streamsx Documentation

Release 1.14.11

IBMStreams

Mar 17, 2020

CONTENTS

1 Python Application API for Streams **3**
- 1.1 streamsx.topology . . . 3
- 1.2 streamsx.topology.topology . . . 6
- 1.3 streamsx.topology.context . . . 37
- 1.4 streamsx.topology.schema . . . 48
- 1.5 streamsx.topology.state . . . 56
- 1.6 streamsx.topology.composite . . . 59
- 1.7 streamsx.topology.tester . . . 62
- 1.8 streamsx.topology.tester_runtime . . . 71
- 1.9 streamsx.ec . . . 71
- 1.10 streamsx.spl.op . . . 77
- 1.11 streamsx.spl.types . . . 89
- 1.12 streamsx.spl.toolkit . . . 93

2 SPL primitive Python operators **95**
- 2.1 streamsx.spl.spl . . . 95

3 Streams Python REST API **111**
- 3.1 streamsx.build . . . 111
- 3.2 streamsx.rest . . . 113
- 3.3 streamsx.rest_primitives . . . 118

4 Scripts **151**
- 4.1 spl-python-extract . . . 151
- 4.2 streamsx-info . . . 152
- 4.3 streamsx-runner . . . 152
- 4.4 streamsx-sc . . . 156
- 4.5 streamsx-service . . . 158
- 4.6 streamsx-streamtool . . . 159

5 Environments **167**
- 5.1 IBM Streaming Analytics service . . . 167
- 5.2 IBM Streams Python setup . . . 170
- 5.3 Indices and tables . . . 172

Python Module Index **173**

Index **175**

Python APIs for use with IBM® Streaming Analytics service on IBM Cloud and on-premises IBM Streams.

CHAPTER

ONE

PYTHON APPLICATION API FOR STREAMS

Module that allows the definition and execution of streaming applications implemented in Python. Applications use Python code to process tuples and tuples are Python objects.

SPL operators may also be invoked from Python applications to allow use of existing IBM Streams toolkits.

See *topology*

streamsx.topology	Python application support for IBM Streams.
streamsx.topology.topology	Streaming application definition.
streamsx.topology.context	Context for submission and build of topologies.
streamsx.topology.schema	Schemas for streams.
streamsx.topology.state	Application state.
streamsx.topology.composite	Composite transformations.
streamsx.topology.tester	Testing support for streaming applications.
streamsx.topology.tester_runtime	Runtime tester functionality.
streamsx.ec	Access to the IBM Streams execution context.
streamsx.spl.op	Integration of SPL operators.
streamsx.spl.types	SPL type definitions.
streamsx.spl.toolkit	SPL toolkit integration.

1.1 streamsx.topology

Python application support for IBM Streams.

1.1.1 Overview

IBM® Streams is an advanced analytic platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of real-time sources. Streams can handle very high data throughput rates, millions of events or messages per second.

With this API Python developers can build streaming applications that can be executed using IBM Streams, including the processing being distributed across multiple computing resources (hosts or machines) for scalability.

IBM Streams is also available on IBM Cloud through *IBM Streaming Analytics service*

1.1.2 Creating Applications

Applications are created by declaring a flow graph contained in a `Topology` instance.

For details see *`streamsx.topology.topology`*.

1.1.3 Extensions

This package (*streamsx*) provides the core functionality to build streaming applications in Python for Streams.

Additional *streamsx.** packages are available that provide adapters to external systems, analytics and streaming primitives. This include:

- Apache Kafka integration - streamsx.kafka
- Database integration - streamsx.database
- Geospatial analytics- streamsx.geospatial
- IBM Event Streams integration - streamsx.eventstreams
- MQTT integration - streamsx.mqtt
- Cloud Object Storage integration - streamsx.objectstorage
- Streaming primitives - streamsx.standard

A full list of available packages is at : https://pypi.org/search?q=streamsx

1.1.4 Microservices

Publish-subscribe provides the abiltity to connect streams between independent IBM Streams applications regardless of their implementation language. This allows a microservice approach where a Streams application acting as a service publishes one or more streams. Subscriber services then subscribe to those streams without requiring any knowledge of how a stream is published.

Publish-subscribe overview

Applications can publish streams to a topic name which can then be subscribed to by other applications (or even the same application). Publish-subscribe works across applications written in SPL and those written using the Java/Scala and Python application APIs.

A subscriber matches a publisher if their topic filter matches a publisher's topic name and the stream type (schema) is an exact match to that of the publisher. It is recommended that a single stream type is used for a topic name.

A topic is a string value (encoded with UTF-8), based upon the MQTT topic style

Topic names for publishing a stream:

- Must be at least one character long.
- Use / as a level separator, zero length topic levels are valid.
- Must not include wild card characters + and #.
- Must not include the Unicode character NULL (U+0000).

Topic filters for subscribing to streams:

- Must be at least one character long.

- Use / as a level separator.
- Must not include the Unicode character NULL (U+0000).
- + is a single-level wildcard character that can be used at any level, but it must occupy the entire level. +, *a/b/+*, *+/b/+* and *+/b* are valid but *a/b/c+* is not valid.
- # is a wildcard character that matches any number of levels including the parent and any number of child levels. The multi-level wildcard character must be specified either on its own or following a topic level separator. In either case it must be the last character specified in the topic filter. # and 'a/b/#' are valid but *a/b/c#* and *a/#/c* are not valid.

Without a wildcard character a topic filter is an exact match for a topic name so that filter *a/b/c* only matches *a/b/c*.

Single-level filter (+) match examples are:

- filter + matches *a* and *b* but not *a/b*
- filter *a/+* matches *a/b*, *a/c* and *a/* but not *a*, *b/c* or *a/b/c*
- filter +/+ matches *a/b*, *b/c*, *d/* and */* but not *a* or *a/b/c*

Multi-level filter (#) match examples are:

- filter # matches every topic name such as *a*, *b/c*, *//*
- filter *a/b/#* matches *a/b* (parent), *a/b/c*, *a/b/d* and *a/b/c/d*

Note: A publish-subscribe match requires the stream type to match as well as the topic filter matching the topic name.

Publish-subscribe is a many to many relationship, any number of publishers can publish to the same topic and stream type, and there can be many subscribers to a topic.

For example a telco ingest microservice/application may process Call Detail Records from network switches and publish processed records on multiple topics, `cdr/voice/normal`, `cdr/voice/dropped`, `cdr/sms`, etc. by publishing each processed stream with its own topic. Then a dropped call analytic microservice would subscribe to the `cdr/voice/dropped` topic.

Publish-subscribe is dynamic, using IBM Streams dynamic connections, an application can be submitted that subscribes to topics published by other already running applications. Once the new application has initialized, it will start consuming tuples from published streams from existing applications. And any stream the new application publishes will be subscribed to by existing applications where the topic and stream type matches.

An application only receives tuples that are published while it is connected, thus tuples are lost during a connection failure.

A Python application publishes streams using *`publish()`* and subscribes using *`subscribe()`*.

A stream of *`Python tuples`* can only be subscribed to by Python Streams applications. All other types (schemas) can be subscribed to by any Streams application.

Module contents

1.2 streamsx.topology.topology

Streaming application definition.

1.2.1 Overview

IBM Streams is an advanced analytic platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of real-time sources. Streams can handle very high data throughput rates, millions of events or messages per second.

With this API Python developers can build streaming applications that can be executed using IBM Streams, including the processing being distributed across multiple computing resources (hosts or machines) for scalability.

1.2.2 Topology

A *Topology* declares a graph of *streams* and *operations* against tuples (data items) on those streams.

After being declared, a Topology is submitted to be compiled into a Streams application bundle (sab file) and then executed. The sab file is a self contained bundle that can be executed in a distributed Streams instance either using the Streaming Analytics service on IBM Cloud or an on-premise IBM Streams installation.

The compilation step invokes the Streams compiler to produce a bundle. This effectively, from a Python point of view, produces a runnable version of the Python topology that includes application specific Python C extensions to optimize performance.

The bundle also includes any required Python packages or modules that were used in the declaration of the application, excluding ones that are in a directory path containing `site-packages`.

The Python standard package tool `pip` uses a directory structure including `site-packages` when installing packages. Packages installed with `pip` can be included in the bundle with *add_pip_package()* when using a build service. This avoids the requirement to have packages be preinstalled in cloud environments.

Local Python packages and modules containing callables used in transformations such as *map()* are copied into the bundle from their local location. The addition of local packages to the bundle can be controlled with *Topology.include_packages* and *Topology.exclude_packages*.

The Streams runtime distributes the application's operations across the resources available in the instance.

Note: *Topology* represents a declaration of a streaming application that will be executed by a Streams instance as a *job*, either using the Streaming Analytics service on IBM Cloud or an on-premises distributed instance. *Topology* does not represent a running application, so an instance of *Stream* class does not contain the tuples, it is only a declaration of a stream.

1.2.3 Stream

A *Stream* can be an infinite sequence of tuples, such as a stream for a traffic flow sensor. Alternatively, a stream can be finite, such as a stream that is created from the contents of a file. When a streams processing application contains infinite streams, the application runs continuously without ending.

A stream has a schema that defines the type of each tuple on the stream. The schema for a stream is either:

- *Python* - A tuple may be any Python object. This is the default when the schema is not explictly or implicitly set.
- *String* - Each tuple is a Unicode string.
- *Binary* - Each tuple is a blob.
- *Json* - Each tuple is a Python dict that can be expressed as a JSON object.
- Structured - A stream that has a structured schema of a ordered list of attributes, with each attribute having a fixed type (e.g. float64 or int32) and a name. The schema of a structured stream is defined using typed named tuple or *StreamSchema*.

A stream's schema is implictly dervied from type hints declared for the callable of the transform that produces it. For example *readings* defined as follows would have a structured schema matching `SensorReading`

```
class SensorReading(typing.NamedTuple):
    sensor_id: str
    ts: int
    reading: float

def reading_from_json(value:dict) -> SensorReading:
    return SensorReading(value['id'], value['timestamp'], value['reading'])

topo = Topology()
json_readings = topo.source(HttpReadings()).as_json()
readings = json_readings.map(reading_from_json)
```

Deriving schemas from type hints can be disabled by setting the topology's `type_checking` attribute to false, for example this would change *readings* in the previous example to have generic Python object schema *Python*

```
topo = Topology()
topo.type_checking = False
```

1.2.4 Stream processing

Callables

A stream is processed to produce zero or more transformed streams, such as filtering a stream to drop unwanted tuples, producing a stream that only contains the required tuples.

Streaming processing is per tuple based, as each tuple is submitted to a stream consuming operators have their processing logic invoked for that tuple.

A functional operator is declared by methods on *Stream* such as *map()* which maps the tuples on its input stream to tuples on its output stream. *Stream* uses a functional model where each stream processing operator is defined in terms a Python callable that is invoked passing input tuples and whose return defines what output tuples are submitted for downstream processing.

The Python callable used for functional processing in this API may be:

- A Python lambda function.
- A Python function.
- An instance of a Python callable class.

For example a stream `words` containing only string objects can be processed by a *`filter()`* using a lambda function:

```
# Filter the stream so it only contains words starting with py
pywords = words.filter(lambda word : word.startswith('py'))
```

When a callable has type hints they are used to:

- define the schema of the resulting transformation, see *Stream*.
- type checking the correctness of the transformation at topology declaration time.

For example if the callable defining the source had type hints that indicated it was an iterator of `str` objects then the schema of the resultant stream would be *`String`*. If this source stream then underwent a *`Stream.map()`* transform with a callable that had a type hint for its argument, a check is made to ensure that the type of the argument is compatible with `str`.

Type hints are maintained through transforms regardless of resultant schema. For example a transform that has a return type hint of `int` defines the schema as *`Python`*, but the type hint is retained even though the schema is generic. Thus an error is raised at topology declaration time if a downstream transformation uses a callable with a type hint that is incompatible with being passed an `int`.

How type hints are used is specific to each transformation, such as *`source()`*, *`map()`*, *`filter()`* etc.

Type checking can be disabled by setting the topology's `type_checking` attribute to false.

When a callable is a lambda or defined inline (defined in the main Python script, a notebook or an interactive session) then a serialized copy of its definition becomes part of the topology. The supported types of captured globals for these callables is limited to avoid increasing the size of the application and serialization failures due non-serializable objects directly or indirectly referenced from captured globals. The supported types of captured globals are constants (`int`, `str`, `float`, `bool`, `bytes`, `complex`), modules, module attributes (e.g. classes, functions and variables defined in a module), inline classes and functions. If a lambda or inline callable causes an exception due to unsupported global capture then moving it to its own module is a solution.

Due to Python bug 36697 a lambda or inline callable can incorrect capture a global variable. For example an inline class using a attribute of `self.model` will incorrectly capture the global `model` even if the global variable `model` is never used within the class. To workaround this bug use attribute or variable names that do not shadow global variables (e.g. `self._model`).

Due to issue 2336 an inline class using `super()` will cause an `AttributeError` at runtime. Workaround is to call the super class's method directly, for example replace this code:

```
class A(X):
    def __init__(self):
        super().__init__()
```

with:

```
class A(X):
    def __init__(self):
        X.__init__(self)
```

or move the class to a module.

Stateful operations

Use of a class instance allows the operation to be stateful by maintaining state in instance attributes across invocations.

Note: For support with consistent region or checkpointing instances should ensure that the object's state can be pickled. See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects

---

Initialization and shutdown

Execution of a class instance effectively run in a context manager so that an instance's `__enter__` method is called when the processing element containing the instance is initialized and its `__exit__` method called when the processing element is stopped. To take advantage of this the class must define both `__enter__` and `__exit__` methods.

Note: Since an instance of a class is passed to methods such as *map()* `__init__` is only called when the topology is *declared*, not at runtime. Initialization at runtime, such as opening connections, occurs through the `__enter__` method.

Example of using `__enter__` to create custom metrics:

```
import streamsx.ec as ec

class Sentiment(object):
    def __init__(self):
        pass

    def __enter__(self):
        self.positive_metric = ec.CustomMetric(self, "positiveSentiment")
        self.negative_metric = ec.CustomMetric(self, "negativeSentiment")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

    def __call__(self):
        pass
```

When an instance defines a valid `__exit__` method then it will be called with an exception when:

- the instance raises an exception during processing of a tuple
- a data conversion exception is raised converting a value to an structutured schema tuple or attribute

If `__exit__` returns a true value then the exception is suppressed and processing continues, otherwise the enclosing processing element will be terminated.

Tuple semantics

Python objects on a stream may be passed by reference between callables (e.g. the value returned by a map callable may be passed by reference to a following filter callable). This can only occur when the functions are executing in the same PE (process). If an object is not passed by reference a deep-copy is passed. Streams that cross PE (process) boundaries are always passed by deep-copy.

Thus if a stream is consumed by two map and one filter callables in the same PE they may receive the same object reference that was sent by the upstream callable. If one (or more) callable modifies the passed in reference those changes may be seen by the upstream callable or the other callables. The order of execution of the downstream callables is not defined. One can prevent such potential non-deterministic behavior by one or more of these techniques:

- Passing immutable objects
- Not retaining a reference to an object that will be submitted on a stream
- Not modifying input tuples in a callable
- Using copy/deepcopy when returning a value that will be submitted to a stream.

Applications cannot rely on pass-by reference, it is a performance optimization that can be made in some situations when stream connections are within a PE.

Application log and trace

IBM Streams provides application trace and log services which are accesible through standard Python loggers from the *logging* module.

See *Application log and trace*.

SPL operators

In addition an application declared by *Topology* can include stream processing defined by SPL primitive or composite operators. This allows reuse of adapters and analytics provided by IBM Streams, open source and third-party SPL toolkits.

See `streamsx.spl.op`

1.2.5 Module contents

1.2.6 Module contents

Classes

`PendingStream`	Pending stream connection.
`Routing`	Defines how tuples are routed to channels in a parallel region.
`Sink`	Termination of a *Stream*.
`Stream`	The Stream class is the primary abstraction within a streaming application.
`SubscribeConnection`	Connection mode between a subscriber and matching publishers.

Continued on next page

Table 2 – continued from previous page

Topology	The Topology class is used to define data sources, and is passed as a parameter when submitting an application.
View	The View class provides access to a continuously updated sampling of data items on a *Stream* after submission.
Window	Declaration of a window of tuples on a *Stream*.

class streamsx.topology.topology.**Routing**

Bases: enum.Enum

Defines how tuples are routed to channels in a parallel region.

A parallel region is started by *parallel()* and ended with *end_parallel()* or *for_each()*.

BROADCAST = 0

Tuples are routed to every channel in the parallel region.

HASH_PARTITIONED = 3

Tuples are routed based upon a hash value so that tuples with the same hash and thus same value are always routed to the same channel. When a hash function is specified it is passed the tuple and the return value is the hash. When no hash function is specified then *hash(tuple)* is used.

Each tuple is only sent to a single channel.

> **Warning:** A consistent hash function is required to guarantee that a tuple with the same value is always routed to the same channel. *hash()* is not consistent in that for types str, bytes and datetime objects are "salted" with an unpredictable random value (Python 3.5). Thus if the processing element is restarted channel routing for a hash based upon a str, bytes or datetime will change. In addition code executing in the channels can see a different hash value to other channels and the execution that routed the tuple due to being in different processing elements.

ROUND_ROBIN = 1

Tuples are routed to maintain an even distribution of tuples to the channels.

Each tuple is only sent to a single channel.

class streamsx.topology.topology.**SubscribeConnection**

Bases: enum.Enum

Connection mode between a subscriber and matching publishers.

New in version 1.9.

See also:

subscribe()

Buffered = 1

Buffered connection between a subscriber and and matching publishers.

With a buffered connection tuples from publishers are placed in a single queue owned by the subscriber. This allows a slower subscriber to handle brief spikes in tuples from publishers.

A subscriber can fully isolate itself from matching publishers by adding a `CongestionPolicy` that drops tuples when the queue is full. In this case when the subscriber is not able to keep up with the tuple rate from all matching subscribers it will have a minimal effect on matching publishers.

Direct = 0

Direct connection between a subscriber and and matching publishers.

When connected directly a slow subscriber will cause back-pressure against the publishers, forcing them to slow tuple processing to the slowest publisher.

class streamsx.topology.topology.**Topology**(*name=None*, *namespace=None*, *files=None*)
Bases: object

The Topology class is used to define data sources, and is passed as a parameter when submitting an application. Topology keeps track of all sources, sinks, and transformations within your application.

Submission of a Topology results in a Streams application that has the name *namespace::name*.

Parameters

- **name** (*str*) – Name of the topology. Defaults to a name dervied from the calling evironment if it can be determined, otherwise a random name.
- **namespace** (*str*) – Namespace of the topology. Defaults to a name dervied from the calling evironment if it can be determined, otherwise a random name.

include_packages
Python package names to be included in the built application. Any package in this list is copied into the bundle and made available at runtime to the Python callables used in the application. By default a Topology will automatically discover which packages and modules are required to be copied, this field may be used to add additional packages that were not automatically discovered. See also *add_pip_package()*. Package names in *include_packages* take precedence over package names in *exclude_packages*.

Type set[str]

exclude_packages
Python top-level package names to be excluded from the built application. Excluding a top-level packages excludes all sub-modules at any level in the package, e.g. *sound* excludes *sound.effects.echo*. Only the top-level package can be defined, e.g. *sound* rather than *sound.filters*. Behavior when adding a module within a package is undefined. When compiling the application using Anaconda this set is pre-loaded with Python packages from the Anaconda pre-loaded set.

Type set[str]

type_checking
Set to false to disable type checking, defaults to True.

Type bool

name_to_runtime_id
Optional callable that returns a runtime identifier for a name. Used to override the default mapping of a name into a runtime identifer. It will be called with *name* and returns a valid SPL identifier or None. If None is returned then the default mapping for *name* is used. Defaults to None indicating the default mapping is used. See *Stream.runtime_id*.

All declared streams in a *Topology* are available through their name using topology[name]. The stream's name is defined by *Stream.name()* and will differ from the name parameter passed when creating the stream if the application uses duplicate names.

Changed in version 1.11: Declared streams available through topology[name].

add_file_dependency(*path*, *location*)
Add a file or directory dependency into an Streams application bundle.

Ensures that the file or directory at *path* on the local system will be available at runtime.

The file will be copied and made available relative to the application directory. Location determines where the file is relative to the application directory. Two values for location are supported *etc* and *opt*. The runtime path relative to application directory is returned.

The copy is made during the submit call thus the contents of the file or directory must remain availble until submit returns.

For example calling `add_file_dependency('/tmp/conf.properties', 'etc')` will result in contents of the local file *conf.properties* being available at runtime at the path *application directory*/etc/conf.properties. This call returns `etc/conf.properties`.

Python callables can determine the application directory at runtime with *get_application_directory()*. For example the path above at runtime is `os.path.join(streamsx.ec.get_application_directory(), 'etc', 'conf.properties')`

Parameters

- **path** (*str*) – Path of the file on the local system.
- **location** (*str*) – Location of the file in the bundle relative to the application directory.

Returns Path relative to application directory that can be joined at runtime with `get_application_directory`.

Return type str

New in version 1.7.

add_pip_package(*requirement*)

Add a Python package dependency for this topology.

If the package defined by the requirement specifier is not pre-installed on the build system then the package is installed using *pip* and becomes part of the Streams application bundle (*sab* file). The package is expected to be available from *pypi.org*.

If the package is already installed on the build system then it is not added into the *sab* file. The assumption is that the runtime hosts for a Streams instance have the same Python packages installed as the build machines. This is always true for IBM Cloud Pak for Data and the Streaming Analytics service on IBM Cloud.

The project name extracted from the requirement specifier is added to *exclude_packages* to avoid the package being added by the dependency resolver. Thus the package should be added before it is used in any stream transformation.

When an application is run with trace level `info` the available Python packages on the running system are listed to application trace. This includes any packages added by this method.

Example:

```
topo = Topology()
# Add dependency on pint package
# and astral at version 0.8.1
topo.add_pip_package('pint')
topo.add_pip_package('astral==0.8.1')
```

Parameters **requirement** (*str*) – Package requirements specifier.

Warning: Only supported when using the build service with a Streams instance in Cloud Pak for Data or Streaming Analytics service on IBM Cloud.

Note: Installing packages through *pip* is preferred to the automatic dependency checking performed on local modules. This is because *pip* will perform a full install of the package including any dependent packages and additional files, such as shared libraries, that might be missed by dependency discovery.

New in version 1.9.

property checkpoint_period

Enable checkpointing for the topology, and define the checkpoint period.

When checkpointing is enabled, the state of all stateful operators is saved periodically. If the operator restarts, its state is restored from the most recent checkpoint.

The checkpoint period is the frequency at which checkpoints will be taken. It can either be a `timedelta` value or a floating point value in seconds. It must be at 0.001 seconds or greater.

A stateful operator is an operator whose callable is an instance of a Python callable class.

Examples:

```
# Create a topology that will checkpoint every thirty seconds
topo = Topology()
topo.checkpoint_period = 30.0
```

```
# Create a topology that will checkpoint every two minutes
topo = Topology()
topo.checkpoint_period = datetime.timedelta(minutes=2)
```

New in version 1.11.

create_submission_parameter(*name*, *default=None*, *type_=None*)

Create a submission parameter.

A submission parameter is a handle for a value that is not defined until topology submission time. Submission parameters enable the creation of reusable topology bundles.

A submission parameter has a *name*. The name must be unique within the topology.

The returned parameter is a *callable*. Prior to submitting the topology, while constructing the topology, invoking it returns `None`.

After the topology is submitted, invoking the parameter within the executing topology returns the actual submission time value (or the default value if it was not set at submission time).

Submission parameters may be used within functional logic. e.g.:

```
threshold = topology.create_submission_parameter('threshold', 100);

# s is some stream of integers
s = ...
s = s.filter(lambda v : v > threshold())
```

Note: The parameter (value returned from this method) is only supported within a lambda expression or a callable that is not a function.

The default type of a submission parameter's value is a *str*. When a *default* is specified the type of the value matches the type of the default.

If *default* is not set, then the type can be set with *type_*.

The types supported are `str`, `int`, `float` and `bool`.

Topology submission behavior when a submission parameter lacking a default value is created and a value is not provided at submission time is defined by the underlying topology execution runtime.

- Submission fails for contexts `DISTRIBUTED`, `STANDALONE`, and `STREAMING_ANALYTICS_SERVICE`.

Parameters

- **name** (*str*) – Name for submission parameter.
- **default** – Default parameter when submission parameter is not set.
- **type_** – Type of parameter value when default is not set. Supported values are *str*, *int*, *float* and *bool*.

New in version 1.9.

property name
Name of the topology.

Returns Name of the topology.

Return type str

property namespace
Namespace of the topology.

Returns Namespace of the topology.

Return type str

source(*func*, *name=None*)
Declare a source stream that introduces tuples into the application.

Typically used to create a stream of tuple from an external source, such as a sensor or reading from an external system.

Tuples are obtained from an iterator obtained from the passed iterable or callable that returns an iterable.

Each tuple that is not None from the iterator is present on the returned stream.

Each tuple is a Python object and must be picklable to allow execution of the application to be distributed across available resources in the Streams instance.

If the iterator's `__iter__` or `__next__` block then shutdown, checkpointing or consistent region processing may be delayed. Having `__next__` return `None` (no available tuples) or tuples to submit will allow such processing to proceed.

A shutdown `threading.Event` is available through *streamsx.ec.shutdown()* which becomes set when a shutdown of the processing element has been requested. This event my be waited on to perform a sleep that will terminate upon shutdown.

Parameters

- **func** (*callable*) – An iterable or a zero-argument callable that returns an iterable of tuples.
- **name** (*str*) – Name of the stream, defaults to a generated name.

Exceptions raised by `func` or its iterator will cause its processing element will terminate.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method.

Suppressing an exception raised by `func.__iter__` causes the source to be empty, no tuples are submitted to the stream.

Suppressing an exception raised by `__next__` on the iterator results in no tuples being submitted for that call to `__next__`. Processing continues with calls to `__next__` to fetch subsequent tuples.

Returns A stream whose tuples are the result of the iterable obtained from *func*.

Return type *Stream*

Type hints

Type hints on *func* define the schema of the returned stream, defaulting to *Python* if no type hints are present.

For example `s_sensor` has a type hint that defines it as an iterable of `SensorReading` instances (typed named tuples). Thus *readings* has a structured schema matching `SensorReading`

```
def s_sensor() -> typing.Iterable[SensorReading] :
    ...

topo = Topology()
readings = topo.source(s_sensor)
```

Simple examples

Finite constant source stream containing two tuples `Hello` and `World`:

```
topo = Topology()
hw = topo.source(['Hello', 'World'])
```

Use of builtin *range* to produce a finite source stream containing 100 *int* tuples from 0 to 99:

```
topo = Topology()
hw = topo.source(range(100))
```

Use of *itertools.count* to produce an infinite stream of *int* tuples:

```
import itertools
topo = Topology()
hw = topo.source(lambda : itertools.count())
```

Use of *itertools* to produce an infinite stream of tuples with a constant value and a sequence number:

```
import itertools
topo = Topology()
hw = topo.source(lambda : zip(itertools.repeat(), itertools.count()))
```

External system examples

Typically sources pull data in from external systems, such as files, REST apis, databases, message systems etc. Such a source will typically be implemented as class that when called returns an iterable.

To allow checkpointing of state standard methods `__enter__` and `__exit__` are implemented to allow creation of runtime objects that cannot be persisted, for example a file handle.

At checkpoint time state is preserved through standard pickling using `__getstate__` and (optionally) `__setstate__`.

Stateless source that polls a REST API every ten seconds to get a JSON object (*dict*) with current time details:

```
import requests
import time

class RestJsonReader(object):
    def __init__(self, url, period):
        self.url = url
        self.period = period
        self.session = None

    def __enter__(self):
        self.session = requests.Session()
        self.session.headers.update({'Accept': 'application/json'})

    def __exit__(self, exc_type, exc_value, traceback):
        if self.session:
            self.session.close()
            self.session = None

    def __call__(self):
        return self

    def __iter__(self):
        return self

    def __next__(self):
        time.sleep(self.period)
        return self.session.get(self.url).json()

    def __getstate__(self):
        # Remove the session from the persisted state
        return {'url':self.url, 'period':self.period}

def main():
    utc_now = 'http://worldclockapi.com/api/json/utc/now'
    topo = Topology()
    times = topo.source(RestJsonReader(10, utc_now))
```

Warning: Source functions that use generators are not supported when checkpointing or within a consistent region. This is because generators cannot be pickled (even when using *dill*).

Changed in version 1.14: Type hints are used to define the returned stream schema.

property streams

Dict of all streams in the topology.

Key is the name of the stream, value is the corresponding *Stream* instance.

The returned value is a shallow copy of current streams in this topology. This allows callers to iterate over the copy and perform operators that would add streams.

Note: Includes all streams created by composites and any internal streams created by topology.

New in version 1.14.

subscribe(*topic*, *schema=<CommonSchema.Python: <streamsx.topology.schema.StreamSchema object>>*, *name=None*, *connect=None*, *buffer_capacity=None*, *buffer_full_policy=None*)

Subscribe to a topic published by other Streams applications. A Streams application may publish a stream to allow other Streams applications to subscribe to it. A subscriber matches a publisher if the topic and schema match.

By default a stream is subscribed as *Python* objects which connects to streams published to topic by Python Streams applications.

Structured schemas are subscribed to using an instance of `StreamSchema`. A Streams application publishing structured schema streams may have been implemented in any programming language supported by Streams.

JSON streams are subscribed to using schema *Json*. Each tuple on the returned stream will be a Python dictionary object created by `json.loads(tuple)`. A Streams application publishing JSON streams may have been implemented in any programming language supported by Streams.

String streams are subscribed to using schema *String*. Each tuple on the returned stream will be a Python string object. A Streams application publishing string streams may have been implemented in any programming language supported by Streams.

Subscribers can ensure they do not slow down matching publishers by using a buffered connection with a buffer full policy that drops tuples.

Parameters

- **topic** (*str*) – Topic to subscribe to.
- **schema** (`StreamSchema`) – schema to subscribe to.
- **name** (*str*) – Name of the subscribed stream, defaults to a generated name.
- **connect** (`SubscribeConnection`) – How subscriber will be connected to matching publishers. Defaults to *Direct* connection.
- **buffer_capacity** (*int*) – Buffer capacity in tuples when *connect* is set to *Buffered*. Defaults to 1000 when *connect* is *Buffered*. Ignored when *connect* is *None* or *Direct*.
- **buffer_full_policy** (*CongestionPolicy*) – Policy when a pulished tuple arrives and the subscriber's buffer is full. Defaults to *Wait* when *connect* is *Buffered*. Ignored when *connect* is *None* or *Direct*.

Returns A stream whose tuples have been published to the topic by other Streams applications.

Return type *Stream*

Changed in version 1.9: *connect*, *buffer_capacity* and *buffer_full_policy* parameters added.

class streamsx.topology.topology.**Stream**(*topology*, *oport*, *other=None*)

Bases: `streamsx._streams._placement._Placement`, `object`

The Stream class is the primary abstraction within a streaming application. It represents a potentially infinite series of tuples which can be operated upon to produce another stream, as in the case of *map()*, or terminate a stream, as in the case of *for_each()*.

aliased_as(*name*)

Create an alias of this stream.

Returns an alias of this stream with name *name*. When invocation of an SPL operator requires an *Expression* against an input port this can be used to ensure expression matches the input port alias regardless of the name of the actual stream.

Example use where the filter expression for a `Filter` SPL operator uses `IN` to access input tuple attribute `seq`:

```
s = ...
s = s.aliased_as('IN')

params =  {'filter': op.Expression.expression('IN.seq % 4ul == 0ul')}
f = op.Map('spl.relational::Filter', stream, params = params)
```

Parameters **name** (*str*) – Name for returned stream.

Returns Alias of this stream with `name` equal to *name*.

Return type *Stream*

New in version 1.9.

as_json(*force_object=True*, *name=None*)

Declares a stream converting each tuple on this stream into a JSON value.

The stream is typed as a *JSON stream*.

Each tuple must be supported by *JSONEncoder*.

If *force_object* is *True* then each tuple that not a *dict* will be converted to a JSON object with a single key *payload* containing the tuple. Thus each object on the stream will be a JSON object.

If *force_object* is *False* then each tuple is converted to a JSON value directly using *json* package.

If this stream is already typed as a JSON stream then it will be returned (with no additional processing against it and *force_object* and *name* are ignored).

Parameters

- **force_object** (*bool*) – Force conversion of non dicts to JSON objects.
- **name** (*str*) – Name of the resulting stream. When *None* defaults to a generated name.

New in version 1.6.1.

Returns Stream containing the JSON representations of tuples on this stream.

