SPL Examples for Beginners

This package (SPL-Examples-For-Beginners.tar.gz) includes 50 different Streams Studio projects. These are simple examples that will help developers who are just beginning to wrap their minds around IBM Streams Processing Language (SPL). Along with a great number of Streams SPL programming documentation files and advanced examples shipped with the Streams product, beginners can use this package of examples as an additional learning aid.

All the examples listed below are compact and every example focuses on a particular aspect of the SPL feature set. These individual project directories can be imported directly into Eclipse-based Streams Studio development tool and executed from there. A procedure to import them into Streams Studio is explained at the end of this file.

The goal here is to make the beginners get their feet wet via these simple SPL examples that produce verifiable results.

[These examples were developed and tested using the Streams Studio IDE installed on top of Eclipse 3.7.1 Indigo running on 64 bit Red Hat Enterprise Linux 5.3 or higher.]

The following is a catalog of all the examples with a short description about what is covered in each one of them.

Use the Eclipse InfoSphere Streams perspective to work with the following projects.

001_hello_world_in_spl

This example is the simplest possible SPL application. It uses a Beacon operator to generate tuples that carry "Hello World" messages. A custom sink operator receives the tuples from Beacon and displays it on the console.

002_source_sink_at_work

This example shows how a FileSource operator can be used to read CSV formatted records from a file and then receive those tuples in a FileSink to be written to a file in the data directory of this application.

003_sink_at_work

This example shows how FileSink and Custom sinks can be employed in applications. It also shows how a Beacon operator can be used to customize tuple attributes. In addition, it introduces the Filter operator to route the incoming tuples by inspecting their attributes using a conditional statement specified in the filter parameter.

004_delay_at_work

This example shows how a Delay standard toolkit operator can be used to delay a stream. This example also introduces the Custom operator that can be used to perform custom logic. You can also notice the use of a state variable that is mutable inside the Custom operator. It also shows how to create a new tuple on the fly and do your own submissions onto the output ports.

005_throttle_at_work

This example shows how a stream can be throttled to flow at a specified rate. This example also mixes other operators such as Beacon, Custom, and FileSink.

006_barrier_at_work

This example shows how to synchronize the incoming tuples using a Barrier operator. It uses a bank deposit/debit scenario to split the deposit/debit requests, perform that account activity, and then combine the post-activity result with the incoming requests. Barrier operator does what is needed to accomplish that i.e. it waits for the streams to arrive at all the configured input ports before emitting an output tuple.

007_split_at_work

This example shows how a Split operator can be used to split the incoming tuples based on a key. In this example, the split condition (which tuples comes out on which port) is pre configured through a text file. Alternatively, one can compute the index of the output port on the fly inside the Split operator parameter section.

008_get_submission_time_value

This example shows how the tuple attributes can be assigned values that were supplied by the user at the application/job submission time. It employs the getSubmissionTimeValue function to obtain different values made of different SPL data types.

009_custom_operator_using_get_submission_time_value

This example demonstrates how to assign tuple attributes at the time of job submission inside a custom operator. When the incoming tuples arrive at the Custom operator in this example, values entered by the user at the application startup are assigned to the tuple attributes.

SPL Examples for Beginners

010_get_compiler_time_value

This example shows how arguments supplied during the application compile time can be accessed inside of the SPL applications. In Streams Studio, you can enter these values by editing the active build configuration. Additional SPL compiler options can be added in the resulting build configuration dialog.

011_compiler_intrinsic_functions

Streams compiler provides several intrinsic functions to query the SPL filename, file path, absolute path of the directory, source code line number, composite instance name etc. This example shows the use of the compiler intrinsic functions inside of a Functor operator.

012_filter_functor_at_work

This example puts the two commonly used standard toolkit operators to work. They are Filter and Functor. Filter allows you to route tuples based on conditional checks. It provides two output ports to send the matched tuples on the first output port and the unmatched tuples on the second output port. Functor operator allows us to transform the incoming tuple attributes and then to send it on many different output ports with different stream schemas.

013_punctor_at_work

This example shows how a Punctor operator could be used in an application. Punctor operator allows us to transform the input tuples and then inject puncuation markers either before or after the output tuple as configured.

014_sort_at_work

This example shows the use of the Sort operator in the context of an application. Sort operator is highly configurable with all kinds of windowing support. In this example, the following window configurations are applied for sorting the incoming tuples.

a) Count-based tumbling window.
b) Time-based tumbling window.
c) Punctuation-based tumbling window.
d) Delta-based tumbling window.
e) Count-based sliding window.

