Building Your First Custom Operator in Scala

Created by Allison Nelson, last modified just a moment ago

The Custom Operator framework allows developers to create new functionality for their Alpine workflows. We have provided a set of samples in Scala and Java to showcase some of the things you can do and describe the format an operator takes. To get started writing your own, first we'll go over the structure of a simple example called Column Filter and talk about how to build it for yourself.

This tutorial will focus on building an operator with Spark and Scala. Tutorials for Database and Java operators will come in later tutorials.

The SDK with examples can be found on Alpine's Github page. This tutorial assumes that you have followed the steps in How To Compile and Run the Sample Operators and can build the code on your machine.

For more in-depth information about the available API methods, please refer to the relevant Scaladoc below.

What Version Should I Use?

There are different versions of the Custom Operator Framework that map to different versions of the Alpine software. For more information, see the page here: How to Update Custom Operators With New SDK Changes

Alpine Version	Custom Operator SDK	Samples Branch Name	Spark Version
5.6.1	Version 1.0, Scaladoc, SDK Source, Samples	`branch-5.6.1`	1.3.1
5.7.x	Version 1.3, Scaladoc, SDK Source, Samples	`branch-5.7`	1.5.1
5.8.x	Version 1.4.2, Scaladoc, SDK Source, Samples	`branch-5.8`	1.5.1
5.9.x	Version 1.5.1, Scaladoc, SDK Source, Samples	`branch-5.9`	1.5.1
6.0	Version 1.6, Scaladoc, SDK Source, Samples	`branch-6.0`	1.6

In order to get the correct version of the samples on your machine, follow these commands:

```
git clone https://github.com/AlpineNow/CustomPlugins
git checkout branch-6.0 # replace with the branch name from the table above
```

Table of Contents

Setting Up Your Environment

1. Open the PluginTemplateProject in IntelliJ.
2. Create a package in `PluginTemplateProject/src/main/scala` to hold your operator code. Use your company's naming scheme for packages if you have one -- we called our package `com.mycompany.plugins`
3. Within that package, create a Scala file called `MyColumnFilter`

> ⚠ Please note, no two operators may have the same classpath – that is, the same package name + class name. Be sure that you and your team use distinct naming for each operator to avoid conflicts.

Your code should look like this so far:

```
package com.mycompany.plugins

class MyColumnFilter {

}
```

This is where we'll write the code for our operator. A custom operator needs to implement three classes: the signature, the GUI node, and the runtime. Each of them defines behavior and information that the Alpine workflow engine will use to execute the operator.

We'll go through these classes one at a time.

The first one to implement is the Signature class. This contains metadata about your operator and where it will show up in Alpine.

Signature

Edit the line that says:

```
class MyColumnFilter {
```

to say:

```
class MyColumnFilterSignature extends OperatorSignature [MyColumnFilterGUINode, MyColumnFilterRuntime] {
```

You will need to add the corresponding import statement for `OperatorSignature`:

```
import com.alpine.plugin.core.OperatorSignature
```

Let's break down what this means:

- Our new class extends `OperatorSignature`, which is the superclass implementing the core functionality of the operator signature.
- We pass `MyColumnFilterGUINode` and `MyColumnFilterRuntime` as type parameters. These are the other two pieces we'll be implementing next.

Within the Signature, we need to override one method: `getMetadata()`. This is the only method within this class. The parameters you will fill in with details about your operator.

- **Name:** the name of the operator
- **Category:** the category the operator will appear in within the Alpine application
- **Author:** the author of this operator
- **Version:** the current version of this operator
- **HelpURL:** if you have a link to a page with more information about your operator, place it here
- **IconNamePrefix:** by default this is blank. This allows you to use a custom icon for your operator. Place the icon images in the resources folder and supply the name of the file without the extension (for example, if your icon is '`test.png`', enter '`test`'.)

```
def getMetadata(): OperatorMetadata = {
 new OperatorMetadata(
  name = "My Column Filter",
  category = "Samples",
  author = "Jane Doe",
  version = 1,
  helpURL = "",
  iconNamePrefix = ""
 )
}
```

Constants Object

Before we go any further, let's set up a Constants object to keep some values for us. By doing this, we can refer to variables across the operator and it makes it easier to maintain and update.

The object is pretty simple:

```
object OperatorConstants {
    val parameterID = "parameterId"
}
```

You can add other variables you want to track here as well. To reference them in our code, you can just call `OperatorConstants.parameterID`. You'll see this throughout the rest of the code in this tutorial.

GUI Node

The next thing we will do is implement the GUI Node. This represents the design-time configuration. It defines configuration parameters that will be shown to the user, describes input/output schemas, and can be used to design visualizations for the results.

All `GUINode` implementations extend `OperatorGUINode`. `OperatorGUINode` has three functions that are important.

1. **`onPlacement:`** This is called at the moment you place your operator on the canvas.
2. **`onInputOrParameterChange:`** If you drag a new “arrow” to this operator or change the parameters in the GUI, this function is called.
3. **`onOutputVisualization:`** This configures what appears in the results console when the operator finishes running.