Return type *Stream*

as_string(*name=None*)

Declares a stream converting each tuple on this stream into a string using *str(tuple)*.

The stream is typed as a *string stream*.

If this stream is already typed as a string stream then it will be returned (with no additional processing against it and *name* is ignored).

Parameters **name** (*str*) – Name of the resulting stream. When *None* defaults to a generated name.

New in version 1.6.

New in version 1.6.1: *name* parameter added.

Returns Stream containing the string representations of tuples on this stream.

Return type *Stream*

autonomous()

Starts an autonomous region for downstream processing. By default IBM Streams processing is executed in an autonomous region where any checkpointing of operator state is autonomous (independent) of other operators.

This method may be used to end a consistent region by starting an autonomous region. This may be called even if this stream is in an autonomous region.

Autonomous is not applicable when a topology is submitted to a STANDALONE contexts and will be ignored.

New in version 1.6.

Returns Stream whose subsequent downstream processing is in an autonomous region.

Return type *Stream*

batch(*size*)

Declares a tumbling window to support batch processing against this stream.

The number of tuples in the batch is defined by *size*.

If *size* is an `int` then it is the count of tuples in the batch. For example, with `size=10` each batch will nominally contain ten tuples. Thus processing against the returned *Window*, such as *aggregate()* will be executed every ten tuples against the last ten tuples on the stream. For example the first three aggregations would be against the first ten tuples on the stream, then the next ten tuples and then the third ten tuples, etc.

If *size* is an *datetime.timedelta* then it is the duration of the batch using wallclock time. With a *timedelta* representing five minutes then the window contains any tuples that arrived in the last five minutes. Thus processing against the returned *Window*, such as *aggregate()* will be executed every five minutes tuples against the batch of tuples arriving in the last five minutes on the stream. For example the first three aggregations would be against any tuples on the stream in the first five minutes, then the next five minutes and then minutes ten to fifteen. A batch can contain no tuples if no tuples arrived on the stream in the defined duration.

Each tuple on the stream appears only in a single batch.

The number of tuples seen by processing against the returned window may be less than *size* (count or time based) when:

- the stream is finite, the final batch may contain less tuples than the defined size,
- the stream is in a consistent region, drain processing will complete the current batch without waiting for it to batch to reach its nominal size.

Examples:

```
# Create batches against stream s of 100 tuples each
w = s.batch(size=100)
```

```
# Create batches against stream s every five minutes
w = s.batch(size=datetime.timedelta(minutes=5))
```

Parameters **size** – The size of each batch, either an *int* to define the number of tuples or *datetime.timedelta* to define the duration of the batch.

Returns Window allowing batch processing on this stream.

Return type *Window*

New in version 1.11.

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

end_low_latency()

Returns a Stream that is no longer guaranteed to run in the same process as the calling stream.

Returns Stream

end_parallel()

Ends a parallel region by merging the channels into a single stream.

Returns Stream for which subsequent transformations are no longer parallelized.

Return type *Stream*

See also:

set_parallel(), *parallel()*

filter(*func*, *name=None*)

Filters tuples from this stream using the supplied callable *func*.

For each stream tuple *t* on the stream `func(t)` is called, if the return evaluates to `True` the tuple will be present on the returned stream, otherwise the tuple is filtered out.

Parameters

- **func** – Filter callable that takes a single parameter for the stream tuple.
- **name** (*str*) – Name of the stream, defaults to a generated name.

If invoking `func` for a stream tuple raises an exception then its processing element will terminate. By default the processing element will automatically restart though tuples may be lost.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method. When an exception is suppressed no tuple is submitted to the filtered stream corresponding to the input tuple that caused the exception.

Returns A Stream containing tuples that have not been filtered out. The schema of the returned stream is the same as this stream's schema.

Return type *Stream*

Type hints

The argument type hint on *func* is used (if present) to verify at topology declaration time that it is compatible with the type of tuples on this stream.

flat_map(*func=None*, *name=None*)

Maps and flatterns each tuple from this stream into 0 or more tuples.

For each tuple on this stream `func(tuple)` is called. If the result is not *None* then the the result is iterated over with each value from the iterator that is not *None* will be submitted to the return stream.

If the result is *None* or an empty iterable then no tuples are submitted to the returned stream.

Parameters

- **func** – A callable that takes a single parameter for the tuple. If not supplied then a function equivalent to `lambda tuple_ : tuple_` is used. This is suitable when each tuple on this stream is an iterable to be flattened.
- **name** (*str*) – Name of the flattened stream, defaults to a generated name.

If invoking `func` for a tuple on the stream raises an exception then its processing element will terminate. By default the processing element will automatically restart though tuples may be lost.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method. When an exception is suppressed no tuples are submitted to the flattened and mapped stream corresponding to the input tuple that caused the exception.

Returns A Stream containing flattened and mapped tuples.

Return type *Stream*

Raises **TypeError** – if *func* does not return an iterator nor None

Changed in version 1.11: *func* is optional.

for_each(*func*, *name=None*)

Sends information as a stream to an external system.

The transformation defined by *func* is a callable or a composite transformation.

Callable transformation

If *func* is callable then for each tuple *t* on this stream `func(t)` is called.

If invoking `func` for a tuple on the stream raises an exception then its processing element will terminate. By default the processing element will automatically restart though tuples may be lost.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method. When an exception is suppressed no further processing occurs for the input tuple that caused the exception.

Composite transformation

A composite transformation is an instance of *ForEach*. Composites allow the application developer to use the standard functional style of the topology api while allowing allowing expansion of a *for_each* transform to multiple basic transformations.

Parameters

- **func** – A callable that takes a single parameter for the tuple and returns None.
- **name** (*str*) – Name of the stream, defaults to a generated name.

Returns Stream termination.

Return type *streamsx.topology.topology.Sink*

Type hints

The argument type hint on *func* is used (if present) to verify at topology declaration time that it is compatible with the type of tuples on this stream.

Changed in version 1.7: Now returns a *Sink* instance.

Changed in version 1.14: Support for type hints and composite transformations.

isolate()

Guarantees that the upstream operation will run in a separate processing element from the downstream operation

Returns Stream whose subsequent immediate processing will occur in a separate processing element.

Return type *Stream*

last(*size=1*)

Declares a slding window containing most recent tuples on this stream.

The number of tuples maintained in the window is defined by *size*.

If *size* is an *int* then it is the count of tuples in the window. For example, with `size=10` the window always contains the last (most recent) ten tuples.

If *size* is an *datetime.timedelta* then it is the duration of the window. With a *timedelta* representing five minutes then the window contains any tuples that arrived in the last five minutes.

Parameters **size** – The size of the window, either an *int* to define the number of tuples or *datetime.timedelta* to define the duration of the window.

Examples:

```
# Create a window against stream s of the last 100 tuples
w = s.last(size=100)
```

```
# Create a window against stream s of tuples
# arrived on the stream in the last five minutes
w = s.last(size=datetime.timedelta(minutes=5))
```

Returns Window of the last (most recent) tuples on this stream.

Return type *Window*

low_latency()

The function is guaranteed to run in the same process as the upstream Stream function. All streams that are created from the returned stream are also guaranteed to run in the same process until end_low_latency() is called.

Returns Stream

map(*func=None*, *name=None*, *schema=None*)

Maps each tuple from this stream into 0 or 1 stream tuples.

The transformation defined by *func* is a callable or a composite transformation.

Callable transformation

For each tuple on this stream `result = func(tuple)` is called. If *result* is not *None* then the result will be submitted as a tuple on the returned stream. If *result* is *None* then no tuple submission will occur.

By default the submitted tuple is `result` without modification resulting in a stream of picklable Python objects. Setting the *schema* parameter changes the type of the stream and modifies each `result` before submission.

- `object` or *Python* - The default: *result* is submitted.
- `str` type or *String* - A stream of strings: `str(result)` is submitted.
- `json` or *Json* - A stream of JSON objects: `result` must be convertable to a JSON object using *json* package.
- *StreamSchema* - A structured stream. *result* must be a *dict* or (Python) *tuple*. When a *dict* is returned the outgoing stream tuple attributes are set by name, when a *tuple* is returned stream tuple attributes are set by position.
- string value - Equivalent to passing `StreamSchema(schema)`

Composite transformation

A composite transformation is an instance of *Map*. Composites allow the application developer to use the standard functional style of the topology api while allowing allowing expansion of a *map* transform to multiple basic transformations.

Parameters

- **func** – A callable that takes a single parameter for the tuple. If not supplied then a function equivalent to `lambda tuple_ : tuple_` is used.
- **name** (*str*) – Name of the mapped stream, defaults to a generated name.
- **schema** (*StreamSchema* | *CommonSchema* | *str*) – Schema of the resulting stream.

If invoking `func` for a tuple on the stream raises an exception then its processing element will terminate. By default the processing element will automatically restart though tuples may be lost.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method. When an exception is suppressed no tuple is submitted to the mapped stream corresponding to the input tuple that caused the exception.

Returns A stream containing tuples mapped by *func*.

Return type *Stream*

Type hints

If *schema* is not set then the return type hint on *func* define the schema of the returned stream, defaulting to *Python* if no type hints are present.

For example *reading_from_json* has a type hint that defines it as returning `SensorReading` instances (typed named tuples). Thus *readings* has a structured schema matching `SensorReading`

```
def reading_from_json(value:dict) -> SensorReading:
    return SensorReading(value['id'], value['timestamp'], value['reading'])

topo = Topology()
json_readings = topo.source(HttpReadings()).as_json()
readings = json_readings.map(reading_from_json)
```

The argument type hint on *func* is used (if present) to verify at topology declaration time that it is compatible with the type of tuples on this stream.

New in version 1.7: *schema* argument added to allow conversion to a structured stream.

New in version 1.8: Support for submitting *dict* objects as stream tuples to a structured stream (in addition to existing support for *tuple* objects).

Changed in version 1.11: *func* is optional.

property name

Unique name of the stream.

When declaring a stream a *name* parameter can be provided. If the supplied name is unique within its topology then it will be used as-is, otherwise a variant will be provided that is unique within the topology.

If a *name* parameter was not provided when declaring a stream then the stream is assigned a unique generated name.

Returns Name of the stream.

Return type str

See also:

aliased_as()

Warning: If the name is not a valid SPL identifier or longer than 80 characters then the name will be converted to a valid SPL identifier at compile and runtime. This identifier will be the name used in the REST api and log/trace.

Visualizations of the runtime graph uses *name* rather than the converted identifier.

A valid SPL identifier consists only of characters `A-Z`, `a-z`, `0-9`, `_` and must not start with a number or be an SPL keyword.

See *runtime_id*.

parallel(*width*, *routing=<Routing.ROUND_ROBIN: 1>*, *func=None*, *name=None*)

Split stream into channels and start a parallel region.

Returns a new stream that will contain the contents of this stream with tuples distributed across its channels.

The returned stream starts a parallel region where all downstream transforms are replicated across *width* channels. A parallel region is terminated by *end_parallel()* or *for_each()*.

Any transform (such as *map()*, *filter()*, etc.) in a parallel region has a copy of its callable executing independently in parallel. Channels remain independent of other channels until the region is terminated.

For example with this topology fragment a parallel region of width 3 is created:

```
s = ...
p = s.parallel(3)
p = p.filter(F()).map(M())
e = p.end_parallel()
e.for_each(E())
```

Tuples from `p` (parallelized `s`) are distributed across three channels, 0, 1 & 2 and are independently processed by three instances of `F` and `M`. The tuples that pass the filter `F` in channel 0 are then mapped by the instance of `M` in channel 0, and so on for channels 1 and 2.

The channels are combined by `end_parallel` and so a single instance of `E` processes all the tuples from channels 0, 1 & 2.

This stream instance (the original) is outside of the parallel region and so any downstream transforms are executed normally. Adding this *map* transform would result in tuples on `s` being processed by a single instance of `N`:

```
n = s.map(N())
```

The number of channels is set by *width* which may be an *int* greater than zero or a submission parameter created by *Topology.create_submission_parameter()*.

With IBM Streams 4.3 or later the number of channels can be dynamically changed at runtime.

Tuples are routed to channels based upon *routing*, see *Routing*.

A parallel region can have multiple termination points, for example when a stream within the stream has multiple transforms against it:

```
s = ...
p = s.parallel(3)
m1p = p.map(M1())
m2p = p.map(M2())
p.for_each(E())

m1 = m1p.end_parallel()
m2 = m2p.end_parallel()
```

Parallel regions can be nested, for example:

```
s = ...
m = s.parallel(2).map(MO()).parallel(3).map(MI()).end_parallel().end_
↪parallel()
```

In this case there will be two instances of `MO` (the outer region) and six (2x3) instances of `MI` (the inner region).

Streams created by *source()* or *subscribe()* are placed in a parallel region by *set_parallel()*.

Parameters

- **width** (*int*) – Degree of parallelism.
- **routing** (`Routing`) – Denotes what type of tuple routing to use.
- **func** – Optional function called when *Routing.HASH_PARTITIONED* routing is specified. The function provides an integer value to be used as the hash that determines the tuple channel routing.
- **name** (*str*) – The name to display for the parallel region.

Returns A stream for which subsequent transformations will be executed in parallel.

Return type *Stream*

See also:

set_parallel(), *end_parallel()*, *split()*

print(*tag=None*, *name=None*)

Prints each tuple to stdout flushing after each tuple.

If *tag* is not *None* then each tuple has "tag: " prepended to it before printing.

Parameters

- **tag** – A tag to prepend to each tuple.
- **name** (*str*) – Name of the resulting stream. When *None* defaults to a generated name.

Returns Stream termination.

Return type *streamsx.topology.topology.Sink*

New in version 1.6.1: *tag*, *name* parameters.

Changed in version 1.7: Now returns a *Sink* instance.

publish(*topic*, *schema=None*, *name=None*)

Publish this stream on a topic for other Streams applications to subscribe to. A Streams application may publish a stream to allow other Streams applications to subscribe to it. A subscriber matches a publisher if the topic and schema match.

By default a stream is published using its schema.

A stream of *Python objects* can be subscribed to by other Streams Python applications.

If a stream is published setting *schema* to `json` or *Json* then it is published as a stream of JSON objects. Other Streams applications may subscribe to it regardless of their implementation language.

If a stream is published setting *schema* to `str` or *String* then it is published as strings. Other Streams applications may subscribe to it regardless of their implementation language.

Supported values of *schema* are only `json`, *Json* and `str`, *String*.

Parameters

- **topic** (*str*) – Topic to publish this stream to.
- **schema** – Schema to publish. Defaults to the schema of this stream.
- **name** (*str*) – Name of the publish operator, defaults to a generated name.

Returns Stream termination.

Return type *streamsx.topology.topology.Sink*

New in version 1.6.1: *name* parameter.

Changed in version 1.7: Now returns a `Sink` instance.

property resource_tags

Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A `Stream` that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a `union()` of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

Warning: If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for `Sink` and `Invoke`.

property runtime_id

Return runtime identifier.

If `name` is not a valid SPL identifier then the runtime identifier will be valid SPL identifier that represents *name*. Otherwise *name* is returned.

The runtime identifier is how the underlying SPL operator or output port is named in the REST api and trace/log files.

If a topology unique name is supplied when creating a stream then runtime identifier is fixed regardless of other changes in the topology.

The algorithm to determine the runtime name (for clients that cannot call this method, for example, remote REST clients gathering metrics) is as follows.

If the length of `name` is less than or equal to 80 and `name` is an SPL identifier then `name` is used. An SPL identifier consists only of the characters `A-Z`, `a-z` `0-9` and `_`, must not start with `0-9` and must not be an SPL keyword.

Otherwise the identifier has the form `prefix_suffix`.

`prefix` is the kind of the SPL operator stripped of its namespace and `::`. For all functional methods the operator kind is the method name with the first character upper-cased.

For example, `Filter` for `filter()`, `Beacon` for `spl::utility::Beacon`.

`suffix` is a hashed version of name, an MD5 digest `d` is calculated from the UTf-8 encoding of `name`. `d` is shortened by having its first eight bytes xor folded with its last eight bytes. `d` is then base64 encoded to produce a string. Padding = and + and / characters are removed from the string.

For example, `s.filter(lambda x : True, name='')` results in a runtime identifier of `Filter_oGwCfhWRg4`.

The default mapping can be overridden by setting *Topology.name_to_runtime_id* to a callable that returns a valid identifier for its single argument. The returned identifier should be unique with the topology. For example usinig a pre-populated *dict* as the mapper:

```
topo = Topology()
names = {'', 'Buses', '':'Trains'}
topo.name_to_runtime_id = names.get

buses = toopo.source(..., name='')
trains = topo.source(..., name=''}

// buses.runtime_id will be Buses
// trains.runtime_id will be Trains
```

Returns Runtime identifier of the stream.

Return type str

New in version 1.14.

set_consistent(*consistent_config*)

Indicates that the stream is the start of a consistent region.

Parameters **consistent_config** (*consistent.ConsistentRegionConfig*) – the configuration of the consistent region.

Returns Returns this stream.

Return type *Stream*

New in version 1.11.

set_parallel(*width*, *name=None*)

Set this source stream to be split into multiple channels as the start of a parallel region.

Calling `set_parallel` on a stream created by *source()* results in the stream having *width* channels, each created by its own instance of the callable:

```
s = topo.source(S())
s.set_parallel(3)
f = s.filter(F())
e = f.end_parallel()
```

Each channel has independent instances of `S` and `F`. Tuples created by the instance of `S` in channel 0 are passed to the instance of `F` in channel 0, and so on for channels 1 and 2.

Callable transforms instances within the channel can use the runtime functions *channel()*, *local_channel()*, *max_channels()* & *local_max_channels()* to adapt to being invoked in parallel. For example a source callable can use its channel number to determine which partition to read from in a partitioned external system.

Calling `set_parallel` on a stream created by *subscribe()* results in the stream having *width* channels. Subscribe ensures that the stream will contain all published tuples matching the topic subscription

and type. A published tuple will appear on one of the channels though the specific channel is not known in advance.

A parallel region is terminated by *end_parallel()* or *for_each()*.

The number of channels is set by *width* which may be an *int* greater than zero or a submission parameter created by *Topology.create_submission_parameter()*.

With IBM Streams 4.3 or later the number of channels can be dynamically changed at runtime.

Parallel regions are started on non-source streams using *parallel()*.

Parameters

- **width** – The degree of parallelism for the parallel region.
- **name** (*str*) – Name of the parallel region. Defaults to the name of this stream.

Returns Returns this stream.

Return type *Stream*

See also:

parallel(), *end_parallel()*

New in version 1.9.

Changed in version 1.11: *name* parameter added.

split(*into*, *func*, *names=None*, *name=None*)

Splits tuples from this stream into multiple independent streams using the supplied callable *func*.

For each tuple on the stream `int(func(tuple))` is called, if the return is zero or positive then the (unmodified) tuple will be present on one, and only one, of the output streams. The specific stream will be at index `int(func(tuple)) % N` in the returned list, where `N` is the number of output streams. If the return is negative then the tuple is dropped.

`split` is used to declare disparate transforms on each split stream. This differs to *parallel()* where each channel has the same logic transforms.

Parameters

- **into** (*int*) – Number of streams the input is split into, must be greater than zero.
- **func** – Split callable that takes a single parameter for the tuple.
- **names** (*list[str]*) – Names of the returned streams, in order. If not supplied or a stream doesn't have an entry in *names* then a generated name is used. Entries are used to generated the field names of the returned named tuple.
- **name** (*str*) – Name of the split transform, defaults to a generated name.

If invoking `func` for a tuple on the stream raises an exception then its processing element will terminate. By default the processing element will automatically restart though tuples may be lost.

If `func` is a callable object then it may suppress exceptions by return a true value from its `__exit__` method. When an exception is suppressed no tuple is submitted to the filtered stream corresponding to the input tuple that caused the exception.

Returns Named tuple of streams this stream is split across. All returned streams have the same schema as this stream.

Return type namedtuple

Type hints

The argument type hint on *func* is used (if present) to verify at topology declaration time that it is compatible with the type of tuples on this stream.

Examples

Example of splitting a stream based upon message severity, dropping any messages with unknown severity, and then performing different transforms for each severity:

```
msgs = topo.source(ReadMessages())
SEVS = {'H':0, 'M':1, 'L':2}
severities = msg.split(3, lambda SEVS.get(msg.get('SEV'), -1),
    names=['high','medium','low'], name='SeveritySplit')

high_severity = severities.high
high_severity.for_each(SendAlert())

medium_severity = severities.medium
medium_severity.for_each(LogMessage())

low_severity = severities.low
low_severity.for_each(Archive())
```

See also:

parallel()

New in version 1.13.

union(*streamSet*)
Creates a stream that is a union of this stream and other streams

Parameters **streamSet** – a set of Stream objects to merge with this stream

Returns

Return type *Stream*

view(*buffer_time=10.0*, *sample_size=10000*, *name=None*, *description=None*, *start=False*)
Defines a view on a stream.

A view is a continually updated sampled buffer of a streams's tuples. Views allow visibility into a stream from external clients such as Jupyter Notebooks, the Streams console, Microsoft Excel or REST clients.

The view created by this method can be used by external clients and through the returned *View* object after the topology is submitted. For example a Jupyter Notebook can declare and submit an application with views, and then use the resultant *View* objects to visualize live data within the streams.

When the stream contains Python objects then they are converted to JSON.

Parameters

- **buffer_time** – Specifies the buffer size to use measured in seconds.
- **sample_size** – Specifies the number of tuples to sample per second.
- **name** (*str*) – Name of the view. Name must be unique within the topology. Defaults to a generated name.
- **description** – Description of the view.

- **start** (*bool*) – Start buffering data when the job is submitted. If *False* then the view starts buffering data when the first remote client accesses it to retrieve data.

Returns View object which can be used to access the data when the topology is submitted.

Return type *streamsx.topology.topology.View*

Note: Views are only supported when submitting to distributed contexts including Streaming Analytics service.

---

class streamsx.topology.topology.**View**(*name*)

Bases: object

The View class provides access to a continuously updated sampling of data items on a *Stream* after submission. A view object is produced by *view()*, and will access data items from the stream on which it is invoked.

For example, a *View* object could be created and used as follows:

```
>>> topology = Topology()
>>> rands = topology.source(lambda: iter(random.random, None))
>>> view = rands.view()
>>> submit(ContextTypes.DISTRIBUTED, topology)
>>> queue = view.start_data_fetch()
>>> for val in iter(queue.get, None):
...     print(val)
...
0.6527
0.1963
0.0512
```

display(*duration=None*, *period=2*)

Display a view within a Jupyter or IPython notebook.

Provides an easy mechanism to visualize data on a stream using a view.

Tuples are fetched from the view and displayed in a table within the notebook cell using a pandas.DataFrame. The table is continually updated with the latest tuples from the view.

This method calls *start_data_fetch()* and will call *stop_data_fetch()* when completed if *duration* is set.

Parameters

- **duration** (*float*) – Number of seconds to fetch and display tuples. If None then the display will be updated until *stop_data_fetch()* is called.
- **period** (*float*) – Maximum update period.

Note: A view is a sampling of data on a stream so tuples that are on the stream may not appear in the view.

Note: Python modules *ipywidgets* and *pandas* must be installed in the notebook environment.

Warning: Behavior when called outside a notebook is undefined.

New in version 1.12.

fetch_tuples(*max_tuples=20*, *timeout=None*)
Fetch a number of tuples from this view.

Fetching of data must have been started with *start_data_fetch()* before calling this method.

If `timeout` is `None` then the returned list will contain `max_tuples` tuples. Otherwise if the timeout is reached the list may contain less than `max_tuples` tuples.

Parameters

- **max_tuples** (*int*) – Maximum number of tuples to fetch.
- **timeout** (*float*) – Maximum time to wait for `max_tuples` tuples.

Returns List of fetched tuples.

Return type list

New in version 1.12.

start_data_fetch()
Starts a background thread which begins accessing data from the remote Stream. The data items are placed asynchronously in a queue, which is returned from this method.

Returns A Queue object which is populated with the data items of the stream.

Return type queue.Queue

stop_data_fetch()
Terminates the background thread fetching stream data items.

class streamsx.topology.topology.**PendingStream**(*topology*)
Bases: `object`

Pending stream connection.

A pending stream is an initially *disconnected* stream. The *stream* attribute can be used as an input stream when the required stream is not yet available. Once the required stream is available the connection is made using *complete()*.

The schema of the pending stream is defined by the stream passed into *complete*.

A simple example is creating a source stream after the filter that will use it:

```
# Create the pending or placeholder stream
pending_source = PendingStream(topology)

# Create a filter against the placeholder stream
f = pending_source.stream.filter(lambda : t : t.startswith("H"))

source = topology.source(['Hello', 'World'])

# Now complete the connection
pending_source.complete(source)
```

Streams allows feedback loops in its flow graphs, where downstream processing can produce a stream that is fed back into the input port of an upstream operator. Typically, feedback loops are used to modify the state of upstream transformations, rather than repeat processing of tuples.

A feedback loop can be created by using a *PendingStream*. The upstream transformation or operator that will end the feedback loop uses `stream` as one of its inputs. A processing pipeline is then created and once the downstream starting point of the feedback loop is available, it is passed to *complete()* to create the loop.

complete(*stream*)
Complete the pending stream.

Any connections made to `stream` are connected to *stream* once this method returns.

Parameters **stream** (`Stream`) – Stream that completes the connection.

is_complete()
Has this connection been completed.

class streamsx.topology.topology.**Window**(*stream*, *window_type*)
Bases: `object`

Declaration of a window of tuples on a *Stream*.

A *Window* enables transforms against collection (or window) of tuples on a stream rather than per-tuple transforms. Windows are created against a stream using *Stream.batch()* or *Stream.last()*.

Supported transforms are:

- *aggregate()* - Aggregate the window contents into a single tuple.

A window is optionally *partitioned* to create independent sub-windows per partition key.

A *Window* can be also passed as the input of an SPL operator invocation to indicate the operator's input port is windowed.

Example invoking the SPL *Aggregate* operator with a sliding window of the last two minutes, triggering every five tuples:

```
win = s.last(datetime.timedelta(minutes=2)).trigger(5)

agg = op.Map('spl.relational::Aggregate', win,
             schema = 'tuple<uint64 sum, uint64 max>')
agg.sum = agg.output('Sum(val)')
agg.max = agg.output('Max(val)')
```

aggregate(*function*, *name=None*)
Aggregates the contents of the window when the window is triggered.

Upon a window trigger, the supplied function is passed a list containing the contents of the window: `function(items)`. The order of the window items in the list are the order in which they were each received by the window. If the function's return value is not *None* then the result will be submitted as a tuple on the returned stream. If the return value is *None* then no tuple submission will occur.

For example, a window that calculates a moving average of the last 10 tuples could be written as follows:

```
win = s.last(10).trigger(1)
moving_averages = win.aggregate(lambda tuples: sum(tuples)/len(tuples))
```

When the window is *partitioned* then each partition is triggered and aggregated using *function* independently.

For example, this partitioned window aggregation will independently call `summarize_sensors` with ten tuples all having the same *id* when triggered. Each partition triggers independently so that `summarize_sensors` is invoked for a specific *id* every time two tuples with that *id* have been inserted into the window partition:

```
win = s.last(10).trigger(2).partition(key='id')
moving_averages = win.aggregate(summarize_sensors)
```

Note: If a tumbling (*batch()*) window's stream is finite then a final aggregation is performed if the window is not empty. Thus `function` may be passed fewer tuples for a window sized using a count. For example a stream with 105 tuples and a batch size of 25 tuples will perform four aggregations with 25 tuples each and a final aggregation of 5 tuples.

Parameters

- **function** – The function which aggregates the contents of the window
- **name** (*str*) – The name of the returned stream. Defaults to a generated name.

Returns A *Stream* of the returned values of the supplied function.

Return type *Stream*

Warning: In Python 3.5 or later if the stream being aggregated has a structured schema that contains a `blob` type then any `blob` value will not be maintained in the window. Instead its `memoryview` object will have been released. If the `blob` value is required then perform a `map()` transformation (without setting `schema`) copying any required blob value in the tuple using `memoryview.tobytes()`.

New in version 1.8.

Changed in version 1.11: Support for aggregation of streams with structured schemas.

Changed in version 1.13: Support for partitioned aggregation.

partition(*key*)

Declare a window with this window's eviction and trigger policies, and a partition.

In a partitioned window, a subwindow will be created for each distinct value received for the attribute used for partitioning. Each subwindow is treated as if it were a separate window, and each subwindow shares the same trigger and eviction policy.

The key may either be a string containing the name of an attribute, or a python callable.

The *key* parameter may be a string only with a structured schema, and the value of the *key* parameter must be the name of a single attribute in the schema.

The *key* parameter may be a python callable object. If it is, the callable is evaluated for each tuple, and the return from the callable determines the partition into which the tuple is placed. The return value must have a `__hash__` method. If checkpointing is enabled, and the callable object has a state, the state of the callable object will be saved and restored in checkpoints. However, `__enter__` and `__exit__` methods may not be called on the callable object.

Parameters **key** – The name of the attribute to be used for partitioning, or the python callable object used for partitioning.

Returns Window that will be triggered.

Return type *Window*

New in version 1.13.

trigger(*when=1*)

Declare a window with this window's size and a trigger policy.

When the window is triggered is defined by *when*.

If *when* is an *int* then the window is triggered every *when* tuples. For example, with `when=5` the window will be triggered every five tuples.

If *when* is an *datetime.timedelta* then it is the period of the trigger. With a *timedelta* representing one minute then the window is triggered every minute.

By default, when *trigger* has not been called on a *Window* it triggers for every tuple inserted into the window (equivalent to `when=1`).

Parameters **when** – The size of the window, either an *int* to define the number of tuples or *datetime.timedelta* to define the duration of the window.

Returns Window that will be triggered.

Return type *Window*

Warning: A trigger is only supported for a sliding window such as one created by `last()`.

class streamsx.topology.topology.**Sink**(*op*)

Bases: streamsx._streams._placement._Placement, object

Termination of a *Stream*.

A *Stream* is terminated by processing that typically sends the tuples to an external system.

Note: A *Stream* may have multiple terminations.

See also:

for_each(), *publish()*, *print()*

New in version 1.7.

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

property resource_tags

Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A *Stream* that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a *union()* of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

> **Warning:** If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for *Sink* and *Invoke*.

1.3 streamsx.topology.context

Context for submission and build of topologies.

1.3.1 Module contents

Functions

build	Build a topology to produce a Streams application bundle.
run	Run a topology in a distributed Streams instance.
submit	Submits a *Topology* (application) using the specified context type.

Classes

ConfigParams	Configuration options which may be used as keys in *submit()* *config* parameter.
ContextTypes	Submission context types.
JobConfig	Job configuration.
SubmissionResult	Passed back to the user after a call to submit.

class streamsx.topology.context.**ContextTypes**

Bases: object

Submission context types.

A *Topology* is submitted using *submit()* and a context type. Submision of a *Topology* generally builds the application into a Streams application bundle (sab) file and then submits it for execution in the required context.

The Streams application bundle contains all the artifacts required by an application such that it can be executed remotely (e.g. on a Streaming Analytics service), including distributing the execution of the application across multiple resources (hosts).

The context type defines which context is used for submission.

The main context types result in a running application and are:

- *STREAMING_ANALYTICS_SERVICE* - Application is submitted to a Streaming Analytics service running on IBM Cloud.
- *DISTRIBUTED* - Application is submitted to an IBM Streams instance.
- *STANDALONE* - Application is executed as a local process, IBM Streams *standalone* application. Typically this is used during development or testing.

The *BUNDLE* context type compiles the application (*Topology*) to produce a Streams application bundle (sab file). The bundle is not executed but may subsequently be submitted to a Streaming Analytics service or an IBM Streams instance. A bundle may be submitted multiple times to services or instances, each resulting in a unique job (running application).

BUILD_ARCHIVE = 'BUILD_ARCHIVE'

Creates a build archive.

This context type produces the intermediate code archive used for bundle creation.

> **Note:** *BUILD_ARCHIVE* is typically only used when diagnosing issues with bundle generation.

BUNDLE = 'BUNDLE'

Create a Streams application bundle.

The *Topology* is compiled to produce Streams application bundle (sab file).

The resultant application can be submitted to:

- Streaming Analytics service using the Streams console or the Streaming Analytics REST api.
- IBM Streams instance using the Streams console, JMX api or command line `streamtool submitjob`.
- Executed standalone for development or testing.

The bundle must be built on the same operating system version and architecture as the intended running environment. For Streaming Analytics service this is currently RedHat/CentOS 7 and *x86_64* architecture.

IBM Cloud Pak for Data integated configuration

Projects (within cluster)

The *Topology* is compiled using the Streams build service for a Streams service instance running in the same Cloud Pak for Data cluster as the Jupyter notebook or script declaring the application.