015_join_at_work

This example shows one of the power-packed standard toolkit operators; i.e. Join. This operator is so versatile that it is hard to do justice in explaining it thoroughly in a simple example such as this one. This example provides coverage to the following Join operator features.

a) Inner Join
b) Inner (Equi) Join
c) Left Outer Join
d) Right Outer Join
e) Full Outer Join

016_aggregate_at_work

This example shows off yet another powerful standard toolkit operator named the Aggregate. It is very good in computing on the fly aggregate values after collecting a set of tuples. Tuples are grouped based on tumbling and sliding windows with partitioned variants. This example also shows how to use the built-in assignment functions provided by this operator to compute regular statistical calculations such as min, max, average, standard deviation etc.

017_filesource_filesink_at_work

We have used the FileSource and the FileSink operators in other examples before. However, this example shows off the following intriguing features that will become handy in a lot of practical situations.

a) Automatic deletion of a file after the FileSource finishes reading all the records.
b) Flushing the sink file on demand after writing a certain number of tuples.
c) Ability of the FileSource to move the file once it reads all the content in that file.
d) Creating a fresh and new output sink file after writing a certain number of tuples.
e) Ability of the FileSource to keep reading from a hot file as new CSV records get written to the end of that file.

018_directory_scan_at_work

This example demonstrates one of the important features desired in the real world (mostly in the Retail banking and in the Telco industries). In many real-world scenarios, they still work via files and such files get dropped into a directory for processing. It is shown here how the DirectoryScan operator picks up a new file as soon as it appears inside an input directory. (Apply caution if huge files are copied to the watch directory. DirectoryScan may detect that big file copy as multiple new files and output multiple tuples with the same file name.)

019_import_export_at_work

This example demonstrates how two different SPL applications can share streams between them. This is an important feature that is elegantly done using two pseudo operators called Export and Import. This application also shows how two different main composites can be part of the same application by using two different namespaces. As an aside, there is also a demonstration of using a Custom operator to customize the Beacon generated tuples by involving state variables.

020_metrics_sink_at_work

This example shows how one can use the MetricsSink standard toolkit operator to create application-specific custom metrics that can be viewed in real-time when the application is running. Viewing of custom metrics is typically done inside Streams Explorer view of the Streams Studio or by using the capturestate option in streamtool.

021_pair_at_work

This example shows off the Pair operator that is used for pairing tuples arriving on different input ports. Only when all the tuples arrive at all the input ports, this operator will emit them one after the other in their order of arrival.

022_deduplicate_at_work

This example describes the use of an important operator that is highly applicable in many Telco scenarios. That operator is called DeDuplicate, which eliminates duplicate tuples for a specified duration of time. It also has an optional second output port on which duplicate tuples could be sent out for additional processing.

023_union_at_work

This example demonstrates an utility operator called Union. This operator combines all the tuples from several input ports as they arrive and emits a single output stream. All the input ports must have a schema that contains attributes of the same name and type as those of the output port. The order of the attributes in the input ports need not match the order in the output port.

024_threaded_split_at_work

This example demonstrates an important standard toolkit operator named ThreadedSplit. It is a multi-threaded split that is different from the other content-based Split operator. ThreadedSplit uses its own algorithm to split the incoming tuples to the available output ports to improve concurrency. This will speed up the distribution of tuples by using individual threads assigned to each of the output ports.

025_dynamic_filter_at_work

This example deals with an interesting standard toolkit operator called DynamicFilter. This operator is a special version of the Filter operator that you have already seen in another example; it decides at runtime which input tuples will be passed through, based on the control input it receives. This operator is applicable in many real-life scenarios. This example also demonstrates using a second composite operator to perform a sub-task that the main composite will make use of. There is also coverage to show how the second composite can take its own operator parameters.

026_gate_at_work

This is an example that uses the Gate operator from the standard toolkit. This operator delays the incoming tuples until a downstream operator signals with an acknowledgment to receive any further tuples. This is a great way to have a feedback through which we can control the rate at which tuples are passed through. (Please refer to another example named 905_gate_load_balancer that shows the effectiveness of the Gate operator in combination with the ThreadedSplit operator to provide load balancing the incoming tuples.)

027_java_op_at_work

This example shows an important operator that brings Java into the C++ dominated world of Streams!!! That operator is called JavaOp, which is used to call out to other operators implemented in Java using the Java Operator API. In this example, we will have a tiny Java logic that will calculate the current time and add that time string to a tuple attribute and output that tuple. There is another example that shows the Java primitive operator that is different from the JavaOp operator.

