AI REGIO Data Pipelines

Deliverable Author:	NISSATECH
Work Package:	WP1 (T1.3)
Date:	MAY 2021
Approved by:	-
Approved by:	-

H2020 Innovation Action - This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement N. 952003

Innovation Action - This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement N. 952003

Contents

1 StreamPipes Installation

1.1 Introduction

This section contains installation and configuration walkthrough of StreamPipes toolbox for Linux Operating System.

Useful content:

- StreamPipes – Getting Started
- apache/incubator-streampipes-installer

1.2 Installation

The easiest way to install StreamPipes would be to follow the guide found on *StreamPipes – Getting Started* page. This will install StreamPipes in “***user***” mode, enabling interaction with its web application and using its functionalities (*managing PEs and pipelines, using SP visualization dashboard, etc.*).

Other option would be to follow guides provided on *apache/incubator-streampipes-installer*, which extensively explains installation process and configuration options. It provides explanation regarding installing StreamPipes in “**developer**” mode, as well. In said mode, all the necessary ports and services are configured in order to enable development of custom pipeline elements and new backend and UI features of StreamPipes. All functionalities of “**user**” mode are preserved, as well.

1.3 Additional configurations

1.3.1 UnknownHostException

During development of custom pipeline elements an *UknownHostException* error can occur. It means that docker container is unable to properly map specified IP address to the appropriate IP address on the host machine.

We recommend that this solution gets applied in order to resolve mentioned error.

Innovation Action - This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 952003

2 StreamPipes Usage

This section contains description of developed StreamPipes Pipeline Elements (PEs) – their purpose, implementation, corresponding use-cases, etc.

It is organized according to their roles into 4 sections – Adapters, Data Sets/Streams, Data Processors and Data Sinks.

Useful content:

- StreamPipes User Guide
- StreamPipes Developer Guide
- apache/incubator-streampipes
- apache/incubator-streampipes-extensions

2.1 Pipeline Elements PEs

Each PE description is organized into following sections – purpose/role, implementation, use-cases it is being used for and additional comments, if required.

2.1.1 Adapters

SP Adapters represent special kind of PEs and are part of SP **Connect Library**. They are not directly employed in pipelines, but are, in fact, used as "templates" to instantiate purpose-specific Data Set/Stream PEs.

Therefore, they are used to create Data Sets/Streams that are going to fetch data from database (PostgreSQL, InfluxDB, MySQL, etc.) or broker (RabbitMQ, etc.), subscribe with messaging protocol (e.g., MQTT), use OPC UA, etc.

Adapters are used to configure specific Data Sets/Streams and define their Event Schemas.

Implementation-wise, important parts of Adapter are its model declaration, the way we start/stop the Adapter and event schema fetching.

2.1.1.1 AAS over HTTP

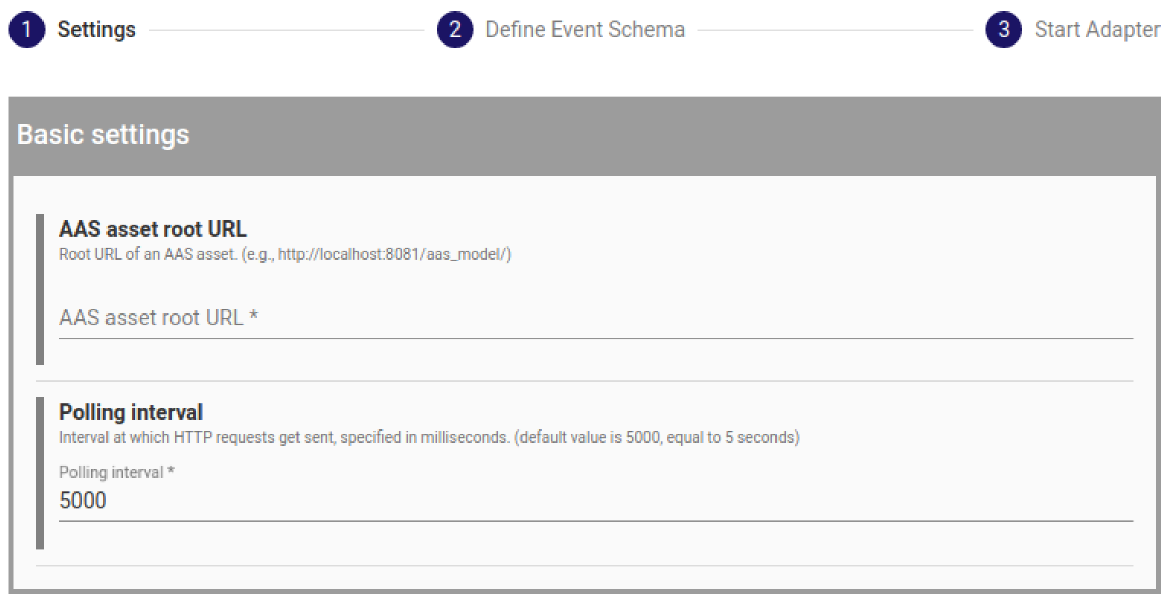
This Adapter is used to instantiate a Data Stream PE that will periodically poll Asset Administration Shell instance over HTTP, in order to fetch values of its properties.

Therefore, user is provided with options to configure the AAS instance to be connected with, as well as the polling interval (Image 2.1.1.1).

Image 2.1.1.1. *Configuration options of "AAS over HTTP" Adapter*

Once created and used in pipeline, corresponding Data Source PE will connect to the specified AAS instance and poll for data as long as said AAS instance is up and running.

Implementation



1 Settings — 2 Define Event Schema — 3 Start Adapter

Basic settings

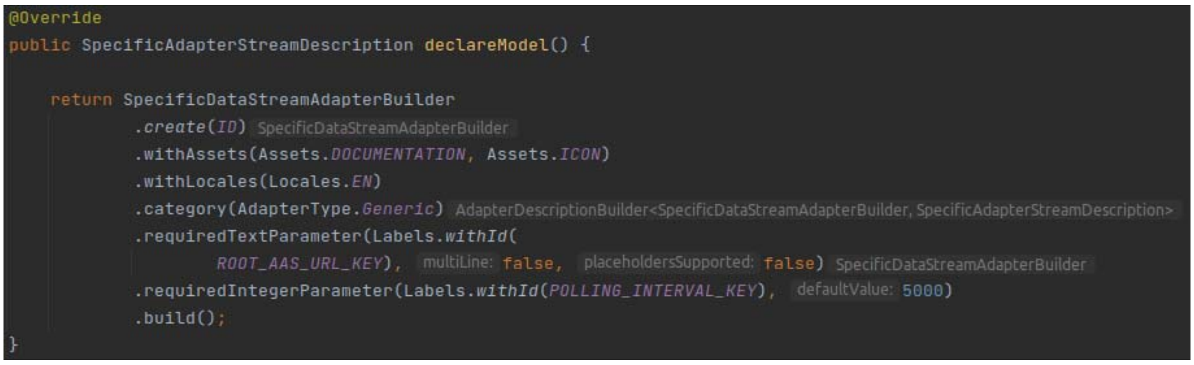
AAS asset root URL
Root URL of an AAS asset. (e.g., http://localhost:8081/aas_model/)

