.js operators can simply require it at runtime and use the SDK as follows:

javascript:

```
const { operator } = require("@sap/vflow-sub-node-sdk");
const operator = Operator.getInstance();
```

If you utilize a dedicated local Node.js development project, you probably want to download and install the SDK, because you need to program against its API.

For more information, see [Availability of the Node.js Subengine SDK \[page 205\]](#)

12.3.3.2 SDK API Reference

getInstance(basePath)

- basePath <string> Optional. The path to the operator descriptor (operator.json) directory. Default set to the directory where the operator's script file is located.

- Returns: <Operator>

Creates an operator instance (singleton).

config

- Returns: <object>

The `operator.config` method returns the configuration of this operator. The runtime configuration is returned, if user changes this operator in the Modeler UI. Otherwise returns the design time configuration which is specified in the `operator.json`.

addShutdownHandler(handler)

- handler <Function> The handler to be called before terminating this operator
- Returns: <void>

The `operator.addShutdownHandler()` method adds a shutdown handler when terminating this process. It accepts a callback function that takes only one optional error parameter, for example, taking an `(err) => ...` callback.

The operator fails if the shutdown handler is not a function or is undefined.

Using a callback function:

```
const handler = (cb: (err?: Error) => void): void => {
  // do something ...
  cb(...); // callback with void or an error
};
const operator: Operator = Operator.getInstance();
operator.addShutdownHandler(handler);
```

done()

- Returns: <void>

Terminates this operator process with exit code 0.

fail(message)

- message <string> Error message to be written out
- Returns: <void>

Terminates this operator process with exit code 1. Writes an error message to `process.stderr`.

getInPort(name)

- name <string> The name of the input port
- Returns: <InPort>

Finds a single input port by its name and returns it.

The operator fails if the port was not found.

getOutPort(name)

- name <string> The name of the output port
- Returns: <OutPort>

Finds a single output port by its name and returns it.

The operator fails if the port was not found.

getInPorts()

- Returns: <Map<string, InPort>>

Returns all input ports of this operator. An empty map is returned when no input port found.

getOutPorts()

- Returns: <Map<string, OutPort>>

Returns all output ports of this operator. An empty map is returned when no output port is found.

logger

- Returns: <Logger>

Returns a logger instance which can be used for logging. See [Logging \[page 213\]](#).

12.3.4 Data Types

Inside a Node.js operator you use JavaScript types and data structures. The following table shows how SAP Data Hub Modeler types are mapped to JavaScript types:

SAP Data Hub Modeler	JavaScript
string	String
blob	Buffer
int64	Number
uint64	Number
float64	Number
byte	Number
message	Object
[]x	Array

The letter x in the last row of the table should be replaced with any of the allowed types (except for message), e.g., []string, []uint64, [[]]blob, etc. The conversion is always done by the Node.js subengine. If data reaches the Node.js operator script, it is already converted in respect to the above table.

i Note

Always use the SAP Data Hub Modeler types to specify the types of the in and out ports in the operator.json. This is necessary, for example, if you develop a Vsolution using an external project.

12.3.5 Additional Type Specific Constraints

JavaScript has no integer type, but only a number type, which is internally represented as a IEEE-754 double precision float. Thus, only integer values up to 53 bits can be exactly represented.

To prevent from silent and undetected problems, the Node.js subengine will intentionally stop with a corresponding error message when receiving or sending an unsafe integer value. If an Node.js operator faces such a problem, the corresponding pipeline will also stop.

Unsafe integer value is defined as follows:

Sample Code

```
Number.isSafeInteger(data) === false
```

The safe integers consist of all integers from $-(2^{53} - 1)$ to $2^{53} - 1$ (inclusive). That means, that the full range of SAP Data Hub Modeler types uint64 and int64 cannot be represented in JavaScript/Node.js

```
type uint64 - range: 0 ... 2**64 -1 (0 through 18446744073709551615)
type int64: -(2**63 -1) ... 2**63 -1 (-9223372036854775808 through
9223372036854775807)
```

If converted into number, the mathematical rules of equality can be violated in JavaScript. See the following examples:

```
( 1 === 2 ) === false // <- this always evaluates to false and is mathematically
exact

// Now, add MAX_SAFE_INTEGER to both sides:
( 1 + Number.MAX_SAFE_INTEGER === 2 + Number.MAX_SAFE_INTEGER ) === true // <-
results to true
// Also a legal expression, it is mathematically incorrect
```

For more detailed information about safe integer, see the external link [developer.mozilla.org, datatype \(IEEE-754\)](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Number#safe_integers).

12.3.6 Develop a Node.js Operator

This section explains how to create Node.js operators outside the Modeler application by using a dedicated Node.js project.

Context

Download the SDK from a SAP Datahub System:

Procedure

1. Log in to SAP Data Hub System Management as a cluster administrator
2. Click on *Files* and then choose *Union View*
3. On the union view, navigate to *vflow/subengines/com/sap/node*
4. Right-click on file *vflow-sub-node-sdk.tar.gz* and choose *Export File*
5. Export the file to your local Node.js project. Save it into an appropriate location inside your project.
6. To make the SDK available to your JavaScript code, install it using the Node.js package manager:

```
npm install --save [path_to]/vflow-sub-node-sdk.tar.gz
```

7. Finally check whether the (runtime-) dependency is added to your package.json. It should look similar to this:

```
{
  "name": ...,
  "version": ...,
  "dependencies": {
    "@sap/vflow-sub-node-sdk": "file:vflow-sub-node-sdk.tar.gz",
    ...
  }
}
```

Related Information

[Requiring Node Modules \[page 210\]](#)

[Using the SAP Data Hub Modeler UI \[page 210\]](#)

12.3.6.1 Requiring Node Modules

Because a Node.js operator runs in its own process, it also can require and use its own individual set and version of node modules. This provides great flexibility and allows you to re-use your own JavaScript libraries as well as third-party Node.js modules.

12.3.6.2 Using the SAP Data Hub Modeler UI

In this section, we explain how to customize the behavior of Node.js operators in the SAP Data Hub user interface. This is recommended if you want to create simple scripts in a quick way.

There are two ways to personalize the behavior of a Node.js operator:

- Drag out the *Node.js Script Operator* to the graph canvas and then right-click on it to open the script editor.
- Create a new operator in the *Repository* tab and select *Node.js Base Operator* as the base operator to be extended. The advantage of the second approach is that you can reuse your new operator across many graphs, while in the first approach, the edited script will be specific to the given graph and operator instance.