The instance is specified in the configuration passed into *submit()*. The code that selects a service instance by name is:

```
from icpd_core import icpd_util
cfg = icpd_util.get_service_instance_details(name='instanceName')

topo = Topology()
...
submit(ContextTypes.BUNDLE, topo, cfg)
```

The resultant *cfg* dict may be augmented with other values such as keys from *ConfigParams*.

External to cluster or project

The *Topology* is compiled using the Streams build service for a Streams service instance running in Cloud Pak for Data.

Environment variables: These environment variables define how the application is built and submitted.

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

IBM Cloud Pak for Data standalone configuration

The *Topology* is compiled using the Streams build service.

Environment variables: These environment variables define how the application is built.

- **STREAMS_BUILD_URL** - Streams build service URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

IBM Streams on-premise 4.2 & 4.3

The *Topology* is compiled using a local IBM Streams installation.

Environment variables: These environment variables define how the application is built.

- **STREAMS_INSTALL** - Location of a local IBM Streams installation.

DISTRIBUTED = 'DISTRIBUTED'
Submission to an IBM Streams instance.

IBM Cloud Pak for Data integated configuration

Projects (within cluster)

The *Topology* is compiled using the Streams build service and submitted to an Streams service instance running in the same Cloud Pak for Data cluster as the Jupyter notebook or script declaring the application.

The instance is specified in the configuration passed into *submit()*. The code that selects a service instance by name is:

```
from icpd_core import icpd_util
cfg = icpd_util.get_service_instance_details(name='instanceName')

topo = Topology()
...
submit(ContextTypes.DISTRIBUTED, topo, cfg)
```

The resultant *cfg* dict may be augmented with other values such as a *JobConfig* or keys from *ConfigParams*.

External to cluster or project

The *Topology* is compiled using the Streams build service and submitted to a Streams service instance running in Cloud Pak for Data.

Environment variables: These environment variables define how the application is built and submitted.

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

IBM Cloud Pak for Data standalone configuration

The *Topology* is compiled using the Streams build service and submitted to a Streams service instance using REST apis.

Environment variables: These environment variables define how the application is built and submitted.

- **STREAMS_BUILD_URL** - Streams build service URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_REST_URL** - Streams SWS service (REST API) URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

IBM Streams on-premise 4.2 & 4.3

The *Topology* is compiled locally and the resultant Streams application bundle (sab file) is submitted to an IBM Streams instance.

Environment variables: These environment variables define how the application is built and submitted.

- **STREAMS_INSTALL** - Location of a IBM Streams installation (4.2 or 4.3).
- **STREAMS_DOMAIN_ID** - Domain identifier for the Streams instance.
- **STREAMS_INSTANCE_ID** - Instance identifier.
- **STREAMS_ZKCONNECT** - (optional) ZooKeeper connection string for domain (when not using an embedded ZooKeeper)
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.

Warning: `streamtool` is used to submit the job with on-premise 4.2 & 4.3 Streams and requires that `streamtool` does not prompt for authentication. This is achieved by using `streamtool genkey`.

See also:

Generating authentication keys for IBM Streams

STANDALONE = 'STANDALONE'

Build and execute locally.

Compiles and executes the *Topology* locally in IBM Streams standalone mode as a separate sub-process. Typically used for devlopment and testing.

The call to *submit()* return when (if) the application completes. An application completes when it has finite source streams and all tuples from those streams have been processed by the complete topology. If the source streams are infinite (e.g. reading tweets) then the standalone application will not complete.

Environment variables: This environment variables define how the application is built.

- **STREAMS_INSTALL** - Location of a IBM Streams installation (4.0.1 or later).

STREAMING_ANALYTICS_SERVICE = 'STREAMING_ANALYTICS_SERVICE'

Submission to Streaming Analytics service running on IBM Cloud.

The *Topology* is compiled and the resultant Streams application bundle (sab file) is submitted for execution on the Streaming Analytics service.

When **STREAMS_INSTALL** is not set or the *submit()* *config* parameter has *FORCE_REMOTE_BUILD* set to *True* the compilation of the application occurs remotely by the service. This allows creation and submission of Streams applications without a local install of IBM Streams.

When **STREAMS_INSTALL** is set and the *submit()* *config* parameter has *FORCE_REMOTE_BUILD* set to *False* or not set then the creation of the Streams application bundle occurs locally and the bundle is submitted for execution on the service.

Environment variables: These environment variables define how the application is built and submitted.

- **STREAMS_INSTALL** - (optional) Location of a IBM Streams installation (4.0.1 or later). The install must be running on RedHat/CentOS 7 and *x86_64* architecture.

TOOLKIT = 'TOOLKIT'
Creates an SPL toolkit.

Topology applications are implemented as an SPL application before compilation into an Streams application bundle. This context type produces the intermediate SPL toolkit that is input to the SPL compiler for bundle creation.

Note: *TOOLKIT* is typically only used when diagnosing issues with bundle generation.

---

class streamsx.topology.context.**ConfigParams**
Bases: object

Configuration options which may be used as keys in *submit()* *config* parameter.

FORCE_REMOTE_BUILD = 'topology.forceRemoteBuild'
Force a remote build of the application.

When submitting to STREAMING_ANALYTICS_SERVICE a local build of the Streams application bundle will occur if the environment variable **STREAMS_INSTALL** is set. Setting this flag to *True* ignores the local Streams install and forces the build to occur remotely using the service.

JOB_CONFIG = 'topology.jobConfigOverlays'
Key for a *JobConfig* object representing a job configuration for a submission.

SC_OPTIONS = 'topology.sc.options'
Options to be passed to IBM Streams sc command.

A topology is compiled into a Streams application bundle (*sab*) using the SPL compiler `sc`.

Additional options to be passed to `sc` may be set using this key. The value can be a single string option (e.g. `--c++std=c++11` to select C++ 11 compilation) or a list of strings for multiple options.

Setting `sc` options may be required when invoking SPL operators directly or testing SPL applications.

Warning: Options that modify the requested submission context (e.g. setting a different main composite) or deprecated options should not be specified.

New in version 1.12.10.

SERVICE_DEFINITION = 'topology.service.definition'
Streaming Analytics service definition. Identifies the Streaming Analytics service to use. The definition can be one of

- The *service credentials* copied from the *Service credentials* page of the service console (not the Streams console). Credentials are provided in JSON format. They contain such as the API key and secret, as well as connection information for the service.
- A JSON object (*dict*) created from the *service credentials*, for example with *json.loads(service_credentials)*
- A JSON object (*dict*) of the form: `{ "type": "streaming-analytics", "name": "service name", "credentials": ... }` with the *service credentials* as the value of the `credentials` key. The value of the `credentials` key can be a JSON object (*dict*) or a *str* copied from the *Service credentials* page of the service console.

This key takes precedence over *VCAP_SERVICES* and *SERVICE_NAME*.

See also:

Service definition

SERVICE_NAME = 'topology.service.name'
Streaming Analytics service name.

Selects the specific Streaming Analytics service from VCAP service definitions defined by the the environment variable **VCAP_SERVICES** or the key *VCAP_SERVICES* in the *submit* config.

See also:

Selecting the service

SSL_VERIFY = 'topology.SSLVerify'
Key for the SSL verification value passed to *requests* as its `verify` option for distributed contexts. By default set to *True*.

Note: Only `True` or `False` is supported. Behaviour is undefined when passing a path to a CA_BUNDLE file or directory with certificates of trusted CAs.

New in version 1.11.

STREAMS_CONNECTION = 'topology.streamsConnection'
Key for a `StreamsConnection` object for connecting to a running IBM Streams instance. Only supported for Streams 4.2, 4.3. Requires environment variable `STREAMS_INSTANCE_ID` to be set.

VCAP_SERVICES = 'topology.service.vcap'
Streaming Analytics service definitions including credentials in **VCAP_SERVICES** format.

Provides the connection credentials when connecting to a Streaming Analytics service using context type *STREAMING_ANALYTICS_SERVICE*. The `streaming-analytics` service to use within the service definitions is identified by name using *SERVICE_NAME*.

The key overrides the environment variable **VCAP_SERVICES**.

The value can be:

- Path to a local file containing a JSON representation of the VCAP services information.
- Dictionary containing the VCAP services information.

See also:

VCAP services

class streamsx.topology.context.**JobConfig**(*job_name=None*, *job_group=None*, *preload=False*, *data_directory=None*, *tracing=None*)

Bases: `object`

Job configuration.

JobConfig allows configuration of job that will result from submission of a `Topology` (application).

A *JobConfig* is set in the *config* dictionary passed to *submit()* using the key *JOB_CONFIG*. *add()* exists as a convenience method to add it to a submission configuration.

A *JobConfig* can also be used when submitting a Streams application bundle through the Streaming Analytics REST API method *submit_job()*.

Parameters

- **job_name** (*str*) – The name that is assigned to the job. A job name must be unique within a Streasm instance When set to *None* a system generated name is used.
- **job_group** (*str*) – The job group to use to control permissions for the submitted job.

- **preload** (*bool*) – Specifies whether to preload the job onto all resources in the instance, even if the job is not currently needed on each. Preloading the job can improve PE restart performance if the PEs are relocated to a new resource.
- **data_directory** (*str*) – Specifies the location of the optional data directory. The data directory is a path within the cluster that is running the Streams instance.
- **tracing** – Specify the application trace level. See *tracing*

Example:

```
# Submit a job with the name NewsIngester
cfg = {}
job_config = JobConfig(job_name='NewsIngester')
job_config.add(cfg)
context.submit('STREAMING_ANALYTICS_SERVICE', topo, cfg)
```

See also:

Job configuration overlays reference

add(*config*)

Add this *JobConfig* into a submission configuration object.

Parameters **config** (*dict*) – Submission configuration.

Returns config.

Return type dict

as_overlays()

Return this job configuration as a complete job configuration overlays object.

Converts this job configuration into the full format supported by IBM Streams. The returned *dict* contains:

- `jobConfigOverlays` key with an array containing a single job configuration overlay.
- an optional `comment` key containing the comment `str`.

For example with this `JobConfig`:

```
jc = JobConfig(job_name='TestIngester')
jc.comment = 'Test configuration'
jc.target_pe_count = 2
```

the returned *dict* would be:

```
{"comment": "Test configuration",
    "jobConfigOverlays":
        [{"jobConfig": {"jobName": "TestIngester"},
        "deploymentConfig": {"fusionTargetPeCount": 2, "fusionScheme": "manual
↪"}}]}
```

The returned overlays object can be saved as JSON in a file using `json.dump`. A file can be used with job submission mechanisms that support a job config overlays file, such as `streamtool submitjob` or the IBM Streams console.

Example of saving a `JobConfig` instance as a file:

```
jc = JobConfig(job_name='TestIngester')
with open('jobconfig.json', 'w') as f:
    json.dump(jc.as_overlays(), f)
```

Returns Complete job configuration overlays object built from this object.

Return type dict

New in version 1.9.

property comment

Comment for job configuration.

The comment does not change the functionality of the job configuration.

Returns Comment text, *None* if it has not been set.

Return type str

New in version 1.9.

static from_overlays(*overlays*)

Create a *JobConfig* instance from a full job configuration overlays object.

All logical items, such as `comment` and `job_name`, are extracted from *overlays*. The remaining information in the single job config overlay in `overlays` is set as `raw_overlay`.

Parameters **overlays** (*dict*) – Full job configuration overlays object.

Returns Instance representing logical view of *overlays*.

Return type *JobConfig*

New in version 1.9.

property raw_overlay

Raw Job Config Overlay.

A submitted job is configured using Job Config Overlay which is represented as a JSON. *JobConfig* exposes Job Config Overlay logically with properties such as `job_name` and `tracing`. This property (as a `dict`) allows merging of the configuration defined by this object and raw representation of a Job Config Overlay. This can be used when a capability of Job Config Overlay is not exposed logically through this class.

For example, the threading model can be set by:

```
jc = streamsx.topology.context.JobConfig()
jc.raw_overlay = {'deploymentConfig': {'threadingModel': 'manual'}}
```

Any logical items set by this object **overwrite** any set with `raw_overlay`. For example this sets the job name to to value set in the constructor (*DBIngest*) not the value in `raw_overlay` (*Ingest*):

```
jc = streamsx.topology.context.JobConfig(job_name='DBIngest')
jc.raw_overlay = {'jobConfig': {'jobName': 'Ingest'}}
```

Note: Contents of `raw_overlay` is a `dict` that is must match a single Job Config Overlay and be serializable as JSON to the correct format.

See also:

Job Config Overlay reference

New in version 1.9.

property submission_parameters
Job submission parameters.

Submission parameters values for the job. A *dict* object that maps submission parameter names to values.

New in version 1.9.

property target_pe_count
Target processing element count.

When submitted against a Streams instance *target_pe_count* provides a hint to the scheduler as to how to partition the topology across processing elements (processes) for the job execution. When a job contains multiple processing elements (PEs) then the Streams scheduler can distributed the PEs across the resources (hosts) running in the instance.

When set to `None` (the default) no hint is supplied to the scheduler. The number of PEs in the submitted job will be determined by the scheduler.

The value is only a target and may be ignored when the topology contains `isolate()` calls.

Note: Only supported in Streaming Analytics service and IBM Streams 4.2 or later.

property tracing
Runtime application trace level.

The runtime application trace level can be a string with value `error`, `warn`, `info`, `debug` or `trace`.

In addition a level from Python `logging` module can be used in with `CRITICAL` and `ERROR` mapping to `error`, `WARNING` to `warn`, `INFO` to `info` and `DEBUG` to `debug`.

Setting tracing to *None* or `logging.NOTSET` will result in the job submission using the Streams instance application trace level.

The value of `tracing` is the level as a string (`error`, `warn`, `info`, `debug` or `trace`) or None.

class streamsx.topology.context.**SubmissionResult**(*results*)
Bases: `object`

Passed back to the user after a call to submit. Allows the user to use dot notation to access dictionary elements.

cancel_job_button(*description=None*)
Display a button that will cancel the submitted job.

Used in a Jupyter IPython notebook to provide an interactive mechanism to cancel a job submitted from the notebook.

Once clicked the button is disabled unless the cancel fails.

A job may be cancelled directly using:

```
submission_result = submit(ctx_type, topology, config)
submission_result.job.cancel()
```

Parameters **description** (*str*) – Text used as the button description, defaults to value based upon the job name.

Warning: Behavior when called outside a notebook is undefined.

New in version 1.12.

property job

REST binding for the job associated with the submitted build.

Returns REST binding for running job or `None` if connection information was not available or no job was submitted.

Return type *Job*

`streamsx.topology.context.`**submit**(*ctxtype*, *graph*, *config=None*, *username=None*, *password=None*)

Submits a *Topology* (application) using the specified context type.

Used to submit an application for compilation into a Streams application and execution within an Streaming Analytics service or IBM Streams instance.

ctxtype defines how the application will be submitted, see *ContextTypes*.

The parameters *username* and *password* are only required when submitting to an IBM Streams instance and it is required to access the Streams REST API from the code performing the submit. Accessing data from views created by *view()* requires access to the Streams REST API.

Parameters

- **ctxtype** (*str*) – Type of context the application will be submitted to. A value from *ContextTypes*.
- **graph** (*Topology*) – The application topology to be submitted.
- **config** (*dict*) – Configuration for the submission.
- **username** (*str*) – Deprecated: Username for the Streams REST api. Use environment variable `STREAMS_USERNAME` if using user-password authentication.
- **password** (*str*) – Deprecated: Password for *username*. Use environment variable `STREAMS_PASSWORD` if using user-password authentication.

Returns Result of the submission. For details of what is contained see the *ContextTypes* constant passed as *ctxtype*.

Return type *SubmissionResult*

`streamsx.topology.context.`**build**(*topology*, *config=None*, *dest=None*, *verify=None*)

Build a topology to produce a Streams application bundle.

Builds a topology using *submit()* with context type *BUNDLE*. The result is a sab file on the local file system along with a job config overlay file matching the application.

The build uses a build service or a local install, see *BUNDLE* for details.

Parameters

- **topology** (*Topology*) – Application topology to be built.
- **config** (*dict*) – Configuration for the build.
- **dest** (*str*) – Destination directory for the sab and JCO files. Default is context specific.
- **verify** – SSL verification used by requests when using a build service. Defaults to enabling SSL verification.

Returns

3-element tuple containing

- **bundle_path** (*str*): path to the bundle (sab file) or `None` if not created.

- **jco_path** (*str*): path to file containing the job config overlay for the application or `None` if not created.
- **result** (*SubmissionResult*): value returned from `submit`.

See also:

BUNDLE for details on how to configure the build service to use.

New in version 1.14.

`streamsx.topology.context.`**`run`**(*topology*, *config=None*, *job_name=None*, *verify=None*, *ctxtype='DISTRIBUTED'*)

Run a topology in a distributed Streams instance.

Runs a topology using *submit()* with context type *DISTRIBUTED* (by default). The result is running Streams job.

Parameters

- **`topology`** (`Topology`) – Application topology to be run.
- **`config`** (*dict*) – Configuration for the build.
- **`job_name`** (*str*) – Optional job name. If set will override any job name in *config*.
- **`verify`** – SSL verification used by requests when using a build service. Defaults to enabling SSL verification.
- **`ctxtype`** (*str*) – Context type for submission.

Returns

2-element tuple containing

- **job** (*Job*): REST binding object for the running job or `None` if no job was submitted.
- **result** (*SubmissionResult*): value returned from `submit`.

See also:

DISTRIBUTED for details on how to configure the Streams instance to use.

New in version 1.14.

1.4 streamsx.topology.schema

Schemas for streams.

1.4.1 Overview

A stream represents an unbounded flow of tuples with a declared schema so that each tuple on the stream complies with the schema. A stream's schema may be one of:

- `StreamsSchema` structured schema - a tuple is a sequence of attributes, and an attribute is a named value of a specific type.
- *Json* a tuple is a JSON object.
- *String* a tuple is a string.
- *Python* a tuple is any Python object, effectively an untyped stream.

1.4.2 Structured schemas

A structured schema is a sequence of attributes, and an attribute is a named value of a specific type. For example a stream of sensor readings can be represented as a schema with three attributes `sensor_id`, `ts` and `reading` with types of `int64`, `int64` and `float64` respectively.

This schema can be declared a number of ways:

Python 3.6:

```
class SensorReading(typing.NamedTuple):
    sensor_id: int
    ts: int
    reading: float

sensors = raw_readings.map(parse_sensor, schema=SensorReading)
```

Python 3:

```
SensorReading = typing.NamedTuple('SensorReading',
    [('sensor_id', int), ('ts', int), ('reading', float)]

sensors = raw_readings.map(parse_sensor, schema=SensorReading)
```

Python 3:

```
sensors = raw_readings.map(parse_sensor,
    schema='tuple<int64 sensor_id, int64 ts, float64 reading>')
```

The supported types are defined by IBM Streams and are listed in *StreamSchema*.

Structured schemas provide type-safety and efficient network serialization when compared to passing a `dict` using *Python* streams.

Streams with structured schemas can be interchanged with any IBM Streams application using *publish()* and *subscribe()* maintaining type safety.

1.4.3 Defining a stream's schema

Every stream within a *Topology* has defined schema. The schema may be defined explictly (for example *map()* or *subscribe()*) or implicity (for example *filter()* produces a stream with the same schema as its input stream).

Explictly defining a stream's schema is flexible and various types of values are accepted as the schema.

- Builtin types as aliases for common schema types:
  - `json` (module) - for *Json*
  - `str` - for *String*
  - `object` - for *Python*
- Values of the enumeration *CommonSchema*
- An instance of `typing.NamedTuple` (Python 3)
- An instance of *StreamSchema*
- A string of the format `tuple<...>` defining the attribute names and types. See *StreamSchema* for details on the format and types supported.

- A string containing a namespace qualified SPL stream type (e.g. `com.ibm.streams.geospatial::FlightPathEncounterTypes.Observation3D`)

1.4.4 Module contents

Functions

is_common	Is *schema* an common schema.

Classes

CommonSchema	Common stream schemas for interoperability within Streams applications.
StreamSchema	Defines a schema for a structured stream.

`streamsx.topology.schema.`**`is_common`**(*schema*)

Is *schema* an common schema.

Parameters **`schema`** – Scheme to test.

Returns `True` if schema is a common schema, otherwise `False`.

Return type bool

`class` `streamsx.topology.schema.`**`StreamSchema`**(*schema*)

Bases: `object`

Defines a schema for a structured stream.

On a structured stream a tuple is a sequence of attributes, and an attribute is a named value of a specific type.

The supported types are defined by IBM Streams and include such types as *int8*, *int16*, *rstring* and *list<float32>*.

A structured schema can be defined using a `typing.NamedTuple` in Python 3, a string with the syntax `tuple<type name [,...]>` or an instance of this class.

typing.NamedTuple:

A `typing.NamedTuple` can be used to define a structured schema with the field names and types mapping to the structured schema attribute names and types.

Python types are mapped to IBM Streams types as follows:

Python type	IBM Streams type
`str`	`rstring`
`bool`	`boolean`
`int`	`int64`
`float`	`float64`
`decimal.Decimal`	`decimal128`
`complex`	`complex64`
`bytes`	`blob`
`streamsx.spl.types.Timestamp`	`timestamp`
`datetime.datetime`	`timestamp`
`typing.List[T]`	`list<T>`
`typing.Set[T]`	`set<T>`
`typing.Mapping[K,V]`	`map<K,V>`
`typing.Optional[T]`	`optional<T>`

Note: Tuples on a stream with a schema defined by a `typing.NamedTuple` instance are passed into callables as instance of a named tuple with the the correct field names and types. This is not guaranteed to be the same class instance as the one used to declare the schema.

Tuple string:

A string of the format *tuple<type name [,. . .]>* can be used to define a structured schema, where *type* is an IBM Streams type.

Example:

```
tuple<rstring id, timestamp ts, float64 value>
```

represents a schema with three attributes suitable for a sensor reading.

IBM Streams types:

Type	Description	Python representation	Conversion from Python
`boolean`	True or False	`bool`	`bool(value)`
`int8`	8-bit signed integer	`int`	`int(value)` truncated to 8 bits
`int16`	16-bit signed integer	`int`	`int(value)` truncated to 16 bits
`int32`	32-bit signed integer	`int`	`int(value)` truncated to 32 bits
`int64`	64-bit signed integer	`int`	`int(value)`
`uint8`	8-bit unsigned integer	`int`	•
`uint16`	16-bit unsigned integer	`int`	•
`uint32`	32-bit unsigned integer	`int`	•

Continued on next page

Table 7 – continued from previous page

Type	Description	Python representation	Conversion from Python
`uint64`	64-bit unsigned integer	`int`	•
`float32`	32-bit binary floating point	`float`	`float(value)` truncated to 32 bits
`float64`	64-bit binary floating point	`float`	`float(value)`
`decimal32`	32-bit decimal floating point	`decimal.Decimal`	`decimal.Decimal(value)` normalized to IEEE 754 decimal32
`decimal64`	64-bit decimal floating point	`decimal.Decimal`	`decimal.Decimal(value)` normalized to IEEE 754 decimal64
`decimal128`	128-bit decimal floating point	`decimal.Decimal`	`decimal.Decimal(value)` normalized to IEEE 754 decimal128
`complex32`	complex with *float32* values	`complex`	`complex(value)` with real and imaginary values truncated to 32 bits
`complex64`	complex with *float64* values	`complex`	`complex(value)`
`timestamp`	Nanosecond timestamp	*`Timestamp`*	•
`rstring`	UTF-8 string	`str`	`str(value)`
`rstring[N]`	Bounded UTF-8 string	`str`	`str(value)`
`ustring`	UTF-16 string	`str`	`str(value)`
`blob`	Sequence of bytes	`memoryview`	•
`list<T>`	List with elements of type *T*	`list`	•
`list<T>[N]`	Bounded list	`list`	•
`set<T>`	Set with elements of type *T*	`set`	•
`set<T>[N]`	Bounded set	`set`	•
`map<K,V>`	Map with typed keys and values	`dict`	•

Continued on next page

Table 7 – continued from previous page

Type	Description	Python representation	Conversion from Python
`map<K,V>[N]`	Bounded map, limted to N pairs	`dict`	•
`optional<T>`	Optional value of type *T*	Value of type *T*, or None	Value of for type `T`
`enum{id [,...]}`	Enumeration	Not supported	Not supported
`xml`	XML value	Not supported	Not supported
`tuple<type name [, ...]>`	Nested tuple	Not supported	Not supported

Note: Type *optional<T>* requires IBM Streams 4.3 or later.

Python representation is how an attribute value in a structured schema is passed into a Python function.

Conversion from Python indicates how a value from Python is converted to an attribute value in a structured schema. For example a value `v` assigned to `float64` attribute is converted as though `float(v)` is called first, thus `v` may be a `float`, `int` or any type that has a `__float__` method.

When a type is not supported in Python it can only be used in a schema used for streams produced and consumed by invocation of SPL operators.

A *StreamSchema* can be created by passing a string of the form `tuple<...>` or by passing the name of an SPL type from an SPL toolkit, for example `com.ibm.streamsx.transportation.vehicle::VehicleLocation`.

Attribute names must start with an ASCII letter or underscore, followed by ASCII letters, digits, or underscores.

When a tuple on a structured stream is passed into a Python callable it is converted to a `dict`, `tuple` or named tuple object containing all attributes of the stream tuple. See *`style()`*, *`as_dict()`* and *`as_tuple()`* for details.

When a Python object is submitted to a structured stream, for example as the return from the function invoked in a *`map()`* with the *schema* parameter set, it must be:

- A Python `dict`. Attributes are set by name using value in the dict for the name. If a value does not exist (the name does not exist as a key) or is set to *None* then the attribute has its default value, zero, false, empty list or string etc.
- A Python `tuple` or named tuple. Attributes are set by position, with the first attribute being the value at index 0 in the Python *tuple*. If a value does not exist (the tuple has less values than the structured schema) or is set to *None* then the attribute has its default value, zero, false, empty list or string etc.

Parameters **`schema`** (*`str`*) – Schema definition. Either a schema definition or the name of an SPL type.

`as_dict`()

Create a structured schema that will pass stream tuples into callables as `dict` instances. This allows a return to the default calling style for a structured schema.

If this instance represents a common schema then it will be returned without modification. Stream tuples with common schemas are always passed according to their definition.

Returns Schema passing stream tuples as `dict` if allowed.

Return type *StreamSchema*

New in version 1.8.

as_tuple(*named=None*)

Create a structured schema that will pass stream tuples into callables as `tuple` instances.

If this instance represents a common schema then it will be returned without modification. Stream tuples with common schemas are always passed according to their definition.

Passing as tuple

When *named* evaluates to `False` then each stream tuple will be passed as a `tuple`. For example with a structured schema of `tuple<rstring id, float64 value>` a value is passed as `('TempSensor', 27.4)` and access to the first attribute is `t[0]` and the second as `t[1]` where `t` represents the passed value..

Passing as named tuple

When *named* is `True` or a `str` then each stream tuple will be passed as a named tuple. For example with a structured schema of `tuple<rstring id, float64 value>` a value is passed as `('TempSensor', 27.4)` and access to the first attribute is `t.id` (or `t[0]`) and the second as `t.value` (`t[1]`) where `t` represents the passed value.

Warning: If an schema's attribute name is not a valid Python identifier or starts with an underscore then it will be renamed as positional name `_n`. For example, with the schema `tuple<int32 a, int32 def, int32 id>` the field names are `a`, `_1`, `_2`.

The value of *named* is used as the name of the named tuple class with `StreamTuple` used when *named* is `True`.

It is not guaranteed that the class of the namedtuple is the same for all callables processing tuples with the same structured schema, only that the tuple is a named tuple with the correct field names.

Parameters **named** – Pass stream tuples as a named tuple. If not set then stream tuples are passed as instances of `tuple`.

Returns Schema passing stream tuples as `tuple` if allowed.

Return type *StreamSchema*

New in version 1.8.

New in version 1.9: Addition of *named* parameter.

extend(*schema*)

Extend a structured schema by another.

For example extending `tuple<rstring id, timestamp ts, float64 value>` with `tuple<float32 score>` results in `tuple<rstring id, timestamp ts, float64 value, float32 score>`.

Parameters **schema** (`StreamSchema`) – Schema to extend this schema by.

Returns New schema that is an extension of this schema.

Return type *StreamSchema*

schema()

Private method. May be removed at any time.

property style

Style stream tuples will be passed into a callable.

For the common schemas the style is fixed:

- `CommonSchema.Python` - `object` - Stream tuples are arbitrary objects.
- `CommonSchema.String` - `str` - Stream tuples are unicode strings.
- `CommonSchema.Json` - `dict` - Stream tuples are a `dict` that represents the JSON object.

For a structured schema the supported styles are:

- `dict` - Stream tuples are passed as a `dict` with the key being the attribute name and and the value the attribute value. This is the default.
  - E.g. with a schema of `tuple<rstring id, float32 value>` a value is passed as `{'id':'TempSensor', 'value':20.3}`.
- `tuple` - Stream tuples are passed as a `tuple` with the value being the attributes value in order. A schema is set to pass stream tuples as tuples using *as_tuple()*.
  - E.g. with a schema of `tuple<rstring id, float32 value>` a value is passed as `('TempSensor', 20.3)`.
- `namedtuple` - Stream tuples are passed as a named tuple (see `collections.namedtuple`) with the value being the attributes value in order. Field names correspond to the attribute names of the schema. A schema is set to pass stream tuples as named tuples using *as_tuple()* setting the *named* parameter.

  Returns Class of tuples that will be passed into callables.

  Return type type

New in version 1.8.

New in version 1.9: Support for namedtuple.

class streamsx.topology.schema.**CommonSchema**

Bases: `enum.Enum`

Common stream schemas for interoperability within Streams applications.

Streams application can publish streams that are subscribed to by other applications. Use of common schemas allow streams connections regardless of the application implementation language.

Python applications publish streams using *publish()* and subscribe using *subscribe()*.

- *Python* - Stream constains Python objects.
- *Json* - Stream contains JSON objects.
- *String* - Stream contains strings.
- *Binary* - Stream contains binary tuples.
- *XML* - Stream contains XML documents.

Binary = <streamsx.topology.schema.StreamSchema object>

Stream where each tuple is a binary object (sequence of bytes).

> **Warning:** *Binary* is not yet supported for Python applications.

Json = <streamsx.topology.schema.StreamSchema object>

Stream where each tuple is logically a JSON object.

Json can be used as a natural interchange format between Streams applications implemented in different programming languages. All languages supported by Streams support publishing and subscribing to JSON streams.

A Python callable receives each tuple as a *dict* as though it was created from `json.loads(json_formatted_str)` where *json_formatted_str* is the JSON formatted representation of tuple.

Python objects that are to be converted to JSON objects must be supported by *JSONEncoder*. If the object is not a *dict* then it will be converted to a JSON object with a single key *payload* containing the value.

Python = <streamsx.topology.schema.StreamSchema object>
Stream where each tuple is a Python object. Each object must be picklable to allow execution in a distributed environment where streams can connect processes running on the same or different resources.

Python streams can only be used by Python applications.

String = <streamsx.topology.schema.StreamSchema object>
Stream where each tuple is a string.

String can be used as a natural interchange format between Streams applications implemented in different programming languages. All languages supported by Streams support publishing and subscribing to string streams.

A Python callable receives each tuple as a *str* object.

Python objects are converted to strings using `str(obj)`.

XML = <streamsx.topology.schema.StreamSchema object>
Stream where each tuple is an XML document.

> **Warning:** *XML* is not yet supported for Python applications.

extend(*schema*)
Extend a structured schema by another.

Parameters **schema** (`StreamSchema`) – Schema to extend this schema by.

Returns New schema that is an extension of this schema.

Return type *StreamSchema*

schema()
Private method. May be removed at any time.

1.5 streamsx.topology.state

Application state.

1.5.1 Overview

Stateful applications are ones that include callables that are classes and thus can maintain state as instance variables.

By default any state is reset to its initial state after a processing element (PE) restart. A restart may occur due to:

- a failure in the PE or its resource,
- a explicit PE restart request,
- or a parallel region width change (IBM Streams 4.3 or later)

The application or a portion of it may be configured to maintain state after a PE restart by one of two mechanisms.

- Consistent region. A consistent region is a subgraph where the states of callables become consistent by processing all the tuples within defined points on a stream. After a PE restart all callables in the region are reset to the last consistent point, so that the state of all callables represents the processing of the same input tuples to the region.
  - *streamsx.topology.topology.Stream.set_consistent()*
  - *ConsistentRegionConfig*
  - Consistent region overview
- Checkpointing, each stateful callable is checkpointed periodically and after a PE restart its callables are reset to their most recent checkpointed state.
  - *streamsx.topology.topology.Topology.checkpoint_period*

1.5.2 Stateful callables

Use of a class instance allows a transformation (for example *map()*) to be stateful by maintaining state in instance attributes across invocations.