AAS asset root URL *

Polling interval
Interval at which HTTP requests get sent, specified in milliseconds. (default value is 5000, equal to 5 seconds)

Polling interval *
5000

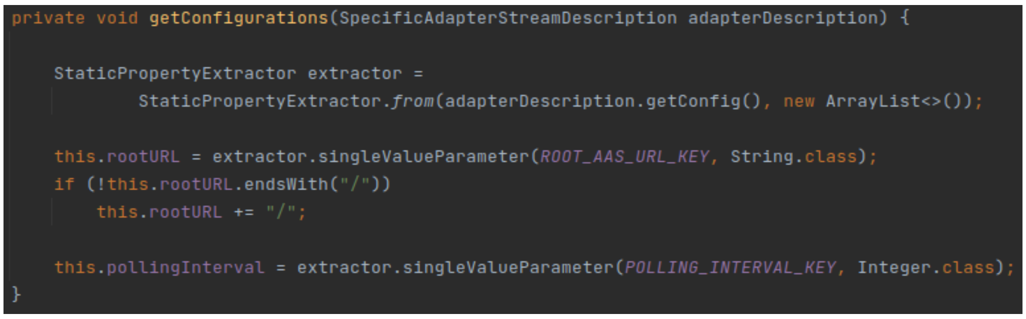
As this Adapter requires URL of a running AAS instance and polling interval to be provided, its model is declared in the following way:



```
@Override
public SpecificAdapterStreamDescription declareModel() {

    return SpecificDataStreamAdapterBuilder
            .create(ID) SpecificDataStreamAdapterBuilder
            .withAssets(Assets.DOCUMENTATION, Assets.ICON)
            .withLocales(Locales.EN)
            .category(AdapterType.Generic) AdapterDescriptionBuilder<SpecificDataStreamAdapterBuilder, SpecificAdapterStreamDescription>
            .requiredTextParameter(Labels.withId(
                    ROOT_AAS_URL_KEY),  multiLine: false,  placeholdersSupported: false) SpecificDataStreamAdapterBuilder
            .requiredIntegerParameter(Labels.withId(POLLING_INTERVAL_KEY),  defaultValue: 5000)
            .build();
}
```

It prompts user to enter text value (*requiredTextParameter with **ROOT_AAS_URL_KEY***) and integer value (*requiredIntegerParameter with **POLLING_INTERVAL_KEY***) representing URL and polling interval, respectively.

User input is fetched in the following way:



```
private void getConfigurations(SpecificAdapterStreamDescription adapterDescription) {

    StaticPropertyExtractor extractor =
            StaticPropertyExtractor.from(adapterDescription.getConfig(), new ArrayList<>());

    this.rootURL = extractor.singleValueParameter(ROOT_AAS_URL_KEY, String.class);
    if (!this.rootURL.endsWith("/"))
        this.rootURL += "/";

    this.pollingInterval = extractor.singleValueParameter(POLLING_INTERVAL_KEY, Integer.class);
}
```

During Adapter configuration, in the *Define Event Schema* step, Adapter sends requests to provided AAS instance URL to fetch available *submodels* and *submodelElements*. Once fetched, user can select ones that are going to be part of events that corresponding Data Source sends out.

When starting corresponding Data Source PE (*starting the pipeline*), user-selected properties of AAS instance during *Define Event Schema* step of Adapter configuration are singled out – properties that are going to be polled for. In addition, a "single thread scheduled executor" gets started that will periodically poll for data of selected properties.

When stopping the corresponding Data Source PE (*stopping the pipeline*), said "single thread scheduled executor" gets stopped, thus ending the polling process.

Comments

- This PE requires version **0.68.0**(-***SNAPSHOT***) of SDK and running StreamPipes instance in order to develop and run this adapter.
- It is planned to update this PE's configuration steps, in order to allow user to select which *submodels* of provided AAS instance Event Schema gets fetched from.

2.1.2 Data Sets/Streams

SP Data Sets/Streams represent a starting point of pipeline. They are used for connecting the pipeline to a data source, whether that is sensor (*e.g., with some messaging protocol*), database, broker, HTTP server, some custom third-party service/system, etc.

Data Set/Stream PEs can be created either as an SP Adapter instance (*preferred way, since it enables custom configuration, e.g., defining which database Data Set/Stream connects to*) or with SP SDK (*good option for creating use-case specific Data Sets/Streams that cannot be made general or be reused*).

Implementation-wise, important parts of Data Processor are its model declaration and logic (*how is data streaming performed*).

2.1.3 Data Processors

SP Data Processors are main part of every pipeline – they are used for data processing, manipulation, analysis, etc. Built-in Data Processor PEs include ones for event manipulation (*remove, change, add property, etc.*), basic trend detection, image manipulation, filtering, etc. In addition, with provided SP SDK, pool of Data Processors can be greatly expanded, thus covering nearly all use-cases (*Machine Learning, Complex Event Processing, Data Analysis, etc.*).

Implementation-wise, important parts of Data Processor are its model declaration and logic (*what happens when new event arrives*).

2.1.3.1 Trend with Numerical Comparison

This Data Processor performs Complex Event Processing using Siddhi engine and outputs results of CEP query execution. Internally, this element uses StreamPipes Siddhi wrapper that maps StreamPipes' input and output events to the Siddhi engine input and output events, respectively. Additionally, it provides means of writing CEP queries with regard to pipeline element configuration.

It detects whether comparison of some numerical property of input event to the one provided in configuration step fulfils condition within given window of consecutive events. Therefore, the query we want to execute looks like this:

*define stream InputStream (**<property>** float);*
define stream CondStream (cond1 bool, cond2 bool);

*from InputStream#window.length(**<window length>**)*
*select count() == **<window_length>** as cond1, and(**<property>** **<comparison operator>** **<comparison value>**) as cond2*
insert into CondStream;

from CondStream
*select cond1 and cond2 as **<output property name>***
insert into OutputStream;