In more detail, the second approach can be achieved by opening the *Repository* tab and expanding the *Operators* section. You can right-click in any subfolder of *Operators* to create your operator under it. Then click *Create Operator*. A dialog will open and ask for the operator name, display name, and base operator. Fill in the fields and choose *Node.js Base Operator* as the base operator. Then, in the *Operator Editor* view, you will be able to create input ports, output ports, tags, configuration parameters, and write your script.

12.3.7 Project Structure

To deploy your operator later to a Data Hub system, you will need to structure your project as follows:

```
| - my_solution/
  | - vsolution.json
  | - vrep/vflow/
    | - subengines/com/sap/node/operators/
      |-- example
        |-- operator.json
        |-- script.js
        |-- icon.svg
```

This structure needs to be archived before you can deploy it to a SAP Data Intelligence system. You can use the tar archiver (on macOS and Linux) to create an archive:

```
tar -C <path to my_solution> -czf my_solution.tar.gz
```

i Note

Double check the structure of your archive. Be sure that the folder 'vrep' is at the top level of the archive, beside the file 'vsolution.json'.

12.3.8 Project Files and Resources

vsolution.json

The vsolution.json describes the solution and looks as follows:

```
{
  "name": "vsolution_vflow_my_solution",
  "version": "n.n.n",
  "description": "my Solution",
  "license": "my license"
}
```

i Note

Be sure to increment the version number after every change. Otherwise the SAP Data Hub system will not overwrite a deployed version.

operator.json

The operator.json specifies the base operator (component), the interface (inports, outports), configuration data, and other details of the operator. See the following example:

☰ Sample Code

```
{
  "component": "com.sap.node.operator", // extend the basic node operator
  "description": "Operator description", // visible title of the operator in
  operators explorer in SAP Data Hub Modeler
  "iconsrc": "icon.svg",
  "inports": [
    {
      "name": "in1",
      "type": "string"
    }
  ],
  "outports": [
    {
      "name": "out1",
      "type": "string"
    }
  ],
  "config": {
    "codelanguage": "javascript",
    "script": "file://script.js"
  }
}
```

Inports and outports can be empty or left out, if the operator does not have any ports. However, port names in either ports must be unique, if there are more than one ports. Moreover, port names should not contain special characters such as white spaces.

The script config must be specified if the script file is not placed in the same root path as the operator.json or is named differently than the operator's. Otherwise it can be left out.

Operator Script

The file script.js is the entry point of your operator. The following example has been taken from the operator 'Node.js Counter' also available in the SAP Data Hub Modeler:

Sample Code

```
const SDK = require("@sap/vflow-sub-node-sdk");
const operator = SDK.Operator.getInstance();
let counter = 0;
/**
 * This operator receives messages on port "in1",
 * increases a counter and forwards the counter value
 * to port "out1".
 */
operator.getInPort("in1").onMessage(msg => {
  // The content of the actual message is ignored.
  // We will only count the number of messages here.
  counter++;
  operator.getOutPort("out1").send(counter.toString());
});
/**
 * A keep alive hook for the node process.
 * @param tick length of a heart beat of the operator
 */
function keepAlive(tick) {
  setTimeout(() => {
    keepAlive(tick);
  }, tick);
}
// keep the operator alive in 1sec ticks
keepAlive(1000);
```

Whenever this operator receives a message at input port 'in1' it increments a counter and sends the current counter value to output port 'out1'. To prevent the Node.js process from terminating immediately after the start, a 'keepAlive' timer has been added. It resets a time-out every 1000 milliseconds.

Operator Documentation

The operator's documentation can be created in a file named README.md in the operator's folder. To create the documentation for the operator *Counter*, for example, you have to create a new README.md file in the following path:

```
my_solution/subengines/com/sap/node/operators/com/mydomain/util/counter/README.md
```

The documentation is written using Markdown syntax (more information see [Mastering Markdown](#)).

Operator Icon

You can add a custom icon (file format: svg, .jpg or .png) for your operator. To do this, copy the icon to the operator's folder, for example:

```
my_solution/subengines/com/sap/node/operators/com/mydomain/util/counter/icon.svg
```

12.3.9 Logging

This library provides different logging APIs that propagate logging info to the Node.js subengine. In local development (not running with Node.js subengine core), the console is used for logging.

Logging Levels

Logging levels use the npm severity ordering: ascending from most important to least important.

```
{
  ERROR: 0,
  WARN: 1,
  INFO: 2,
  DEBUG: 3,
}
```

Setting logging levels

By default, the logging level is set to *INFO*. This can be changed as below:

```
const operator: Operator = Operator.getInstance();
operator.logger.logLevel = "ERROR"; // case insensitive
```

Usage of the logging API

The logging APIs are directly accessible from the operator instance.

```
const operator: Operator = Operator.getInstance();
// if not running in subengine, log is outputted to console
operator.logger.info("message", ...); // log level INFO
// DEBUG level
operator.logger.debug("message", ...);
// WARNING level
operator.logger.warn("message", ...);
// ERROR level
operator.logger.error("message", ...);
```

You can also create an own logger instance in your operators.

```
import { Logger } from "@sap/vflow-sub-node-sdk";
const logger: Logger = new Logger("LOG_LEVEL");
logger.info("message");
```

Message formatting

Log message can take zero or more placeholder tokens. Each placeholder is replaced with the converted value from the corresponding argument. The log message is then a print-f like format string. For supported placeholders, please see nodejs.org.

```
operator.logger.debug("debug message %j", { foo: "bar" });
// 2018-11-22T13:03:01.088Z - debug: Operator "sampleiotdevicegeneratingdata1"
(pid: 11912)|debug message { "foo": "bar" }
```

12.4 Working with Flowagent Subengine to Connect to Databases

Partitioning the source data allows us to load data in chunks there by overcome memory pressure and by doing it in parallel we can load data faster and improve overall performance.

Partitioning

Below are supported partitioning methods with the [Connectivity \(via Flowagent\)](#) operators.

Logical Partition

Logical Partitions are user-defined partitions on how to read data from source. For Table Consumer, the user can choose the partition type then add the partition specification as described below.

- `defaultPartition`: The partition which will fetch all rows which were not filtered by the other partitions (you can disable it by setting `defaultPartition` to false);
- `conditions`: Each condition represents a set of filters which will be performed on a certain column;
 - `columnExpression`: The column where the filters will be applied. It can be either the column name or a function on it (i.e., `TRUNC(EMPLOYEE_NAME)`);
 - `type`: The filter type. It can be either "LIST" (to filter exact values) or "RANGE" (to filter a range of values);
 - `dataType`: The data type of the elements. It can be either "STRING", "NUMBER" or "EXPRESSION";
 - `elements`: Each element represents a different filter, and its semantics depends on the "type" (see above).