When the callable is in a consistent region or checkpointing then it is serialized using *dill*. The default serialization may be modified by using the standard Python pickle mechanism of `__getstate__` and `__setstate__`. This is required if the state includes objects that cannot be serialized, for example file descriptors. For details see See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects .

If the callable as `__enter__` and `__exit__` context manager methods then `__enter__` is called after the object has been deserialized by *dill*. Thus `__enter__` is used to recreate runtime objects that cannot be serialized such as open files or sockets.

1.5.3 Module contents

Classes

ConsistentRegionConfig	A *ConsistentRegionConfig* configures a consistent region.

class streamsx.topology.state.**ConsistentRegionConfig**(*trigger=None*, *period=None*, *drain_timeout=180*, *reset_timeout=180*, *max_consecutive_attempts=5*)

Bases: `object`

A *ConsistentRegionConfig* configures a consistent region.

The recommended way to create a *ConsistentRegionConfig* is to call either *operator_driven()* or *periodic()*.

Parameters

- **trigger** (ConsistentRegionConfig.Trigger) – Determines how the drain/checkpoint cycle of the consistent region is triggered.
- **period** – The trigger period. If the trigger is *PERIODIC*, this must be specified, otherwise it may not be specfied. This may be either a `datetime.timedelta` value or the number of seconds as a *float*.
- **drain_timeout** – Indicates the maximum time in seconds that the drain and checkpoint of the region is allotted to finish processing. If the process takes longer than the specified time, a failure is reported and the region is reset to the point of the previously successfully established consistent state. The value must be specified as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **reset_timeout** – Indicates the maximum time in seconds that the reset of the region is allotted to finish processing. If the process takes longer than the specified time, a failure is reported and another reset of the region is attempted. The value must be specified as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **max_consecutive_attempts** (*int*) – Indicates the maximum number of consecutive attempts to reset a consistent region. After a failure, if the maximum number of attempts is reached, the region stops processing new tuples. After the maximum number of consecutive attempts is reached, a region can be reset only with manual intervention or with a program with a call to a method in the consistent region controller. This must be an integer value between 1 and 2147483647, inclusive. If not specified, the default value is 5.

Example:

```
# set source to be a the start of an operator driven consistent region
# with a drain timeout of five seconds and a reset timeout of twenty seconds.
source.set_consistent(ConsistentRegionConfig.operatorDriven(drain_timeout=5,␣
↪reset_timeout=20))
```

See also:

set_consistent()

New in version 1.11.

class Trigger

Bases: `enum.Enum`

Defines how the drain-checkpoint cycle of a consistent region is triggered. .. versionadded:: 1.11

OPERATOR_DRIVEN = 1

Region is triggered by the start operator.

PERIODIC = 2

Region is triggered periodically.

static operator_driven(*drain_timeout=180*, *reset_timeout=180*, *max_consecutive_attempts=5*)

Define an operator-driven consistent region configuration. The source operator triggers drain and checkpoint cycles for the region.

Parameters

- **drain_timeout** – The drain timeout, as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **reset_timeout** – The reset timeout, as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **max_consecutive_attempts** (*int*) – The maximum number of consecutive attempts to reset the region. This must be an integer value between 1 and 2147483647, inclusive. If not specified, the default value is 5.

Returns the configuration.

Return type *ConsistentRegionConfig*

static periodic(*period*, *drain_timeout=180*, *reset_timeout=180*, *max_consecutive_attempts=5*)

Create a periodic consistent region configuration. The IBM Streams runtime will trigger a drain and checkpoint the region periodically at the time interval specified by *period*.

Parameters

- **period** – The trigger period. This may be either a `datetime.timedelta` value or the number of seconds as a *float*.
- **drain_timeout** – The drain timeout, as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **reset_timeout** – The reset timeout, as either a `datetime.timedelta` value or the number of seconds as a *float*. If not specified, the default value is 180 seconds.
- **max_consecutive_attempts** (*int*) – The maximum number of consecutive attempts to reset the region. This must be an integer value between 1 and 2147483647, inclusive. If not specified, the default value is 5.

Returns the configuration.

Return type *ConsistentRegionConfig*

1.6 streamsx.topology.composite

Composite transformations.

New in version 1.14.

1.6.1 Module contents

Classes

Composite	Composite transformations support a single logical transformation being a composite of one or more basic transformations.
ForEach	Abstract composite for each transformation.
Map	Abstract composite map transformation.
Source	Abstract composite source.

class streamsx.topology.composite.**Composite**

Bases: `abc.ABC`

Composite transformations support a single logical transformation being a composite of one or more basic transformations.

A composite transformation is implemented as a sub-class of *Source*, *Map* or *ForEach* whose `populate` method populates the topology with the required basic transformations. For example a `Source` composite might have use `source()` followed by a `filter()` to filter out unwanted events and then a `map()` to parse the event into a structured schema.

Composites may use other composites during `populate`.

Composites can control how the basic transformations are visually represented. By default any transformations within a composite are grouped visually. A composite may alter this using these attributes of the composite instance:

- `kind` - Sets the name of operator kind for a group or single operator. Defaults to a combination of the module and class name of the composite, e.g. `streamsx.standard.utility::Sequence`. Set to a false value to disable any modification of the visual representation of the composite's transformations.
- `group` - Set to a false value to disable any grouping of multiple transformations. Defaults to `True` to enable grouping.

The values of `kind` and `group` are checked after the expansion of the composite using `populate`.

class `streamsx.topology.composite.`**Source**

Bases: *streamsx.topology.composite.Composite*

Abstract composite source.

An instance of a subclass can be passed to `source()` to create a source stream that is composed of one or more basic transformations.

Example assuming `RawTweets` is Python iterable that produces raw tweets:

```
class Tweets(streamsx.topology.composite.Source):
    def __init__(self, track):
        self.track = track

    def populate(self, topology, name, **options):
        # get all the tweets
        tweets = topology.source(RawTweets(track=self.track), name=name)
        # filter so that only with a message are returned
        return tweets.filter(lambda tweet : tweet['text'])
```

This class can then be used as follows:

```
topo = Topology()
gf_tweets = topo.source(Tweets(track=['glutenfree', 'gf']))
```

abstract populate(*topology*, *name*, ***options*)

Populate the topology with this composite source.

Parameters

- **topology** (*Topology*) – Topology containing the source.
- **name** (`Optional[str]`) – Name passed into `source`.
- ****options** – Future options passed to `source`.

Returns Single stream representing the source.

Return type *Stream*

class streamsx.topology.composite.**Map**
Bases: *streamsx.topology.composite.Composite*

Abstract composite map transformation.

An instance of a subclass can be passed to *map()* to create a stream that is composed of one or more basic transformations of an input stream.

Example:

```
class WordCount(streamsx.topology.composite.Map):
    def __init__(self, period, update):
        self.period = period
        self.update = update

    def populate(self, topology, stream, schema, name, **options):
        words = stream.flat_map(lambda line : line.split())
        win = words.last(size=self.period).trigger(self.update).partition(lambda
↪s : s)
        return win.aggregate(lambda values : (values[0], len(values)))
```

abstract populate(*topology*, *stream*, *schema*, *name*, ***options*)
Populate the topology with this composite map transformation.

Parameters

- **topology** (*Topology*) – Topology containing the composite map.
- **stream** (*Stream*) – Stream to be transformed.
- **schema** (Union[*StreamSchema*, *CommonSchema*, str, NamedTuple]) – Schema passed into map.
- **name** (Optional[str]) – Name passed into map.
- ****options** – Future options passed to map.

Returns Single stream representing the transformation of *stream*.

Return type *Stream*

class streamsx.topology.composite.**ForEach**
Bases: *streamsx.topology.composite.Composite*

Abstract composite for each transformation.

An instance of a subclass can be passed to *for_each()* to create a sink (stream termination) that is composed of one or more basic transformations of an input stream.

abstract populate(*topology*, *stream*, *name*, ***options*)
Populate the topology with this composite for each transformation.

Parameters

- **topology** (*Topology*) – Topology containing the composite map.
- **stream** (*Stream*) – Stream to be transformed.
- **name** (Optional[str]) – Name passed into for_each.
- ****options** – Future options passed to for_each.

Returns Termination for this composite transformation of *stream*.

Return type *Sink*

1.7 streamsx.topology.tester

Testing support for streaming applications.

1.7.1 Overview

Allows testing of a streaming application by creation conditions on streams that are expected to become valid during the processing. *Tester* is designed to be used with Python's *unittest* module.

A complete application may be tested or fragments of it, for example a sub-graph can be tested in isolation that takes input data and scores it using a model.

Supports execution of the application on *STREAMING_ANALYTICS_SERVICE*, *DISTRIBUTED* or *STANDALONE*.

A *Tester* instance is created and associated with the `Topology` to be tested. Conditions are then created against streams, such as a stream must receive 10 tuples using *tuple_count()*.

Here is a simple example that tests a filter correctly only passes tuples with values greater than 5:

```
import unittest
from streamsx.topology.topology import Topology
from streamsx.topology.tester import Tester

class TestSimpleFilter(unittest.TestCase):

    def setUp(self):
        # Sets self.test_ctxtype and self.test_config
        Tester.setup_streaming_analytics(self)

    def test_filter(self):
        # Declare the application to be tested
        topology = Topology()
        s = topology.source([5, 7, 2, 4, 9, 3, 8])
        s = s.filter(lambda x : x > 5)

        # Create tester and assign conditions
        tester = Tester(topology)
        tester.contents(s, [7, 9, 8])

        # Submit the application for test
        # If it fails an AssertionError will be raised.
       tester.test(self.test_ctxtype, self.test_config)
```

A stream may have any number of conditions and any number of streams may be tested.

A *local_check()* is supported where a method of the unittest class is executed once the job becomes healthy. This performs checks from the context of the Python unittest class, such as checking external effects of the application or using the REST api to monitor the application.

A test fails-fast if any of the following occur:

- Any condition fails. E.g. a tuple failing a *tuple_check()*.
- The *local_check()* (if set) raises an error.
- **The job for the test:**
  - Fails to become healthy.
  - Becomes unhealthy during the test run.

- Any processing element (PE) within the job restarts.

A test timeouts if it does not fail but its conditions do not become valid. The timeout is not fixed as an absolute test run time, but as a time since "progress" was made. This can allow tests to pass when healthy runs are run in a constrained environment that slows execution. For example with a tuple count condition of ten, progress is indicated by tuples arriving on a stream, so that as long as gaps between tuples are within the timeout period the test remains running until ten tuples appear.

Note: The test timeout value is not configurable.

Note: The submitted job (application under test) has additional elements (streams & operators) inserted to implement the conditions. These are visible through various APIs including the Streams console raw graph view. Such elements are put into the *Tester* category.

Note: The package streamsx.testing provides nose plugins to provide control over tests without having to modify their source code.

Changed in version 1.9: - Python 2.7 supported (except with Streaming Analytics service).

1.7.2 Module contents

Classes

Tester	Testing support for a Topology.

class streamsx.topology.tester.**Tester**(*topology*)

Bases: object

Testing support for a Topology.

Allows testing of a Topology by creating conditions against the contents of its streams.

Conditions may be added to a topology at any time before submission.

If a topology is submitted directly to a context then the graph is not modified. This allows testing code to be inserted while the topology is being built, but not acted upon unless the topology is submitted in test mode.

If a topology is submitted through the test method then the topology may be modified to include operations to ensure the conditions are met.

Warning: For future compatibility applications under test should not include intended failures that cause a processing element to stop or restart. Thus, currently testing is against expected application behavior.

Parameters **topology** – Topology to be tested.

add_condition(*stream*, *condition*)

Add a condition to a stream.

Conditions are normally added through *tuple_count()*, *contents()* or *tuple_check()*.

This allows an additional conditions that are implementations of `Condition`.

Parameters

- **stream** (`Stream`) – Stream to be tested.
- **condition** (`Condition`) – Arbitrary condition.

Returns stream

Return type *Stream*

contents(*stream*, *expected*, *ordered=True*)

Test that a stream contains the expected tuples.

Parameters

- **stream** (`Stream`) – Stream to be tested.
- **expected** (*list*) – Sequence of expected tuples.
- **ordered** (*bool*) – True if the ordering of received tuples must match expected.

Returns stream

Return type *Stream*

eventual_result(*stream*, *checker*)

Test a stream reaches a known result or state.

Creates a test condition that the tuples on a stream eventually reach a known result or state. Each tuple on *stream* results in a call to `checker(tuple_)`.

The return from *checker* is handled as:

- `None` - The condition requires more tuples to become valid.
- *true value* - The condition has become valid.
- *false value* - The condition has failed. Once a condition has failed it can never become valid.

Thus *checker* is typically stateful and allows ensuring that condition becomes valid from a set of input tuples. For example in a financial application the application under test may need to achieve a final known balance, but due to timings of windows the number of tuples required to set the final balance may be variable.

Once the condition becomes valid any false value, except `None`, returned by processing of subsequent tuples will cause the condition to fail.

Returning `None` effectively never changes the state of the condition.

Parameters

- **stream** (`Stream`) – Stream to be tested.
- **checker** (*callable*) – Callable that returns evaluates the state of the stream with result to the result.

New in version 1.11.

static get_streams_version(*test*)

Returns IBM Streams product version string for a test.

Returns the product version corresponding to the test's setup. For `STANDALONE` and `DISTRIBUTED` the product version corresponds to the version defined by the environment variable `STREAMS_INSTALL`.

Parameters **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.

local_check(*callable*)

Perform local check while the application is being tested.

A call to *callable* is made after the application under test is submitted and becomes healthy. The check is in the context of the Python runtime executing the unittest case, typically the callable is a method of the test case.

The application remains running until all the conditions are met and *callable* returns. If *callable* raises an error, typically through an assertion method from *unittest* then the test will fail.

Used for testing side effects of the application, typically with *STREAMING_ANALYTICS_SERVICE* or *DISTRIBUTED*. The callable may also use the REST api for context types that support it to dynamically monitor the running application.

The callable can use *submission_result* and *streams_connection* attributes from *Tester* instance to interact with the job or the running Streams instance. These REST binding classes can be obtained as follows:

- *Job* - `tester.submission_result.job`
- *Instance* - `tester.submission_result.job.get_instance()`
- *StreamsConnection* - `tester.streams_connection`

Simple example of checking the job is healthy:

```
import unittest
from streamsx.topology.topology import Topology
from streamsx.topology.tester import Tester

class TestLocalCheckExample(unittest.TestCase):
    def setUp(self):
        Tester.setup_distributed(self)

    def test_job_is_healthy(self):
        topology = Topology()
        s = topology.source(['Hello', 'World'])

        self.tester = Tester(topology)
        self.tester.tuple_count(s, 2)

        # Add the local check
        self.tester.local_check = self.local_checks

        # Run the test
        self.tester.test(self.test_ctxtype, self.test_config)


    def local_checks(self):
        job = self.tester.submission_result.job
        self.assertEqual('healthy', job.health)
```

Warning: A local check must not cancel the job (application under test).

Warning: A local check is not supported in standalone mode.

Parameters **callable** – Callable object.

static minimum_streams_version(*test*, *required_version*)
Checks test setup matches a minimum required IBM Streams version.

Parameters

- **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.
- **required_version** (*str*) – VRMF of the minimum version the test requires. Examples are `'4.3'`, `4.2.4`.

Returns True if the setup fulfills the minimum required version, false otherwise.

Return type bool

static require_streams_version(*test*, *required_version*)
Require a test has minimum IBM Streams version.

Skips the test if the test's setup is not at the required minimum IBM Streams version.

Parameters

- **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.
- **required_version** (*str*) – VRMF of the minimum version the test requires. Examples are `'4.3'`, `4.2.4`.

resets(*minimum_resets=10*)
Create a condition that randomly resets consistent regions. The condition becomes valid when each consistent region in the application under test has been reset *minimum_resets* times by the tester.

The resets are performed at arbitrary intervals scaled to the period of the region (if it is periodically triggered).

Note: A region is reset by initiating a request though the Job Control Plane. The reset is not driven by any injected failure, such as a PE restart.

Parameters **minimum_resets** (*int*) – Minimum number of resets for each region.

New in version 1.11.

run_for(*duration*)
Run the test for a minimum number of seconds.

Creates a test wide condition that becomes *valid* when the application under test has been running for *duration* seconds. Maybe be called multiple times, the test will run as long as the maximum value provided.

Can be used to test applications without any externally visible streams, or streams that do not have testable conditions. For example a complete application may be tested by runnning it for for ten minutes and use *local_check()* to test any external impacts, such as messages published to a message queue system.

Parameters **duration** (*float*) – Minimum number of seconds the test will run for.

static setup_distributed(*test*, *verbose=None*)
Set up a unittest.TestCase to run tests using IBM Streams distributed mode.

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config - Test configuration.

Parameters

- **test** (*unittest.TestCase*) – Test case to be set up to run tests using Tester
- **verbose** (*bool*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

Returns: None

Cloud Pak for Data integrated instance configuration

These environment variables define how the test is built and submitted.

- `CP4D_URL` - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*.
- `STREAMS_INSTANCE_ID` - Streams service instance name.
- `STREAMS_USERNAME` - (optional) User name to submit the test as, defaulting to the current operating system user name.
- `STREAMS_PASSWORD` - Password for authentication.

Cloud Pak for Data standalone instance configuration

These environment variables define how the test is built and submitted.

- `STREAMS_BUILD_URL` - Endpoint for the Streams build service.
- `STREAMS_REST_URL` - Endpoint for the Streams SWS (REST) service.
- `STREAMS_USERNAME` - (optional) User name to submit the test as, defaulting to the current operating system user name.
- `STREAMS_PASSWORD` - Password for authentication.

Streams 4.2 & 4.3 instance configuration

Requires a local IBM Streams install define by the `STREAMS_INSTALL` environment variable. If `STREAMS_INSTALL` is not set then the test is skipped.

The Streams instance to use is defined by the environment variables:

- `STREAMS_ZKCONNECT` - Zookeeper connection string (optional)
- `STREAMS_DOMAIN_ID` - Domain identifier
- `STREAMS_INSTANCE_ID` - Instance identifier

The user used to submit and monitor the job is set by the optional environment variables:

- `STREAMS_USERNAME` - User name defaulting to *streamsadmin*.
- `STREAMS_PASSWORD` - User password defaulting to *passw0rd*.

The defaults match the setup for testing on a IBM Streams Quick Start Edition (QSE) virtual machine.

Warning: `streamtool` is used to submit the job and requires that `streamtool` does not prompt for authentication. This is achieved by using `streamtool genkey`.

See also:

Generating authentication keys for IBM Streams

static setup_standalone(*test*, *verbose=None*)

Set up a unittest.TestCase to run tests using IBM Streams standalone mode.

Requires a local IBM Streams install define by the `STREAMS_INSTALL` environment variable. If `STREAMS_INSTALL` is not set, then the test is skipped.

A standalone application under test will run until a condition fails or all the streams are finalized or when the *`run_for()`* time (if set) elapses. Applications that include infinite streams must include set a run for time using *`run_for()`* to ensure the test completes

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config- Test configuration.

Parameters

- **test** (*`unittest.TestCase`*) – Test case to be set up to run tests using Tester
- **verbose** (*`bool`*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

Returns: None

static setup_streaming_analytics(*test*, *service_name=None*, *force_remote_build=False*, *verbose=None*)

Set up a unittest.TestCase to run tests using Streaming Analytics service on IBM Cloud.

The service to use is defined by:

- VCAP_SERVICES environment variable containing *streaming_analytics* entries.
- service_name which defaults to the value of STREAMING_ANALYTICS_SERVICE_NAME environment variable.

If VCAP_SERVICES is not set or a service name is not defined, then the test is skipped.

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config - Test configuration.

Parameters

- **test** (*`unittest.TestCase`*) – Test case to be set up to run tests using Tester
- **service_name** (*`str`*) – Name of Streaming Analytics service to use. Must exist as an entry in the VCAP services. Defaults to value of STREAMING_ANALYTICS_SERVICE_NAME environment variable.
- **force_remote_build** (*`bool`*) – Force use of the Streaming Analytics build service. If *false* and `STREAMS_INSTALL` is set then a local build will be used if the local environment is suitable for the service, otherwise the Streams application bundle is built using the build service.
- **verbose** (*`bool`*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

If run with Python 2 the test is skipped,.

Returns: None

test(*ctxtype*, *config=None*, *assert_on_fail=True*, *username=None*, *password=None*, *always_collect_logs=False*)
Test the topology.

Submits the topology for testing and verifies the test conditions are met and the job remained healthy through its execution.

The submitted application (job) is monitored for the test conditions and will be canceled when all the conditions are valid or at least one failed. In addition if a local check was specified using *local_check()* then that callable must complete before the job is cancelled.

The test passes if all conditions became valid and the local check callable (if present) completed without raising an error.

The test fails if the job is unhealthy, any condition fails or the local check callable (if present) raised an exception. In the event that the test fails when submitting to the *STREAMING_ANALYTICS_SERVICE* context, the application logs are retrieved as a tar file and are saved to the current working directory. The filesystem path to the application logs is saved in the tester's result object under the *application_logs* key, i.e. *tester.result['application_logs']*

Parameters

- **ctxtype** (*str*) – Context type for submission.
- **config** – Configuration for submission.
- **assert_on_fail** (*bool*) – True to raise an assertion if the test fails, False to return the passed status.
- **username** (*str*) – **Deprecated**
- **password** (*str*) – **Deprecated**
- **always_collect_logs** (*bool*) – True to always collect the console log and PE trace files of the test.

result
The result of the test. This can contain exit codes, application log paths, or other relevant test information.

submission_result
Result of the application submission from *submit()*.

streams_connection
Connection object that can be used to interact with the REST API of the Streaming Analytics service or instance.
Type *StreamsConnection*

Returns *True* if test passed, *False* if test failed if *assert_on_fail* is *False*.

Return type bool

Deprecated since version 1.8.3: `username` and `password` parameters. When required for a distributed test use the environment variables `STREAMS_USERNAME` and `STREAMS_PASSWORD` to define the Streams user.

tuple_check(*stream*, *checker*)
Check each tuple on a stream.

For each tuple `t` on *stream* `checker(t)` is called.

If the return evaluates to *False* then the condition fails. Once the condition fails it can never become valid. Otherwise the condition becomes or remains valid. The first tuple on the stream makes the condition valid if the checker callable evaluates to *True*.

The condition can be combined with *tuple_count()* with `exact=False` to test a stream map or filter with random input data.

An example of combining *tuple_count* and *tuple_check* to test a filter followed by a map is working correctly across a random set of values:

```
def rands():
    r = random.Random()
    while True:
        yield r.random()

class TestFilterMap(unittest.testCase):
# Set up omitted

    def test_filter(self):
        # Declare the application to be tested
        topology = Topology()
        r = topology.source(rands())
        r = r.filter(lambda x : x > 0.7)
        r = r.map(lambda x : x + 0.2)

        # Create tester and assign conditions
        tester = Tester(topology)
        # Ensure at least 1000 tuples pass through the filter.
        tester.tuple_count(r, 1000, exact=False)
        tester.tuple_check(r, lambda x : x > 0.9)


        # Submit the application for test
        # If it fails an AssertionError will be raised.
        tester.test(self.test_ctxtype, self.test_config)
```

Parameters

- **stream** (Stream) – Stream to be tested.
- **checker** (*callable*) – Callable that must evaluate to True for each tuple.

tuple_count(*stream*, *count*, *exact=True*)

Test that a stream contains a number of tuples.

If *exact* is *True*, then condition becomes valid when *count* tuples are seen on *stream* during the test. Subsequently if additional tuples are seen on *stream* then the condition fails and can never become valid.

If *exact* is *False*, then the condition becomes valid once *count* tuples are seen on *stream* and remains valid regardless of any additional tuples.

Parameters

- **stream** (Stream) – Stream to be tested.
- **count** (*int*) – Number of tuples expected.
- **exact** (*bool*) – *True* if the stream must contain exactly *count* tuples, *False* if the stream must contain at least *count* tuples.

Returns stream

Return type *Stream*

1.8 streamsx.topology.tester_runtime

Runtime tester functionality.

1.8.1 Overview

Module containing runtime functionality for *streamsx.topology.tester*.

When test is executed any specified *Condition* instances are executed in the context of the application under test (and not the `unittest` class instance). This module separates out the runtime execution code from the test definition module *tester*.

1.8.2 Module contents

Classes

Condition	A condition for testing.

class `streamsx.topology.tester_runtime.`**Condition**(*name=None*)

Bases: `object`

A condition for testing.

Parameters **name** (*str*) – Condition name, must be unique within the tester.

1.9 streamsx.ec

Access to the IBM Streams execution context.

1.9.1 Overview

This module (*streamsx.ec*) provides access to the execution context when Python code is running in a Streams application.

A Streams application runs distributed or standalone.

Distributed

Distributed is used when an application is submitted to the Streaming Analytics service on IBM Cloud or a IBM Streams distributed instance.

With distributed a running application is a *job* that contains one or more processing elements (PEs). A PE corresponds to a Linux operating system process. The PEs in a job may be distributed across the resources (hosts) in the Streams instance.

Standalone

Standalone is a mode where the complete application is run as a single PE (process) outside of a Streams instance.

Standalone is typically used for ad-hoc testing of an application.

1.9.2 Application log and trace

IBM Streams provides application trace and log services.

Application log

The *Streams application log* service is for application logging, where logging is defined as the recording of serviceability information pertaining to application or operator events. The purpose of logging is to provide an administrator with enough information to do problem determination for items they can potentially control. In general, very few events are logged in the normal running scenario of an application or operator. Events pertinent to the failure or partial failure of application runtime scenarios should be logged.

When running in distributed or standalone the *com.ibm.streams.log* logger has a handler that records messages to the *Streams application log* service. The level of the logger and its handler are set to the configured application log level at PE start up.

This logger and handler discard any message with level below *INFO* (20).

Python application code can log a message suitable for an administrator by using the *com.ibm.streams.log* logger or a child logger that has `logger.propagate` evaulating to `True`. Example of logging a file exception:

```
try:
    import numpy
except ImportError as e:
    logging.getLogger('com.ibm.streams.log').error(e)
    raise
```

Application code must not modify the *com.ibm.streams.log* logger, if additional handlers or different levels are required a child logger should be used.

Application trace

The *Streams application trace* service is for application tracing, where tracing is defined as the recording of application or operator internal events and data. The purpose of tracing is to allow application or operator developers to debug their applications or operators.

When running in distributed or standalone the root logger has a handler that records messages to the *Streams application trace* service. The level of the logger and its handler are set to the configured application trace level at PE start up.

Python application code can trace a message using the root logger or a child logger that has `logger.propagate` evaulating to `True`. Example of logging a trace message:

```
trace = logging.getLogger(__name__)

...

    trace.info("Threshold set to %f", val)
```

Any existing logging performed by modules will automatically become Streams trace messages if the application is using the *logging* package.

Application code must not modify the root logger, if additional handlers or different levels are required a child logger should be used.

1.9.3 Execution Context

This module (*streamsx.ec*) provides access to the execution context when Python code is running in a Streams application.

Access is only supported when running:

- Streams 4.2 or later

This module may be used by Python functions or classes used in a *Topology* or decorated SPL operators.

Most functionality is only available when a Python class is being invoked in a Streams application.

Changed in version 1.9: Support for Python 2.7

1.9.4 Module contents

Functions

channel	Return the parallel region global channel number *obj* is executing in.
domain_id	Return the instance identifier.
get_application_configuration	Get a named application configuration.
get_application_directory	Get the application directory.
instance_id	Return the instance identifier.
is_active	Tests is code is active within a IBM Streams exection context.
is_standalone	Is the execution context standalone.
job_id	Return the job identifier.
local_channel	Return the parallel region local channel number *obj* is executing in.
local_max_channels	Return the local maximum number of channels for the parallel region *obj* is executing in.
max_channels	Return the global maximum number of channels for the parallel region *obj* is executing in.
pe_id	Return the PE identifier.
shutdown	Return the processing element (PE) shutdown event.

Classes

CustomMetric	Create a custom metric.
MetricKind	Enumeration for the kind of a metric.

streamsx.ec.**is_active**()
Tests is code is active within a IBM Streams exection context.

Returns a true value when called from within a IBM Streams distributed job or standalone execution.

Can be used to only run code required at runtime, such as importing a module that is only needed at runtime and not topology declaration time.

Returns True if running in a IBM Streams context false otherwise.

Return type bool

New in version 1.11.

streamsx.ec.**shutdown**()
Return the processing element (PE) shutdown event.

The event is set when the PE is being shutdown. Can be used in source iterators that need to block by sleeping:

```
# Sleep for 60 seconds unless the PE is being shutdown
if streamsx.ec.shutdown.wait(60.0):
    return None
```

Code must not call `set()` on the returned event.

Returns Event object representing PE shutdown.

Return type threading.Event

New in version 1.11.

streamsx.ec.**domain_id**()
Return the instance identifier.

streamsx.ec.**instance_id**()
Return the instance identifier.

streamsx.ec.**job_id**()
Return the job identifier.

streamsx.ec.**pe_id**()
Return the PE identifier.

streamsx.ec.**is_standalone**()
Is the execution context standalone.

Returns True if the execution context is standalone, False if it is distributed.

Return type boolean

streamsx.ec.**get_application_directory**()
Get the application directory.

Returns The application directory.

Return type str

New in version 1.7.

streamsx.ec.**get_application_configuration**(*name*)

Get a named application configuration.

An application configuration is a named set of securely stored properties where each key and its value in the property set is a string.

An application configuration object is used to store information that IBM Streams applications require, such as:

- Database connection data
- Credentials that your applications need to use to access external systems
- Other data, such as the port numbers or URLs of external systems

Parameters **name** (*str*) – Name of the application configuration.

Returns Dictionary containing the property names and values for the application configuration.

Return type dict

Raises **ValueError** – Application configuration does not exist.

streamsx.ec.**channel**(*obj*)

Return the parallel region global channel number *obj* is executing in.

The channel number is in the range of 0 to `max_channel(obj)`.

When the parallel region is not nested this is the same value as `local_channel(obj)`.

If the parallel region is nested the value will be between zero and `(width*N - 1)` where N is the number of times the parallel region has been replicated due to nesting.

Parameters **obj** – Instance of a class executing within Streams.

Returns Parallel region global channel number or -1 if not located in a parallel region.

Return type int

streamsx.ec.**local_channel**(*obj*)

Return the parallel region local channel number *obj* is executing in.

The channel number is in the range of zero to `local_max_channel(obj)`.

Parameters **obj** – Instance of a class executing within Streams.

Returns Parallel region local channel number or -1 if not located in a parallel region.

Return type int

streamsx.ec.**max_channels**(*obj*)

Return the global maximum number of channels for the parallel region *obj* is executing in.

When the parallel region is not nested this is the same value as `local_max_channels(obj)`.

If the parallel region is nested the value will be `(width*N)` where N is the number of times the parallel region has been replicated due to nesting.

Parameters **obj** – Instance of a class executing within Streams.

Returns Parallel region global maximum number of channels or 0 if not located in a parallel region.

Return type int

streamsx.ec.**local_max_channels**(*obj*)

Return the local maximum number of channels for the parallel region *obj* is executing in.

The maximum number of channels corresponds to the width of the region.

Parameters **obj** – Instance of a class executing within Streams.

Returns Parallel region local maximum number of channels or 0 if not located in a parallel region.

Return type int

class streamsx.ec.**MetricKind**

Bases: enum.Enum

Enumeration for the kind of a metric.

The kind of the metric only indicates the behavior of value, it does not impose any semantics on the value. The kind is typically used by tooling applications.

Counter = 1

A counter metric observes a value that represents a count of an occurrence.

Gauge = 0

A gauge metric observes a value that is continuously variable with time.

Time = 2

A time metric represents a point in time or duration. The recommended unit of time is milliseconds, using the standard epoch of 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970 to represent a point in time.

class streamsx.ec.**CustomMetric**(*obj*, *name*, *description=None*, *kind=<MetricKind.Counter: 1>*, *initialValue=0*)

Bases: object

Create a custom metric.

A custom metric holds a 64 bit signed integer value that represents a *Counter*, *Gauge* or *Time* metric.

Custom metrics are exposed through the IBM Streams monitoring APIs.

Metric `name` is unique within the execution context of the callable `obj`. Attempts to create multiple metrics with the same name but different kinds will raise an exception. Multiple creations of a metric of the same name and kind all refer to the same metric, the first creation is the only one that will set the initial value.

The metric's value is assigned through the `value` property and can be modified through `+=` and `-=`. `CustomMetric` can also be converted to an `int`.