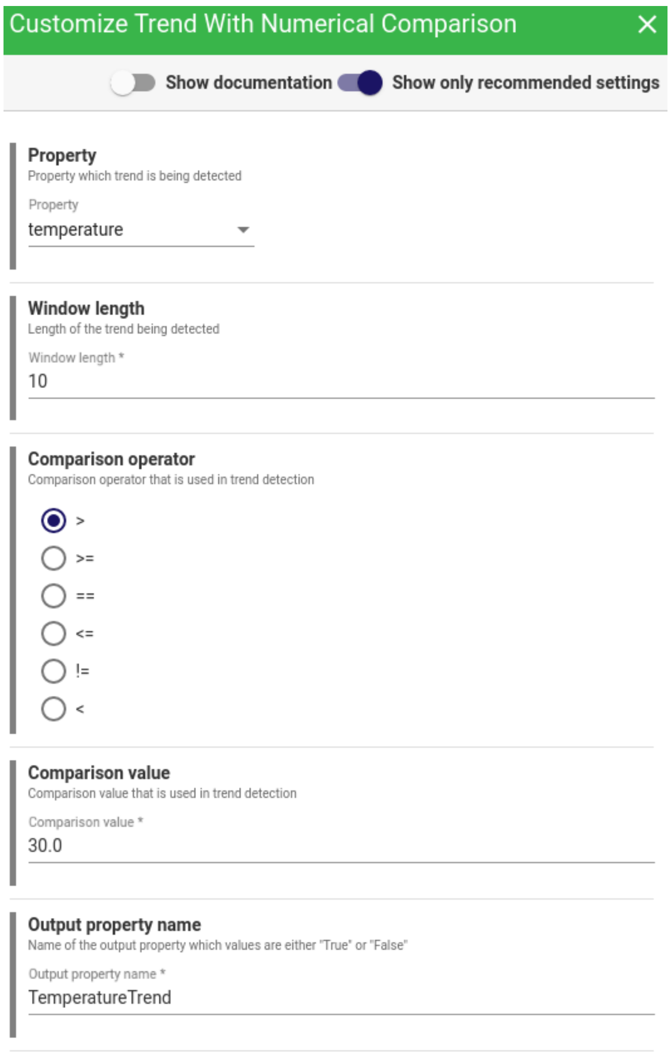
Therefore, user is provided with options to configure which event's numerical value – named **<property>**, is going to be taken into account for specified trend detection, value it is going to be compared with – named **<comparison value>**, operator that is going to be used for comparison – named **<comparison operator>**, number of consecutive events that are going to be used – named **<window length>** and name of the output property – named **<output property name>** (Image 2.3.1.1.).

Image 2.3.1.1. *Configuration options of "Trend with Numerical comparison" Data Processor with demo values*

When used in pipeline, this PE looks upon "window length" number of last arrived events and determines whether all of them fulfil comparison condition. If they do, it outputs "*true*", otherwise, it outputs "*false*".

Implementation

As this Data Processor requires numerical event property, comparison value and operator, window length and output property name to be specified, its model is declared in the following way:



Customize Trend With Numerical Comparison

Show documentation
Show only recommended settings

Property
Property which trend is being detected
Property: temperature

Window length
Length of the trend being detected
Window length *: 10

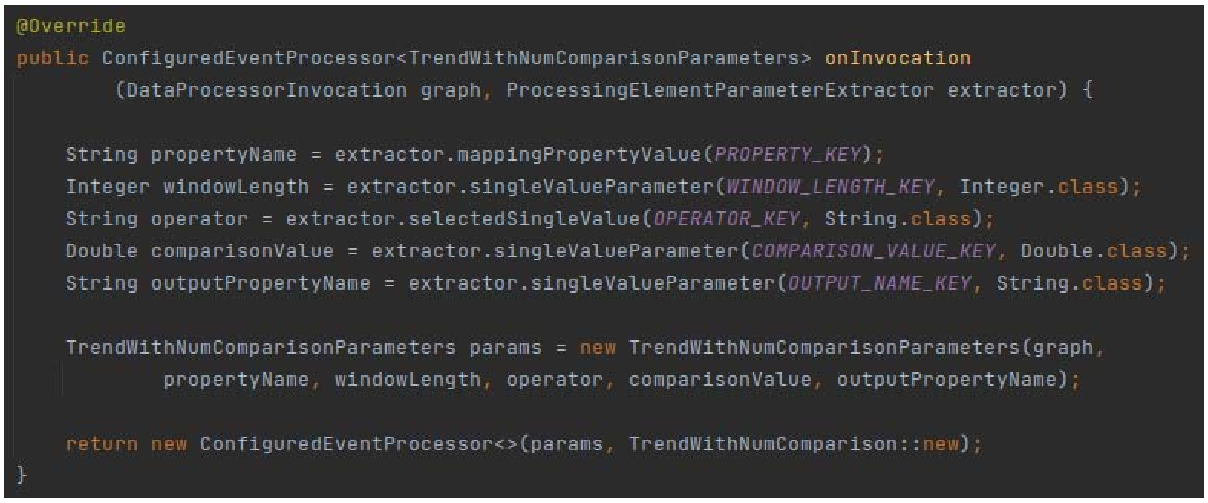
Comparison operator
Comparison operator that is used in trend detection
[✓ >]
>=
==
<=
!=
<

Comparison value
Comparison value that is used in trend detection
Comparison value *: 30.0

Output property name
Name of the output property which values are either "True" or "False"
Output property name *: TemperatureTrend

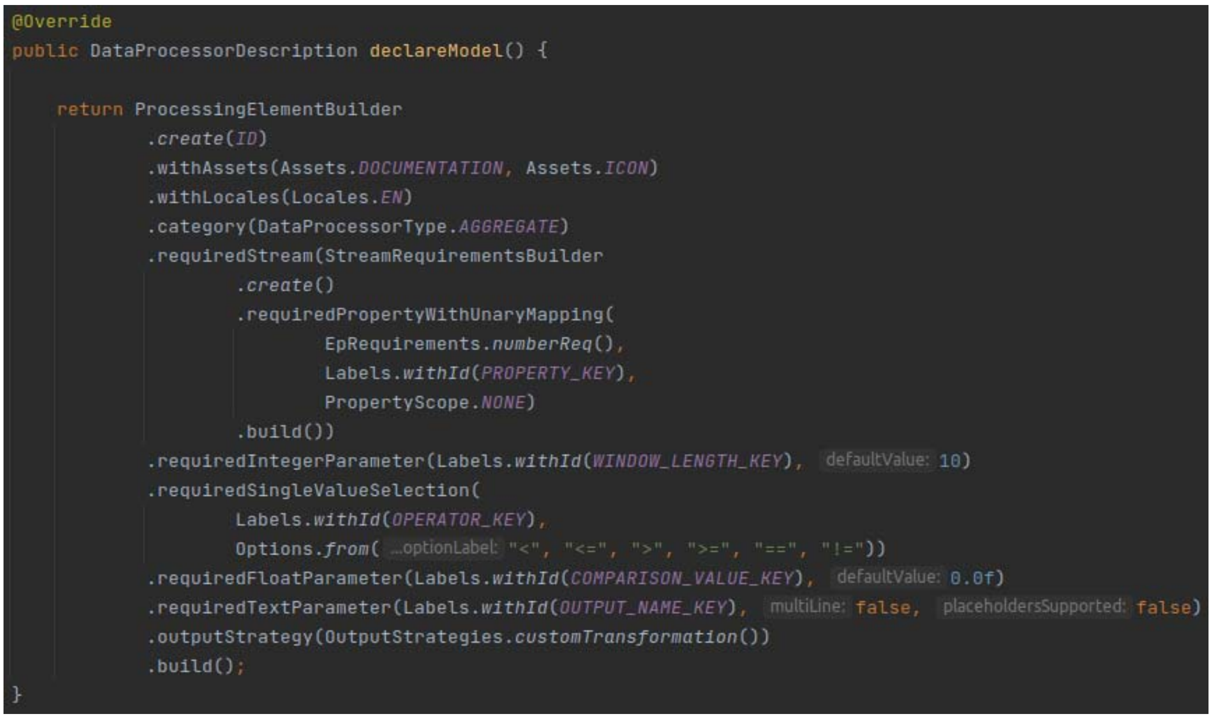
It prompts user to select numerical event property (*requiredPropertyWithUnaryMapping with* ***PROPERTY_KEY,*** *of requiredStream*), upon which trend detection is going to be performed, and operator (*requiredSingleValueSelection with* ***OPERATOR_KEY***) that is going to be used for comparison. Additionally, it prompts user to enter integer value (*requiredIntegerParameter with* ***WINDOW_LENGTH_KEY***), float value (*requiredFloatParameter with* ***COMPARISON_VALUE_KEY***) and text value (*requiredTextParameter with* ***OUTPUT_NAME_KEY***) representing window length, value used for comparison and name of output property, respectively.