- When more than one condition is presented, it will be performed a cartesian product among the elements of each condition, i.e.:
 - Condition A: [C1, C2]
 - Condition B: [C3, C4]
 - Resulting filters: [(C1, C3), (C1, C4), (C2, C3), (C2, C4)]

Ex: Let's assume we have a source table called EMPLOYEE which has the following schema:

```
CREATE TABLE EMPLOYEE (
  EMPLOYEE_ID NUMBER(5) PRIMARY KEY,
  EMPLOYEE_NAME VARCHAR(32),
  EMPLOYEE_ADMISSION_DATE DATE
);
```

RANGE

A JSON representing a RANGE partition is shown below, where the elements represent ranges, thus, they need to be sorted in ascending order.

Numeric Partitions

```
{
  "defaultPartition": true,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_ID",
      "type": "RANGE",
      "dataType": "NUMBER",
      "elements": [
        "10",
        "20",
        "30"
      ]
    }
  ]
}
```

String Partitions

```
{
  "defaultPartition": true,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_NAME",
      "type": "RANGE",
      "dataType": "STRING",
      "elements": [
        "M",
        "T"
      ]
    }
  ]
}
```

Expression Partitions

```
{
  "defaultPartition": true,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_ADMISSION_DATE",
      "type": "RANGE",

```

```

        "dataType": "EXPRESSION",
        "elements": [
            "TO_DATE('2012-01-01', 'YYYY-MM-DD')",
            "TO_DATE('2015-01-01', 'YYYY-MM-DD')"
        ]
    }
]
}

```

LIST

List partitions can be used to filter exact values. Each element represents a different filter.

Numeric partitions

```

{
  "defaultPartition": true,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_ID",
      "type": "LIST",
      "dataType": "NUMBER",
      "elements": [
        "10",
        "20",
        "50"
      ]
    }
  ]
}

```

String partitions

```

{
  "defaultPartition": false,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_NAME",
      "type": "LIST",
      "dataType": "STRING",
      "elements": [
        "Jhon",
        "Ana",
        "Beatrice"
      ]
    }
  ]
}

```

Expression partitions

```

{
  "defaultPartition": false,
  "conditions": [
    {
      "columnExpression": "TRUNC(EMPLOYEE_ADMISSION_DATE)",
      "type": "LIST",
      "dataType": "EXPRESSION",
      "elements": [
        "TO_DATE('2012-07-17', 'YYYY-MM-DD')"
      ]
    }
  ]
}

```


COMBINED

```
{
  "defaultPartition": true,
  "conditions": [
    {
      "columnExpression": "EMPLOYEE_NAME",
      "type": "LIST",
      "dataType": "NUMBER",
      "elements": [
        "Beatrice",
        "Ana"
      ]
    },
    {
      "columnExpression": "EMPLOYEE_ADMISSION_DATE",
      "type": "RANGE",
      "dataType": "EXPRESSION",
      "elements": [
        "TO_DATE('2015-01-01', 'YYYY-MM-DD')",
        "TO_DATE('2016-01-01', 'YYYY-MM-DD')"
      ]
    }
  ]
}
```

Note

- Operator auto-generates the default partition, so the user does not need to define it, but it can be disabled by setting the property "defaultPartition": false.
- For range partition the default partition is greater than last element. So ensure that the elements are ordered in ascending order.

Physical Partition

Physical partitions are partitions which are defined directly on the source, using Oracle partitioning concept.

Limitations:

- Hash partitioning is not supported.
- Sub-partitioning is not supported.

Row ID Partition

Row ID partitions are partitions which are generated automatically based on the row id of the columns. The user must supply the number of partitions, and the range of row id partitions will be generated automatically based on it.

13 Create Dockerfiles

Create Dockerfiles that contains all commands, which you may call on the command line to assemble a docker image.

Context

In the Modeler application, you can create a library of Dockerfiles. These Dockerfiles provide a predefined runtime environment to execute the operators in a graph.

Procedure

1. Start the SAP Data Hub Modeler.
2. In the navigation pane, choose the *Repository* tab.
The tool displays all graphs, operators, and Dockerfiles available in your repository. These are grouped under the tabs, *Graphs*, *Operators*, *Docker Files* respectively.
3. Create a folder.
 - a. Right-click the *Docker Files* section and choose *Create Folder*.
The application creates a new folder within which you can create Dockerfiles.
 - b. Provide a name for the root folder and choose *OK*.

i Note



If you want to create a folder structure (subdirectories) within the root folder, right-click the root folder and choose *Create Folder*.

4. Create a Dockerfile.
 - a. Right-click the folder in which you want to create the operator and choose the *Create Docker File* menu option.
 - b. In the *Name* text field, provide a name for the new Dockerfile.
 - c. Choose *OK*.
 - d. In the editor, write the script that defines your Dockerfile.


i Note

For security reasons, we recommend you to start the image with the non-root user.

5. Create tags.
Tags describe the runtime requirements of operators and are the annotations of Dockerfiles that you create.

- a. In the editor toolbar, choose  (Show Configuration) to open the configuration pane.
 - b. In the right-pane, choose + (Add Tag) to add one or more tags.
 - c. In the dropdown list, select a tag.
 - d. Provide a version.
6. In the editor toolbar, choose  (Save) to save your changes.
 7. Build a docker image.

A docker image is a stand-alone, executable package that includes all the code, a runtime, libraries, environment variables, and config files necessary to execute the operators in a graph.

 - a. In the editor toolbar, choose  (Build) to build a docker image for the Dockerfile.

Next Steps

In future releases, for security reasons, the Modeler application will stop supporting images that run with a root user.

The image descriptions provided by SAP are being modified to create and use non-root users. If your image doesn't inherit from one of those users, you must add the creation of a user and a USER directive in your Dockerfile. For example:

Sample Code

```
RUN groupadd -g 1972 vflow && useradd -g 1972 -u 1972 -m vflow
USER 1972:1972
```

Note

It is mandatory to use numeric IDs in the USER directive instead of using and group names. The graph execution will fail otherwise.

Ensure the commands that you execute after the USER directive work with the new environment:

- Installing new Python packages with `pip` has to have the `--user` flag for local installation.
- For software that are not installed via package managers and for those that do support user local installation, you must manually install them inside the user's home directory. For example in, `/home/vflow`. Do the same for any other files added to the image.