Parameters

- **obj** – Instance of a class executing within Streams.
- **name** (*str*) – Name of the custom metric.
- **kind** (`MetricKind`) – Kind of the metric.
- **description** (*str*) – Description of the metric.
- **initialValue** – Initial value of the metric.

Examples:

Simple example used as an instance to `Stream.filter`:

```
class MyF:
    def __init__(self, substring):
        self.substring = substring
        pass

    def __call__(self, tuple):
        if self.substring in str(tuple)
```

(continues on next page)

(continued from previous page)

```
            self.my_metric += 1
        return True

    # Create the metric when the it is running
    # in the Streams execution context
    def __enter__(self):
        self.my_metric = ec.CustomMetric(self, "count_" + self.substring)

    # must supply __exit__ if using __enter__
    def __exit__(self, exc_type, exc_val, exc_tb):
        pass

    def __getstate__(self):
        # Remove metric from saved state.
        state = self.__dict__.copy()
        if 'my_metric' in state:
            del state['my_metric']
        return state

    def __setstate__(self, state):
        self.__dict__.update(state)
```

property value
Current value of the metric.

1.10 streamsx.spl.op

Integration of SPL operators.

1.10.1 Invoking SPL Operators

IBM Streams supports *Stream Processing Language* (SPL), a domain specific language for streaming analytics. SPL creates an application by building a graph of operator invocations. These operators are declared in an SPL toolkit.

SPL streams have a structured schema, such as `tuple<rstring id, timestamp ts, float64 value>` for a sensor reading with a sensor identifier, timestamp and value. A schema is defined using *StreamSchema*.

A Python topology application can take advantage of SPL operators by using streams with structured schemas. A stream of Python objects can be converted to a structured stream using *map()* with the *schema* parameter set:

```
# s is stream of Python objects representing a sensor
s = ...

# map s to a structured stream using a lambda function
# for each sensor reading r a Python tuple is created
# with the required values matching the order of the
# structured schema.
s2 = s.map(lambda r : (r.sensor_id, r.reading_time, r.reading),
     schema='tuple<rstring id, timestamp ts, float64 value>'
```

An SPL operator is invoked in an application by creating an instance of:

- *Invoke* - Invocation of an arbitrary SPL operator.
- *Source* - Invocation of an SPL source operator with one input port.

- *Map* - Invocation of an SPL map operator with one input port and one output port.
- *Sink* - Invocation of an SPL sink operator with one output port.

In SPL, operator invocation supports a number of clauses that are supported in Python.

Values for operator clauses

When an operator clause requires a value, the value may be passed as a constant, an input attribute (passed using the *attribute* method of the invocation), or an arbitrary SPL expression (passed as a string or an *Expression*). Because a string is interpreted as an SPL expression, a string constant should be passed by enclosing the quoted string in outer quotes (for example, '"a string constant"').

SPL is strictly typed so when passing a constant as a value the value may need to be strongly typed.

- `bool`, `int`, `float` and `str` values map automatically to SPL *boolean*, *int32*, *float64* and *rstring* respectively.
- `Enum` values map to an operator custom literal using the symbolic name of the value. For custom literals only the symbolic name needs to match a value expected by the operator, the class name and other values are arbitrary.
- The module *streamsx.spl.types* provides functions to create typed SPL expressions from values.

An optional type may be set to SPL *null* by passing either Python *None* or the value returned from *null()*.

Param clause

Operator parameterization is through operator parameters that configure and modify the operator for the specific application.

Parameters are passed as a *dict* containing the parameter names and their values (see *Values for operator clauses*).

Examples

To invoke a *Beacon* operator from the SPL standard toolkit producing 100 tuples at the rate of two per second:

```
schema = StreamSchema('tuple<uint64 seq>')
beacon = op.Source(topology, 'spl.utility::Beacon', schema,
    params = {'iterations':100, 'period':0.5})
```

To use an `IntEnum` to pass a custom literal to the `Parse` operator:

```
from enum import IntEnum

class DataFormats(IntEnum):
    csv = 0
    txt = 1

...

params['format'] = DataFormats.csv
```

To create a *count* parameter of type *uint64* for the SPL *DeDuplicate* operator:

```
params['count'] = streamsx.spl.types.uint64(20)
```

After the instance representing the operator invocation has been created, additional parameters may be added through the *params* attribute. If the value is an expression that is only valid in the context of the operator invocation then the parameter must be added after the operator invocation has been created.

For example, the *Filter* operator uses an expression that is usually dependent on the context, filtering tuples based upon their attribute values:

```
fs = op.Map('spl.relational::Filter', beacon)
fs.params['filter'] = fs.expression('seq % 2ul == 0ul')
```

Output clause

The operator output clause defines the values of attributes on outgoing tuples on the operator invocation's output ports.

When a tuple is submitted by an operator invocation each of its attributes is set in one of three ways:

- By the operator based upon its state and input tuples. For example, a US ZIP code operator might set the *zipcode* attribute based upon its lookup of the ZIP code from the address details in the input tuple.
- By the operator implicitly setting output attributes from matching input attributes when those attributes have not been explicitly set elsewhere. Many streaming operators implicitly set output attributes to allow attributes to flow through the operator without any explicit coding. This only occurs when an output attribute is not explicitly set by the operator, or the output clause, and the input tuple has an attribute that matches the output attribute (same name and type, or same name and same type as the underlying type of an output attribute with an optional type). For example, in the US ZIP code operator, if the output tuple included attributes of `rstring city, rstring state` that matched input attributes, then they would be implicitly copied from the input tuple to the output tuple.
- By an output clause in the operator invocation. In this case the application invoking the operator is explicitly setting attributes using SPL expressions. An operator may provide output functions that return values based upon the operator's state and input tuples. For example, the US ZIP code operator might provide a `ZIPCode()` output function rather than explicitly setting an output attribute. Then the application is free to use any attribute name to represent the ZIP code in its output tuple.

In Python an output tuple attribute is set by creating an attribute in the operator invocation instance that is set to a return from the *output* method. The attribute value passed to the *output* method is passed as described in *Values for operator clauses*.

For example, invoking an SPL *Beacon* operator using an output function to set the sequence number of a tuple and an SPL expression to set the timestamp:

```
schema = StreamSchema('tuple<uint64 seq, timestamp ts>')
beacon = op.Source(topology, 'spl.utility::Beacon', schema, params = {'period':0.1})

# Set the seq attribute using an output function provided by Beacon
beacon.seq = beacon.output('IterationCount()')

# Set the ts attribute using an SPL function that returns the current time
beacon.ts = beacon.output('getTimestamp()')
```

See also:

Streams Processing Language (SPL) Reference Reference documentation.

Developing Streams applications Developing Streams applications.

Operator invocations Operator invocations from the SPL reference documentation.

1.10.2 Module contents

Functions

main_composite	Wrap a main composite invocation as a *Topology*.

Classes

Expression	An SPL expression.
Invoke	Declaration of an invocation of an SPL operator in a Topology.
Map	Declaration of an invocation of an SPL *map* operator.
Sink	Declaration of an invocation of an SPL sink operator.
Source	Declaration of an invocation of an SPL *source* operator.

class streamsx.spl.op.**Invoke**(*topology*, *kind*, *inputs=None*, *schemas=None*, *params=None*, *name=None*)

Bases: streamsx._streams._placement._Placement, streamsx.topology.exop.ExtensionOperator

Declaration of an invocation of an SPL operator in a Topology.

An SPL operator has an arbitrary of input ports and an arbitrary number of output ports. The kind of the operator places constraints on how many input and output ports it supports, and potentially the schemas for those ports. For example, `spl.relational::Filter` has a single input port and one or two output ports, in addition the schemas of the ports must be identical.

When the operator has output ports an instance of `SPLOperator` has an `outputs` attributes which is a list of `Stream` instances.

Parameters

- **topology** (Topology) – Topology that will invoke the operator.
- **kind** (*str*) – SPL operator kind, e.g. `spl.utility::Beacon`.
- **inputs** – Streams to connect to the operator. If not set or set to *None* or an empty collection then the operator has no input ports. Otherwise a list or tuple of `Stream` instances where the number of items is the number of input ports.
- **schemas** – Schemas of the output ports. If not set or set to *None* or an empty collection then the operator has no outut ports. Otherwise a list or tuple of schemas where the number of items is the number of output ports.
- **params** – Operator parameters.
- **name** – Name of the operator. When *None* defaults to a name derived from the operator kind.

attribute(*stream*, *name*)

Expression for an input attribute.

An input attribute is an attribute on one of the input ports of the operator invocation. *stream* must have been used to declare this invocation.

Parameters

- **stream** (Stream) – Stream the attribute is from.

- **name** (*str*) – Name of the attribute.

Returns Expression representing the input attribute.

Return type *Expression*

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

expression(*value*)

SPL expression.

An arbitrary expression that is valid in the context of this operator.

Parameters **value** (*str*) – Arbitrary SPL expression.

Returns Expression that is valid in the context of this operator.

Return type *Expression*

output(*stream*, *value*)

SPL output port assignment expression.

Parameters

- **stream** (*Stream*) – Output stream the assignment is for.
- **value** (*str*) – SPL expression used for an output assignment. This can be a string, a constant, or an *Expression*.

Returns Output assignment expression that is valid as a the context of this operator.

Return type *Expression*

property params
Parameters for the operator invocation.

property resource_tags
Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A *Stream* that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a *union()* of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

Warning: If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for *Sink* and *Invoke*.

class streamsx.spl.op.**Source**(*topology*, *kind*, *schema*, *params=None*, *name=None*)
Bases: *streamsx.spl.op.Invoke*

Declaration of an invocation of an SPL *source* operator.

Source operators typically bring external data into a Streams application as a stream. A source operator has no input ports and a single output port.

An instance of Source has an attribute `stream` that is `Stream` produced by the operator.

This is a utility class that allows simple invocation of the common case of a operator with a single output port.

Parameters

- **topology** (Topology) – Topology that will invoke the operator.
- **kind** (*str*) – SPL operator kind, e.g. `spl.utility::Beacon`.
- **schema** – Schema of the output port.
- **params** – Operator parameters.
- **name** – Name of the operator. When *None* defaults to a generated name.

attribute(*stream*, *name*)
Expression for an input attribute.

An input attribute is an attribute on one of the input ports of the operator invocation. *stream* must have been used to declare this invocation.

Parameters

- **stream** (Stream) – Stream the attribute is from.

- **name** (*str*) – Name of the attribute.

Returns Expression representing the input attribute.

Return type *Expression*

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

expression(*value*)

SPL expression.

An arbitrary expression that is valid in the context of this operator.

Parameters **value** (*str*) – Arbitrary SPL expression.

Returns Expression that is valid in the context of this operator.

Return type *Expression*

output(*value*)

SPL output port assignment expression.

Parameters **value** (*str*) – SPL expression used for an output assignment. This can be a string, a constant, or an *Expression*.

Returns Output assignment expression that is valid as a the context of this operator.

Return type *Expression*

property params

Parameters for the operator invocation.

property resource_tags
Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A *Stream* that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a *union()* of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

Warning: If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for *Sink* and *Invoke*.

property stream
Stream produced by the operator invocation.

Returns Stream produced by the operator invocation.

Return type *Stream*

class streamsx.spl.op.**Map**(*kind*, *stream*, *schema=None*, *params=None*, *name=None*)
Bases: *streamsx.spl.op.Invoke*

Declaration of an invocation of an SPL *map* operator.

Map operators have a single input port and single output port.

An instance of Map has an attribute `stream` that is `Stream` produced by the operator.

This is a utility class that allows simple invocation of the common case of a operator with a single input stream and single output stream.

Parameters

- **kind** (*str*) – SPL operator kind, e.g. `spl.relational::Filter`.
- **stream** – Stream to connect to the operator.
- **schema** – Schema of the output stream. If set to *None* then the output schema is the same as the schema of *stream*.
- **params** – Operator parameters.
- **name** – Name of the operator. When *None* defaults to a generated name.

attribute(*name*)
Expression for an input attribute.

An input attribute is an attribute on the input port of the operator invocation.

Parameters **name** (*str*) – Name of the attribute.

Returns Expression representing the input attribute.

Return type *Expression*

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

expression(*value*)

SPL expression.

An arbitrary expression that is valid in the context of this operator.

Parameters **value** (*str*) – Arbitrary SPL expression.

Returns Expression that is valid in the context of this operator.

Return type *Expression*

output(*value*)

SPL output port assignment expression.

Parameters **value** (*str*) – SPL expression used for an output assignment. This can be a string, a constant, or an *Expression*.

Returns Output assignment expression that is valid as a the context of this operator.

Return type *Expression*

property params

Parameters for the operator invocation.

property resource_tags
Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A *Stream* that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a *union()* of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

Warning: If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for *Sink* and *Invoke*.

property stream
Stream produced by the operator invocation.

Returns Stream produced by the operator invocation.

Return type *Stream*

class streamsx.spl.op.**Sink**(*kind*, *stream*, *params=None*, *name=None*)
Bases: *streamsx.spl.op.Invoke*

Declaration of an invocation of an SPL sink operator.

Source operators typically send data on a stream to an external system. A sink operator has a single input port and no output ports.

This is a utility class that allows simple invocation of the common case of a operator with a single input port.

Parameters

- **kind** (*str*) – SPL operator kind, e.g. `spl.adapter::FileSink`.
- **input** – Stream to connect to the operator.
- **params** – Operator parameters.
- **name** – Name of the operator. When *None* defaults to a generated name.

attribute(*stream*, *name*)
Expression for an input attribute.

An input attribute is an attribute on one of the input ports of the operator invocation. *stream* must have been used to declare this invocation.

Parameters

- **stream** (Stream) – Stream the attribute is from.

- **name** (*str*) – Name of the attribute.

Returns Expression representing the input attribute.

Return type *Expression*

property category

Category for this processing logic.

An arbitrary application label allowing grouping of application elements by category.

Assign categories based on common function. For example, *database* is a common category that you can use to group all database sinks in an application.

A category is not required and defaults to `None` meaning no assigned category.

Streams console supports visualization based upon categories.

Raises **TypeError** – No directly associated processing logic.

Note: A category has no affect on the execution of the application.

New in version 1.9.

colocate(*others*)

Colocate this processing logic with others.

Colocating processing logic requires execution in the same Streams processing element (operating system process).

When a job is submitted Streams may colocate (fuse) processing logic into the same processing element based upon flow analysis and current resource usage. This call instructs that this logic and *others* must be executed in the same processing element.

Parameters **others** – Processing logic such as a *Stream* or *Sink*. A single value can be passed or an iterable, such as a list of streams.

Returns This logic.

Return type self

expression(*value*)

SPL expression.

An arbitrary expression that is valid in the context of this operator.

Parameters **value** (*str*) – Arbitrary SPL expression.

Returns Expression that is valid in the context of this operator.

Return type *Expression*

output(*stream*, *value*)

SPL output port assignment expression.

Parameters

- **stream** (*Stream*) – Output stream the assignment is for.
- **value** (*str*) – SPL expression used for an output assignment. This can be a string, a constant, or an *Expression*.

Returns Output assignment expression that is valid as a the context of this operator.

Return type *Expression*

property params
Parameters for the operator invocation.

property resource_tags
Resource tags for this processing logic.

Tags are a mechanism for differentiating and identifying resources that have different physical characteristics or logical uses. For example a resource (host) that has external connectivity for public data sources may be tagged *ingest*.

Processing logic can be associated with one or more tags to require running on suitably tagged resources. For example adding tags *ingest* and *db* requires that the processing element containing the callable that created the stream runs on a host tagged with both *ingest* and *db*.

A *Stream* that was not created directly with a Python callable cannot have tags associated with it. For example a stream that is a *union()* of multiple streams cannot be tagged. In this case this method returns an empty *frozenset* which cannot be modified.

See https://www.ibm.com/support/knowledgecenter/en/SSCRJU_4.2.1/com.ibm.streams.admin.doc/doc/tags.html for more details of tags within IBM Streams.

Returns Set of resource tags, initially empty.

Return type set

Warning: If no resources exist with the required tags then job submission will fail.

New in version 1.7.

New in version 1.9: Support for *Sink* and *Invoke*.

class streamsx.spl.op.**Expression**(*_type*, *_value*)
Bases: object

An SPL expression.

static expression(*value*)
Create an SPL expression.

Parameters **value** – Expression as a string or another *Expression*. If value is an instance of *Expression* then a new instance is returned containing the same type and value.

Returns SPL expression from *value*.

Return type *Expression*

streamsx.spl.op.**main_composite**(*kind*, *toolkits=None*, *name=None*)
Wrap a main composite invocation as a *Topology*.

Provides a bridge between an SPL application (main composite) and a *Topology*. Create a *Topology* that contains just the invocation of the main composite defined by *kind*.

The returned *Topology* may be used like any other topology instance including job configuration, tester or even addition of SPL operator invocations or functional transformations.

Note: Since a main composite by definition has no input or output ports any functionality added to the topology cannot interact directly with its invocation.

When *name* is None and no additions or tests are made to the topology then SPL compilation uses *kind* directly. Otherwise the main composite invocation is invoked within a generated main composite.

Parameters

- **kind** (*str*) – Kind of the main composite operator invocation.
- **toolkits** (*list[str]*) – Optional list of toolkits the main composite depends on.
- **name** (*str*) – Invocation name for the main composite.

Returns

tuple containing:

- **Topology**: Topology with main composite invocation.
- **Invoke**: Invocation of the main composite

Return type tuple

1.11 streamsx.spl.types

SPL type definitions.

1.11.1 Overview

SPL is strictly typed, thus when invoking SPL operators using classes from `streamsx.spl.op` then any parameters must use the SPL type required by the operator.

1.11.2 Module contents

Functions

float32	Create an SPL `float32` value.
float64	Create an SPL `float64` value.
int16	Create an SPL `int16` value.
int32	Create an SPL `int32` value.
int64	Create an SPL `int64` value.
int8	Create an SPL `int8` value.
null	Return an SPL `null`.
rstring	Create an SPL `rstring` value.
uint16	Create an SPL `uint16` value.
uint32	Create an SPL `uint32` value.
uint64	Create an SPL `uint64` value.
uint8	Create an SPL `uint8` value.

Classes

Timestamp	SPL native timestamp type with nanosecond resolution.

class streamsx.spl.types.**Timestamp**

Bases: streamsx.spl.runtime.Timestamp

SPL native timestamp type with nanosecond resolution.

Common usage is to store the seconds and nanoseconds since the Unix Epoch (Jan 1, 1970), but this is not enforced by the *Timestamp* class.

Machine identifier is an optional application defined identifier for the machine the timestamp was created on. It is the responsibility of the application to set the machine identifier if required. The machine identifier may be used to detect if two timestamps were created on the same machine, as there may be variations in the clocks on different machines.

A instance can be created by passing seconds, nanoseconds and optionally machine identifier:

```
# Timestamp with the current time in seconds
# discarding any fractional seconds.
ts = Timestamp(time.time(), 0)

# Timestamp set to a specific time with a machine identifier
ts = Timestamp(1516500542, 9511447, 4)
```

A *Timestamp* is a namedtuple with three fields *seconds*, *nanoseconds* and *machine_id*.

A *Timestamp* acts as a `datetime.datetime` instance (duck typing) with the exception of:

- *time()* - returns an `int` instead of `datetime.time`
- `datetime.datetime` operations (`+`, `-`, `<`) are not supported
- string representation (uses `Timestamp` representation)
- is not an instance of `datetime.datetime`

The value of the equivalent `datetime.datetime` is identical to the instance returned by *datetime()*.

seconds
Seconds since epoch.

Type int

nanoseconds
Nanosecond component.

Type int

machine_id
Optional machine identifier, defaults to zero.

Type int

Warning: Implementation of *Timestamp* changed with 1.8.3 to be a *namedtuple* maintaining the existing class API.

Changed in version 1.14: `Timestamp` acts as a `datetime.datetime`.

count()
Return number of occurrences of value.

datetime()
Return the UTC datetime corresponding to the POSIX timestamp.

This is identical to `datetime.datetime.utcfromtimestamp(self.time())`. Nanosecond resolution may be lost.

Returns Timestamp converted to a *datetime.datetime*.

Return type datetime.datetime

static from_datetime(*dt*, *machine_id=0*)
Convert a datetime to an SPL *Timestamp*.

Parameters

- **dt** (*datetime.datetime*) – Datetime to be converted.
- **machine_id** (*int*) – Machine identifier.

Returns Datetime converted to Timestamp.

Return type *Timestamp*

static from_time(*t*, *machine_id=0*)
Convert seconds since epoch to a Timestamp.

The time argument matches the return from `time.time()`.

Parameters

- **t** (*float*) – Time to be converted.
- **machine_id** (*int*) – Machine identifier.

Returns Time converted to Timestamp.

Return type *Timestamp*

New in version 1.8.3.

index()
Return first index of value.

Raises ValueError if the value is not present.

property machine_id
Alias for field number 2

property nanoseconds
Alias for field number 1

static now(*machine_id=0*)
Timestamp representing the current time.

Parameters **machine_id** (*int*) – Machine identifier.

Returns Current time.

Return type *Timestamp*

New in version 1.8.3.

property seconds
Alias for field number 0

time()
Get the time in seconds since the epoch.

Returns time in seconds since the epoch.

Return type float

streamsx.spl.types.**int8**(*value*)
Create an SPL `int8` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**int16**(*value*)
Create an SPL `int16` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**int32**(*value*)
Create an SPL `int32` value.

Returns Expression representing the value.

Return type *Expression*

Parameters **value** (*int*) – Value to be types as `int32`.

streamsx.spl.types.**int64**(*value*)
Create an SPL `int64` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**uint8**(*value*)
Create an SPL `uint8` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**uint16**(*value*)
Create an SPL `uint16` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**uint32**(*value*)
Create an SPL `uint32` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**uint64**(*value*)
Create an SPL `uint64` value.

Returns Expression representing the value.

Return type *Expression*

streamsx.spl.types.**float32**(*value*)
Create an SPL `float32` value.

Returns Expression representing the value.

Return type *Expression*

`streamsx.spl.types.`**`float64`**(*value*)
Create an SPL `float64` value.

Returns Expression representing the value.

Return type *Expression*

`streamsx.spl.types.`**`rstring`**(*value*)
Create an SPL `rstring` value.

Returns Expression representing the value.

Return type *Expression*

`streamsx.spl.types.`**`null`**()
Return an SPL `null`.

Returns Expression representing an SPL null value.

Return type *Expression*

New in version 1.10.

1.12 streamsx.spl.toolkit

SPL toolkit integration.

1.12.1 Overview

SPL operators are defined by an SPL toolkit. When a `Topology` contains invocations of SPL operators, their defining toolkit must be made known using *`add_toolkit()`*.

Toolkits shipped with the IBM Streams product under `$STREAMS_INSTALL/toolkits` are implictly known and must not be added through `add_toolkit`.

1.12.2 Module contents

Functions

`add_toolkit`	Add an SPL toolkit to a topology.
`add_toolkit_dependency`	Add a version dependency on an SPL toolkit to a topology.

`streamsx.spl.toolkit.`**`add_toolkit`**(*topology*, *location*)
Add an SPL toolkit to a topology.

Parameters

- **`topology`** (`Topology`) – Topology to include toolkit in.
- **`location`** (*`str`*) – Location of the toolkit directory.

`streamsx.spl.toolkit.`**`add_toolkit_dependency`**(*topology*, *name*, *version*)

Add a version dependency on an SPL toolkit to a topology.

To specify a range of versions for the dependent toolkits, use brackets (`[]`) or parentheses. Use brackets to represent an inclusive range and parentheses to represent an exclusive range. The following examples describe how to specify a dependency on a range of toolkit versions:

- `[1.0.0, 2.0.0]` represents a dependency on toolkit versions 1.0.0 - 2.0.0, both inclusive.
- `[1.0.0, 2.0.0)` represents a dependency on toolkit versions 1.0.0 or later, but not including 2.0.0.
- `(1.0.0, 2.0.0]` represents a dependency on toolkits versions later than 1.0.0 and less than or equal to 2.0.0.
- `(1.0.0, 2.0.0)` represents a dependency on toolkit versions 1.0.0 - 2.0.0, both exclusive.

Parameters

- **topology** (Topology) – Topology to include toolkit in.
- **name** (*str*) – Toolkit name.
- **version** (*str*) – Toolkit version dependency.

See also:

Toolkit information model file

New in version 1.12.

CHAPTER

TWO

SPL PRIMITIVE PYTHON OPERATORS

SPL primitive Python operators provide the ability to perform tuple processing using Python in an SPL application.

A Python function or class is simply turned into an SPL primitive operator through provided decorators.

SPL (Streams Processing Language) is a domain specific language for streaming analytics supported by Streams.

streamsx.spl.spl	SPL Python primitive operators.

2.1 streamsx.spl.spl

SPL Python primitive operators.

2.1.1 Overview

SPL primitive operators that call a Python function or class methods are created by decorators provided by this module.

The name of the function or callable class becomes the name of the operator.

A decorated function is a stateless operator while a decorated class is an optionally stateful operator.

These are the supported decorators that create an SPL operator:

- *@spl.source* - Creates a source operator that produces tuples.
- *@spl.filter* - Creates a operator that filters tuples.
- *@spl.map* - Creates a operator that maps input tuples to output tuples.
- *@spl.for_each* - Creates a operator that terminates a stream processing each tuple.
- *@spl.primitive_operator* - Creates an SPL primitive operator that has an arbitrary number of input and output ports.

Decorated functions and classes must be located in the directory `opt/python/streams` in the SPL toolkit. Each module in that directory will be inspected for operators during extraction. Each module defines the SPL namespace for its operators by the function `spl_namespace`, for example:

```
from streamsx.spl import spl

def spl_namespace():
    return 'com.example.ops'

@spl.map()
```

(continues on next page)

(continued from previous page)

```
def Pass(*tuple_):
    return tuple_
```

creates a pass-through operator `com.example.ops::Pass`.

SPL primitive operators are created by executing the extraction script *spl-python-extract* against the SPL toolkit. Once created the operators become part of the toolkit and may be used like any other SPL operator.

2.1.2 Python classes as SPL operators

Overview

Decorating a Python class creates a stateful SPL operator where the instance fields of the class are the operator's state. An instance of the class is created when the SPL operator invocation is initialized at SPL runtime. The instance of the Python class is private to the SPL operator and is maintained for the lifetime of the operator.

If the class has instance fields then they are the state of the operator and are private to each invocation of the operator.

If the *__init__* method has parameters beyond the first *self* parameter then they are mapped to operator parameters. Any parameter that has a default value becomes an optional parameter to the SPL operator. Parameters of the form **args* and ***kwargs* are not supported.

Warning: Parameter names must be valid SPL identifers, SPL identifiers start with an ASCII letter or underscore, followed by ASCII letters, digits, or underscores. The name also must not be an SPL keyword.

Parameter names `suppress` and `include` are reserved.

The value of the operator parameters at SPL operator invocation are passed to the *__init__* method. This is equivalent to creating an instance of the class passing the operator parameters into the constructor.

For example, with this decorated class producing an SPL source operator:

```
@spl.source()
class Range(object):
  def __init__(self, stop, start=0):
    self.start = start
    self.stop = stop

  def __iter__(self):
      return zip(range(self.start, self.stop))
```

The SPL operator *Range* has two parameters, *stop* is mandatory and *start* is optional, defaulting to zero. Thus the SPL operator may be invoked as:

```
// Produces the sequence of values from 0 to 99
//
// Creates an instance of the Python class
// Range using Range(100)
//
stream<int32 seq> R = Range() {
  param
    stop: 100;
}
```

or both operator parameters can be set:

```
// Produces the sequence of values from 50 to 74
//
// Creates an instance of the Python class
// Range using Range(75, 50)
//
stream<int32 seq> R = Range() {
  param
    start: 50;
    stop: 75;
}
```

Operator state

Use of a class allows the operator to be stateful by maintaining state in instance attributes across invocations (tuple processing).

When the operator is in a consistent region or checkpointing then it is serialized using *dill*. The default serialization may be modified by using the standard Python pickle mechanism of `__getstate__` and `__setstate__`. This is required if the state includes objects that cannot be serialized, for example file descriptors. For details see See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects .

If the class has `__enter__` and `__exit__` context manager methods then `__enter__` is called after the instance has been deserialized by *dill*. Thus `__enter__` is used to recreate runtime objects that cannot be serialized such as open files or sockets.

Operator initialization & shutdown

Execution of an instance for an operator effectively run in a context manager so that an instance's `__enter__` method is called when the processing element containing the operator is initialized and its `__exit__` method called when the processing element is stopped. To take advantage of this the class must define both `__enter__` and `__exit__` methods.

Note: Initialization such as opening files should be in `__enter__` in order to support stateful operator restart & checkpointing.

Example of using `__enter__` and `__exit__` to open and close a file:

```
import streamsx.ec as ec

@spl.map()
class Sentiment(object):
    def __init__(self, name):
        self.name = name
        self.file = None

    def __enter__(self):
        self.file = open(self.name, 'r')

    def __exit__(self, exc_type, exc_value, traceback):
        if self.file is not None:
            self.file.close()

    def __call__(self):
        pass
```

When an instance defines a valid `__exit__` method then it will be called with an exception when:

- the instance raises an exception during processing of a tuple
- a data conversion exception is raised converting a Python value to an SPL tuple or attribute

If `__exit__` returns a true value then the exception is suppressed and processing continues, otherwise the enclosing processing element will be terminated.

Application log and trace

IBM Streams provides application trace and log services which are accesible through standard Python loggers from the *logging* module.

See *Application log and trace*.

2.1.3 Python functions as SPL operators

Decorating a Python function creates a stateless SPL operator. In SPL terms this is similar to an SPL Custom operator, where the code in the Python function is the custom code. For operators with input ports the function is called for each input tuple, passing a Python representation of the SPL input tuple. For an SPL source operator the function is called to obtain an iterable whose contents will be submitted to the output stream as SPL tuples.

Operator parameters are not supported.

An example SPL sink operator that prints each input SPL tuple after its conversion to a Python tuple:

```
@spl.for_each()
def PrintTuple(*tuple_):
    "Print each tuple to standard out."
     print(tuple_, flush=True)
```

2.1.4 Processing SPL tuples in Python

SPL tuples are converted to Python objects and passed to a decorated callable.

Overview

For each SPL tuple arriving at an input port a Python function is called with the SPL tuple converted to Python values suitable for the function call. How the tuple is passed is defined by the tuple passing style.

Tuple Passing Styles

An input tuple can be passed to Python function using a number of different styles:

- *dictionary*
- *tuple*
- *attributes by name* **not yet implemented**
- *attributes by position*

Dictionary

Passing the SPL tuple as a Python dictionary is flexible and makes the operator independent of any schema. A disadvantage is the reduction in code readability for Python function by not having formal parameters, though getters such as `tuple['id']` mitigate that to some extent. If the function is general purpose and can derive meaning from the keys that are the attribute names then `**kwargs` can be useful.

When the only function parameter is `**kwargs` (e.g. `def myfunc(**tuple_):`) then the passing style is *dictionary*.

All of the attributes are passed in the dictionary using the SPL schema attribute name as the key.

Tuple

Passing the SPL tuple as a Python tuple is flexible and makes the operator independent of any schema but is brittle to changes in the SPL schema. Another disadvantage is the reduction in code readability for Python function by not having formal parameters. However if the function is general purpose and independent of the tuple contents `*args` can be useful.

When the only function parameter is `*args` (e.g. `def myfunc(*tuple_):`) then the passing style is *tuple*.

All of the attributes are passed as a Python tuple with the order of values matching the order of the SPL schema.

Attributes by name

(**not yet implemented**)

Passing attributes by name can be robust against changes in the SPL scheme, e.g. additional attributes being added in the middle of the schema, but does require that the SPL schema has matching attribute names.

When *attributes by name* is used then SPL tuple attributes are passed to the function by name for formal parameters. Order of the attributes and parameters need not match. This is supported for function parameters of kind `POSITIONAL_OR_KEYWORD` and `KEYWORD_ONLY`.

If the function signature also contains a parameter of the form `**kwargs` (`VAR_KEYWORD`) then any attributes not bound to formal parameters are passed in its dictionary using the SPL schema attribute name as the key.

If the function signature also contains an arbitrary argument list `*args` then any attributes not bound to formal parameters or to `**kwargs` are passed in order of the SPL schema.

If there are only formal parameters any non-bound attributes are not passed into the function.

Attributes by position

Passing attributes by position allows the SPL operator to be independent of the SPL schema but is brittle to changes in the SPL schema. For example a function expecting an identifier and a sensor reading as the first two attributes would break if an attribute representing region was added as the first SPL attribute.

When *attributes by position* is used then SPL tuple attributes are passed to the function by position for formal parameters. The first SPL attribute in the tuple is passed as the first parameter. This is supported for function parameters of kind *POSITIONAL_OR_KEYWORD*.