User input is fetched in the following way:

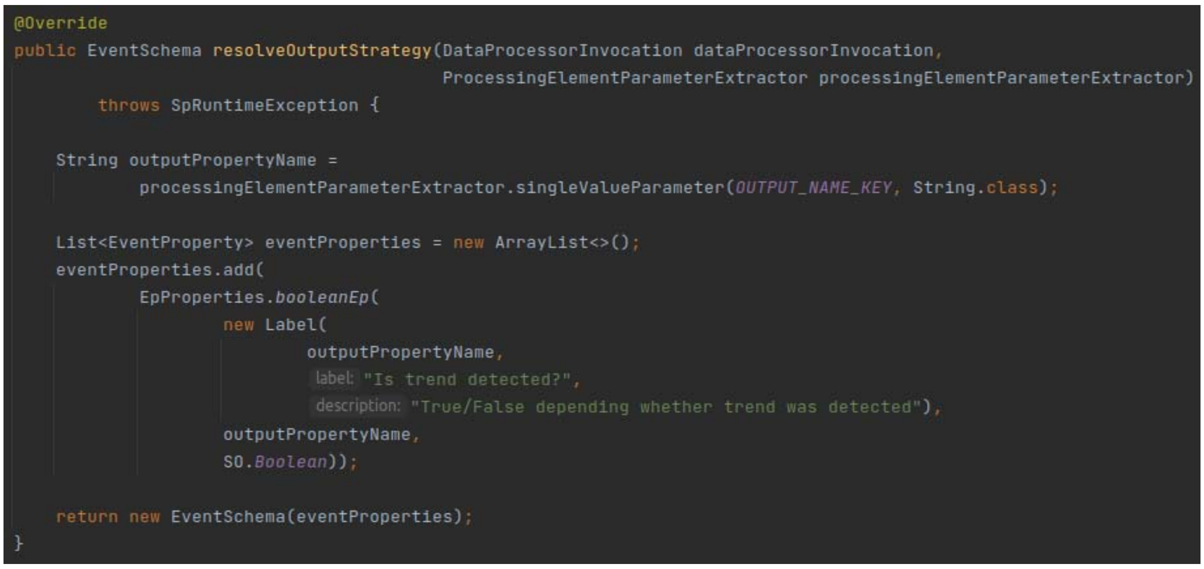


```
@Override
public ConfiguredEventProcessor<TrendWithNumComparisonParameters> onInvocation
        (DataProcessorInvocation graph, ProcessingElementParameterExtractor extractor) {

    String propertyName = extractor.mappingPropertyValue(PROPERTY_KEY);
    Integer windowLength = extractor.singleValueParameter(WINDOW_LENGTH_KEY, Integer.class);
    String operator = extractor.selectedSingleValue(OPERATOR_KEY, String.class);
    Double comparisonValue = extractor.singleValueParameter(COMPARISON_VALUE_KEY, Double.class);
    String outputPropertyName = extractor.singleValueParameter(OUTPUT_NAME_KEY, String.class);

    TrendWithNumComparisonParameters params = new TrendWithNumComparisonParameters(graph,
            propertyName, windowLength, operator, comparisonValue, outputPropertyName);

    return new ConfiguredEventProcessor<>(params, TrendWithNumComparison::new);
}
```

Because we want to allow user to specify the name of the output property, we declared the model with *customTransforamtion outputStrategy*. Therefore, this PE's output strategy gets inferred after configuration step, in the following manner:



```
@Override
public DataProcessorDescription declareModel() {

    return ProcessingElementBuilder
            .create(ID)
            .withAssets(Assets.DOCUMENTATION, Assets.ICON)
            .withLocales(Locales.EN)
            .category(DataProcessorType.AGGREGATE)
            .requiredStream(StreamRequirementsBuilder
                    .create()
                    .requiredPropertyWithUnaryMapping(
                            EpRequirements.numberReq(),
                            Labels.withId(PROPERTY_KEY),
                            PropertyScope.NONE)
                    .build())
            .requiredIntegerParameter(Labels.withId(WINDOW_LENGTH_KEY),  defaultValue: 10)
            .requiredSingleValueSelection(
                    Labels.withId(OPERATOR_KEY),
                    Options.from( ...optionLabel: "<", "<=", ">", ">=", "==", "!="))
            .requiredFloatParameter(Labels.withId(COMPARISON_VALUE_KEY),  defaultValue: 0.0f)
            .requiredTextParameter(Labels.withId(OUTPUT_NAME_KEY),  multiLine: false,  placeholdersSupported: false)
            .outputStrategy(OutputStrategies.customTransformation())
            .build();
}
```



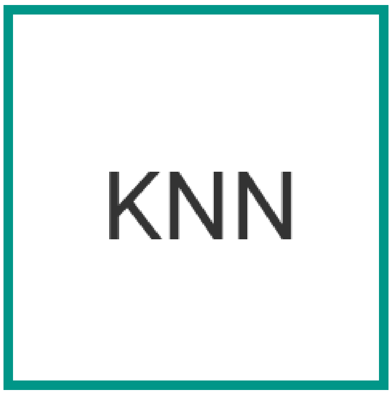
```
@Override
public EventSchema resolveOutputStrategy(DataProcessorInvocation dataProcessorInvocation,
                                         ProcessingElementParameterExtractor processingElementParameterExtractor)
        throws SpRuntimeException {

    String outputPropertyName =
            processingElementParameterExtractor.singleValueParameter(OUTPUT_NAME_KEY, String.class);

    List<EventProperty> eventProperties = new ArrayList<>();
    eventProperties.add(
            EpProperties.booleanEp(
                    new Label(
                            outputPropertyName,
                            label: "Is trend detected?",
                            description: "True/False depending whether trend was detected"),
                    outputPropertyName,
                    SO.Boolean));

    return new EventSchema(eventProperties);
}
```

Every new event that gets fed to this Data Processor, gets stored inside window of specified length. Then, for each event inside said window, it gets determined whether it satisfies specified condition or not – if they do, this PE outputs "*true*", otherwise, it outputs "*false*".