Related Information

[Docker Inheritance \[page 220\]](#)

13.1 Docker Inheritance

It is possible to inherit from other dockerfiles defined on the modeler by using the "FROM \$<dockerfile_id>" syntax.

For example, if you want to create a new dockerfile to use the numpy library in a [Python3Operator](#), you can inherit from the existing `com.sap.sles.base` docker image, which already satisfies the requirements of the [Python3Operator](#).

The content of the dockerfile could look like this:

```
FROM $com.sap.sles.base
RUN pip3.6 install --user numpy
```

Add the following tags to the new dockerfile:

```
{ "numpy36": "" }
```

At runtime, the new dockerfile tags are automatically expanded to `{"sles": "", "python36": "", "tornado": "5.0.2", "numpy36": ""}` due to the automatic tag inheritance mechanism (the parent dockerfile `com.sap.sles.base` contains the tags `{"sles": "", "python36": "", "tornado": "5.0.2"}`).

You must also add the new `numpy36` tag to your custom [Python3 Operator](#) or to the configuration of the group that contains it.

i Note

If you install a new package with zypper, it inherits from the **com.sap.opensuse.golang.zypper** dockerfile. The **com.sap.opensuse.golang.zypper** dockerfile contains the following packages: `{"opensuse": "", "python36": "", "tornado": "5.0.2", "sapgolang": "1.12.1-bin", "zypper": ""}`. Alternatively, you can extend from a publicly available online docker image. In this case, you don't need to use the \$ sign after the FROM clause (for example, `FROM opensuse/leap`). When you extend from a public image, you must create all tags from scratch, because no tags can be inherited from them.

14 Create Types

Types are JSON files that enable you to define properties and bind them with data types. You can also associate the properties with certain validations, define its UI behavior, and more.

Context

Types are based on JSON schema. The Modeler application provides a form-based UI to create types. After creating a type, you can reuse them, for example, in the type definition to define the operator configurations or of specific parameters within the operator configuration definition.

The application also provides global types, which are associated with the configuration parameters of certain default operators (base operators) available within the application. If you want to access the global types, in the navigation pane, choose the *Types* tab.

Procedure

1. Start the SAP Data Hub Modeler.
 2. In the navigation pane, choose the *Types* tab.
 3. In the navigation pane toolbar, choose + (Create Type).
The application opens an empty type editor in a separate tab, where you can define the type.
 4. In the *Description* text field, provide a description for the type.
 5. Create a property.
 - a. In the *Properties* pane, choose + (Add property).
Types can have more than one property.
 - b. In the *Name* text field, provide a name for the type property.
 - c. In the *Title* text field, provide a name for the property.
The application uses the value in the *Title* as the display name for the property in the UI.
- i Note**

If you do not provide a *Title*, then the application uses the value in the *Name* text field as the display name.
- d. In the *Description* text field, provide a description for the property.
6. Define the property.

You can define the data type, UI behavior, and UI validations for the property.

 - a. In the *Data Type* dropdown list, select the required data type value.

Value	Description
String	<p>For properties of data type string, you can define helpers. These helpers enable you to identify the property type and define values accordingly. In the <i>Validation</i> dropdown list, select a value. The application format, predefined values, and services as helpers.</p> <ul style="list-style-type: none"> In the <i>Format</i> dropdown list, select the required format. The application supports the formats, date and time, URL, password, or e-mail for properties of data type string. In the <i>Value Help</i> dropdown list, select a value. <p>Predefined Values: You can preconfigure the property with a list of values that users can choose from the UI. The application displays the property in the UI as a dropdown list of values. In the <i>Values</i> text field, provide a list of values.</p> <p>Service: You can specify a URL to obtain the property values from the REST API. The application displays the response from the service call as auto-suggestions to users. In the <i>Url</i> text field, specify the service URL. The response from the REST API can be an array of strings or an array of objects. If the response is an array of objects, in the <i>Value Path</i> field, provide the name of an object property. The application renders the values of this object property in a dropdown list in the UI to define the operator configuration.</p> <div style="border: 1px solid orange; padding: 5px; margin-top: 10px;"> <p>! Restriction</p> <p>The URL should be of the same origin. Cross-origin requests are not supported.</p> </div>
Object	For properties of data type object, the application lets specify the schema of the object by drilling down into the object definition. In the <i>Properties</i> section, double-click the property to drilldown further and define the object schema.
Array	<p>For properties of data type array, you can specify the data types of items in the array. In the <i>Item Type</i> dropdown list, select a value. The application supports string, number, object, and custom as data types for array items.</p> <p>If the <i>Item Type</i> is <i>Object</i>, in the <i>Properties</i> section, double-click the property to drilldown further and define the object schema.</p> <p>If the <i>Item Type</i> is <i>Custom</i>, select and reuse any of the predefined types for the property definition.</p>
Number	For properties of data type number, you can provide numbered values to the property.
Boolean	For properties of data type Boolean, you can provide Boolean values to the property.
Integer	For properties data type integer, you can provide integer values to the property.
Custom Type	Custom data types enable you to set the data type of a property to another user-defined type. In the <i>Type</i> dropdown list, select a value. The application populates the dropdown list with the global schema types.

- b. If you want to configure the property to mandatorily accept values, enable the *Required* toggle button. If enabled, users must mandatorily provide values to the property in the UI. The property value cannot be empty.
- c. If you want to configure the property as read-only, enable the *ReadOnly* toggle button.

If enabled, the application does not allow any edits to the property from the UI.


7. (Optional) Control property visibility.

You can choose to configure a property as *Visible* or *Hidden* in the UI. Additionally, you can also define one or more conditions to control the visibility based on the values of other properties.

- a. If you want to display the property in the UI based on certain conditions, select *Conditional*.
- b. Define the required property and value pairs.
- c. If you want to define another condition, choose + (Add dependency).


The application performs AND operation of all conditions to determine whether the property should be Visible or Hidden in the UI.

8. Save type.

- a. After creating a type, in the editor toolbar, choose  (Save) to save the type definition.
- b. Choose the *Save* menu option.
- c. Provide a name along with the fully qualified path to the type.

For example, `com.sap.others.<typename>`.

- d. Choose *OK*.

The types are stored in a folder structure within the Modeler repository. For example, `com.sap.others.typeName`. If you want to save another instance of the type, in the editor toolbar, choose  (Save As). Provide a name along with the fully qualified path to the type.

Related Information

[Creating Operators \[page 78\]](#)

15 Creating Scenes for Graphs

SAP Data Hub Modeler provides graphical capabilities that help users to create Scenes - visualizations for graphs.