028_multiple_composites_at_work

This example shows the use of multiple composites in a single application. There is a main composite that in turn uses two other composites. This application shows how the additional composites in different namespaces get included into the main composite via the "use" directive. It also demonstrates how the additional composites can accept their own operator parameters. It teaches the basics of an important feature that will come handy when big applications need to be componentized.

029_spl_functions_at_work

This example shows how helper and utility functions can be written using the SPL language. It also shows how such SPL functions can be put to use inside the context of an application. Learning this simple concept will go a long way in doing a lot of neat stuff in real-world applications.

030_spl_config_at_work

This example introduces one of the must-learn features of the SPL language. SPL language offers an extensive list of options to do configuration at the operator level as well as at the composite level. This application attempts to sprinkle many of the available configuration parameters as shown below.

a) host
b) hostColocation
c) partitionColocation
d) placement
e) threadedPort and queue
f) relocatable and many more.

In addition, this example shows how to make this application toolkit dependent on another (025_dynamic_filter_at_work) SPL toolkit project.

031_spl_mixed_mode_at_work

This example shows a cool SPL feature called mixed-mode support. In this, developers can mix PERL code islands inside of an SPL application. Mixed-mode enables the easy parameterization of SPL applications. This example gives a slight flavor of how a PERL code snippet inter-mixed with SPL allows us to parameterize the SPL Stream names and the number of output stream definitions for an SPL operator.

032_native_function_at_work

This application shows how native functions written in C++ can be called within an SPL application. Before working through this example, you may want to install the Eclipse CDT plug-in (C Development Tool) from the CDT update site (http://www.eclipse.org/cdt/downloads.php).

There are two ways in which native functions can be written in C++.

1) Code for the C++ functions can be written in a C++ header file.

2) C++ functions can be written outside of the SPL project and packaged into a shared library (.so) file. All the SPL developer will have to work with are an .so file and a C++ header file.

This application demonstrates incorporating native functions built in both of those ways.

[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED NativeFunctionLib THAT IS DESCRIBED BELOW.]

033_java_primitive_operator_at_work

This example shows how a Java primitive operator is created from scratch. Java primitive operator is different from JavaOp that you have seen earlier in a different example. Java primitive operator is a first class operator in SPL, whereas JavaOp only permits a callout to another Java operator. In addition, Java primitive operator has the advantage of keeping its name as the operator's runtime instance name.

[THIS EXAMPLE HAS A COMPANION JAVA PROJECT NAMED RSS_Reader_Primitive THAT IS DESCRIBED BELOW.]

034_odbc_adapters_for_db2_at_work

This example shows the use of the three Streams ODBC adapters. Those operators are ODBCSource, ODBCAppend, and ODBCEnrich. The code in this example is written to access a particular test DB2 database inside IBM. You have to create your own DB2 database and tables to make this application work in your environment. After creating your own database and tables, you have to change the etc/connections.xml file in this application's directory to match your database/table names, userid, and password. You also have to make changes in the SPL code using your database information for all the three ODBC operator invocations.

035_c++_primitive_operator_at_work

This example shows the steps required to create a C++ primitive operator from scratch. In this application, a C++ primitive operator model XML file can be explored to learn how the different fields in that file are configured. Then, the code generation template header and implementation files (*_h.cgt and *_cpp.cgt) can be browsed to learn about the primitive operator logic. Additionally, this example demonstrates about including a Java operator and a C++ primitive operator as part of the application flow.

036_shared_lib_primitive_operator_at_work

This example demonstrates two important techniques that will be commonly used in real-world use cases.

1) Creating a C++ primitive operator.

2) Calling a function available inside a .so shared library from the C++ primitive operator logic.

Application logic here is to receive input tuples as hostnames and then make the C++ primitive operator logic invoke a shared library function that does a name server lookup.

[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED PrimitiveOperatorLib THAT IS DESCRIBED BELOW.]

037_odbc_adapters_for_solid_db_at_work

This example shows the use of the three Streams ODBC adapters for connecting to a SolidDB in-memory database. Those operators are ODBCSource, ODBCAppend, and ODBCEnrich. The code in this example is written to access a particular test SolidDB database inside IBM. You have to create your own SolidDB database and tables to make this application work in your environment.

After creating your own database and tables, you have to change the ./etc/connections.xml file in this application's directory to match your database/table names, userid, and password. You also have to make changes in the SPL code using your database information for all the three ODBC operator invocations.