Comments

- Siddhi wrapper used for implementation of this PE is not yet fully developed and lacks certain features, such as some extensions. Updates are planned for version 0.68.0.
- This PE was implemented with version **0.68.0**(***-SNAPSHOT***) of SDK. Therefore, it requires the same version of running StreamPipes instance in order to run this PE.

2.1.3.2 Keras Neural Network



This Data Processor performs inference with loaded HDF5-formated Neural Network model and outputs its result. It is required that the loaded NN model performs binary classification; i.e., it has one output neuron that states probability that input instance belongs to certain class.

Internally, this element uses DeepLearning4J library (*dedicated to Deep Learning in Java*) to load NN model and perform inference.

Upon connecting this PE to the pipeline, user is provided with options to configure which event properties are going to be used as input for the NN model, what will their order be (*i.e., which property corresponds to which input neuron*) and which NN model file is going to be used (Image 2.3.2.1.).

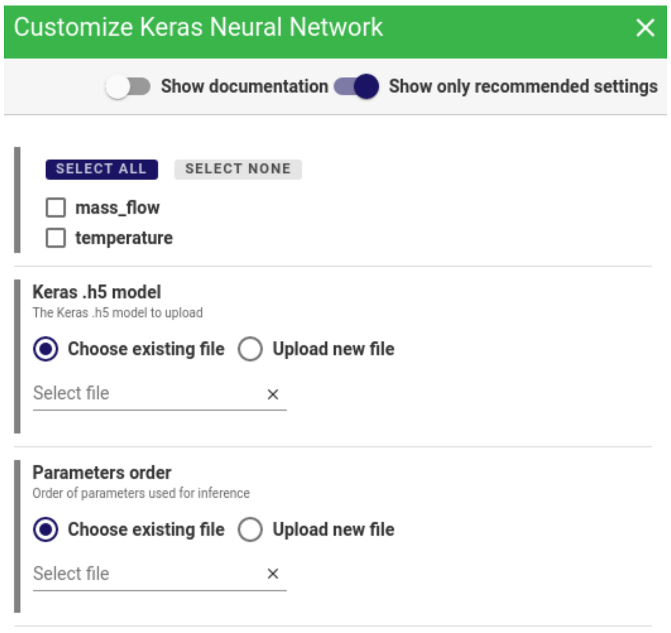


Image 2.3.2.1. *Configuration options of “Keras Neural Network” Data Processor with demo Event Schema*

This PE requires a .h5 file to be provided in order for the NN model to be properly loaded. Additionally, user must provide an order of event properties that are going to be used as input to NN model in JSON format, like the one presented below:

```
{
  "order": ["temperature", "mass_flow"]
```

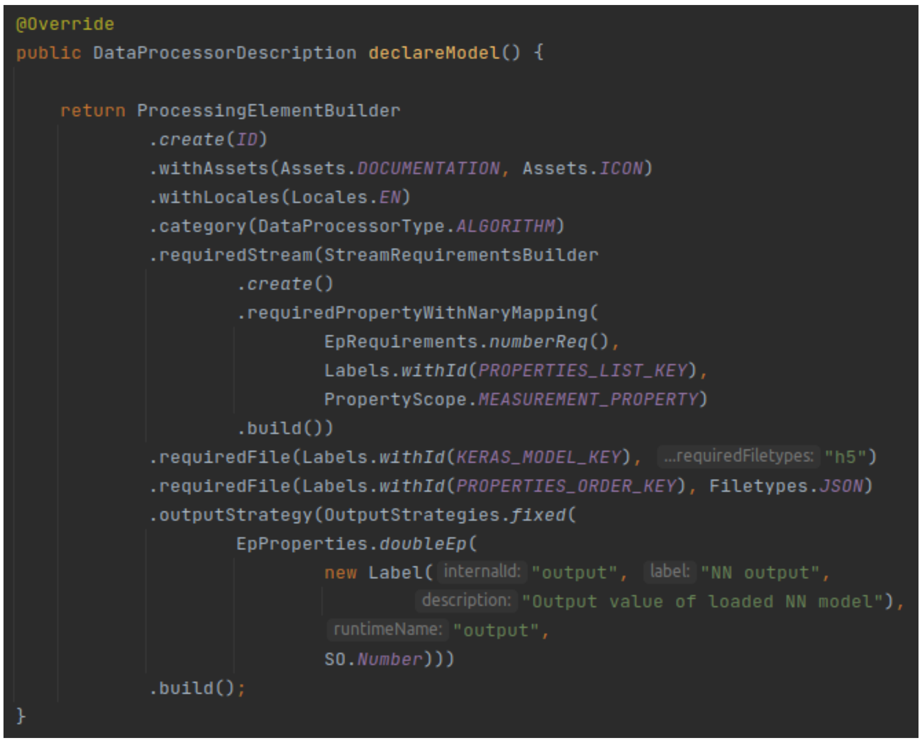
}

According to the presented order, value of "temperature" property will be used as input for first neuron, while value of "mass_flow" property will be used as input for second neuron.

When used in pipeline, this PE loads selected NN model and, for each received event, orders its properties according to the provided order and performs inference, outputting its result when completed.