In this guide, you will find information to:

- [Understand Key Concepts \[page 224\]](#)
- [Use the Scene Editor \[page 229\]](#)
- [Develop Controls \[page 233\]](#)
- [Write Scripts \[page 236\]](#)

If you're looking for a list of the available *Controls*, refer to the [Repository Objects Reference for SAP Data Hub](#).

Related Information

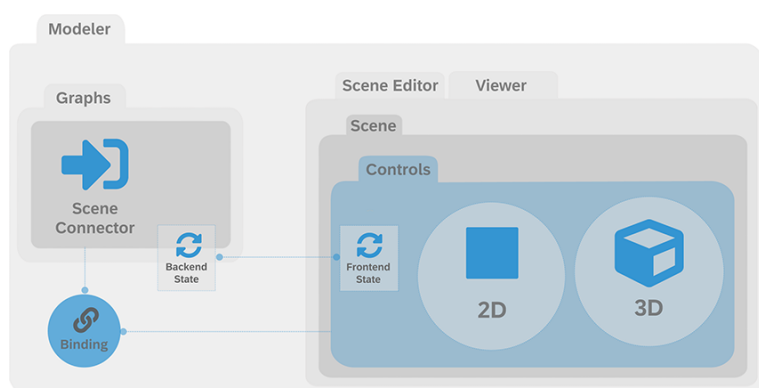
[Understand Key Concepts \[page 224\]](#)

[Use the Scene Editor \[page 229\]](#)

[Develop Controls \[page 233\]](#)

15.1 Understand Key Concepts

You can design and create your Scenes through the Scene Editor. This guide will show you a few important concepts that will enable you to use the Scene Editor.



Scenes

Scenes are 2D or 3D visualizations you create for a graph using the *Scene Editor*.

A 2D scene is the equivalent of a pre-configured HTML page where you add HTML elements.

A **3D Scene** is the equivalent of the canvas where the **3D elements** are modeled. When you add a 3D element to the scene, the application creates a **3D environment** (a tile) for the element in the canvas. Within this environment, you can add only 3D elements.

Scene Editor

The *Scene Editor* is a canvas within the *Modeler* where you design and create *Scenes*.

To learn more, go to [Use the Scene Editor \[page 229\]](#) and see instructions for elements in the Editor.

Controls

SAP Data Hub Modeler provides built-in elements within the *Scene Editor* that you can drag-and-drop to the canvas to create a *Scene* for the graph. These elements are referred to as *Controls*.

A **Control** can be anything from a simple `<input>` tag in an HTML **context** to a very complex **3D animated object** with effects. In the *Controls* tab, you can access them to view, edit, or drag and drop them to the *Scene Editor*.

Control Editor

The *Control Editor* is a tool available within SAP Data Hub Modeler to create and edit *Controls*.

See [Use the Scene Editor \[page 229\]](#) to learn more.


Viewer

The Modeler provides both an embedded viewer and a standalone viewer. The viewer allows you to preview your interface receiving and sending data in real time when designing the scene.

i Note

Opening the viewer starts your graph if it's not running.

Embedded Viewer: In the scene editor toolbar, choose  (Viewer) to open the running scene on top of the scene editor.

Standalone Viewer: In the scene editor toolbar, choose  to open a new page with only your interface. You can use it to test the interface in different browsers or in VR headsets. You can also share the link to allow others to access your interface. You can also access the standalone viewer from the [Status](#) pane.

i Note

If you change the [Scene](#), you don't have to stop and start a graph to see those changes. You can reopen the [Viewer](#) or refresh the standalone page.

Scene Status Panel

[Scene Status](#) is a panel at the bottom that displays the status of the current [Scene](#) or [Control](#) you are working on.

At the top of the panel, you should see the name of the [Scene](#) and the [Graph](#) or the name of the [Control](#) you are editing at the moment.

Information on the [Scene](#) is shown in a table with columns [type](#) and [message](#).

i Note

If you are not editing any [Scene](#) or [Control](#), a message will be displayed warning you so.

Scene Connector Operator


The Scene Connector operator in the Modeler helps establish a connection between the running graph and the scene. The controls in the scene know how to receive and send data to this operator.

See [Scene Connector \(Beta\)](#) to learn more.

States

[State](#) is the consolidation of data by the [Modeler](#) when you execute a graph and it flows through the operators in it. It exists in two places: in the [Scene Connector](#) operator and in the [Controls](#). The [States](#) help, for example, if the [Scene](#) is opened multiple times for a single graph. In such cases, the data that the [State](#) holds are shared between all open instances. If you modify the [Scene](#), then the modifications are reflected in all other open interfaces.

Binding

You can add a *Binding* to link a *Control* property and the *State*. For the selected control, in its *Configuration* panel, click the  icon to create a *Binding*. The *Control* receives the state changes through the binding and, if needed, can adapt the data to show the visualization that the user requires.

Property Types

When creating properties and bindings, you need to deal with types, just like you do when creating operators and ports in the Modeler.

Since Scenes work with JavaScript, you will find the same types that JavaScript works with when editing the properties of a Control:

- String
- Number
- Boolean
- Object
- Array

You can also use Custom Types. For example, one default Custom Type that is provided is `com.sap.scenes.color`, to set a color value in 3D elements. Internally, this type is an object with properties `r`, `g`, and `b`, for red, green and blue values of the color.

i Note

When creating a Binding, the system will only allow you to do so between properties of matching types. In case you are binding two Controls, the types are straightforward, but between a Control and a Scene Connector, there is an important difference, as the types are not perfect matches.

For Control types string and number, you should use types `string` and `float64` in operators.

For all other types, you should use `string.<type>`. For example, `boolean` would become a port of type `string.boolean` in the Scene Connector. That is because the flow does not recognize the type `boolean`, so the value has to be encoded as a string.

Table 15:

Scene Connector	Selected Control
Port Name = <code>input</code>	Property Name = <code>input</code>
Type = <code>string.boolean</code>	Type = <code>boolean</code>

It should follow the rules below:

Table 16:

Scene Connector	Selected Control
<code>string</code>	<code>string</code>

Scene Connector	Selected Control
float64	number
string.<OTHER_TYPE>	<OTHER_TYPE>

Assets

In the *Configuration* pane, choose the *Resources* tab to manage scene assets and scripts.



Assets are elements that you can upload and use in the Scene or Control, including imported scripts, images, videos, 3D models, CSS sheets, JSON files, or any other document relevant to your scenario.