038_spl_built_in_functions_at_work

This is a very simple example that showcases a random collection of powerful built-in SPL functions that are available out of the box. This application demonstrates how time, math, and collection type functions can be used inside of an SPL application.

039_application_set_at_work

This example shows how multiple SPL applications can be grouped together so that they can be started, monitored, and stopped together. There is no code that needs to be written to accomplish this grouping.

One can simply create an SPL application set project inside the Streams Studio and then right-click on this project to get a dialog box displaying all possible applications in your workspace eligible to be grouped. You can now select the SPL projects that need to be grouped together. It is important to note that only those applications with Distributed active build configuration are eligible for the application set grouping.

In this example, you will notice that the 019_import_export_at_work and the 030_spl_config_at_work projects are added to the application set. You can now right click on the 039_application_set project and build/start/stop all of its member applications together.

040_ingest_data_generation_in_spl

This example shows how SPL provides rich features to generate synthetic data required for large scale testing. Many real-life applications in the Telco and the Retail Banking sectors consume large amounts of daily business data through CSV formatted text files. There could be huge amounts of CDR data from several telecom circles or daily transaction data for millions of accounts in a retail bank.

While building and testing the SPL applications, it will become necessary to generate such ingest data files with artificial data that is close enough to be realistic. This application shows how such large amounts of data in several thousands of files can be created very quickly using the SPL standard toolkit operators as well as the SPL file IO and math random built-in functions.

Use the Eclipse Java perspective to work with the following projects.

RSS_Reader_Standalone

This is a stand-alone Java project that encapsulates the RSS reader function. Before the RSS reader logic can be added to an SPL Java primitive operator, this stand-alone Java project is used to verify and test the business logic. Once its functionally is working, the core logic is ready to be moved into the process function of an SPL Java primitive operator.

RSS_Reader_Primitive

This is a companion Java project for the SPL project (033_java_primitive_operator_at_work) present in the same Eclipse workspace. This Java class extends from a Streams AbstractOperator class. It receives an input tuple with information about an RSS provider. Then, it queries that RSS server for the currently available RSS feeds. It parses the RSS XML data and converts that into an output tuple and sends it via its output port.

Use the Eclipse CDT perspective to work with the following projects.

NativeFunctionLib

This is a companion project for the 032_native_function_at_work SPL project present in the same Eclipse workspace. This C++ project gets compiled into a shared library (.so) file so that an SPL project can access the native C++ functions available in the .so file. In the SPL project, there is a native function model that will define the location of the .so file, C++ function name and the C++ namespace. The C++ include file, CPP file, and the .so files should be manually copied into the impl directory of the SPL project. Read the commentary in the main composite of the SPL project to learn more about the exact locations where the files from the C++ project files need to be copied.

PrimitiveOperatorLib

This is a companion project for the 036_shared_lib_primitve_operator_at_work SPL project present in the same Eclipse workspace. This C++ project gets compiled into a shared library (.so) file so that the SPL C++ primitve operator can access the C++ functions available in the .so file. In the SPL project, there will be an operator model that will define the location of the .so file, C++ function name and the C++ namespace. The C++ include file, CPP file, and the .so files should be manually copied into the impl directory of the SPL project. Read the commentary in the main composite of the SPL project to learn more about the exact locations where the files from the C++ project files need to be copied.

UsePrimitiveOperatorLib

This is a client a.k.a test driver program that uses a shared library. This stand-alone test application calls a function available inside the PrimitiveOperatorLib shared (.so) library. Before using the shared library inside an SPL C++ primitive operator, it is recommended to test it first using a test driver.

In CDT, there are specific project build properties for a test driver program to link with the .so library. Refer to the commentary at the top of the test driver CPP file.

The following SPL examples are directly taken out of the SPL Introductory Tutorial PDF file. They are also added to the mix inside the same tar.gz file. This package containing a total of 50 examples is targeted to assist the newbie enthusiasts of the InfoSphere Streams SPL language. Please refer to the SPL Tutorial PDF file for a detailed description about the applications listed below.

901_cat_example

902_word_count

903_unique

904_primitive_round_robin_split

905_gate_load_balancer

A **detailed procedure to import the "SPL-Examples-For-Beginners"** into Streams Studio is described below.