If the function signature also contains an arbitrary argument list **args* (*VAR_POSITIONAL*) then any attributes not bound to formal parameters are passed in order of the SPL schema.

The function signature must not contain a parameter of the form `**kwargs` (*VAR_KEYWORD*).

If there are only formal parameters any non-bound attributes are not passed into the function.

The SPL schema must have at least the number of positional arguments the function requires.

Selecting the style

For signatures only containing a parameter of the form `*args` or `**kwargs` the style is implicitly defined:

- `def f(**tuple_)` - *dictionary* - `tuple_` will contain a dictionary of all of the SPL tuple attribute's values with the keys being the attribute names.
- `def f(*tuple_)` - *tuple* - `tuple_` will contain all of the SPL tuple attribute's values in order of the SPL schema definition.

Otherwise the style is set by the `style` parameter to the decorator, defaulting to *attributes by name*. The style value can be set to:

- `'name'` - *attributes by name* (the default)
- `'position'` - *attributes by position*

Examples

These examples show how an SPL tuple with the schema and value:

```
tuple<rstring id, float64 temp, boolean increase>
{id='battery', temp=23.7, increase=true}
```

is passed into a variety of functions by showing the effective Python call and the resulting values of the function's parameters.

Dictionary consuming all attributes by `**kwargs`:

```
@spl.map()
def f(**tuple_)
    pass
# f({'id':'battery', 'temp':23.7, 'increase': True})
#     tuple_={'id':'battery', 'temp':23.7, 'increase':True}
```

Tuple consuming all attributes by `*args`:

```
@spl.map()
def f(*tuple_)
    pass
# f('battery', 23.7, True)
#     tuple_=('battery',23.7, True)
```

Attributes by name consuming all attributes:

```
@spl.map()
def f(id, temp, increase)
    pass
# f(id='battery', temp=23.7, increase=True)
#     id='battery'
#     temp=23.7
#     increase=True
```

Attributes by name consuming a subset of attributes:

```
@spl.map()
def f(id, temp)
    pass
# f(id='battery', temp=23.7)
#    id='battery'
#    temp=23.7
```

Attributes by name consuming a subset of attributes in a different order:

```
@spl.map()
def f(increase, temp)
    pass
# f(temp=23.7, increase=True)
#    increase=True
#    temp=23.7
```

Attributes by name consuming *id* by name and remaining attributes by `**kwargs`:

```
@spl.map()
def f(id, **tuple_)
    pass
# f(id='battery', {'temp':23.7, 'increase':True})
#    id='battery'
#    tuple_={'temp':23.7, 'increase':True}
```

Attributes by name consuming *id* by name and remaining attributes by `*args`:

```
@spl.map()
def f(id, *tuple_)
    pass
# f(id='battery', 23.7, True)
#    id='battery'
#    tuple_=(23.7, True)
```

Attributes by position consuming all attributes:

```
@spl.map(style='position')
def f(key, value, up)
     pass
# f('battery', 23.7, True)
#    key='battery'
#    value=23.7
#    up=True
```

Attributes by position consuming a subset of attributes:

```
@spl.map(style='position')
def f(a, b)
   pass
# f('battery', 23.7)
#    a='battery'
#    b=23.7
```

Attributes by position consuming *id* by position and remaining attributes by `*args`:

```
@spl.map(style='position')
def f(key, *tuple_)
```

(continues on next page)

(continued from previous page)

```
    pass
# f('battery', 23.7, True)
#    key='battery'
#    tuple_=(23.7, True)
```

In all cases the SPL tuple must be able to provide all parameters required by the function. If the SPL schema is insufficient then an error will result, typically an SPL compile time error.

The SPL schema can provide a subset of the formal parameters if the remaining attributes are optional (having a default).

Attributes by name consuming a subset of attributes with an optional parameter not matched by the schema:

```
@spl.map()
def f(id, temp, pressure=None)
   pass
# f(id='battery', temp=23.7)
#     id='battery'
#     temp=23.7
#     pressure=None
```

2.1.5 Submission of SPL tuples from Python

The return from a decorated callable results in submission of SPL tuples on the associated outut port.

A Python function must return:

- `None`
- a Python tuple
- a Python dictionary
- a list containing any of the above.

None

When `None` is return then no tuple will be submitted to the operator output port.

Python tuple

When a Python tuple is returned it is converted to an SPL tuple and submitted to the output port.

The values of a Python tuple are assigned to an output SPL tuple by position, so the first value in the Python tuple is assigned to the first attribute in the SPL tuple:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,b,a+b)

# The SPL output will be:
# All values explictly set by returned Python tuple
# based on the x,y values from the input tuple
```

(continues on next page)

(continued from previous page)

```
# x is set to: x
# y is set to: y
# z is set to: x+y
```

The returned tuple may be *sparse*, any attribute value in the tuple that is `None` will be set to their SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,None,a+b)

# The SPL output will be:
# x is set to: x (explictly set by returned Python tuple)
# y is set to: y (set by matching input SPL attribute)
# z is set to: x+y
```

When a returned tuple has fewer values than attributes in the SPL output schema the attributes not set by the Python function will be set to their SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return a,

# The SPL output will be:
# x is set to: x (explictly set by returned Python tuple)
# y is set to: y (set by matching input SPL attribute)
# z is set to: 0 (default int32 value)
```

When a returned tuple has more values than attributes in the SPL output schema then the additional values are ignored:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,b,a+b,a/b)

# The SPL output will be:
# All values explictly set by returned Python tuple
# based on the x,y values from the input tuple
# x is set to: x
# y is set to: y
# z is set to: x+y
#
# The fourth value in the tuple a/b = x/y is ignored.
```

Python dictionary

A Python dictionary is converted to an SPL tuple for submission to the associated output port. An SPL attribute is set from the dictionary if the dictionary contains a key equal to the attribute name. The value is used to set the attribute, unless the value is `None`.

If the value in the dictionary is `None`, or no matching key exists, then the attribute value is set to its SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind.

Any keys in the dictionary that do not map to SPL attribute names are ignored.

Python list

When a list is returned, each value is converted to an SPL tuple and submitted to the output port, in order of the list starting with the first element (position 0). If the list contains *None* at an index then no SPL tuple is submitted for that index.

The list must only contain Python tuples, dictionaries or *None*. The list can contain a mix of valid values.

The list may be empty resulting in no tuples being submitted.

2.1.6 Module contents

Functions

extracting	Is a module being loaded by `spl-python-extract`.
ignore	Decorator to ignore a Python function.

Classes

PrimitiveOperator	Primitive operator super class.
filter	Decorator that creates a filter SPL operator from a callable class or function.
for_each	Creates an SPL operator with a single input port.
input_port	Declare an input port and its processor method.
map	Decorator to create a map SPL operator from a callable class or function.
primitive_operator	Creates an SPL primitive operator with an arbitrary number of input ports and output ports.
source	Create a source SPL operator from an iterable.

class streamsx.spl.spl.**source**(*docpy=True*)

Bases: object

Create a source SPL operator from an iterable. The resulting SPL operator has a single output port.

When decorating a class the class must be iterable having an `__iter__` function. When the SPL operator is invoked an instance of the class is created and an iteration is created using `iter(instance)`.

When decoratiing a function the function must have no parameters and must return an iterable or iteration. When the SPL operator is invoked the function is called and an iteration is created using `iter(value)` where

`value` is the return of the function.

For each value in the iteration SPL zero or more tuples are submitted to the output port, derived from the value, see *Submission of SPL tuples from Python*.

If the iteration completes then no more tuples are submitted and a window punctuation mark followed by final punctuation mark are submitted to the output port.

Example definition:

```
@spl.source()
class Range(object):
    def __init__(self, stop, start=0):
        self.start = start
        self.stop = stop

    def __iter__(self):
        return zip(range(self.start, self.stop))
```

Example SPL invocation:

```
stream<int32 seq> R = Range() {
    param
        stop: 100;
}
```

If `__iter__` or `__next__` block then shutdown, checkpointing or consistent region processing may be delayed. Having `__next__` return `None` (no available tuples) or tuples to submit will allow such processing to proceed.

A shutdown `threading.Event` is available through *`streamsx.ec.shutdown()`* which becomes set when a shutdown of the processing element has been requested. This event my be waited on to perform a sleep that will terminate upon shutdown.

Parameters **docpy** – Copy Python docstrings into SPL operator model for SPLDOC.

Exceptions raised by `__iter__` and `__next__` can be suppressed when this decorator wraps a class with context manager `__enter__` and `__exit__` methods.

If `__exit__` returns a true value when called with an exception then the exception is suppressed.

Suppressing an exception raised by `__iter__` results in the source producing an empty iteration. No tuples will be submitted.

Suppressing an exception raised by `__next__` results in the source not producing any tuples for that invocation. Processing continues with a call to `__next__`.

Data conversion errors of the value returned by `__next__` can also be suppressed by `__exit__`. If `__exit__` returns a true value when called with the exception then the exception is suppressed and the value that caused the exception is not submitted as an SPL tuple.

class `streamsx.spl.spl.`**map**(*style=None*, *docpy=True*)

Bases: `object`

Decorator to create a map SPL operator from a callable class or function.

Creates an SPL operator with a single input port and a single output port. For each tuple on the input port the callable is called passing the contents of the tuple.

The value returned from the callable results in zero or more tuples being submitted to the operator output port, see *Submission of SPL tuples from Python*.

Example definition:

```
@spl.map()
class AddSeq(object):
"""Add a sequence number as the last attribute."""
def __init__(self):
    self.seq = 0

def __call__(self, *tuple_):
    id = self.seq
    self.seq += 1
    return tuple_ + (id,)
```

Example SPL invocation:

```
stream<In, tuple<uint64 seq>> InWithSeq = AddSeq(In) { }
```

Parameters

- **style** – How the SPL tuple is passed into Python callable or function, see *Processing SPL tuples in Python*.
- **docpy** – Copy Python docstrings into SPL operator model for SPLDOC.

Exceptions raised by `__call__` can be suppressed when this decorator wraps a class with context manager `__enter__` and `__exit__` methods. If `__exit__` returns a true value when called with the exception then the exception is suppressed and the tuple that caused the exception is dropped.

Data conversion errors of the value returned by `__call__` can also be suppressed by `__exit__`. If `__exit__` returns a true value when called with the exception then the exception is suppressed and the value that caused the exception is not submitted as an SPL tuple.

class streamsx.spl.spl.**filter**(*style=None*, *docpy=True*)

Bases: object

Decorator that creates a filter SPL operator from a callable class or function.

A filter SPL operator has a single input port and one mandatory and one optional output port. The schema of each output port must match the input port. For each tuple on the input port the callable is called passing the contents of the tuple. if the function returns a value that evaluates to True then it is submitted to mandatory output port 0. Otherwise it it submitted to the second optional output port (1) or discarded if the port is not specified in the SPL invocation.

Parameters

- **style** – How the SPL tuple is passed into Python callable or function, see *Processing SPL tuples in Python*.
- **docpy** – Copy Python docstrings into SPL operator model for SPLDOC.

Example definition:

```
@spl.filter()
class AttribThreshold(object):
    """
    Filter based upon a single attribute being
    above a threshold.
    """
    def __init__(self, attr, threshold):
        self.attr = attr
        self.threshold = threshold
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, **tuple_):
        return tuple_[self.attr] > self.threshold:
```

Example SPL invocation:

```
stream<rstring id, float64 voltage> Sensors = ...
stream<Sensors> InterestingSensors = AttribThreshold(Sensors) {
    param
      attr: "voltage";
      threshold: 225.0;
}
```

Exceptions raised by `__call__` can be suppressed when this decorator wraps a class with context manager `__enter__` and `__exit__` methods. If `__exit__` returns a true value when called with the exception then the expression is suppressed and the tuple that caused the exception is dropped.

class streamsx.spl.spl.**for_each**(*style=None*, *docpy=True*)

Bases: object

Creates an SPL operator with a single input port.

A SPL operator with a single input port and no output ports. For each tuple on the input port the decorated callable is called passing the contents of the tuple.

Example definition:

```
@spl.for_each()
def PrintTuple(*tuple_):
"""Print each tuple to standard out."""
    print(tuple_, flush=True)
```

Example SPL invocation:

```
() as PT = PrintTuple(SensorReadings) { }
```

Parameters

- **style** – How the SPL tuple is passed into Python callable, see *Processing SPL tuples in Python*.
- **docpy** – Copy Python docstrings into SPL operator model for SPLDOC.

Exceptions raised by `__call__` can be suppressed when this decorator wraps a class with context manager `__enter__` and `__exit__` methods. If `__exit__` returns a true value when called with the exception then the expression is suppressed and the tuple that caused the exception is ignored.

class streamsx.spl.spl.**PrimitiveOperator**

Bases: object

Primitive operator super class. Classes decorated with *@spl.primitive_operator* must extend this class if they have one or more output ports. This class provides the *submit* method to submit tuples to specified otuput port.

New in version 1.8.

all_ports_ready()

Notifcation that the operator can submit tuples.

Called when the primitive operator can submit tuples using *submit()*. An operator must not submit tuples until this method is called or until a port processing method is called.

Any implementation must not block. A typical use is to start threads that submit tuples.

An implementation must return a value that allows the SPL runtime to determine when an operator completes. An operator completes, and finalizes its output ports when:

- All input ports (if any) have been finalized.
- All background processing is complete.

The return from `all_ports_ready` defines when background processing, such as threads started by `all_ports_ready`, is complete. The value is one of:

- A value that evaluates to *False* - No background processing exists.
- A value that evaluates to *True* - Background processing exists and never completes. E.g. a source operator that processes real time events.
- A callable - Background processing is complete when the callable returns. The SPL runtime invokes the callable once (passing no arguments) when the method returns background processing is assumed to be complete.

For example if an implementation starts a single thread then *Thread.join* is returned to complete the operator when the thread completes:

```
def all_ports_ready(self):
    submitter = threading.Thread(target=self._find_and_submit_data)
    submitter.start()
    return submitter.join

def _find_and_submit_data(self):
    ...
```

Returns Value indicating active background processing.

This method implementation does nothing and returns `None`.

submit(*port_id*, *tuple_*)

Submit a tuple to the output port.

The value to be submitted (`tuple_`) can be a `None` (nothing will be submitted), `tuple`, `dict` or `list` of those types. For details on how the `tuple_` is mapped to an SPL tuple see *Submission of SPL tuples from Python*.

Parameters

- **port_id** – Identifier of the port specified in the `output_ports` parameter of the `@spl.primitive_operator` decorator.
- **tuple_** – Tuple (or tuples) to be submitted to the output port.

class streamsx.spl.spl.**input_port**(*style=None*)

Bases: `object`

Declare an input port and its processor method.

Instance methods within a class decorated by *spl.primitive_operator* declare input ports by decorating methods with this decorator.

Each tuple arriving on the input port will result in a call to the processor method passing the stream tuple converted to a Python representation depending on the style. The style is determined by the method signature or the *style* parameter, see *Processing SPL tuples in Python*.

The order of the methods within the class define the order of the ports, so the first port is the first method decorated with *input_port*.

Parameters **style** – How the SPL tuple is passed into the method, see *Processing SPL tuples in Python*.

New in version 1.8.

class streamsx.spl.spl.**primitive_operator**(*output_ports=None*, *docpy=True*)
Bases: object

Creates an SPL primitive operator with an arbitrary number of input ports and output ports.

Input ports are declared by decorating an instance method with *input_port()*. The method is the process method for the input port and is called for each tuple that arrives at the port. The order of the decorated process methods defines the order of the ports in the SPL operator, with the first process method being the first port at index zero.

Output ports are declared by the output_ports parameter which is set to a list of port identifiers. The port identifiers are arbitrary but must be hashable. Port identifiers allow the ability to submit tuples "logically' rather than through a port index. Typically a port identifier will be a *str* or an *enum*. The size of the list defines the number of output ports with the first identifier in the list coresponding to the first output port of the operator at index zero. If the list is empty or not set then the operator has no output ports.

Tuples are submitted to an output port using *submit()*.

When an operator has output ports it must be a sub-class of *PrimitiveOperator* which provides the *submit()* method and the ports ready notification mechanism *all_ports_ready()*.

Example definition of an operator with a single input port and two output ports:

```
@spl.primitive_operator(output_ports=['MATCH', 'NEAR_MATCH'])
class SelectCustomers(spl.PrimitiveOperator):
    """ Score customers using a model.
    Customers that are a good match are submitted to port 0 ('MATCH')
    while customers that are a near match are submitted to port 1 ('NEAR_MATCH').

    Customers that are not a good or near match are not submitted to any port.
    """
    def __init__(self, match, near_match):
        self.match = match
        self.near_match = near_match

    @spl.input_port()
    def customers(self, **tuple_):
         customer_score = self.score(tuple_)
         if customer_score >= self.match:
             self.submit('MATCH', tuple_)
         elif customer_score >= self.near_match:
             self.submit('NEAR_MATCH', tuple_)

    def score(self, **customer):
        # Actual model scoring omitted
        score = ...
        return score
```

Example SPL invocation:

```
(stream<Customers> MakeOffer; stream<Customers> ImproveOffer>) =␣
↪SelectCustomers(Customers) {
    param
        match: 0.9;
        near_match: 0.8;
}
```

Parameters

- **output_ports** (*list*) – List of identifiers for output ports.
- **docpy** – Copy Python docstrings into SPL operator model for SPLDOC.

New in version 1.8.

streamsx.spl.spl.**extracting**()

Is a module being loaded by `spl-python-extract`.

This can be used by modules defining SPL primitive operators using decorators such as *@spl.map*, to avoid runtime behavior. Typically not importing modules that are not available locally. The extraction script loads the module to determine method signatures and thus does not invoke any methods.

For example if an SPL toolkit with primitive operators requires a package `extras` and is using `opt/python/streams/requirements.txt` to include it, then loading it at extraction time can be avoided by:

```
from streamsx.spl import spl

def spl_namespace():
    return 'myns.extras'

if not spl.extracting():
    import extras

@spl.map():
def myextras(*tuple_):
    return extras.process(tuple_)
```

New in version 1.11.

streamsx.spl.spl.**ignore**(*wrapped*)

Decorator to ignore a Python function.

If a Python callable is decorated with `@spl.ignore` then function is ignored by `spl-python-extract.py`.

Parameters **wrapped** – Function that will be ignored.

CHAPTER

THREE

STREAMS PYTHON REST API

Module that allows interaction with an running Streams instance or service through HTTPS REST APIs.

streamsx.build	REST API bindings for IBM® Streams Cloud Pak for Data build service.
streamsx.rest	REST API bindings for IBM® Streams & Streaming Analytics service.
streamsx.rest_primitives	Primitive objects for REST bindings.

3.1 streamsx.build

REST API bindings for IBM® Streams Cloud Pak for Data build service.

3.1.1 Streams Build REST API

The REST Build API provides programmatic support for creating, submitting and managing Streams builds. You can use the REST Build API from any application that can establish an HTTPS connection to the server that is running the build service. The current support includes only methods for managing toolkits in the build service.

Cloud Pak for Data

of_endpoint() is the entry point to using the Streams Build REST API bindings, returning an *BuildService*.

See also:

IBM Streaming Analytics service

3.1.2 Module contents

Classes

BuildService	IBM Streams build service.

class streamsx.build.**BuildService**(*username=None*, *password=None*, *resource_url=None*, *auth=None*)

Bases: streamsx.rest._AbstractStreamsConnection

IBM Streams build service.

A instance of a *BuildService* is created using `of_endpoint()`.

New in version 1.13.

get_resources()
Retrieves a list of all known Streams high-level Build REST resources.

Returns List of all Streams high-level Build REST resources.

Return type `list` of *RestResource*

get_toolkit(*id*)
Retrieves available toolkit matching a specific toolkit ID.

Parameters **id** (*str*) – Toolkit identifier to retrieve. This is the name and version of a toolkit. For sample, *com.ibm.streamsx.rabbitmq-1.1.3*

Returns Toolkit matching *id*.

Return type *Toolkit*

Raises **ValueError** – No matching toolkit exists.

get_toolkits(*name=None*)
Retrieves a list of all installed Streams Toolkits.

Returns List of all Toolkit resources.

Return type `list` of *Toolkit*

Parameters **name** (*str*) – Return toolkits matching name as a regular expression.

static of_endpoint(*endpoint=None*, *service_name=None*, *username=None*, *password=None*, *verify=None*)
Connect to a Cloud Pak for Data IBM Streams build service instance.

Two configurations are supported.

Integrated configuration

The build service is bound to a Streams instance and is defined using the Cloud Pak for Data deployment endpoint (URL) and the Streams service name.

The endpoint is passed in as *endpoint* defaulting the the environment variable `CP4D_URL`. An example is *https://cp4d_server:31843*.

The Streams service name is passed in as *service_name* defaulting to the environment variable `STREAMS_INSTANCE_ID`.

Standalone configuration

A build service is independent of a Streams instance and is defined using the build service endpoint.

The endpoint is passed in as *endpoint* defaulting the the environment variable `STREAMS_BUILD_URL`. An example is *https://build_service:34679*.

No service name is specified thus *service_name* should be passed as `None` or not set.

Parameters

- **endpoint** (*str*) – Endpoint defining the build service.

- **service_name** (*str*) – Streams instance name for a integrated configuration. This value is ignored for a standalone configuration.
- **username** (*str*) – User name to authenticate as. Defaults to the environment variable `STREAMS_USERNAME` or the operating system identifier if not set.
- **password** (*str*) – Password for authentication. Defaults to the environment variable `STREAMS_PASSWORD` or the operating system identifier if not set.
- **verify** – SSL verification. Set to `False` to disable SSL verification. Defaults to SSL verification being enabled.

Returns Connection to Streams build service or `None` of insufficient configuration was provided.

Return type *BuildService*

property resource_url
Endpoint URL for IBM Streams REST build API.

Type str

upload_toolkit(*path*)
Upload a toolkit from a directory in the local filesystem to the Streams build service.

Multiple versions of a toolkit may be uploaded as long as each has a unique version. If a toolkit is uploaded with a name and version matching an existing toolkit, it will not replace the existing toolkit, and `None` will be returned.

Parameters **path** (*str*) – The path to the toolkit directory in the local filesystem.

Returns The created Toolkit, or `None` if it was not uploaded.

Return type *Toolkit*

3.2 streamsx.rest

REST API bindings for IBM® Streams & Streaming Analytics service.

3.2.1 Streams REST API

The Streams REST API provides programmatic access to configuration and status information for IBM Streams objects such as domains, instances, and jobs.

IBM Cloud Pak for Data (Streams 5)

Integrated configuration within project

of_service() is the entry point to using the Streams REST API bindings, returning an *Instance*. The configuration required to connect is obtained from `ipcd_util.get_service_details` passing in the IBM Streams service instance name.

Integrated & standalone configurations

of_endpoint() is the entry point to using the Streams REST API bindings, returning an *Instance*.

IBM Streams On-premises (4.2, 4.3)

StreamsConnection is the entry point to using the Streams REST API bindings. Through its functions and the returned objects status information can be obtained for items such as *instances* and *jobs*.

3.2.2 Streaming Analytics REST API

You can use the Streaming Analytics REST API to manage your service instance and the IBM Streams jobs that are running on the instance. The Streaming Analytics REST API is accessible from IBM Cloud applications that are bound to your service instance or from an application outside of IBM Cloud that is configured with the service instance VCAP information.

StreamingAnalyticsConnection is the entry point to using the Streaming Analytics REST API. The function *get_streaming_analytics()* returns a *StreamingAnalyticsService* instance which is the wrapper around the Streaming Analytics REST API. This API allows functions such as *start* and *stop* the service instance.

In addition *StreamingAnalyticsConnection* extends from *StreamsConnection* and thus provides access to the Streams REST API for the service instance.

See also:

IBM Streams REST API overview Reference documentation for the Streams REST API.

Streaming Analytics REST API Reference documentation for the Streaming Analytics service REST API.

See also:

IBM Streaming Analytics service

3.2.3 Module contents

Classes

StreamingAnalyticsConnection	Creates a connection to a running Streaming Analytics service and exposes methods to retrieve the state of the service and its instance.
StreamsConnection	Creates a connection to a running distributed IBM Streams instance and exposes methods to retrieve the state of that instance.

class streamsx.rest.**StreamsConnection**(*username=None*, *password=None*, *resource_url=None*, *auth=None*)

Bases: streamsx.rest._AbstractStreamsConnection

Creates a connection to a running distributed IBM Streams instance and exposes methods to retrieve the state of that instance.

Streams maintains information regarding the state of its resources. For example, these resources could include the currently running Jobs, Views, PEs, Operators, and Domains. The *StreamsConnection* provides methods to retrieve that information.

Parameters

- **username** (*str*) – Username of an authorized Streams user. If `None`, the username is taken from the `STREAMS_USERNAME` environment variable. If the `STREAMS_USERNAME` environment variable is not set, the default *streamsadmin* is used.
- **password** (*str*) – Password for *username* If `None`, the password is taken from the `STREAMS_PASSWORD` environment variable. If the `STREAMS_PASSWORD` environment variable is not set, the default *passw0rd* is used to match the Streams Quick Start edition setup.
- **resource_url** (*str*) – Root URL for IBM Streams REST API. If `None`, the URL is taken from the `STREAMS_REST_URL` environment variable. If the `REST_URL` environment variable is not set, then `streamtool geturl --api` is used to obtain the URL.

Example

```
>>> resource_url = "https://streamsqse.localdomain:8443/streams/rest/resources"
>>> sc = StreamsConnection("streamsadmin", "passw0rd", resource_url)
>>> sc.session.verify=False  # manually disable SSL verification, if needed
>>> instances = sc.get_instances()
>>> jobs_count = 0
>>> for instance in instances:
>>>     jobs_count += len(instance.get_jobs())
>>> print("There are {} jobs across all instances.".format(jobs_count))
There are 10 jobs across all instances.
```

session
Requests session object for making REST calls.

Type `requests.Session`

get_domain(*id*)
Retrieves available domain matching a specific domain ID

Parameters **id** (*str*) – domain ID

Returns Domain matching *id*

Return type *Domain*

Raises **ValueError** – No matching domain exists.

get_domains()
Retrieves available domains.

Returns List of available domains

Return type `list` of *Domain*

get_installations()
Retrieves a list of all known Streams installations.

Returns List of all Installation resources.

Return type `list` of *Installation*

get_instance(*id*)
Retrieves available instance matching a specific instance ID.

Parameters **id** (*str*) – Instance identifier to retrieve.

Returns Instance matching *id*.

Return type *Instance*

Raises **ValueError** – No matching instance exists or multiple matching instances exist.

get_instances()
Retrieves available instances.

Returns List of available instances

Return type list of *Instance*

get_resources()
Retrieves a list of all known Streams high-level REST resources.

Returns List of all Streams high-level REST resources.

Return type list of *RestResource*

property resource_url
Root URL for IBM Streams REST API

Type str

class streamsx.rest.**StreamingAnalyticsConnection**(*vcap_services=None*, *service_name=None*)
Bases: *streamsx.rest.StreamsConnection*

Creates a connection to a running Streaming Analytics service and exposes methods to retrieve the state of the service and its instance.

Parameters

- **vcap_services** (*str, optional*) – VCAP services (JSON string or a filename whose content contains a JSON string). If not specified, it uses the value of **VCAP_SERVICES** environment variable.
- **service_name** (*str, optional*) – Name of the Streaming Analytics service. If not specified, it uses the value of **STREAMING_ANALYTICS_SERVICE_NAME** environment variable.

Example

```
>>> # Assume environment variable VCAP_SERVICES has correct information
>>> sc = StreamingAnalyticsConnection(service_name='Streaming-Analytics')
>>> print(sc.get_streaming_analytics().get_instance_status())
{'plan': 'Standard', 'state': 'STARTED', 'enabled': True, 'status': 'running'}
```

get_domain(*id*)
Retrieves available domain matching a specific domain ID

Parameters **id** (*str*) – domain ID

Returns Domain matching *id*

Return type *Domain*

Raises **ValueError** – No matching domain exists.

get_domains()
Retrieves available domains.

Returns List of available domains

Return type list of *Domain*

get_installations()
Retrieves a list of all known Streams installations.

Returns List of all Installation resources.

Return type list of *Installation*

get_instance(*id*)
Retrieves available instance matching a specific instance ID.

Parameters **id** (*str*) – Instance identifier to retrieve.

Returns Instance matching *id*.

Return type *Instance*

Raises **ValueError** – No matching instance exists or multiple matching instances exist.

get_instances()
Retrieves available instances.

Returns List of available instances

Return type list of *Instance*

get_resources()
Retrieves a list of all known Streams high-level REST resources.

Returns List of all Streams high-level REST resources.

Return type list of *RestResource*

get_streaming_analytics()
Returns a *StreamingAnalyticsService* to allow further interaction with the Streaming Analytics service.

Returns Object for interacting with the Streaming Analytics service.

Return type *StreamingAnalyticsService*

static of_definition(*service_def*)
Create a connection to a Streaming Analytics service.

The single service is defined by *service_def* which can be one of

- The *service credentials* copied from the *Service credentials* page of the service console (not the Streams console). Credentials are provided in JSON format. They contain such as the API key and secret, as well as connection information for the service.
- A JSON object (*dict*) of the form: `{ "type": "streaming-analytics", "name": "service name", "credentials": {...} }` with the *service credentials* as the value of the `credentials` key.

Parameters **service_def** (*dict*) – Definition of the service to connect to.

Returns Connection to defined service.

Return type *StreamingAnalyticsConnection*

property resource_url
Root URL for IBM Streams REST API

Type str

3.3 streamsx.rest_primitives

Primitive objects for REST bindings.

3.3.1 Overview

Contains classes representing primitive Streams objects, such as *Instance*, *Job*, *PE*, etc.

3.3.2 Module contents

Functions

get_view_obj

Classes

ActiveService	Domain or instance service.
ActiveVersion	Contains IBM Streams installation information
ApplicationBundle	Application bundle tied to an instance.
ApplicationConfiguration	An application configuration.
Domain	IBM Streams domain.
ExportedStream	Stream exported stream by a job.
Host	Resource in a Streams domain or instance.
ImportedStream	Stream imported by a job.
Installation	IBM Streams installation.
Instance	IBM Streams instance.
Job	A running streams application.
JobGroup	A job group definition.
Metric	Streams custom or system metric.
Operator	An operator invocation within a job.
OperatorConnection	Connection between operators.
OperatorInputPort	Operator input port.
OperatorOutputPort	Operator output port.
PE	Processing element (PE) within a job.
PEConnection	Stream connection between two PEs.
PublishedTopic	Metadata for a published topic.
Resource	A resource available to a IBM Streams domain.
ResourceAllocation	A resource that is allocated to an IBM Streams instance.
ResourceTag	Resource tag defined in a Streams domain
RestResource	HTTP REST resource identifier.
StreamingAnalyticsService	Streaming Analytics service running on IBM Cloud.
Toolkit	IBM Streams toolkit.
View	View on a stream.
ViewItem	A stream tuple in view.

class streamsx.rest_primitives.**ActiveService**(*json_rep*, *rest_client*)

Bases: streamsx.rest_primitives._ResourceElement

Domain or instance service.

resourceType
Identifies the REST resource type, which is *activeService*.

Type str

leader
If *True*, this service is a standby service.

Type bool

processId
Process ID of this service.

Type str

startTime
Epoch time when this service started.

Type long

status
Status of this service. Some possible values include *stopped*, *running*, *failed*, and *unknown*.

Type str

type
Type of this service.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> services = instances.get_active_services()
>>> print(services[0].resourceType)
activeService
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**ActiveVersion**(*json_active_version*)
Bases: object

Contains IBM Streams installation information

architecture
Hardware architecture on which product is installed.

Type str

build_version
Product build ID.

Type str

edition_name
Product edition.

Type str

full_product_version
Full product version, including any hot fix.

Type str

minimum_os_base_version
Minimum operating system version requirement.

Type str

minimum_os_patch_version
Minimum operating system patch requirement.

Type str

product_name
Product name.

Type str

product_version
Product version.

Type str

class streamsx.rest_primitives.**ApplicationBundle**(*_delegator*, *instance*, *json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Application bundle tied to an instance.

New in version 1.11.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

submit_job(*job_config=None*)
Submit this Streams Application Bundle (sab file) to its associated instance.

Parameters **job_config** (JobConfig) – a job configuration overlay

Returns Resulting job instance.

Return type *Job*

class streamsx.rest_primitives.**ApplicationConfiguration**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

An application configuration.

Application configurations are used for secure storage and retrieval of name/value pairs.

An application configuration maintains a set of properties that an application can access at runtime. These are typically used to maintain connection endpoint and credentials for sources and sinks.

name
Name of the configuration.

Type str

description
Description for the configuration.