Implementation

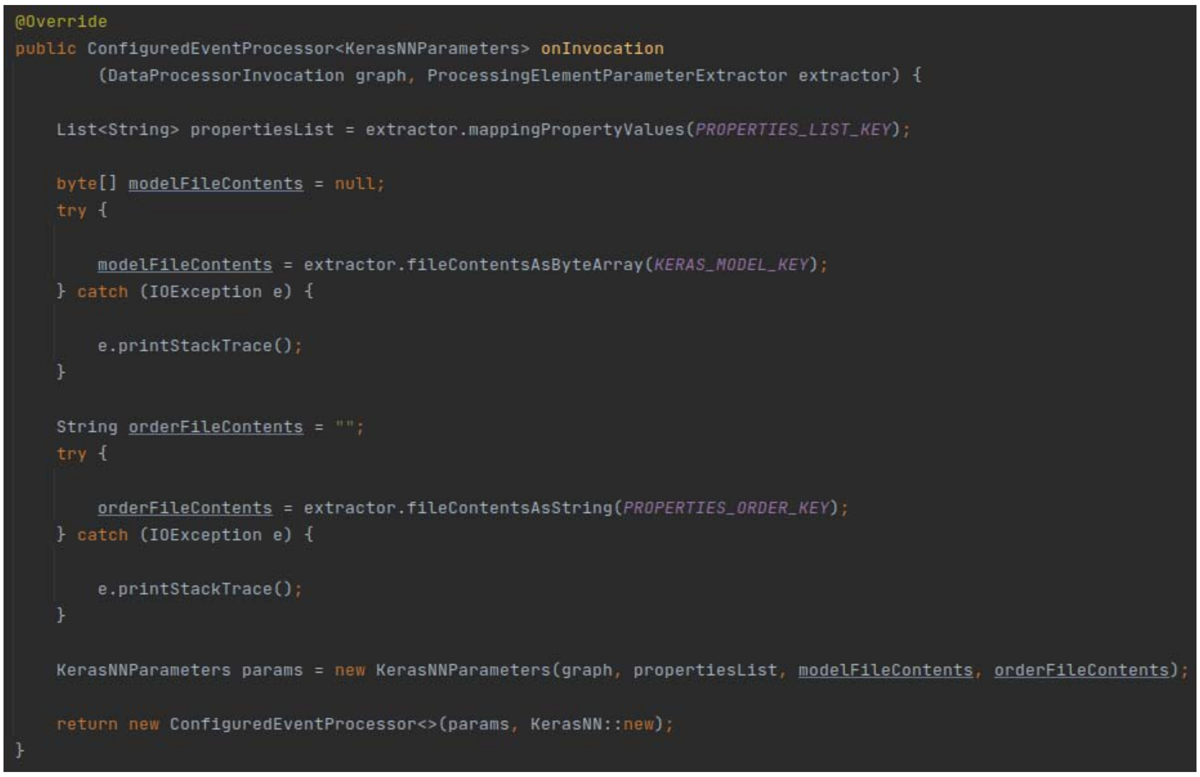
This Data Processor requires numerical event properties, .h5 file of a NN model and JSON file containing order of event properties to be specified. Therefore, its model is declared in the following way:



```
@Override
public DataProcessorDescription declareModel() {

    return ProcessingElementBuilder
            .create(ID)
            .withAssets(Assets.DOCUMENTATION, Assets.ICON)
            .withLocales(Locales.EN)
            .category(DataProcessorType.ALGORITHM)
            .requiredStream(StreamRequirementsBuilder
                    .create()
                    .requiredPropertyWithNaryMapping(
                            EpRequirements.numberReq(),
                            Labels.withId(PROPERTIES_LIST_KEY),
                            PropertyScope.MEASUREMENT_PROPERTY)
                    .build())
            .requiredFile(Labels.withId(KERAS_MODEL_KEY), ...requiredFiletypes: "h5")
            .requiredFile(Labels.withId(PROPERTIES_ORDER_KEY), Filetypes.JSON)
            .outputStrategy(OutputStrategies.fixed(
                    EpProperties.doubleEp(
                            new Label( internalId: "output",  label: "NN output",
                                    description: "Output value of loaded NN model"),
                             runtimeName: "output",
                            SO.Number)))
            .build();
}
```

It prompts user to select numerical event properties (*requiredPropertyWithNaryMapping with* ***PROPERTIES_LIST_KEY,*** *of requiredStream*), that are going to be used as input for NN model. Additionally, user is required to provide .h5 file (*requiredFile with* ***KERAS_MODEL_KEY***) and JSON file (*requiredFile with* ***PROPERTIES_ORDER_KEY***), representing NN model and order of properties, respectively.

User input is fetched in the following way:



```
@Override
public ConfiguredEventProcessor<KerasNNParameters> onInvocation
        (DataProcessorInvocation graph, ProcessingElementParameterExtractor extractor) {

    List<String> propertiesList = extractor.mappingPropertyValues(PROPERTIES_LIST_KEY);

    byte[] modelFileContents = null;
    try {

        modelFileContents = extractor.fileContentsAsByteArray(KERAS_MODEL_KEY);
    } catch (IOException e) {

        e.printStackTrace();
    }

    String orderFileContents = "";
    try {

        orderFileContents = extractor.fileContentsAsString(PROPERTIES_ORDER_KEY);
    } catch (IOException e) {

        e.printStackTrace();
    }

    KerasNNParameters params = new KerasNNParameters(graph, propertiesList, modelFileContents, orderFileContents);

    return new ConfiguredEventProcessor<>(params, KerasNN::new);
}
```

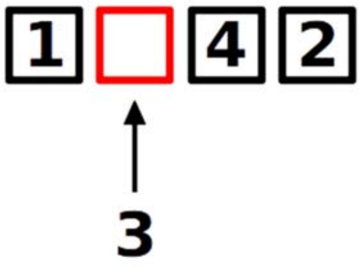
We used *fixed outputStrategy* to define output schema, because we know what output to expect from loaded NN model.

Comments

- It is rather difficult to implement a fully general and reusable PE that will use any NN model because of the mappings between input and output of said model and PE. Therefore, we fixed this PE’s output to one value and use it with NN models that have one output neuron.
- Other PEs should be implemented if there is a need for different types of NNs.

- StreamPipes developers are working on a more reusable approach to using NNs within StreamPipes.

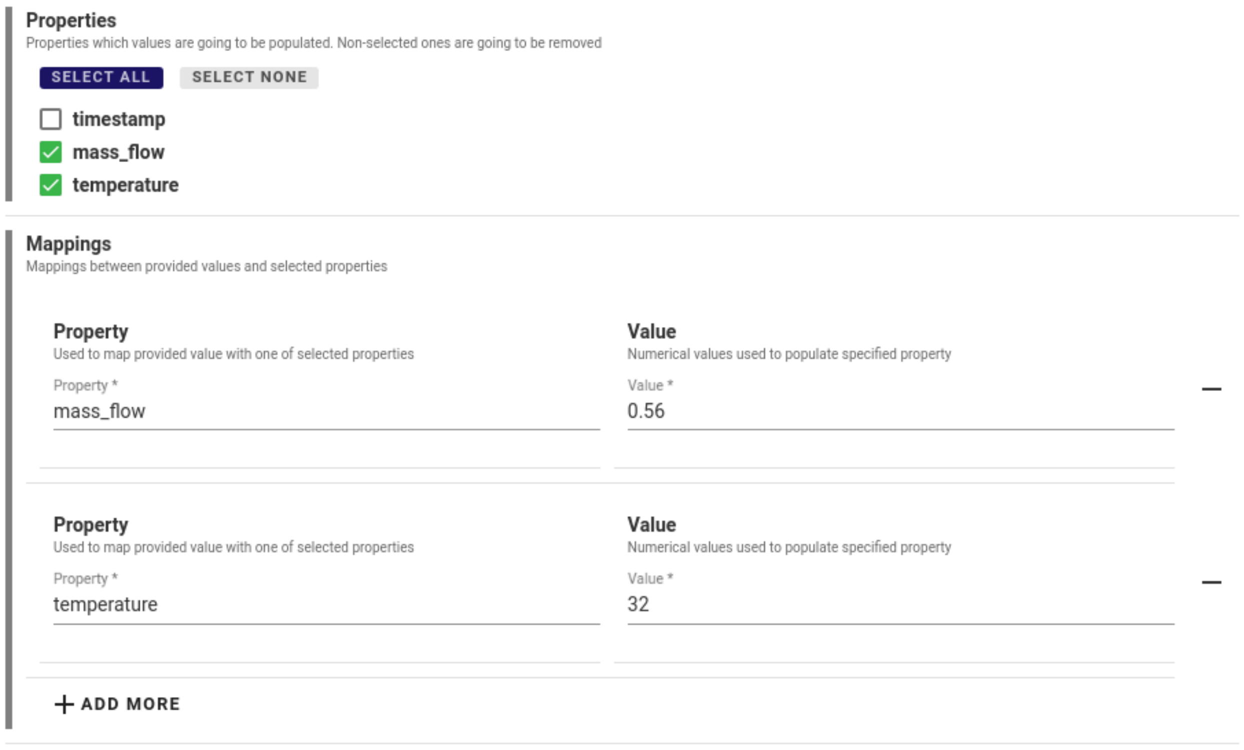
2.1.3.3 Populate Missing Values



Sometimes, data acquisition services can fail to properly collect data (e.g., sensor fails to capture measurement), leaving us with "holes" in our datasets which can have negative impact on further analysis.