Assets that you upload are available for the entire Scene, so adding an asset to a scene makes it available for every control. However, you can also add assets for a specific control by using the control editor, and it will be available in a scene every time that control is used there.

Continue to [Use the Scene Editor \[page 229\]](#) to see instructions on how to use the Scene Editor.

15.2 Use the Scene Editor

The Scene Editor in the SAP Data Hub Modeler helps you create Scenes for graphs. It provides a canvas where you can design visualizations and offers other graphical capabilities.

I want to	How
Open the <i>Scene Editor</i>	<p>The <i>Scene Editor</i> is available within the SAP Data Hub Modeler application,</p> <ol style="list-style-type: none">1. Start the SAP Data Hub Modeler.2. In the navigation pane, choose the <i>Graphs</i> tab.3. Select and open the required graph for which you want to create the <i>Scene</i> in a graph editor.4. In the editor toolbar, choose  (Scenes) and select the <i>Create Scene</i> menu option. If you have already created a scene, click the arrow next to Scenes  and select which you'd like to open.5. In the <i>Create Scene</i> dialog box, provide a name and description for the scene, and select the scene type (2D or 3D). <div data-bbox="807 1016 1401 1167"><p>i Note</p><p>This action only selects the default root container of the Scene, but you can swap that container out if you wish.</p></div> <ol style="list-style-type: none">6. Choose <i>OK</i>. <p>Result: The scene editor opens in a new tab in the Modeler with an empty canvas.</p>
Create a <i>Scene</i>	<p>You can drag and drop <i>Controls</i> to the empty canvas and design a Scene for the selected graph.</p> <ol style="list-style-type: none">1. In the navigation pane, choose the <i>Controls</i> tab.2. Drag and Drop the required controls from the <i>Controls</i> tab to the canvas and design the interface.

I want to

How

Use the *Control Editor*

The *Control Editor* is a tool available within SAP Data Hub Modeler to create and edit *Controls*.

To create a new *Control*:

1. In the navigation pane, choose the *Controls tab*;
2. In the upper toolbar of the navigation pane, choose the + button. A dialog will appear;
3. Fill in the details for the new *Control* and choose OK;
4. A new tab will appear showing the *Control*;

To edit an existing *Control*:

1. In the navigation pane, choose the *Controls tab*;
2. Using the search box or the pane itself, find the *Control* you want to edit;
3. Right-click on the icon of the *Control* and choose *Edit*;
4. A new tab will appear showing the *Control Editor* and the existing properties and details of the selected *Control*.

From the *Control Editor*, you can apply the concepts listed under [Develop Controls \[page 233\]](#) and [Write Scripts \[page 236\]](#) to build your *Control*.

The configuration panel on the right is also available for you to:

- Edit the properties in the *Properties tab*;
 - Add scripts and other resources under the *Resources tab*;
 - Choose default settings to be imported to a *Scene Connector* in the *Scene Connector Settings* tab;
 - Play around with your *Control* and see how it interacts with other *Controls* by dragging and dropping them to the *Scene* and moving them around in the *Navigator tab*.
-

I want to

How

Create and manage camera positions

Camera positions allow you to save specific moments of the camera in a 3D environment. You can use these saved positions to animate between them and reposition the 3D environment to very specific places.

To add a new camera position:

1. In the canvas, click on a `Three.js Container` control to open its *Configuration* pane.
2. Under the *Bindable Properties* section, turn *Editing* on.
3. Scroll down on the *Configuration* pane until you see *Camera Positions*.
4. Click the **+** icon. The tool will save the values of the current camera in your `Three.js Container`.
5. Save your scene

To rename a Camera Position:

1. In the canvas, click on a `Three.js Container` control to open its *Configuration* pane.
2. Under the *Bindable Properties* section, turn *Editing* on.
3. Scroll down on the *Configuration* pane until you see *Camera Positions*.
4. Click the name of an existing *Camera Position*
5. Type the new name of the position and press or click elsewhere
6. Save your scene

To delete a Camera Position:

1. In the canvas, click on a `Three.js Container` control to open its *Configuration* pane.
2. Under the *Bindable Properties* section, turn *Editing* on.
3. Scroll down on the *Configuration* pane until you see *Camera Positions*.
4. Click the *trashcan* icon next to an existing *Camera Position*
5. Save your scene

Provide values to configuration properties

The controls, like the graphs and operators, are associated with various configuration properties.

1. In the canvas, click the required control to open its *Configuration* pane.
The tool displays the predefined configuration properties available for the control.
2. Provide values to the properties.


I want to

How

Create bindings

Bindings help link the control property with the State of the Scene Creator operator or with other Controls. In the *Configuration* pane, under the *Bindable Properties* section, the application displays all properties that you can bind with the State.


To create a binding for a property:

1. Choose .
2. You will see a dialog. In the *Source* dropdown menu, choose the source for your binding.
3. In the *Property* dropdown menu, choose the property for your binding.
4. Click *OK* to confirm your settings.

i Note


You will see an *Advanced* checkbox in the dialog. Select it if you wish to bind an internal property of a complex object.


Auto-binding is a feature that tries to match property names and types to create several bindings at once. To bind the properties automatically with the State:

1. Choose  (Auto bind) under the *Bindable Properties* section.
2. You will see a dialog with the *Source* dropdown menu. Choose the source for your bindings.
3. Once you click *OK*, the system shows you one of the messages to inform:
 - **Success:** Matches were found and bindings were created.
 - **Failure:** No matches were found and no bindings were created. In this case, create the bindings manually.

Create and edit control properties

To edit the property type or definition:

1. Choose  (Edit custom properties) under *Properties*.
2. You will see the *Type Editor*, where you can edit and create custom properties.

I want to	How
View the Scene outline	<p>In the <i>Configuration</i> pane, choose the <i>Navigation</i> tab to view a tree hierarchy representation of all controls in the Scene. This representation also provides information on the relationships between the controls in the scene (parent, child, or sibling).</p> <p>In the <i>Navigation</i> tab, you can also navigate to the control and edit its definition by clicking the required control in the tree hierarchy. Additionally, if you have saved the changes, you can export the Scene and as an application in the <i>Launchpad</i>.</p> <ol style="list-style-type: none"> 1. Choose <i>Export Scene</i>. 2. Provide a display name that the application must use for the scene in the <i>Launchpad</i>. 3. If you want to view the JSON definition for the scene, choose <i>Get JSON</i>. 4. Select whether you want to export it is to the tenant workspace or to the user workspace. 5. Choose <i>Export to Launchpad</i>.
Save Changes	<p>To save the Scene that you designed, in the scene editor toolbar, choose  (Save).</p>
Preview the <i>Scene</i>	<p>After designing the Scene, you can preview it:</p> <ol style="list-style-type: none"> 1. In the editor toolbar, choose  . This operation does not start the graph. It only opens the preview editor that provides a static preview of the Scene. 2. To stop the preview, choose  .
Run a Scene using the Viewer	<p>The <i>Viewer</i> helps you visualize your Scene.</p> <ol style="list-style-type: none"> 1. If you want to preview the Scene on top of the running Scene, choose  . 2. In the <i>Scene Editor toolbar</i>, choose  to open the stand-alone viewer in a new page with only your Scene.