Type str

properties
Property values stored for the configuration.

Type dict

creationTime
Epoch time when this configuraiton was created.

Type long

lastModifiedTime
Epoch time when this configuration was last modified.

Type long

delete()
Delete this application configuration.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

update(*properties=None*, *description=None*)
Update this application configuration.

To create or update a property provide its key-value pair in *properties*.

To delete a property provide its key with the value `None` in properties.

Parameters

- **properties** (*dict*) – Property values to be updated. If `None` the properties are unchanged.
- **description** (*str*) – Description for the configuration. If `None` the description is unchanged.

Returns self

Return type *ApplicationConfiguration*

class streamsx.rest_primitives.**Domain**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

IBM Streams domain. A domain contains instances that support running Streams applications as jobs.

id
Unique ID for this domain.

Type str

resourceType
Identifies the REST resource type, which is *domain*.

Type str

creationTime
Epoch time when this domain was created.

Type long

creationuser
User ID that created this domain.

Type str

status
Status of this domain. Some possible values include *running*, *stopping*, *stopped*, *starting*, *removing*, and *unknown*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> domains = sc.get_domains()
>>> print (domains[0].resourceType)
domain
```

get_active_services()
Get the list of *ActiveService* elements associated with this domain.

Returns List of ActiveService elements associated with this domain.

Return type list(*ActiveService*)

get_hosts()
Get the list of *Host* elements associated with this domain.

Returns List of Host elements associated with this domain.

Return type list(*Host*)

get_instances()
Get the list of *Instance* elements associated with this domain.

Returns List of Instance elements associated with this domain.

Return type list(*Instance*)

get_resource_allocations()
Get the list of *ResourceAllocation* elements associated with this domain.

Returns List of ResourceAllocation elements associated with this domain.

Return type list(*ResourceAllocation*)

get_resources()
Get the list of *Resource* elements associated with this domain.

Returns List of Resource elements associated with this domain.

Return type list(*Resource*)

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**ExportedStream**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Stream exported stream by a job.

resourceType
Identifies the REST resource type, which is *exportedStream*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> exportedstreams = instances[0].get_exported_streams()
>>> print (exportedstreams[0].resourceType)
exportedStream
```

get_operator_output_port()
Get the output port of this exported stream.

Returns Output port of this exported stream.

Return type *OperatorOutputPort*

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**Host**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Resource in a Streams domain or instance.

name
Configuration name for the IBM Streams resource.

Type str

resourceType
Identifies the REST resource type, which is *host*.

Type str

ipAddress
IP address for the IBM Streams resource.

Type str

processorCount
Number of processors on the IBM Streams resource.

Type int

restrictedTags
Set of resource tags that processing elements (PEs) must have to run on the IBM Streams resource.

Type list(str)

services
Name and status of each domain service that is designated to run on the IBM Streams resource.

Type list(dict)

status
Status of the IBM Streams resource.

Type str

tag
Names of each tag that is assigned to the IBM Streams resource.

Type list(str)

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> domains = sc.get_domains()
>>> hosts = domains[0].get_hosts()
>>> print (hosts[0].resourceType)
host
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**ImportedStream**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Stream imported by a job.

resourceType
Identifies the REST resource type, which is *importedStream*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> importedstreams = instances[0].get_imported_streams()
>>> print (importedstreams[0].resourceType)
importedStream
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**Installation**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

IBM Streams installation.

resourceType
Identifies the REST resource type, which is *installation*.

Type str

architecture
Hardware architecture on which product is installed.

Type str

buildVersion
Product build ID.

Type str

editionName
Product edition.

Type str

fullProductVersion
Full product version, including any hot fix.

Type str

minimumOSBaseVersion
Minimum operating system version requirement.

Type str

minimumOSPatchVersion
Minimum operating system patch requirement.

Type str

productName
Product name.

Type str

productVersion
Product version.

Type str

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**Instance**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

IBM Streams instance.

id
Unique ID for this instance.

Type str

resourceType
Identifies the REST resource type, which is *instance*.

Type str

creationTime
Epoch time when this instance was created.

Type long

creationuser
User ID that created this instance.

Type str

health
Summarize status of the jobs in the instance. Some possible values include *healthy*, *partiallyHealthy*, *partiallyUnhealthy*, *unhealthy*, and *unknown*.

Type str

owner
User ID that owns this instance.

Type str

startTime
Epoch time when this instance was started.

Type long

status
Status of this instance. Some possible values include *running*, *failed*, *stopped*, and *unknown*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> print (instances[0].resourceType)
instance
```

create_application_configuration(*name*, *properties*, *description=None*)
Create an application configuration.

Parameters **name** (*str, optional*) – Only return application configurations containing property **name** that matches *name*. *name* can be a

get_active_services()
Get the list of *ActiveService* elements associated with this instance.

Returns List of ActiveService elements associated with this instance.

Return type list(*ActiveService*)

get_application_configurations(*name=None*)
Retrieves application configurations for this instance.

Parameters **name** (*str, optional*) – Only return application configurations containing property **name** that matches *name*. *name* can be a regular expression. If *name* is not supplied, then all application configurations are returned.

Returns A list of application configurations matching the given *name*.

Return type list(*ApplicationConfiguration*)

get_domain()
Get the Streams domain that owns this instance.

Returns Streams domain owning this instance.

Return type *Domain*

get_exported_streams()
Get the list of *ExportedStream* elements associated with this instance.

Returns List of ExportedStream elements associated with this instance.

Return type list(*ExportedStream*)

get_hosts()
Get the list of *Host* element associated with this instance.

Returns List of Host element associated with this instance.

Return type list(*Host*)

get_imported_streams()
Get the list of *ImportedStream* elements associated with this instance.

Returns List of ImportedStream elements associated with this instance.

Return type list(*ImportedStream*)

get_job(*id*)
Retrieves a job matching the given *id*

Parameters **id** (*str*) – Job *id* to match.

Returns Job matching the given *id*

Return type *Job*

Raises **ValueError** – No resource matches given *id* or multiple resources matching given *id*

get_job_groups(*name=None*)
Retrieves job groups defined in this instance.

Parameters **name** (*str, optional*) – Only return job groups containing property **name** that matches *name*. *name* can be a regular expression. If *name* is not supplied, then all job groups are returned.

Returns A list of job groups matching the given *name*.

Return type list(JobGroup)

Only supported for Streams 5.0 and later.

New in version 1.13.13.

get_jobs(*name=None*)
Retrieves jobs running in this instance.

Parameters **name** (*str, optional*) – Only return jobs containing property **name** that matches *name*. *name* can be a regular expression. If *name* is not supplied, then all jobs are returned.

Returns A list of jobs matching the given *name*.

Return type list(*Job*)

Retrieving a list of jobs whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instance = sc.get_instances()[0]
>>> jobs = instance.get_jobs(name=".*temperatureApplication*")
```

get_operator_connections()
Get the list of *OperatorConnection* elements associated with this instance.

Returns List of OperatorConnection elements associated with this instance.

Return type list(*OperatorConnection*)

get_operators(*name=None*)
Get the list of *Operator* elements associated with this instance.

Parameters **name** (*str*) – Only return operators matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all operators for this instance are returned.

Returns List of Operator elements associated with this instance.

Return type list(*Operator*)

Retrieving a list of operators whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instance = sc.get_instances()[0]
>>> operators = instance.get_operators(name=".*temperatureSensor*")
```

Changed in version 1.9: *name* parameter added.

get_pe_connections()
Get the list of *PEConnection* elements associated with this instance.

Returns List of PEConnection elements associated with this instance.

Return type list(*PEConnection*)

get_pes()
Get the list of *PE* elements associated with this instance resource.

Returns List of PE elements associated with this instance.

Return type list(*PE*)

get_published_topics()
Get a list of published topics for this instance.

Streams applications publish streams to a a topic that can be subscribed to by other applications. This allows a microservice approach where publishers and subscribers are independent of each other.

A published stream has a topic and a schema. It is recommended that a topic is only associated with a single schema.

Streams may be published and subscribed by applications regardless of the implementation language. For example a Python application can publish a stream of JSON tuples that are subscribed to by SPL and Java applications.

Returns List of currently published topics.

Return type list(*PublishedTopic*)

get_resource_allocations()
Get the list of *ResourceAllocation* elements associated with this instance.

Returns List of ResourceAllocation elements associated with this instance.

Return type list(*ResourceAllocation*)

get_views(*name=None*)
Get the list of *View* elements associated with this instance.

Parameters

- **name** (*str, optional*) – Returns view(s) matching *name*. *name* can be a regular expression. If *name*
- **not supplied, then all views associated with this instance are returned.** (*is*) –

Returns List of views matching *name*.

Return type list(*streamsx.rest_primitives.View*)

Retrieving a list of views whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instance = sc.get_instances()[0]
>>> view = instance.get_views(name=".*temperatureSensor*")
```

static of_endpoint(*endpoint=None*, *service_name=None*, *username=None*, *password=None*, *verify=None*)

Connect to a Cloud Pak for Data IBM Streams instance.

Two configurations are supported.

Integrated configuration

The Streams instance is defined using the Cloud Pak for Data deployment endpoint (URL) and the Streams service name.

The endpoint is passed in as *endpoint* defaulting the the environment variable `CP4D_URL`. An example is *https://cp4d_server:31843*.

The Streams service name is passed in as *service_name* defaulting to the environment variable `STREAMS_INSTANCE_ID`.

Standalone configuration

The Streams instance is defined using its Streams REST api endpoint, which is its SWS service.

The endpoint is passed in as *endpoint* defaulting the the environment variable `STREAMS_REST_URL`. An example is *https://streams_sws_service:34679*.

No service name is specified thus *service_name* should be passed as `None` or not set.

Parameters

- **endpoint** (*str*) – Endpoint defining the Streams instance.
- **service_name** (*str*) – Streams instance name for a integrated configuration. This value is ignored for a standalone configuration.
- **username** (*str*) – User name to authenticate as. Defaults to the environment variable `STREAMS_USERNAME` or the operating system identifier if not set.
- **password** (*str*) – Password for authentication. Defaults to the environment variable `STREAMS_PASSWORD` or the operating system identifier if not set.
- **verify** – SSL verification. Set to `False` to disable SSL verification. Defaults to SSL verification being enabled.

Returns Connection to Streams instance or `None` of insufficient configuration was provided.

Return type *Instance*

New in version 1.13.

static of_service(*config*)

Connect to an IBM Streams service instance running in Cloud Pak for Data.

The instance is specified in *config*. The configuration may be code injected from the list of services in a Jupyter notebook running in ICPD or manually created. The code that selects a service instance by name is:

```
# Two lines are code injected in a Jupyter notebook by selecting the service␣
↪instance
from icpd_core import ipcd_util
cfg = icpd_util.get_service_details(name='instanceName')

instance = Instance.of_service(cfg)
```

SSL host verification is disabled by setting *SSL_VERIFY* to `False` within *config* before calling this method:

```
cfg[ConfigParams.SSL_VERIFY] = False
instance = Instance.of_service(cfg)
```

Parameters **config** (*dict*) – Configuration of IBM Streams service instance.

Returns Instance representing for IBM Streams service instance.

Return type *Instance*

Note: Only supported when running within the ICPD cluster, for example in a Jupyter notebook within a ICPD project.

New in version 1.12.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

submit_job(*bundle*, *job_config=None*)
Submit a application to be run in this instance.

Parameters

- **bundle** (*str*) – path to a Streams application bundle (sab file) containing the application to be submitted
- **job_config** (JobConfig) – a job configuration overlay

Returns Resulting job instance.

Return type *Job*

New in version 1.11.

upload_bundle(*bundle*)
Upload a Streams application bundle (sab) to the instance.

Uploading a bundle allows job submission from the returned *ApplicationBundle*.

Parameters **bundle** (*str*) – path to a Streams application bundle (sab file) containing the application to be uploaded.

Returns Application bundle representing the uploaded bundle.

Return type *ApplicationBundle*

Note: When an instance does not support uploading a bundle the returned *ApplicationBundle* represents the local file `bundle` tied to this instance. The returned object may still be used for job submission.

New in version 1.11.

class streamsx.rest_primitives.**Job**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

A running streams application.

id
job ID.

Type str

name
Name of the job.

Type str

resourceType
Identifies the REST resource type, which is *job*.

Type str

health
Health indicator for the job. Some possible values for this property include *healthy*, *partiallyHealthy*, *partiallyUnhealthy*, *unhealthy*, and *unknown*.

Type str

applicationName
Name of the streams processing application that this job is running.

Type str

jobGroup
Streams 4.2/4.3 only. Identifies the job group to which this job belongs.

Type str

startedBy
Identifies the user ID that started this job.

Type str

status
Status of this job. Some possible values for this property include *canceling*, *running*, *canceled*, and *unknown*.

Type str

submitTime
Epoch time when this job was submitted.

Type long

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> jobs = instances[0].get_jobs()
>>> print (jobs[0].health)
healthy
```

cancel(*force=False*)
Cancel this job.

Parameters **force** (*bool, optional*) – Forcefully cancel this job.

Returns True if the job was cancelled, otherwise False if an error occurred.

Return type bool

get_domain()
Get the Streams domain that owns this job.

Returns Streams domain that owns this job.

Return type *Domain*

get_hosts()
Get the list of *Host* elements associated with this job.

Returns List of Host elements associated with this job.

Return type list(*Host*)

get_instance()
Get the Streams instance that owns this job.

Returns Streams instance that owns this job.

Return type *Instance*

get_job_group()
Get the job group associated with this job.

New in version 1.13.13.

get_operator_connections()
Get the list of *OperatorConnection* elements associated with this job.

Returns List of OperatorConnection elements associated with this job.

Return type list(*OperatorConnection*)

get_operators(*name=None*)
Get the list of *Operator* elements associated with this job.

Parameters **name** (*str*) – Only return operators matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all operators for this job are returned.

Returns List of Operator elements associated with this job.

Return type list(*Operator*)

Retrieving a list of operators whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> job = instances[0].get_jobs()[0]
>>> operators = job.get_operators(name=".*temperatureSensor*")
```

Changed in version 1.9: *name* parameter added.

get_pe_connections()
Get the list of *PEConnection* elements associated with this job.

Returns List of PEConnection elements associated with this job.

Return type list(*PEConnection*)

get_pes()
Get the list of *PE* elements associated with this job.

Returns List of PE elements associated with this job.

Return type list(*PE*)

get_resource_allocations()
Get the list of *ResourceAllocation* elements associated with this job.

Returns List of ResourceAllocation elements associated with this job.

Return type list(*ResourceAllocation*)

get_views(*name=None*)
Get the list of *View* elements associated with this job.

Parameters

- **name** (*str, optional*) – Returns view(s) matching *name*. *name* can be a regular expression. If *name*
- **not supplied, then all views associated with this instance are returned.** (*is*) –

Returns List of views matching *name*.

Return type list(*streamsx.rest_primitives.View*)

Retrieving a list of views that contain the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> job = instances[0].get_jobs()[0]
>>> views = job.get_views(name = ".*temperatureSensor*")
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

retrieve_log_trace(*filename=None*, *dir=None*)
Retrieves the application log and trace files of the job and saves them as a compressed tar file.

An existing file with the same name will be overwritten.

Parameters

- **filename** (*str*) – name of the created tar file. Defaults to *job_<id>_<timestamp>.tar.gz* where *id* is the job identifier and *timestamp* is the number of seconds since the Unix epoch, for example `job_355_1511995995.tar.gz`.
- **dir** (*str*) – a valid directory in which to save the archive. Defaults to the current directory.

Returns the path to the created tar file, or `None` if retrieving a job's logs is not supported in the version of IBM Streams to which the job is submitted.

Return type str

New in version 1.8.

update_operators(*job_config*)
Adjust a job configuration while the job is running

Parameters **{JobConfig} -- a job configuration overlay**(*job_config*) –

Returns [JSON] – The result of applying the new jobConfig?

class streamsx.rest_primitives.**Metric**(*json_rep*, *rest_client*)

Bases: streamsx.rest_primitives._ResourceElement

Streams custom or system metric.

name
Name of this metric.

Type str

resourceType
Identifies the REST resource type, which is *metric*.

Type str

description
Describes this metric.

Type str

lastTimeRetrieved
Epoch time when the metric was most recently retrieved.

Type str

metricKind
Kind of metric. Some possible values include *counter*, *gauge*, *time* and *unknown*.

Type str

metricType
Type of metric. Some possible values include *system*, *custom* and *unknown*.

Type str

value
Value for the metric.

Type int

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> operators = instances[0].get_operators()
>>> metrics = operators[0].get_metrics()
>>> print (metrics[0].resourceType)
metric
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**OperatorConnection**(*json_rep*, *rest_client*)

Bases: streamsx.rest_primitives._ResourceElement

Connection between operators.

id
Unique ID of this operator connection within the instance.

Type str

resourceType
Identifies the REST resource type, which is *operator*.

Type str

required
Indicates whether the connection is required.

Type bool

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> operatorconnections = instances[0].get_operator_connections()
>>> print (operatorconnections[0].resourceType)
operatorConnection
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**OperatorInputPort**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Operator input port.

name
Name of this input port.

Type str

resourceType
Identifies the REST resource type, which is *operatorInputPort*.

Type str

indexWithinOperator
Index of the input port within the operator.

Type int

New in version 1.9.

get_connections()
Get the list of *OperatorConnection* elements associated with this port.

Returns List of OperatorConnection elements associated with this port.

Return type list(*OperatorConnection*)

New in version 1.13.

get_metrics(*name=None*)
Get metrics for this input port.

Parameters **name** (*str, optional*) – Only return metrics matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all metrics for this input port are returned.

Returns List of matching metrics.

Return type list(*Metric*)

Retrieving a list of metrics whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> operator = instances[0].get_operators()[0]
>>> input_port = operator.get_input_ports()[0]
>>> metrics = input_port.get_metrics(name='*temperatureSensor*')
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**OperatorOutputPort**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Operator output port.

name
Name of this output port.

Type str

resourceType
Identifies the REST resource type, which is *operatorOutputPort*.

Type str

indexWithinOperator
Index of the output port within the operator.

Type int

streamName
Name of the stream that is associated with this output port.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> exportedstreams = instances[0].get_exported_streams()
>>> operatoroutputport = exportedstreams[0].get_operator_output_port()
>>> print (operatoroutputport.resourceType)
operatorOutputPort
```

get_connections()
Get the list of *OperatorConnection* elements associated with this port.

Returns List of OperatorConnection elements associated with this port.

Return type list(*OperatorConnection*)

New in version 1.13.

get_metrics(*name=None*)
Get metrics for this output port.

Parameters **name** (*str, optional*) – Only return metrics matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all metrics for this output port are returned.

Returns List of matching metrics.

Return type list(*Metric*)

Retrieving a list of metrics whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> exportedstreams = instances[0].get_exported_streams()
>>> operatoroutputport = exportedstreams[0].get_operator_output_port()
>>> operatoroutputport.get_metrics(name='*temperatureSensor*')
```

New in version 1.9.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**Operator**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

An operator invocation within a job.

name
Operator name.

Type str

resourceType
Identifies the REST resource type, which is *operator*.

Type str

operatorKind
SPL primitive operator type for this operator.

Type str

indexWithinJob
Index of this operator within the job.

Type int

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> operators = instances[0].get_operators()
>>> print (operators[0].resourceType)
operator
```

get_host()

Get resource this operator is currently executing in. If the operator is running on an externally managed resource None is returned.

Returns Resource this operator is running on.

Return type *Host*

New in version 1.9.

get_input_ports()
Get list of input ports for this operator.

Returns Input ports for this operator.

Return type list(*OperatorInputPort*)

New in version 1.9.

get_job()
Get the Streams job that owns this operator.

Returns Streams Job owning this operator.

Return type *Job*

get_metrics(*name=None*)
Get metrics for this operator.

Parameters **name** (*str, optional*) – Only return metrics matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all metrics for this operator are returned.

Returns List of matching metrics.

Return type list(*Metric*)

Retrieving a list of metrics whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> operator = instances[0].get_operators()[0]
>>> metrics = op.get_metrics(name='*temperatureSensor*')
```

get_output_ports()
Get list of output ports for this operator.

Returns Output ports for this operator.

Return type list(*OperatorOutputPort*)

New in version 1.9.

get_pe()
Get the Streams processing element this operator is executing in.

Returns Processing element for this operator.

Return type *PE*

New in version 1.9.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**PEConnection**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Stream connection between two PEs.

id
PE connection ID.

Type str

resourceType
Identifies the REST resource type, which is *peConnection*.

Type str

required
Indicates whether this connection is required.

Type bool

status
Status of this connection. Some possible values include *connected*, *disconnected*, and *unknown*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> peconnections = instances.get_pe_connections()
>>> print(peconnections[0].resourceType)
peConnection
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**PE**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

Processing element (PE) within a job. A processing element hosts one or more operators within a single job.

id
PE ID.

Type str

resourceType
Identifies the REST resource type, which is *pe*.

Type str

health
Health indicator for this PE. Some possible values include *healthy*, *partiallyHealthy*, *partiallyUnhealthy*, *unhealthy*, and *unknown*.

Type str

indexWithinJob
Index of the PE within the job.

Type int

launchCount
Number of times this PE was started manually or automatically because of failures.

Type int

optionalConnections
Status of optional connections for this PE. Some possible values include *connected*, *disconnected*, *partiallyConnected*, and *unknown*.

Type str

pendingTracingLevel
Describes a pending change to the granularity of the trace information that is stored for this PE. Some possible values include *off*, *error*, *debug* and *trace*. The value is *None*, if no change is pending.

Type str

processId
Operating system process ID for this PE.

Type str

relocatable
Indicates whether this PE can be relocated to a different resource.

Type bool

requiredConnections
Status of the required connections for this PE. Some possible values include *connected*, *disconnected*, *partiallyConnected*, and *unknown*.

Type str

restartable
Indicates whether this PE can be restarted.

Type bool

status
Status of this PE.

Type str

statusReason
Additional information for the status of this PE.

Type str

tracingLevel
Granularity of the trace information. Some possible values include *off*, *error*, *debug* and *trace*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> pes = instances.get_pes()
>>> print(pes[0].resourceType)
pe
```

get_host()

Get resource this processing element is currently executing in. If the processing element is running on an externally managed resource `None` is returned.

Returns Resource this processing element is running on.

Return type *Host*

New in version 1.9.

get_job()
Get the Streams job that owns this PE.

Returns Streams Job owning this PE.

Return type *Job*

get_metrics(*name=None*)
Get metrics for this PE.

Parameters **name** (*str, optional*) – Only return metrics matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all metrics for this PE are returned.

Returns List of matching metrics.

Return type list(*Metric*)

Retrieving a list of metrics whose name contains the string "temperatureSensor" could be performed as followed .. rubric:: Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> pe = instances.get_pes()[0]
>>> metrics = pe.get_metrics(name='*temperatureSensor*')
```

New in version 1.9.

get_resource()
Get resource this processing element is currently executing in.

Returns Resource this processing element is running on.

Return type *Host*

New in version 1.13.13.

get_resource_allocation()
Get the *ResourceAllocation* element tance.

Returns Resource allocation used to access information about the resource where this PE is running.

Return type *ResourceAllocation*

New in version 1.9.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

retrieve_console_log(*filename=None*, *dir=None*)
Retrieves the application console log (standard out and error) files for this PE and saves them as a plain text file.

An existing file with the same name will be overwritten.

Parameters

- **filename** (*str*) – name of the created file. Defaults to *pe_<id>_<timestamp>.stdouterr* where *id* is the PE identifier and *timestamp* is the number of seconds since the Unix epoch, for example `pe_83_1511995995.trace`.
- **dir** (*str*) – a valid directory in which to save the file. Defaults to the current directory.

Returns the path to the created file, or None if retrieving a job's logs is not supported in the version of streams to which the job is submitted.

Return type str

New in version 1.9.

retrieve_trace(*filename=None*, *dir=None*)

Retrieves the application trace files for this PE and saves them as a plain text file.

An existing file with the same name will be overwritten.

Parameters

- **filename** (*str*) – name of the created file. Defaults to *pe_<id>_<timestamp>.trace* where *id* is the PE identifier and *timestamp* is the number of seconds since the Unix epoch, for example `pe_83_1511995995.trace`.
- **dir** (*str*) – a valid directory in which to save the file. Defaults to the current directory.

Returns the path to the created file, or None if retrieving a job's logs is not supported in the version of streams to which the job is submitted.

Return type str

New in version 1.9.

class streamsx.rest_primitives.**PublishedTopic**(*topic*, *schema*)

Bases: `object`

Metadata for a published topic.

topic

Published topic

Type str

schema

Schema of topic

Type str

class streamsx.rest_primitives.**ResourceAllocation**(*json_rep*, *rest_client*)

Bases: `streamsx.rest_primitives._ResourceElement`

A resource that is allocated to an IBM Streams instance.

resourceType

Identifies the REST resource type, which is *resourceAllocation*.

Type str

applicationResource

Indicates whether this resource is an application resource, which is used to run streams processing applications.

Type bool

schedulerStatus

Indicates whether this resource is schedulable for the instance.

Type str

status
Status of this resource for the instance.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> allocations = instances.get_resource_allocations()
>>> print(allocations[0].resourceType)
resourceAllocation
```

get_jobs(*name=None*)
Retrieves jobs running on this resource in its instance.

Parameters **name** (*str, optional*) – Only return jobs containing property **name** that matches *name*. *name* can be a regular expression. If *name* is not supplied, then all jobs are returned.

Returns A list of jobs matching the given *name*.

Return type list(*Job*)

Note: If `applicationResource` is *False* an empty list is returned.

New in version 1.9.

get_pes()
Get the list of *PE* running on this resource in its instance.

Returns List of PE running on this resource.

Return type list(*PE*)

Note: If `applicationResource` is *False* an empty list is returned.

New in version 1.9.

get_resource()
Get the *Resource* of the resource allocation.

Returns Resource for this allocation.

Return type *Resource*

New in version 1.9.

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**Resource**(*json_rep*, *rest_client*)
Bases: `streamsx.rest_primitives._ResourceElement`

A resource available to a IBM Streams domain.

id
Resource identifier.

Type str

displayName
Resource display name.

Type str

ipAddress
IP address.

Type str

status
Resource status.

Type str

tags
Tags assigned to resource.

Type list[str]

New in version 1.9.

get_metrics(*name=None*)
Get metrics for this resource.

Parameters **name** (*str, optional*) – Only return metrics matching *name*, where *name* can be a regular expression. If *name* is not supplied, then all metrics for this resource are returned.

Returns List of matching metrics.

Return type list(*Metric*)

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**ResourceTag**(*json_resource_tag*)
Bases: object

Resource tag defined in a Streams domain

definition_format_properties
Indicates whether the resource definition consists of one or more properties.

Type bool

description
Tag description.

Type str

name
Tag name.

Type str

properties_definition
Contains the properties of the resource definition. Only present if *definition_format_properties* is *True*.

Type list(str)

reserved
If *True*, this tag is defined by IBM Streams, and cannot be modified.

Type bool

class streamsx.rest_primitives.**RestResource**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

HTTP REST resource identifier.

name
Resource name.

Type str

resource
A string that identifies the URI for the resource.

Type str

Changed in version 1.9: Changed to *RestResource* from *Resource*.

get_resource()
Make a request against this REST resource. :returns: JSON response. :rtype: dict

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**StreamingAnalyticsService**(*rest_client*, *credentials*)
Bases: object

Streaming Analytics service running on IBM Cloud.

cancel_job(*job_id=None*, *job_name=None*)
Cancel a running job.

Parameters

- **job_id** (*str, optional*) – Identifier of job to be canceled.
- **job_name** (*str, optional*) – Name of job to be canceled.

Returns JSON response for the job cancel operation.

Return type dict

get_instance_status()
Get the status the instance for this Streaming Analytics service.

Returns JSON response for the instance status operation.

Return type dict

start_instance()
Start the instance for this Streaming Analytics service.

Returns JSON response for the instance start operation.

Return type dict

stop_instance()
Stop the instance for this Streaming Analytics service.

Returns JSON response for the instance start operation.

Return type dict

submit_job(*bundle*, *job_config=None*)
Submit a Streams Application Bundle (sab file) to this Streaming Analytics service.

Parameters

- **bundle** (*str*) – path to a Streams application bundle (sab file) containing the application to be submitted
- **job_config** (JobConfig) – a job configuration overlay

Returns

JSON response from service containing 'name' field with unique job name assigned to submitted job, or, 'error_status' and 'description' fields if submission was unsuccessful.

Return type dict

class streamsx.rest_primitives.**Toolkit**(*json_rep*, *rest_client*)

Bases: streamsx.rest_primitives._ResourceElement

IBM Streams toolkit.

id
Unique ID for this instance.

Type str

resourceType
Identifies the REST resource type, which is *toolkit*.

Type str

name
The name of the toolkit.

Type str

version
The version of the toolkit.

Type str

requiredProductVersion
The earliest version of Streams required by the toolkit.

Type str

path
The full path to the toolkit.

Type str

Example

```
>>> from streamsx.build import BuildService
>>> build_service = BuildService.of_endpoint()
>>> toolkits = build_service.get_toolkits()
>>> print (toolkits[0].resourceType)
toolkit
```

New in version 1.13.

class Dependency(*name*, *version*)

Bases: object

The name, and range of versions, of a toolkit required by another toolkit.

name
the name of the required toolkit

Type str

version
the range of versions required of the toolkit
Type str

property dependencies
Find all the dependencies for this toolkit.

Returns List of dependencies of this toolkit. If this toolkit does not have any dependencies, this will be an empty list.

Return type list(*Dependency*)

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**ViewItem**(*json_rep*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

A stream tuple in view.

collectionTime
Epoch time when this viewItem is collected from the stream.

Type long

data
Content of this viewItem.

Type dict

resourceType
Identifies the REST resource type, which is *viewItem*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> views = instances[0].get_views()
>>> viewitems = views[0].get_view_items()
>>> print (viewitems[0].resourceType)
viewItem
```

refresh()
Refresh the resource and update the attributes to reflect the latest status.

class streamsx.rest_primitives.**View**(*json_view*, *rest_client*)
Bases: streamsx.rest_primitives._ResourceElement

View on a stream.

id
An unique identifier for the view.

Type str

name
View name.

Type str

description
Description of the view.

Type str

resourceType
Identifies the REST resource type, which is *view*.

Type str

activateOption
Indicate when the view starts buffering data.

Type str

maximumTupleRate
Maximum Number of tuples at which the view collects per second.

Type int

logicalOperatorName
The logical name of the operator that contains the output port on which the view is created.

Type str

bufferCapacitySeconds
Buffer size measured in seconds.

Type int

bufferCapacityTuples
Buffer size measured in number of tuples.

Type int

bufferCapacityUnits
Indicates whether the buffer capacity for the view is determined by *seconds*, *tuples* or *unknown*.

Type str

Example

```
>>> from streamsx import rest
>>> sc = rest.StreamingAnalyticsConnection()
>>> instances = sc.get_instances()
>>> views = instances[0].get_views()
>>> print (views[0].resourceType)
view
```

display(*duration=None*, *period=2*)
Display a view within a Jupyter or IPython notebook.

Provides an easy mechanism to visualize data on a stream using a view.

Tuples are fetched from the view and displayed in a table within the notebook cell using a `pandas.DataFrame`. The table is continually updated with the latest tuples from the view.

This method calls *`start_data_fetch()`* and will call *`stop_data_fetch()`* when completed if *duration* is set.

Parameters

- **duration** (*float*) – Number of seconds to fetch and display tuples. If `None` then the display will be updated until *stop_data_fetch()* is called.
- **period** (*float*) – Maximum update period.

Note: A view is a sampling of data on a stream so tuples that are on the stream may not appear in the view.

Note: Python modules *ipywidgets* and *pandas* must be installed in the notebook environment.

Warning: Behavior when called outside a notebook is undefined.

New in version 1.12.

fetch_tuples(*max_tuples=20*, *timeout=None*)
Fetch a number of tuples from this view.

Fetching of data must have been started with *start_data_fetch()* before calling this method.

If `timeout` is `None` then the returned list will contain `max_tuples` tuples. Otherwise if the timeout is reached the list may contain less than `max_tuples` tuples.

Parameters

- **max_tuples** (*int*) – Maximum number of tuples to fetch.
- **timeout** (*float*) – Maximum time to wait for `max_tuples` tuples.

Returns List of fetched tuples.

Return type list

New in version 1.12.

get_domain()
Get the Streams domain for the instance that owns this view.

Returns Streams domain for the instance owning this view.

Return type *Domain*

get_instance()
Get the Streams instance that owns this view.

Returns Streams instance owning this view.

Return type *Instance*

get_job()
Get the Streams job that owns this view.

Returns Streams Job owning this view.