This element is developed to solve this problem – Its purpose is to fill/populate missing measurements with some predefined values.

Upon connecting this PE to the pipeline, user is provided with options to configure which event properties are going to be forwarded by this element and which of these properties are going to be populated with which values (Image 2.3.3.1.). This is specified with list of property name and value pairs.



Properties
Properties which values are going to be populated. Non-selected ones are going to be removed

SELECT ALL SELECT NONE

- [ ] timestamp
- [x] mass_flow
- [x] temperature

Mappings
Mappings between provided values and selected properties

Property	Value
Used to map provided value with one of selected properties	Numerical values used to populate specified property
Property * mass_flow	Value * 0.56
Used to map provided value with one of selected properties	Numerical values used to populate specified property
Property * temperature	Value * 32

+ ADD MORE

Image 2.3.3.1. *Configuration options of "Populate Missing Values" Data Processor with demo Event Schema*

It is important to configure appropriate values that are going to be used for populating properties, e.g., use mean value of existing measurements for chosen property, with the goal being to ensure that the populated measurements are as close as possible to the real ones and to follow behavior patterns of said property.

2.1.4 Data Sinks

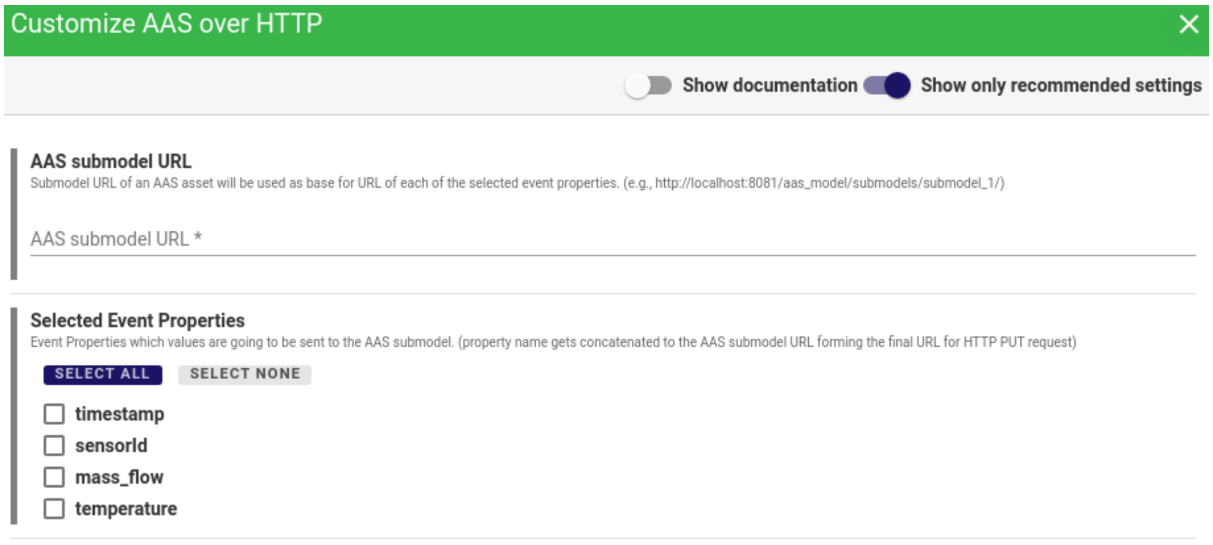
SP Data Sinks represent an endpoint of pipeline. They are used for sending notifications/alerts, writing data into database, sending data to broker, HTTP endpoint, visualization system, etc. Data Sink can be a part of StreamPipes (*SP Notification system or SP Dashboard visualization system*) or it can be connected to a third-party system/service (*PostgreSQL, MySQL, InfluxDB, Email, RabbitMQ, Kafka, etc., or some custom implemented system/service*).

Implementation-wise, important parts of Data Sink are its model declaration and logic (*what happens when new event arrives*).

2.1.4.1 AAS over HTTP

This Data Sink is used to update *submodelElements*’ values of specified *submodel of* Asset Administration Shell instance over HTTP.

Therefore, user is provided with options to configure the *submodel* of AAS instance and its *submodelElements* which values are going to be updated (Image 2.4.1.1).



Customize AAS over HTTP

Show documentation | Show only recommended settings

AAS submodel URL
Submodel URL of an AAS asset will be used as base for URL of each of the selected event properties. (e.g., http://localhost:8081/aas_model/submodels/submodel_1/)

AAS submodel URL *

Selected Event Properties
Event Properties which values are going to be sent to the AAS submodel. (property name gets concatenated to the AAS submodel URL forming the final URL for HTTP PUT request)

SELECT ALL | SELECT NONE

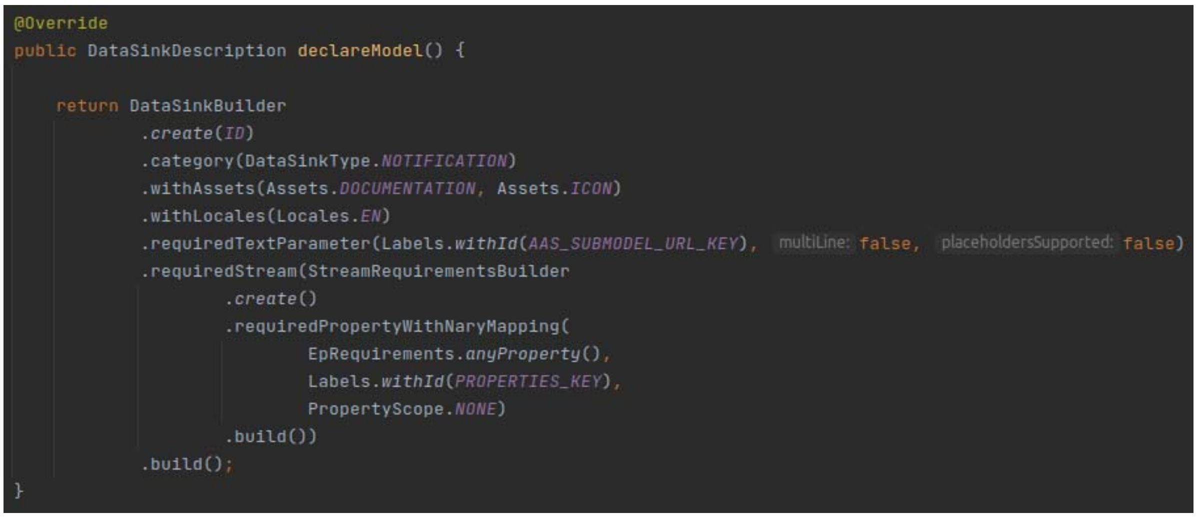
- ☐ timestamp
- ☐ sensorId
- ☐ mass_flow
- ☐ temperature

Image 2.1.1.1. *Configuration options of “AAS over HTTP” Data Sink with demo Event Schema*

When used in pipeline, this PE will send selected event data to AAS instance for every incoming event.