15.3 Develop Controls

Developing Controls is nothing different from normal front-end development with a modern framework.

It has its limitations, but we are always improving the experience and adding new features. Our goal is to have, in the future, a tool that exempts having to understand how to develop a control, and we hope each iteration of our tool will be more robust and complete than the previous.

Templates

Every `Control` has an `HTML` snippet that serves as basis for how the interface element will behave on the screen. This snippet is called a *Template*, and it can not only contain standard `HTML`. Advanced templating techniques are available through the template engine.

The current engine for **Scene** is a framework called `Aurelia`, which provides functions that enable **Scene** to interpret the templates and works under changes in the element during design time.

To learn more about `Aurelia` templates, you can read the official documentation at <https://aurelia.io/docs/templating/>.

i Note

Please, be aware that, while we use `Aurelia`, this framework is subject to change for next releases. If you built your control with the current version, follow the upcoming upgrade guides to understand how to adapt your controls and scenes to a new version.

Scripts

`Scripts` define settings of a control, from how it initializes to how it handles data updates that come from the bindings. `Scripts` are written in `JavaScript`, following the `ES5` syntax for compatibility with Internet Explorer 11.

For more information about how scripts work and which function hooks are available, go to [Write Scripts \[page 236\]](#).

Custom Properties

When you first create a `Control`, it will only have `Visible` as a default property, allowing you to hide or show the `Control` with a switch. That is also available in its `State`, which enables you to change it dynamically using bindings.

If you want to further add properties to your control and its state:

1. Go to the *Bindable Properties* collapsible;
2. Press the *Edit* button on the right;
3. Reach the *Type Editor*;
4. Click the **+** button
 1. There, you can choose a name, a **title**, a **type** and a **Scene UI Control**. This property allows you to choose how the property will be represented in the *Configuration Panel*. For example, if the type of your new property is `string`, but rather than a free text field you want that to be the `URL` to an asset, you can pick the **Asset Chooser**, and you will see a helper in the *Configuration Panel* to choose an asset from.
5. Save your new type and the new properties should be reflected in the *Configuration Panel*

This customization is available when creating a `Control` and after adding one to a `Scene`, which means you can create default properties for a `Control` and extend them once they are part of a `Scene`.

CSS


You can use the `style` attribute on the `HTML` template, a `<style>` tag, or you can define your own `CSS` file with the rules needed and import it as an asset for the control and load it in the `init()` of the element.

i Note

At the moment, `Controls` are not independent from the *Modeler UI*, which means that if your `CSS` file changes a basic rule (such as adding a font-size in the body element), it will change how the *Modeler* looks and can potentially make the tool unusable. If that happens, review your rules and make sure they target your specific elements on the scene.

3D Controls

For 3D Controls or Scenes, we support `three.js`, which is one of the standard libraries for using `WebGL` in modern browsers.

You can read more about `three.js` at <https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene> .

i Note

For the current version, we used `.` Always check the upgrade guide when changing your version, to make sure your controls and scenes are ready for the new version.

Related Information

[Write Scripts \[page 236\]](#)

15.3.1 Write Scripts

Important Concepts

Exports Object vs Control Class

In the previous implementation of scripts, you had to use a few function hooks defined by our API.

For instance:

- If you wanted to execute any logic in control construction, you had to add the following to your script:
`exports.init = function(){<some logic>}`
This way, the Scene Editor core would call this function.
- The Scene Editor core would always parse the script every time a scene ran and extract the function defined in the `exports` object to call it respectively in the *Control* lifecycle.
- Property formats had to be exported with multiple functions (`init`, `afterSetState`, `destroy`, ...).

In the current implementation, there is a single property that can be exported (`Control`), which should be done through the `exports` object.

This property must be a constructor function - which works similarly to classes for older `ecmascript` implementations; and the lifecycle functions must be defined as methods of this "class".

Example of script before refactor

```
exports.init = function () {
  ui.loadCSS('service/v1/repository/files/vgui/assets/2D/globalStyles.css');
}
exports.clickMe = function () {
  operator.state.pressed = !operator.state.pressed;
};
```

Example of script after refactor

```
exports.Control = Control;
function Control(context){
  this.state = context.node.control.state;
  this.viewModel = {
    clickMe: clickMe.bind(this)
  };
}
Control.prototype.getViewModel = function(){
  return this.viewModel;
}
function clickMe() {
  this.state.isPressed = !this.state.isPressed;
}
```

It is important to point out that the lifecycle functions still exist, but now they should be defined as methods of the `Control` class instead of directly exported functions.

Individual Script vs Script Composition/Extension

In the previous implementation, the multiple scripts defined in a control couldn't communicate among themselves.

Now, every exported class extends the class defined in the previous script.

In the base control, the first script doesn't extend any other script. However, in the Scene control, the first extends the last class from the base control following the order **top to bottom** - the first script is on top and the last is at the bottom.

Scripts Order

Since in the current implementation the order in which the scripts are defined matter, you can now reorder the scripts in the [Resource tab](#) by:

1. Clicking on the script to select it;
2. Using the arrows to move it up or down.

Overwrite vs Extending Base

Previously, every script created in the base control could be edited in the scene. Every time you edited a script, a copy of the base script was created in the scene with the editions made, which caused an outdated instance of controls with the edited script.

Now, the base scripts cannot be edited in the scene. With the new standard of using classes and extending the base control, it is possible to extend the base control and change only the desired part of the code with polymorphism.

Operations in the Action vs On Click of the Save Button

In the old implementation of scripts, the actions of editing, creating or deleting scripts were only saved when the scene/control was saved. If the scene/control editor was closed without saving, the changes in the scripts would be lost.