You can implement all three of these manually, or we have provided templates to wrap some of this functionality for you. For this tutorial we’re going to extend `SparkDataFrameGUINode`.

We’ll add another class, this time called `MyColumnFilterGUINode` that will extend `SparkDataFrameGUINode`.

Add the code:

```
class MyColumnFilterGUINode extends SparkDataFrameGUINode [MyColumnFilterJob] {
}
```

`MyColumnFilterJob` is the name of our Spark job, which we will implement later in this tutorial. The `SparkDataFrame` template makes it easy to create a Spark transformation that takes in a dataset and outputs a dataset. It provides a variety of convenience functions to make it easier to create your operator.

The `onPlacement()` method

Whether you are using the Spark data frame template or not, you will need to override the `onPlacement()` method. This describes the `OperatorDialog`’s parameters in the GUI. The other two methods, `onInputorParameterChange` and `onOutputVisualization`, will be implemented in a future tutorial. They are

not needed for this basic example.

The `onPlacement()` method looks like:

```
override def onPlacement(operatorDialog: OperatorDialog,
    operatorDataSourceManager: OperatorDataSourceManager,
    operatorSchemaManager: OperatorSchemaManager): Unit = {
   // code for operatorDialog goes here
}
```

Building the Operator Dialog

We'll be using the `operatorDialog` object to define parameters for our operator. These can be textboxes, radiobuttons, or any other basic form element.

For all parameters, we'll always define the ID of the parameter (a string that you will use to refer to the parameter in the code) and the label. The label is what shows up to the user on the configuration options for the operator. The rest of the options vary depending on the parameter.

For our Column Filter example, we want to build a checklist of the columns that the user can filter by. To do this, we'll utilize the `addTabularDatasetColumnCheckboxes` method. It looks like this:

```
operatorDialog.addTabularDatasetColumnCheckboxes(
 OperatorConstants.parameterID,      // the ID of the operator
 "Columns to keep",                   // the label of the operator
                                      // (user-visible)
 ColumnFilter.All,                       // this means users can select
                                      // all of the columns but this
                                      // can also be changed to allow
                                      // for only numeric/categorical
 "main"                               // this is the selectionGroupID,
                                      // which is used for validating
                                      // groups of parameters
)
```

Since we're using the `SparkDataFrame` template, we can take advantage of their storage configuration by calling

```
super.onPlacement(operatorDialog, operatorDataSourceManager, operatorSchemaManager)
```

after the `operatorDialog` customizations. This calls the superclass (`SparkDataFrameGUINode`) and applies some default storage and parameter configurations.

Defining the Output Schema

To ensure our output displays in a consistent fashion, we need to define the output schema for our data frame. This has to be set in both the runtime and the GUI node class, and they must match. The GUI node needs to know about the output schema in order to properly format any output visualization and to let subsequent operators know what this operator's dataset schema looks like.

To define the output schema, override the `defineOutputSchemaColumns` method from `SparkDataFrameGUINode`. It takes in the input schema and the parameters, and returns a `Seq` containing `ColumnDefs`. In our case, it will contain a `Seq` containing all of the rows the user selected to filter by.

```
override def defineOutputSchemaColumns(inputSchema: TabularSchema,
                                       parameters: OperatorParameters): Seq[ColumnDef] = {
  val columnsToKeep = parameters.getTabularDatasetSelectedColumns(OperatorConstants.parameterID)._2
  inputSchema.getDefinedColumns.filter(colDef => columnsToKeep.contains(colDef.columnName))
 }
```

The first line pulls the column names selected from the `TabularDatasetColumnCheckbox` parameter defined in the `operatorDialog`. The second line filters the available columns by the selected column names and returns those columns as the output schema.

Your code should now look like:

```
class MyColumnFilterGUINode extends SparkDataFrameGUINode[MyColumnFilterJob]{
 override def onPlacement(operatorDialog: OperatorDialog,
                          operatorDataSourceManager: OperatorDataSourceManager,
                          operatorSchemaManager: OperatorSchemaManager): Unit = {

  operatorDialog.addTabularDatasetColumnCheckboxes(
 OperatorConstants.parameterID,      // the ID of the operator
 "Columns to keep", // the label of the operator (user-visible)
 ColumnFilter.All,     // this means users can select all of the columns
                   // but this can also be changed to allow for only
                   // numeric or categorical columns
 "main"            // this is the selectionGroupId,
                   // which is used for validating groups of parameters
  super.onPlacement(operatorDialog, operatorDataSourceManager, operatorSchemaManager)
 }


 override def defineOutputSchemaColumns(inputSchema: TabularSchema,
                                        parameters: OperatorParameters): Seq[ColumnDef] = {
```

```
    val columnsToKeep = parameters.getTabularDatasetSelectedColumns(OperatorConstants.parameterID)._2
    inputSchema.getDefinedColumns.filter(colDef => columnsToKeep.contains(colDef.columnName))
  }
}
```

Runtime

Now that you have created a signature and a GUI node, let's move on to the runtime. This is where the meat of the operator occurs -- you'll be defining the Spark job that performs the data transformation here. While you could extend the base class `OperatorRuntime` and define output steps manually, we'll be using `SparkDataFrameRuntime`. This allows us to easily write a Spark job that performs a data transformation and uses a set of predefined methods on the backend to package the results and return them to the Alpine application.