Implementation

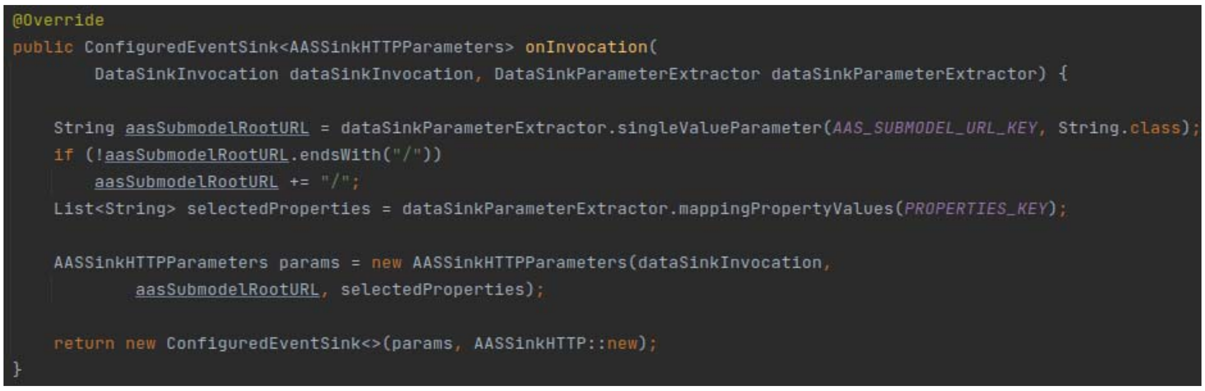
As this Data Sink requires URL of *submodel* of running AAS instance and list of event properties which values are going to be sent to AAS instance, its model is declared in the following way:



```
@Override
public DataSinkDescription declareModel() {

    return DataSinkBuilder
            .create(ID)
            .category(DataSinkType.NOTIFICATION)
            .withAssets(Assets.DOCUMENTATION, Assets.ICON)
            .withLocales(Locales.EN)
            .requiredTextParameter(Labels.withId(AAS_SUBMODEL_URL_KEY),  multiLine: false,  placeholdersSupported: false)
            .requiredStream(StreamRequirementsBuilder
                    .create()
                    .requiredPropertyWithNaryMapping(
                            EpRequirements.anyProperty(),
                            Labels.withId(PROPERTIES_KEY),
                            PropertyScope.NONE)
                    .build())
            .build();
}
```

It prompts user to enter text value (*requiredTextParameter with* ***AAS_SUBMODEL_URL_KEY***) and to select event properties (*requiredPropertyWithNaryMapping with* ***PROPERTIES_KEY****, of requiredStream*) representing URL and list of desired event properties, respectively.

User input is fetched in the following way:



```
@Override
public ConfiguredEventSink<AASSinkHTTPParameters> onInvocation(
        DataSinkInvocation dataSinkInvocation, DataSinkParameterExtractor dataSinkParameterExtractor) {

    String aasSubmodelRootURL = dataSinkParameterExtractor.singleValueParameter(AAS_SUBMODEL_URL_KEY, String.class);
    if (!aasSubmodelRootURL.endsWith("/"))
        aasSubmodelRootURL += "/";
    List<String> selectedProperties = dataSinkParameterExtractor.mappingPropertyValues(PROPERTIES_KEY);

    AASSinkHTTPParameters params = new AASSinkHTTPParameters(dataSinkInvocation,
            aasSubmodelRootURL, selectedProperties);

    return new ConfiguredEventSink<>(params, AASSinkHTTP::new);
}
```

For each new event that gets fed to this Data Sink, an HTTP GET request gets sent for each of the selected event properties, thus updating corresponding *submodelElements* of *submodel*.

Comments

- This PE was implemented with version **0.68.0**(-***SNAPSHOT***) of SDK. Therefore, it requires the same version of running StreamPipes instance in order to run this PE.
- Currently, this PE supports only submodelElements of type *Property*. It is planned to update this PE’s configuration steps and logic, in order to support more types of *sudmodelElements*, once they get integrated into AAS.

3 Data4AI Platform methods

3.1 Introduction

This section contains list of Data Quality methods in Data4AI Platform, implemented with StreamPipes.

3.2 Cleaning methods

remove_params

This method is used to remove all parameters that, for example, did not pass the profiling test or were deemed as not important.

remove_stage

This method is used to remove all parameters that belong to specified stage.

remove_products

This method removes instances that have missing values for a given stage and parameter.

make_new_param

This method is used to create new parameter based on the existing ones. It requires a formula (expression) to be provided, based on which new parameter will be calculated.

remove_outliers

This method is used to remove parameters from measurements and timeseries that represent outliers, based on the provided lower and upper boundaries.

resample_parameter(s)

This method performs resampling of specified timeseries to a desired frequency.

remove_products_with_less_datapoints

This method removes instances for which number of recorded values, for any of the specified parameter, is smaller than the specified threshold. In the case of timeseries data (one of the specified *parameters of an instance*) of length *M* and threshold *T,* if *M* is smaller than *T*, instance gets removed. In the case of measurements data, if one of the specified parameters is missing its value, instance gets removed.

remove_products_missing_parameter

remove_products_with_missing_values

This method removes instances for which number of recorded values is smaller than the specified threshold. In the case of an instance with *N* timeseries parameters of lengths $M_{1,\ 2,\ \dots,\ N}$ and threshold *T,* if $N * M_{1,\ 2,\ \dots,\ N}$ is smaller than *T*, instance gets removed. In the case of an instance with *N* recorded measurements data values, if *N* is smaller than *T*, instance gets removed.

fill_param_with_value

This method fills in missing values of measurement or timeseries with a given value. For example, with average value for measurement or last recorded value for timeseries.
Whenever there is an empty property of an input event, it would get filled in according to the user-specified option.

truncate_timeseries

This method is used to remove all datapoints in timeseries that were recorded before or after specified time value.

3.3 Preparation methods

scale_data

This method performs scaling of data, according to required function (*e.g., normalization, standardization, etc.*) and specified function parameters (*minimum and maximum values for normalization, average and standard deviation values for standardization, etc.*).