In the current implementation, all scripts are saved in the action, meaning that the script will be stored when you click the [Create Script](#) button. It will also be deleted from the storage when you hit the [Delete](#) button and edited in the storage when you click [Save](#) in the text editor.

i Note

This makes the script deletion **unreversible**, but closing an unsaved control or scene will not discard changes in the scripts.

Opening an Outdated Scene

Scenes with outdated scripts in their controls won't run. In that case, you might see an error message in the Scene status panel next to one or more controls stating: *Missing "Control" property in the exported module.*

This means that the control is not exporting the `CONTROL` class which is not exported because the control is in an old format. To check, follow these steps:

1. Select the control with the error message in the [Navigation](#) tab;
2. Select the [Resources](#) tab;
3. In the [Script](#) section, you will see a subsection called [Outdated Scripts](#), showing the scripts in the old format.

Exporting

You can define anything in your operator's script, but if you need to use any functionality described in the following sections, exporting it to the correct association is necessary. For example, if you want to export your control's constructor function, you can either write:

```
js
exports.Control = function Control(context) { ... }
```

Or

```
js
exports.Control = Control;
function Control(key, newValue, oldValue) { ... }
```

This way, it will be properly used by the Scene runtime.

Parameters

There are some parameters that are passed to the controls constructor. They provide access to different objects in the scene. Feel free to use them however it may prove useful.

⚠ Caution

Be aware that improper handling might have unintended consequences and cause the scene to break.

Script Scope Properties

There are also two properties defined in the scripts scope for creating the `Control` constructor function:

- **Exports:** The object which contains all your exported functions and variables. Currently only `Control` is used).
- **Super:** The `Super` constructor function is the exported constructor of the previous script in the compiling pipeline, as described in the Important Concepts section. They can be used for extending the previous control constructor or calling one of its methods.

Control Constructor Parameters

The control constructor receives 3 parameters:

```
function Control(controlContext, api, familyContext){
  ...
}
```

- **controlContext:** An object with properties related to every control. It has the following properties:
 - **node:** The object that contains the state and the default lifecycle functions. The instance of the visual wrapper of the control. If in a 2D scene, it will be a `Node2D`, whereas in a 3D scene, it will contain a `Node3D`. This object contains several utility functions for advanced customization of an element and is intended for experienced developers.
 - **player:** The instance of the current `ScenePlayer`.

- **dataHandler**: The object responsible for sending and receiving data through the websocket.
- **editor**: The instance of the current `SceneEditor`.
- **sceneContainer**: The HTML element in which the scene is rendered.
- **api**: Object that contains useful functions in the API:
 - **log(message, type?)**: A message in the modeler's log panel.
 - message: A string with the message to be added
 - type?: A string defining the type of the message. Should be either *success* or *error*.
 - **animateTo(stepName)**: Scene camera animation from the current point to the specified step (`stepName`). Only available in 3D tiles.
 - **navigate(graphName, sceneName)**: An instance of `graphName` in the same window as the running scene. If the specified graph already has an instance running, it will be switched to the scene of that instance. Otherwise, it creates a new instance of the graph and is switched to it.
 - graphName: A string with the path of the graph to navigate.
 - sceneName: A string with the path of the scene to navigate.
If the scene name is not passed, the opened scene will be the default one.
If `graphName` is not passed, the scene opened will be a scene in the current graph.
 - **loadJS(pathToFile, reload?)**: JS scripts in runtime.
 - pathToFile: A string with the whole path for the asset to be loaded.
 - reload?: A boolean that if set to true will reload the file if already loaded. The default is false.
 A good example of use case is to load `three.js` shaders:

```
js
ui.loadJS('service/v1/repository/files/controls/com/sap/vgui/water1/assets/WaterShader.js').then(function () {
  water = new THREE.Water(...);
});
```

Bear in mind that `loadJS` only supports global modules, so every package that you import must assign itself to the `window` object.

Internally, this uses `SystemJS.load()`, which returns a `Promise`, allowing for `then` usage.

- **loadCSS(pathToFile)**: CSS stylesheets in runtime. Returns a `Promise` that is resolved when the stylesheet is done loading.

i Note

Since styles are loaded for the entire window in the browser, if the imported stylesheets create any rules with global or non-strict selectors, it might affect the look and feel of the entire UI.

If you load a filepath multiple times, it will be reloaded and refreshed.

- **ensureUI5()**: Responsible for loading the UI5 library and returning a `Promise` that resolves when it's done loading. It's also used to check if UI5 is already loaded, since the `Promise` will resolve right away.
- **createUI5View(mSettings)**: Used to create UI5 views. The `mSettings` parameter references the object with the same name in the). The most important settings are `viewContent` and `controller`. Returns a `Promise` that is resolved when the view is done creating.
- **familyContext**: This parameter has an object with properties of the control in relation to its container. E.g.: For controls with the family `three`, `familyContext` has the following properties:
 - scene: The Three.js scene.
 - renderer: The Three.js renderer.
 - camera: The 3D camera used in the scene.

- **controls**: The three js object responsible for controlling the rotation and movement of the camera.
- **target**: The DOM element in which the threejs scene is rendered.
- **steps**: An array with the camera steps.

Event Hooks

Those are the event hooks defined as methods in the `Control` property that the Scene Editor's core will call.

Lifecycle

- **init()**: Called when the operator is created. Useful for setting initial values or listeners.
- **destroy()**: Called when the object is removed. Useful for freeing resources or listeners.
- **start()**: Called when the scene starts.
- **stop()**: Called when the scene stops.
- **getViewMode()**: Called in when building the view model for the 2D scene binding system. Should return an object with the desired properties to be used in template for binding.
- **update(info)**: Update loop, called at every frame.
 - **info (object)**: Contains the time in which the function has been called and a delta, which is the time difference between the current time and the time of the last update call.
- **beforeSetState(key, newValue, oldValue)**
- **afterSetState(property, newValue, oldValue)**
- **resize()**: Only for 2D scenes: Called when the tile is resized. To be used if layout management in your operator according to the container size is needed.

User Interaction

Capture events in the 3D environment.



- **keydown(event)**: Called when the `keydown` event occurs.
- **keyup(event)**: Called when the `keyup` event occurs.
- **mouseup(event)**: Called when the `mouseup` event occurs.
- **mousemove(event)**: Called when the `mousemove` event occurs.
- **touchstart(event)**: called when the `touchstart` event occurs.
- **touchend(event)**: Called when the `touchend` event occurs.
- **touchmove(event)**: Called when the `touchmove` event occurs.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.

About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
 - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
 - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.

The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

© 2021 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <https://www.sap.com/about/legal/trademark.html> for additional trademark information and notices.