We'll call this `MyColumnFilterRuntime` and extend `SparkDataFrameRuntime`, passing in our Spark job `MyColumnFilterJob` as a type parameter. For our Column Filter operator, we'll be submitting a Spark job called `MyColumnFilterJob`. To use the default implementation which launches a Spark job according to your default Spark settings, you will not need to add any code to the body of the `MyColumnFilterRuntime` class. The code should look like this:

```
class MyColumnFilterRuntime extends SparkDataFrameRuntime[MyColumnFilterJob] {

}
```

Finally, we need to implement the Spark job that we referenced in the previous class. Create a class called `MyColumnFilterJob` and extend `SparkDataFrameJob`. Override the `transform()` method to start writing your Spark code.

Here's an example of what this looks like:

```
class MyColumnFilterJob extends SparkDataFrameJob {
 override def transform(parameters: OperatorParameters,
         dataFrame: DataFrame,
         sparkUtils: SparkRuntimeUtils,
         listener: OperatorListener): DataFrame = {

         }
}
```

The parameters are passed from the parameters the user selects at the design time steps. Once the operator starts running, the parameter information is passed to the runtime for us. The parameter `dataFrame` is the input, and the function should return a data frame when the code completes.

To perform the Column Filter operation, we need to grab a list of the parameters the user selected then return a data frame with only those columns included. First, we'll access the user-selected columns using `parameters.getTabularDatasetSelectedColumns(OperatorConstants.parameterID)`.

`OperatorConstants.parameterID` is the value referenced in the Constants class. It points to the columns that you chose in the `OperatorDialog` step. This will return a collection of column names that were selected. After that, we apply Spark's `col()` method across the collection of column names. In code, you'll write:

```
class MyColumnFilterJob extends SparkDataFrameJob {

 override def transform(parameters: OperatorParameters,
                        dataFrame: DataFrame,
                        sparkUtils: SparkRuntimeUtils,
                        listener: OperatorListener): DataFrame = {
   // grab the selected column names
   val columnNamesToKeep = parameters.getTabularDatasetSelectedColumns(OperatorConstants.parameterID)._2
   // returns the columns from the DF containing the column names
   val columnsToKeep = columnNamesToKeep.map(dataFrame.col)
   // returns a data frame with only the selected columns included
   dataFrame.select(columnsToKeep: _*)

 }
}
```

When Your Operator Runs

When the operator is executed, a dataframe with the selected columns is returned from the Spark job. After the job finishes, the `SparkDataFrameJob` class saves the results as a file using the storage parameters defined in the `onPlacement()` method. If you did not specify, this defaults to storing on HDFS. This information is then passed to the Alpine plugin engine so that your results can be visualized.

For basic Spark transformations using `SparkDataFrameGUINode`, the result console output is automatically configured to show a preview of the output table.

Preparing Your Operator to Build

Once you have completed your code, there's only a few things left to do before uploading it to the Alpine web application. You'll need to edit two more files: `pom.xml` and `plugins.xml`. You can also customize the icon that will be displayed for your operator at this point.

First, find the `pom.xml` file in the `PluginTemplateProject` folder. Find the line that says:

```
<artifactId>plugin-template</artifactId>
```

and replace it with the name you want the compiled jar file to have. That is the only thing you need to modify in this file. The other code in `pom.xml` does the work of downloading dependencies based on Alpine's Custom Operator SDK.

So, to call your output jar `MyColumnFilter.jar`, you would edit the `artifactId` line to say:

```
<artifactId>MyColumnFilter</artifactId>
```

When you update your `pom.xml` file, IntelliJ may show a pop-up saying to import changes to Maven Projects. Before proceeding, click 'Import Changes', as it will refresh the dependencies needed to build your operator.

Now you need to specify the signature class to the `plugins.xml` file. This will tell Alpine where to find information about the operator after uploading the JAR. Open `PluginTemplateProject/src/main/resources/plugins.xml`

It should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<alpine-plugins>
</alpine-plugins>
```

You only need to add the Signature class (the one with the metadata) to this file. Add it like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<alpine-plugins>
 <plugin>
   <signature-class>com.mycompany.plugins.MyColumnFilterSignature</signature-class>
 </plugin>
</alpine-plugins>
```

Notice that you'll need to use the full package name to refer to your signature class.

⚠ Don't forget to fill in those two fields, or your operator won't build!

Finally, follow the instructions listed in How To Compile and Run the Sample Operators to build your operators and upload them to the Alpine web application.

At this point you should have a fully functioning Column Filter operator, using the new Custom Operator framework in Spark and Scala.