1) On a 64 bit Redhat Enterprise Linux 5.3 or above, unzip the file "SPL-Examples-For-Beginners.tar.gz".

```
tar -xvzf SPL-Examples-For-Beginners.tar.gz
```

2) Install the following in your home directory.

a) 64 bit version of InfoSphere Streams
b) 64 bit version of Eclipse 3.7.1 or higher
c) Streams Studio plugins inside of your Eclipse installation
d) 64 bit version of Eclipse CDT (C Development Tool)
   Eclipse CDT update site is at the following URL:
   http://www.eclipse.org/cdt/downloads.php

3) In an xterm window, run the following command to set up the correct Streams environment variables.
source ~/<Your_Streams_Install_Directory>/bin/streamsprofile.sh

4) From that same xterm window, start Eclipse 3.7.1 or higher with Streams Studio and Eclipse CDT installed.

5) If you see the Eclipse Welcome screen, close that and you will be taken to the Java perspective.

6) Change to the "InfoSphere Streams" perspective: Click on "Window->Open Perspective->Other".

7) In the resulting dialog box, select "InfoSphere Streams" and click OK.

8) In the left pane, select the "Streams Explorer" tab.

9) Right click on "Toolkit Locations" and select "Add Toolkit Location".

10) In the resulting dialog box, click the "Directory" button.

11) In the resulting dialog box, navigate until you can see the "<Your_Streams_Install_Directory>/toolkits/com.ibm.streams.db" directory.

12) Select "com.ibm.streams.db" and click OK twice. After this, on the left pane, select the "Project Explorer" tab.

13) Let us now import the "SPL-Examples-For-Beginners" project artifacts. Select "File->Import".

14) In the resulting dialog box, expand "General" and select "Existing Projects into Workspace". Click Next.

15) In the next dialog box, click on "Select root directory" and then click on the "Browse" button.

16) Navigate until you can see the "SPL-Examples-For-Beginners" directory.

17) Now, select the "SPL-Examples-For-Beginners" directory and click OK.

18) In the next dialog box, all projects in the "SPL-Examples For-Beginners" directory should be selected.

19) Ensure that an option named "Copy projects into workspace" is selected (It is located below the list of projects).

20) Click "Finish".

21) You should now see all the SPL, Java, and C++ projects getting imported into your workspace.

22) Shortly after the import finishes, all the SPL projects a.k.a toolkits will be automatically indexed and built. You can watch the "SPL Build" console to monitor the progress of the SPL projects that are being built.

23) Approximately after 15 minutes, all the SPL and Java projects will be in "Fully built" state.

24) However, three C++ projects (NativeFunctionLib, PrimitiveOperatorLib, UsePrimitiveOperatorLib) may not build correctly.

Let us resolve the errors in one C++ project at a time.

i) NativeFunctionLib

a) Right click on the NativeFunctionLib project and select "Properties".

b) In the resulting dialog, expand "C/C++ Build" in the left pane and click on "Settings".

c) At the top right of the right pane, click on "Manage Configurations" button.

d) In the resulting dialog box, select "Release", click "Set Active" button, and then click OK.

e) At the top left of the right pane, click on the "Configuration" combo box and select "Release [Active]".

f) In the right pane, expand the "GCC C++ Compiler" section and select "Miscellaneous".

g) In the "Other flags" field, you have to append the resulting text by executing the following command in a terminal window.

```
$STREAMS_INSTALL/bin/dst-pe-pkg-config.sh --cflags dst-spl-pe-install
```

h) Click "Apply" and then the OK button.

ii) PrimitiveOperatorLib
   a) Right click on the PrimitiveOperatorLib project and select "Properties".
   b) In the resulting dialog, expand "C/C++ Build" in the left pane and click on "Settings".
   c) At the top right of the right pane, click on "Manage Configurations" button.
   d) In the resulting dialog box, select "Release", click "Set Active" button, and then click OK.
   e) At the top left of the right pane, click on the "Configuration" combo box and select "Release [Active]".
   f) Click "Apply" and then the OK button.

iii) UsePrimitiveOperatorLib
   a) Right click on the UsePrimitiveOperatorLib project and select "Properties".
   b) In the resulting dialog, expand "C/C++ Build" in the left pane and click on "Settings".
   c) At the top right of the right pane, click on "Manage Configurations" button.
   d) In the resulting dialog box, select "Release", click "Set Active" button, and then click OK.
   e) At the top left of the right pane, click on the "Configuration" combo box and select "Release [Active]".
   f) Click "Apply" and then the OK button.

25) At this point, you have imported all 50 examples into your Eclipse IDE. What comes next? It is time now to immerse in Streams. A popular spot to begin your deep dive is 001_hello_world_in_spl project!!!

HAPPY SPLASHING IN STREAMS