Hortonworks Streaming Analytics Manager 3

Creating Custom Builder Components

Date of Publish: 2020-12-15

https://docs.cloudera.com/

Contents

Adding Custom Builder Components

You can use the SAM SDK to add custom components to your SAM applications.

Adding Custom Processors

To add custom processors to SAM, create the processors and then register it with SAM.

Creating Custom Processors

Create a custom processor using the SDK, and package it into a jar file with all of its dependencies.

About this task

Create a new maven project using this maven pom file as an example.

Procedure

1. To implement a custom processor, implement the following interface:

```
org.apache.streamline.streams.runtime.CustomProcessorRuntime
```

2. Package the jar file with all dependencies, by running the following commands:

```
mvn clean package
mvn assembly:assembly
```

3. In the target directory you should have an uber jar that ends with jar-with-dependencies.jar. You need this jar file when you register your custom processor with SAM.

Example

The PhoenixEnrichmentProcessor is a good example of a new custom processor implementation.

Registering Custom Processors with SAM

You have to register each custom processor in SAM before you can use it for the first time.

Procedure

1. From the left-hand SAM Global menu, hover over the **Configuration** menu, click **Application Resources**, and then click the **Custom Processor** tab.
2. Click the + icon to add a new processor.
3.

Results

It might take a few minutes to upload the jar file to the server. Do not navigate away until you see a response. If you do not see a response, return to the Custom Processor page again; do not click Save again.

Creating a Custom Streaming Application

After you have registered your custom processor, create a new stream application.

Procedure

1. From **My Applications** click the + icon and launch the **Add Application** dialog.

2. Find your new processor in the **Processor Toolbar**, drag it onto the canvas, and configure it.

Results

When you double-click on your new custom processor, the configuration fields are exposed. Notice that the configuration is based on the “Config Fields” settings specified during the registration process.

Adding Custom Functions

User Defined Aggregate Functions (UDAF) allow you to add custom aggregate functions to SAM. Once you create and register UADFs they are available for use in the Aggregate processor.

User Defined Functions (UDFs) allow you to do simple transformations on event streams. This is used in the Projection processor.

This section provides information on how to create, build, and upload these custom functions.

Creating UDAFs

User Defined Aggregate Functions (UDAF) allow you to add custom aggregate functions to SAM. Once you create and register UADFs they are available for use in the Aggregate processor. Use these steps to create a new UADF.

Procedure

1. Create a UADF by implement the following interface:

```
public interface UDAF<A, V, R> {
   A init();
   A add(A aggregate, V val);
   R result(A aggregate);
}
```

   Where:

   - A – Is the type of the aggregate that is used to aggregate the values. init returns the initial value for the aggregate.
   - V – is the type of the values we are processing. The add method is invoked with the current aggregate and the value for each of the events in the window. add is expected to aggregate the current value and return the updated aggregate.
   - R – is the result type and the result function takes the final aggregated value and returns the result.

2. For aggregate functions that requires two parameters, the UDAF2 interface also requires implementation. The only difference is that the add function is passed the current value of the aggregate and two values instead of one.

```
public interface UDAF2<A, V1, V2, R> {
   A init();
   A add(A aggregate, V1 val1, V2 val2);
   R result(A aggregate);
}
```

Example

In this example, you want to compute the average values of a particular field for events within a window. To do that, define an average aggregate function by implementing the UDAF interface as shown below:

```
// Here the aggregate is a pair that holds the running sum and the count of
 elements seen so far
// The values are integers and the result is a double.
public class MyAvg implements UDAF<Pair<Integer, Integer>, Integer, Double>
 {
```

```
// Here we initialize the aggregate and return its initial value (sum = 0
 and count = 0).
@Override
public Pair<Integer, Integer> init() { return Pair.of(0, 0); }

// Here we update the sum and count values in the aggregate and return the
 updated aggregate
@Override
public Pair<Integer, Integer> add(Pair<Integer, Integer> agg, Integer val) {
       return Pair.of(agg.getKey() + val, agg.getValue() + 1);
   }

// Here we return the value of the sum divided by the count which is the
 average of the aggregated values.
   @Override
   public Double result(Pair<Integer, Integer> agg) {
       return (double) agg.getKey() / agg.getValue();
   }
}
```

Creating UDFs

User Defined Functions (UDFs) allow you to do simple transformations on event streams. This is used in the Projection processor.

Procedure

1. Create a UDF by implement the following interface:

```
public interface UDF<O, I> {
   O evaluate(I i);
}
```

   Where:

   - I – Is the input type.
   - O – Is the output type.
   - The evaluate method is invoked with the corresponding field value for each event in the stream.

2. For functions that accept two or more parameters, the there are corresponding UDF interfaces (UDF2 to UDF7).

```
public interface UDF2<O, I1, I2> {
   O evaluate(I1 input1, I2 input2);
}
```

Example

Example 1

The ConvertToTimestampLong UDF is a good example of a new UDF implementation.

Example

Example 2

In this example, you to concatenate the values of two fields of an event. To do this, define a MyConcat function by implementing the UDF2 interface as shown below

```
pubic class MyConcat implements UDF2<String, String, String> {
   public String evaluate(String s1, String s2) {
       return s1.concat(s2);
   }
```

```
}
```

Building Custom Functions

Once you have created a UDAF, create a new maven project and build the .jar files to add to SAM. You can have multiple UDAFs in a single maven project. All of them are bundled into a single jar which can be uploaded.

Procedure

1. Create a new maven project and add streamline-sdk. A sample pom.xml file is provided below.
2. Generate the UDAF .jar file by running:

```
mvn clean install
```

Results

The UDAF .jar file is created and you are ready to upload it to SAM.

Example

Example pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <groupId>test</groupId>
    <version>0.1</version>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>my-custom-functions</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.hortonworks.streamline</groupId>
            <artifactId>streamline-sdk</artifactId>
            <version>0.1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

Uploading Custom Functions to SAM

Once you have created and built the UDAF, upload it to SAM so that it is available in the Aggregate processor.

Procedure

1. From the left-hand menu, select **Configuration**, then **Application Resources**.
2. Click the **UDF** tab. You use the UDF tab to handle both UDFs and UDAFs.
3. Click the **Add** icon to display the **Add UDF**.
4. Supply the following information, and click **Ok**.
   - Name – This is the internal name of the UDAF. This needs to be unique and should not conflict with any of the built in aggregate functions.
   - Display Name – This is what gets displayed in the list of aggregate functions in the Aggregate processor UI.
   - Description – This can be any textual description of the function to assist the user.
   - Type – This should be AGGREGATE for UDAFs, or FUNCTION for UDFs.

- Classname – This is the full qualified class name of the UDAF that gets packaged in the Jar.
- UDF JAR – Browse and select the jar file that you built using the maven project.

Results

Your new UDF or UDAF displays in the list of available functions.