Return type *Job*

get_view_items()
Get a list of *ViewItem* elements associated with this view.

Returns List of ViewItem(s) associated with this view.

Return type list(*ViewItem*)

refresh()
Refresh the resource and update the attributes to reflect the latest status.

start_data_fetch()
Starts a thread that fetches data from the Streams view server.

Each item in the returned *Queue* represents a single tuple on the stream the view is attached to.

Returns Queue containing view data.

Return type queue.Queue

Note: This is a queue of the tuples coverted to Python objects, it is not a queue of *ViewItem* objects.

---

stop_data_fetch()
Stops the thread that fetches data from the Streams view server.

CHAPTER

FOUR

SCRIPTS

The *streamsx* package provides a number of command line scripts.

4.1 spl-python-extract

4.1.1 Overview

Extracts SPL Python primitive operators from decorated Python classes and functions.

Executing this script against an SPL toolkit creates the SPL primitive operator meta-data required by the SPL compiler (*sc*).

4.1.2 Usage

```
spl-python-extract [-h] -i DIRECTORY [--make-toolkit] [-v]

Extract SPL operators from decorated Python classes and functions.

optional arguments:
  -h, --help            show this help message and exit
  -i DIRECTORY, --directory DIRECTORY
                        Toolkit directory
  --make-toolkit        Index toolkit using spl-make-toolkit
  -v, --verbose         Print more diagnostics
```

4.1.3 SPL Python primitive operators

SPL operators that call a Python function or callable class are created by decorators provided by the *streamsx* package.

To create SPL operators from Python functions or classes one or more Python modules are created in the `opt/python/streams` directory of an SPL toolkit.

`spl-python-extract` is a Python script that creates SPL operators from Python functions and classes contained in modules under `opt/python/streams`.

The resulting operators embed the Python runtime to allow stream processing using Python.

Details on how to implement SPL Python primitive operators see *streamsx.spl.spl*.

4.2 streamsx-info

4.2.1 Overview

Information about streamsx package and environment.

Prints to standard out information about the *streamsx* package and environment variables used to support Python in IBM Streams and Streaming Analytics service.

A Python warning is issued if a mismatch is detected between the installed *streamsx* package and its modules. This is typically due to having a different version of the modules accessible through the environment variable `PYTHONPATH`.

Warning: When using the *streamsx* package ensure that the environment variable `PYTHONPATH` does **not** include a path ending with `com.ibm.streamsx.topology/opt/python/packages`. The IBM Streams environment configuration script `streamsprofile.sh` modifies or sets `PYTHONPATH` to include the Python support from the SPL topology toolkit shipped with the product. This was to support Python before the *streamsx* package was available. The recommendation is to unset `PYTHONPATH` or modify it not to include the path to the topology toolkit.

Output is subject to change in the order and information displayed. Intended as an ad-hoc tool to help diagnose issues with *streamsx*.

Script may also be run as Python module:

```
python -m streamsx.scripts.info
```

4.2.2 Usage

```
usage: streamsx-info [-h]

    Prints support information about streamsx package and environment.

optional arguments:
    -h, --help  show this help message and exit
```

4.3 streamsx-runner

4.3.1 Overview

Submits or builds a Streams application to the Streaming Analytics service.

The application to be submitted can be:

- A Python application defined through *Topology* using the `--topology` flag.
- An SPL application (main composite) using the `--main-composite` flag.
- A Streams application bundle (`sab` file) using the `--bundle` flag.

4.3.2 Streaming Analytics service

The Streaming Analytics service is defined by:

- Service name - `--service-name` defaulting to environment variable `STREAMING_ANALYTICS_SERVICE_NAME`. The service name must exist in the vcap services.
- Vcap services - Environment variable `VCAP_SERVICES` containing JSON representation of the service definitions or a file name containing the service definitions.

4.3.3 Job submission

Job submission occurs unless `--create-bundle` is set.

4.3.4 Bundle creation

When `-create-bundle` is specified with `-main-composite` or `--topology` then a Streams application bundle (sab file) is created.

If environment variable *STREAMS_INSTALL* is set the the build is local otherwise the build occurs in the IBM Cloud using the Streaming Analytics service.

When `STREAMS_INSTALL` is not set then *streamsx-runner* can be executed with no local Streams install.

When compiling an SPL application (`--main-composite`) then the path to the application toolkit containing the main composite must be listed with `--toolkits`.

Any other required local toolkits must be listed with with `--toolkits`.

4.3.5 Usage

```
streamsx-runner [-h] [--service-name SERVICE_NAME] | [--create-bundle]
              (--topology TOPOLOGY | --main-composite MAIN_COMPOSITE | --bundle BUNDLE)
              [--toolkits TOOLKITS [TOOLKITS ...]] [--job-name JOB_NAME]
              [--preload] [--trace {error,warn,info,debug,trace}]
              [--submission-parameters SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...
↪]]
              [--job-config-overlays file]

Execute a Streams application using a Streaming Analytics service.

optional arguments:
  -h, --help            show this help message and exit
  --service-name SERVICE_NAME
                        Submit to Streaming Analytics service
  --create-bundle       Create a bundle (sab file). No job submission occurs.
  --topology TOPOLOGY   Topology to call
  --main-composite MAIN_COMPOSITE
                        SPL main composite (namespace::composite_name)
  --bundle BUNDLE       Streams application bundle (sab file) to submit to
                        service

Build options:
  Application build options
```

(continues on next page)

(continued from previous page)

```
  --toolkits TOOLKITS [TOOLKITS ...]
                        SPL toolkit path containing the main composite and any
                        other required SPL toolkit paths.

Job options:
  Job configuration options

  --job-name JOB_NAME   Job name
  --preload             Preload job onto all resources in the instance
  --trace {error,warn,info,debug,trace}
                        Application trace level
  --submission-parameters SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...], -p␣
↪SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...]
                        Submission parameters as name=value pairs
  --job-config-overlays file
                        Path to file containing job configuration overlays
                        JSON. Overrides any job configuration set by the
                        application.
```

4.3.6 Submitting to Streaming Analytics service

An application is submitted to a Streaming Analytics service using `--service-name SERVICE_NAME`. The named service must exist in the VCAP services definition pointed to by the `VCAP_SERVICES` environment variable.

The application is submitted as source (except `--bundle`) and compiled into a Streams application bundle (`sab` file) using the build service before being submitted as a running job to the service instance.

See also:

Accessing a service

Python applications

To submit a Python application a Python function must be defined that returns the application (and optionally its configuration) to be submitted. The fully qualified name of this function is specified using the `--topology` flag.

For example, an application can be submitted as:

```
streamsx-runner --service-name Streaming-Analytics-xd \
    --topology com.example.apps.sensor_ingester
```

The function returns one of:

- a *Topology* instance defining the application
- **a `tuple` containing two values, in order:**
  - a *Topology* instance defining the application
  - **job configuration, one of:**
    - *JobConfig* instance
    - `dict` corresponding to the configuration object passed into *submit()*

For example the above function might be defined as:

```
def _create_sensor_ingester_app():
   topo = Topology('SensorIngesterApp')

   # Application declaration omitted
   ...

   return topo

def sensor_ingester():
    return (_create_sensor_ingester_app(), JobConfig(job_name='SensorIngester'))
```

Thus when this application is submitted using the *sensor_ingester* function it is always submitted with the same job name *SensorIngester*.

The function must be accessible from the current Python path (typically through environment variable `PYTHONPATH`).

SPL applications

The main composite that defines the application is specified using the `-main-composite` flag specifing the fully namespace qualified name.

Any required local SPL toolkits, *including the one containing the main composite*, must be indivdually specified by location to the `--toolkits` flag. Any SPL toolkit that is present on the IBM Cloud service need not be included.

For example, an application that uses the Slack toolkit might be submitted as:

```
streamsx-runner --service-name Streaming-Analytics-xd \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack
```

where `$HOME/app/alerters` is the location of the SPL application toolkit containing the `com.example.alert::SlackAlerter` main composite.

Warning: The main composite name must be namespace qualified. Use of the default namespace for a main composite is not recommended as it increases the chance of a name clash with another SPL toolkit.

Streams application bundles

A Streams application bundle is submitted to a service instance using `--bundle`. The argument to `--bundle` is a locally accessible file that will be uploaded to the service.

The bundle must have been created on using an IBM Streams install whose architecture and OS version matches the service instance. Currently this is `x86_64` and RedHat/CentOS 6 or 7 depending on the service instance.

The `--toolkits` flag must not be specified when submitting a bundle.

Job options

Job options, such as `--job-name`, configure the running job.

For `--topology` job options set as arguments to `streamsx-runner` override any configuration returned from the function defining the application.

4.3.7 Creating Streams application bundles

`--create-bundle` uses a local IBM Streams install to attempt to mimic the build that would occur with `-topology` or `--main-composite`. Differences between the local environment and the IBM Cloud Streaming Analytics build environment may cause build failures in one and not the other.

This can be used as a mechanism to perform a local test build before using the service, or as a valid mechanism to create bundles for later upload with `--bundle`.

For example simply changing the `--service-name name` to `--create-bundle` perfoms a local build of the same application:

```
# Submit to an Streaming Analytics service
streamsx-runner --service-name Streaming-Analytics-xd \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack

# Build the same application locally
streamsx-runner --create-bundle \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack
```

4.4 streamsx-sc

4.4.1 Overview

SPL compiler for IBM Streams running on IBM Cloud Pak for Data.

`streamsx-sc` replicates a sub-set of Streams 4.3 `sc` options.

`streamsx-sc` is supported for Streams 5.x (Cloud Pak for Data). A local install of Streams is **not** required, simply the installation of the *streamsx* package. All functionality is implemented through the Cloud Pak for Data and Streams build service REST apis.

Cloud Pak for Data configuration

Integrated configuration

The Streams instance (and its build service) and authentication are defined through environment variables:

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*.
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

Standalone configuration

The Streams build service and authentication are defined through environment variables:

- **STREAMS_BUILD_URL** - Streams build service URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

4.4.2 Usage

```
streamsx-sc [-h] --main-composite name [--spl-path SPL_PATH]
        [--optimized-code-generation] [--no-optimized-code-generation]
        [--prefer-facade-tuples] [--ld-flags LD_FLAGS]
        [--cxx-flags CXX_FLAGS] [--c++std C++STD]
        [--data-directory DATA_DIRECTORY]
        [--output-directory OUTPUT_DIRECTORY] [--disable-ssl-verify]
        [--static-link] [--standalone-application]
        [--set-relax-fusion-relocatability-restartability]
        [--checkpoint-directory path] [--profiling-sampling rate]
        [compile-time-args [compile-time-args ...]]
```

Options and arguments

compile-time-args: Pass named arguments each in the format *name=value* to the compiler. The name cannot contain the character = but otherwise is a free form string. It matches the name parameter that is specified in calls that are made to the compile-time argument access functions from within SPL code. The value can be any string. See Compile-time arguments .

-M,–main-composite: SPL Main composite

-t,–spl-path: Set the toolkit lookup paths. Separate multiple paths with `:`. Each path is a toolkit directory or a directory of toolkit directories. This path overrides the `STREAMS_SPLPATH` environment variable.

-a,–optimized-code-generation: Generate optimized code with less runtime error checking

—no-optimized-code-generation: Generate non-optimized code with more runtime error checking. Do not use with the –optimized-code- generation option.

-k,–prefer-facade-tuples: Generate the facade tuples when it is possible.

-w,–ld-flags: Pass the specified flags to ld while linking occurs.

-x,–cxx-flags: Pass the specified flags to the C++ compiler during the build.

–c++std: Specify the language level for the underlying C++ compiles.

–data-directory: Specifies the location of the data directory to use.

–output-directory: Specifies a directory where the application artifacts are placed.

–disable-ssl-verify: Disable SSL verification against the build service

Deprecated arguments Arguments supported by *sc* but deprecated. They have no affect on compilation.

-s,–static-link

-T,–standalone-application

-O,–set-relax-fusion-relocatability-restartability

-K,–checkpoint-directory

-S,–profiling-sampling

4.4.3 Toolkits

The application toolkit is defined as the working directory of *streamsx-sc*.

Local toolkits are found through the toolkit path set by *–spl-path* or environment variable `STREAMS_SPLPATH`. Local toolkits are included in the build code archive sent to the build service if:

- the toolkit is defined as a dependent of the application toolkit including recursive dependencies of required local toolkits.
- and a toolkit of a higher version within the required dependency range does not exist locally or remotely on the build service.

The toolkit path for the compilation on the build service includes:

- the application toolkit
- local tookits included in the build code archive
- all toolkits uploaded on the Streams build service
- all product toolkits on the Streams build service

The application toolkit and local toolkits included in the build archive are processed prior to the actual compilation by:

- having any Python SPL primitive operators extracted using `spl-python-extract`
- indexed using `spl-make-toolkit`

New in version 1.13.

4.5 streamsx-service

4.5.1 Overview

Control commands for a Streaming Analytics service.

4.5.2 Usage

```
streamsx-service [-h] [--service-name SERVICE_NAME] [--full-response]
              {start,status,stop} ...

Control commands for a Streaming Analytics service.

positional arguments:
  {start,status,stop}   Supported commands
    start               Start the service instance
    status              Get the service status.
    stop                Stop the instance for the service.

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  --service-name SERVICE_NAME
                        Streaming Analytics service name
  --full-response       Print the full JSON response.

service.py stop [-h] [--force]

optional arguments:
  -h, --help  show this help message and exit
  --force     Stop the service even if jobs are running.
```

4.5.3 Controlling a Streaming Analytics service

The Streaming Analytics service to control is defined using `--service-name SERVICE_NAME`. If not provided then the service name is defined by the environment variable `STREAMING_ANALYTICS_SERVICE_NAME`.

The named service must exist in the VCAP services definition pointed to by the `VCAP_SERVICES` environment variable.

The response from making the control request is printed to standard out in JSON format. By default a minimal response is printed including the status of the service and the job count. The complete response from the service REST API is printed if the option `--full-response` is given.

4.6 streamsx-streamtool

4.6.1 Overview

Command line interface for IBM Streams running on IBM Cloud Pak for Data.

`streamsx-streamtool` replicates a sub-set of Streams `streamtool` commands focusing on supporting DevOps for streaming applications.

`streamsx-streamtool` is supported for Streams Cloud Pak for Data (5.x) instances A local install of Streams is **not** required, simply the installation of the *streamsx* package. All functionality is implemented through Cloud Pak for Data and Streams REST apis.

Cloud Pak for Data configuration

The Streams instance and authentication are defined through environment variables, the details depend on if the Streams instance is running in integrated or standalone configuration.

Integrated configuration

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*.
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name. Overridden by the `--User` option.
- **STREAMS_PASSWORD** - Password for authentication.

Standalone configuration

- **STREAMS_REST_URL** - Streams SWS service (REST API) URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_BUILD_URL** - Streams build service (REST API) URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*. Required for *lstoolkit* and *rmtoolkit*.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

4.6.2 Usage

```
streamsx-streamtool submitjob [-h] [--jobConfig file-name]
        [--jobname job-name] [--jobgroup jobgroup-name]
        [--outfile file-name] [--P parameter-name]
        [--User user]
        sab-pathname

streamsx-streamtool canceljob [-h] [--force] [--collectlogs]
        [--jobs job-id | --jobnames job-names | --file file-name]
        [--User user]
        [jobid [jobid ...]]

streamsx-streamtool lsjobs [-h] [--jobs job-id] [--users user]
        [--jobnames job-names] [--fmt format-spec]
        [--xheaders] [--long] [--showtimestamp]
        [--User user]

streamsx-streamtool lsappconfig [-h] [--fmt format-spec] [--User user]

streamsx-streamtool mkappconfig [-h] [--property name=value]
        [--propfile property-file]
        [--description description] [--User user]
        config-name

streamsx-streamtool rmappconfig [-h] [--noprompt] [--User user] config-name

streamsx-streamtool chappconfig [-h] [--property name=value]
        [--description description] [--User user]
        config-name

streamsx-streamtool getappconfig [-h] [--User user] config-name

streamsx-streamtool lstoolkit [-h]
        (--all | --id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]

streamsx-streamtool rmtoolkit [-h]
        (--toolkitid toolkit-id | --toolkitname toolkit-name | --toolkitregex toolkit-
↪regex)
        [--User user]

streamsx-streamtool uploadtoolkit [-h] --path toolkit-path [--User user]
```

(continues on next page)

(continued from previous page)

```
streamsx-streamtool updateoperators [-h] [--jobname job-name]
        [--jobConfig file-name]
        [--parallelRegionWidth parallelRegionName=width]
        [--force] [--User user]
        [jobid]
```

4.6.3 submitjob

The streamtool submitjob command previews or submits one job.

Description:

A submitted job runs an application that is defined by an application bundle. Application bundles are created by the Stream Processing Language (SPL) compiler. A job consists of one or more processing elements (PEs). The PEs are placed on one or more of the application resources for the instance. The submission fails if the PE placement constraints can't be met.

Jobs remain in the system until they are canceled or the instance is stopped.

```
streamsx-streamtool submitjob [-h] [--jobConfig file-name]
        [--jobname job-name] [--jobgroup jobgroup-name]
        [--outfile file-name] [--P parameter-name]
        [--User user]
        sab-pathname
```

Options and arguments

sab-pathname Specifies the path name for the application bundle file. If you do not specify an absolute path, the command seeks the file in the directory where you ran the command. Alternatively, you can specify the path name for the application description language (ADL) file if the application bundle file exists in the same directory.

-g,–jobConfig: Specifies the name of an external file that defines a job configuration overlay. You can use a job configuration overlay to set the job configuration when the job is submitted or to change the configuration of a running job.

-P,–P: Specifies a submission-time parameter and value for the job. You can specify this option multiple times in the command.

-J,–jobgroup: Specifies the job group. If you do not specify this option, the command uses the following job group: default.

—jobname: Specifies the name of the job.

—outfile: Specifies the path and file name of the output file in which the command writes the list of submitted job IDs. The path can be an absolute or relative path. If you do not specify a path, the file is created in the directory where you run the command.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.4 canceljob

The streamtool canceljob command cancels one or more jobs.

This command stops the processing elements (PEs) for the job and removes knowledge of the jobs and their PEs from the instance. The log files for the processing elements are scheduled for removal.

If you specify to collect the PE logs before they are removed, the operation can time out waiting for the termination of PEs. If such a timeout occurs, the operation fails and the jobs or PEs are still in the system. The canceljob command can be run again later to cancel them.

You can use the –force option to ignore a PE termination timeout and force the job to cancel.

```
streamsx-streamtool canceljob [-h] [--force] [--collectlogs]
        [--jobs job-id | --jobnames job-names | --file file-name]
        [--User user]
        [jobid [jobid ...]]
```

Options and arguments

jobid Specifies a list of job IDs.

-f,–file: Specifies the file that contains a list of job IDs, one per line.

-j,–jobs: Specifies a list of job IDs, which are delimited by commas.

—jobnames: Specifies a list of job names, which are delimited by commas.

—collectlogs: Specifies to collect the log and trace files for each processing element that is associated with the job.

—force: Specifies to quickly cancel a job and remove the job from the Streams data table.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.5 lsjobs

The streamtool lsjobs command lists the jobs in the instance.

The streamtool lsjobs command provides a health summary for each job. The health summary is an aggregation of the PE health summaries for the job. If all of the PEs for a job are reported as healthy, the job is reported as healthy. Otherwise, the job is reported as not healthy. Use the streamtool lspes command to determine the health of PEs.

The command also reports the status of each job. For more information about job states, see the IBM Streams product documentation.

The date and time that the job was submitted are presented in local time with the iso8601 format: yyyy-mm-ddThh:mm:ss+/-hhmm, where the final hhmm values are the local offset from UTC. For example: 2010-03-16T13:41:53-0500.

When job selection options are specified, selected jobs must meet all of the selection criteria. After a cancel request for a job is processed, this command no longer reports the job or its processing elements (PEs).

```
streamsx-streamtool lsjobs [-h] [--jobs job-id] [--users user]
        [--jobnames job-names] [--fmt format-spec]
        [--xheaders] [--long] [--showtimestamp]
        [--User user]
```

Options and arguments

-j,–jobs: Specifies a list of job IDs, which are delimited by commas.

—jobnames: Specifies a list of job names, which are delimited by commas.

-u,–users: Specifies to select from this list of user IDs, which are delimited by commas.

—xheaders: Specifies to exclude headings from the report.

-l,–long: Reports launch count, full host names, and all of the operator instance names for the PEs.

—fmt: Specifies the presentation format. The command supports the following values:

- %Mf: Multiline record format. One line per field.
- %Nf: Name prefixed field table format. One line per job.
- %Tf: Standard table format, which is the default. One line per job.

—showtimestamp: Specifies to show a time stamp in the output to indicate when the command was run.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.6 lsappconfig

The streamtool lsappconfig command lists the available configurations that enable connections to an external application.

Retrieve a list of configurations for making a connection to an external application.

```
streamsx-streamtool lsappconfig [-h] [--fmt format-spec] [--User user]
```

Options and arguments

—fmt: Specifies the presentation format. The command supports the following values:

- %Mf: Multiline record format. One line per field.
- %Nf: Name prefixed field table format. One line per cfgname.
- %Tf: Standard table format, which is the default. One line per cfgname.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.7 mkappconfig

The streamtool mkappconfig command creates a configuration that enables connection to an external application.

Operators can retrieve the configuration information to make a connection to an external application, such as an Internet Of Things application. The properties include items that the application needs at runtime, like connection information and credentials.

Use this command to register properties or a properties file. Create the property file using a name=value syntax.

```
streamsx-streamtool mkappconfig [-h] [--property name=value]
        [--propfile property-file]
        [--description description] [--User user]
        config-name
```

Options and arguments

config-name: Name of the app config

—description: Specifies a description for the application configuration. The description can be 1024 characters in length. If the description contains blank characters, it must be enclosed in single or double quotation marks. Quotation marks within the description must be preceded by a backslash ().

—property: Specifies a property name and value pair to add to or change in the configuration. This option can be specified multiple times and has an additive effect.

—propfile: Specifies the path to a file that contains a list of application configuration properties for connecting to an external application. The properties are listed as name=value pairs, each on a separate line. Use this option as a way to include multiple configuration properties when you create an application configuration. Options that you specify at the command line override values that are specified in this property file.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.8 rmappconfig

The streamtool rmappconfig command removes a configuration that enables connection to an external application.

This command removes a configuration that is used for making a connection to an external application.

```
streamsx-streamtool rmappconfig [-h] [--noprompt] [--User user] config-name
```

Options and arguments

config-name: Name of the app config

—noprompt: Specifies to suppress confirmation prompts.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.9 chappconfig

The streamtool chappconfig command updates a configuration that enables connection to an external application.

Use this command to change the configuration properties that are used to make a connection to an external application, such as an Internet Of Things application. You can change the values of properties or add new properties.

```
streamsx-streamtool chappconfig [-h] [--property name=value]
        [--description description] [--User user]
        config-name
```

Options and arguments

config-name: Name of the app config

—description: Specifies a description for the application configuration. The description can be 1024 characters in length. If the description contains blank characters, it must be enclosed in single or double quotation marks. Quotation marks within the description must be preceded by a backslash ().

—property: Specifies a property name and value pair to add to or change in the configuration. This option can be specified multiple times and has an additive effect.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.10 getappconfig

The streamtool getappconfig command displays the properties of a configuration that enables connection to an external application.

This command retrieves the properties and values of a specific configuration for connecting to an external application.

```
streamsx-streamtool getappconfig [-h] [--User user] config-name
```

Options and arguments

config-name: Name of the app config

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.11 lstoolkit

List toolkits from a build service.

```
streamsx-streamtool lstoolkit [-h]
        (--all | --id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]
```

Options and arguments

-a,–all: List all toolkits

-i,–id: List a specific toolkit given its toolkit id

-n,–name: List all toolkits with this name

-r,–regex: List all toolkits where the name matches the given regex pattern

4.6.12 rmtoolkit

Remove toolkits from a build service.

```
streamsx-streamtool rmtoolkit [-h]
        (--id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]
```

Options and arguments

-i,–id: Specifies the id of the toolkit to delete

-n,–name: Remove all toolkits with this name

-r,–regex: Remove all toolkits where the name matches the given regex pattern

4.6.13 uploadtoolkit

Upload a toolkit to a build service.

```
streamsx-streamtool uploadtoolkit [-h] --path toolkit-path [--User user]
```

Options and arguments

-p,–path: Specifies the path of the indexed toolkit to upload

New in version 1.13.

4.6.14 updateoperators

Adjust a job configuration while the job is running in order to improve the job performance

```
streamsx-streamtool updateoperators [-h] [--jobname job-name]
        [--jobConfig file-name]
        [--parallelRegionWidth parallelRegionName=width]
        [--force] [--User user]
        [jobid]
```

Options and arguments

jobid: Specifies a job ID

—jobname: Specifies the name of the job

-g,–jobConfig: Specifies the name of an external file that defines a job configuration overlay. You can use a job configuration overlay to set the job configuration when the job is submitted or to change the configuration of a running job.

—parallelRegionWidth: Specifies a parallel region name and its width.

—force: Specifies whether to automatically stop the PEs that need to be stopped.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

CHAPTER

FIVE

ENVIRONMENTS

5.1 IBM Streaming Analytics service

5.1.1 Overview

IBM® Streaming Analytics for IBM Cloud is powered by IBM® Streams, an advanced analytic platform that you can use to ingest, analyze, and correlate information as it arrives from different types of data sources in real time. When you create an instance of the Streaming Analytics service, you get your own instance of IBM® Streams running in IBM® Cloud, ready to run your IBM® Streams applications.

See also:

Overview at ibm.com

IBM Cloud catalog

Streaming Analytics service documentation

5.1.2 Package support

This *streamsx* package supports :

- Developing streaming applications in Python that can be submitted to a Streaming Analytics service. See *streamsx.topology.topology*, *STREAMING_ANALYTICS_SERVICE*.
- Submitting streaming applications written in Python or SPL to a Streaming Anlaytics service. See *Python applications*, *SPL applications*.
- Submitting a pre-compiled Streams application bundle (`sab` file) Python or SPL to a Streaming Anlaytics service. See *Streams application bundles*.
- Python bindings to the IBM Streams REST API and the Streaming Analytics REST API. See *streamsx.rest*

5.1.3 Accessing a service

In order to use a Streaming Analytics service you must have access to credentials for the service. There are two mechanisms used by this package, VCAP services and direct use of Streaming Analytics credentials.

VCAP services

This is the format used by Cloud Foundry for bindable services. The service key for Streaming Analytics service is `streaming-analytics`, the value of that key in the VCAP services is a list of accessible services, each service represented by a separate object.

Each streaming analytics object must have these keys:

- `name` identifying the name of the service.
- `credentials` identifying the connection credentials for the service.

Example VCAP services containing two Streaming Analytics services *sa-test* and *sa-prod* (with the specific connection details elided):

```
{
"streaming-analytics": [
{
  "name": "sa-test",
  "credentials":
  {
     "apikey": "...",
     "iam_apikey_description": "Auto generated apikey during resource-key operation␣
↪for Instance - ...",
     "iam_apikey_name": "auto-generated-apikey-...",
     "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
     "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity ...",
      "v2_rest_url": "https://streams-app-service.ng.bluemix.net/v2/streaming_
↪analytics/..."
  }
},
{
  "name": "sa-prod",
  "credentials":
  {
     "apikey": "...",
     "iam_apikey_description": "Auto generated apikey during resource-key operation␣
↪for Instance - ...",
     "iam_apikey_name": "auto-generated-apikey-...",
     "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
     "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity ...",
      "v2_rest_url": "https://streams-app-service.ng.bluemix.net/v2/streaming_
↪analytics/..."
  }
}
]
}
```

Note: The specific keys in the credentials may differ depending on the service plan.

See also:

https://docs.run.pivotal.io/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES

Cloud Foundry applications

When a Streaming Analytics service is bound to a Cloud Foundry Python application the environment variable `VCAP_SERVICES` is automatically defined and contains a string representation of the JSON VCAP services information.

Client applications

Client applications are ones that run outside of the IBM Cloud, for example on a local laptop, or applications that are not bound to a service.

Client applications running must define a valid VCAP services in its JSON format as either:

- In the environment variable `VCAP_SERVICES` containing a string representation of the JSON VCAP services information.
- In a file containing a string representation of the JSON VCAP services information and have the file's absolute path in either:
  - the environment variable `VCAP_SERVICES`
  - the configuration property *`VCAP_SERVICES`* when submitting an application using *`submit()`* with context type *`STREAMING_ANALYTICS_SERVICE`*. This overrides the environment variable *VCAP_SERVICES*.

The contents of the file must be manually created, the credentials for the `credentials` key are obtained from the Streaming Analytics manage console. Select the *Service Credentials* page and then copy the required credentials. You may need to first create credentials. You can an copy the credentials by taking the *View credentials* action and then clicking the *copy to clipboard* icon on the right hand side.

Warning: The credential information in VCAP services is in plain text. Ensure that the any file containing the information or setting the environment variable has suitable permissions set. For example only readable by the intended user.

Selecting the service

The Streaming Analyitcs service to use is specifed by its name, the required service much exist in the VCAP service information using the `name` key.

The name of the service to use is set by:

- the environment variable `STREAMING_ANALYTICS_SERVICE_NAME`.
- the configuration property *`SERVICE_NAME`* when submitting an application using *`submit()`* with context type *`STREAMING_ANALYTICS_SERVICE`*. This overrides the environment variable *STREAMING_ANALYTICS_SERVICE_NAME*.
- the `--service-name` option to `streamsx-runner`.

Service definition

The Streaming Analytics service to use may be specified solely using its credentials. The credentials are specified:

- with the configuration property *SERVICE_DEFINITION* when submitting an application using *submit()* with context type *STREAMING_ANALYTICS_SERVICE*.
- when using *streamsx.rest.StreamingAnalyticsConnection.of_definition()* to create a REST connection.

Credentials obtained from the Streaming Analytics manage console. Select the *Service Credentials* page and then copy the required credentials. You may need to first create credentials. You can an copy the credentials by taking the *View credentials* action and then clicking the *copy to clipboard* icon on the right hand side.

5.2 IBM Streams Python setup

5.2.1 Developer setup

Developers install the *streamsx* package Python Package Index (PyPI) to use this functionality:

```
pip install streamsx
```

If already installed upgrade to the latest version is recommended:

```
pip install --upgrade streamsx
```

A local install of IBM Streams is **not** required when:

- Using the Streams and Streaming Analytics REST bindings *streamsx.rest*.
- Devloping and submitting streaming applications using *streamsx.topology.topology* to Cloud Pak for Data or Streaming Analytics service on IBM Cloud.
  - The environment variable `JAVA_HOME` must reference a Java 1.8 JRE or JDK/SDK.

A local install of IBM Streams is required when:

- Developing and submitting streaming applications using *streamsx.topology.topology* to IBM Streams 4.2, 4.3 distributed or standalone contexts.
  - If set the environment variable `JAVA_HOME` must reference a Java 1.8 JRE or JDK/SDK, otherwise the Java install from `$STREAMS_INSTALL/java` is used.
- Creating SPL toolkits with Python primitive operators using *streamsx.spl.spl* decorators for use with 4.2, 4.3 distributed or standalone applications.

Warning: When using the *streamsx* package ensure that the environment variable `PYTHONPATH` does **not** include a path ending with `com.ibm.streamsx.topology/opt/python/packages`. The IBM Streams environment configuration script `streamsprofile.sh` modifies or sets `PYTHONPATH` to include the Python support from the SPL topology toolkit shipped with the product. This was to support Python before the *streamsx* package was available. The recommendation is to unset `PYTHONPATH` or modify it not to include the path to the topology toolkit.

Note: The *streamsx* package is self-contained and does not depend on any SPL topology toolkit (`com.ibm.streamsx.topology`) installed under `$STREAMS_INSTALL/toolkits` or on the SPL compiler's (`sc`) toolkit

path. This is true at SPL compilation time and runtime.

5.2.2 Streaming Analytics service

The service instance has Anaconda installed with Python 3.6 as the runtime environment and has `PYTHONHOME` Streams application environment variable pre-configured.

Any streaming applications using Python must use Python 3.6 when submitted to the service instance. The *streamsx* package must be installed locally and applications are submitted to the *STREAMING_ANALYTICS_SERVICE* context.

5.2.3 IBM Cloud Pak for Data

An IBM Streams service instance within Cloud Pak for Data has Anaconda installed with Python 3.6 as the runtime environment and has `PYTHONHOME` Streams application environment variable pre-configured.

Any streaming applications using Python must use Python 3.6 when submitted to the service instance.

Streaming applications can be submitted through Jupyter notebooks running in Cloud Pak for Data projects. The *streamsx* package is preinstalled and applications are submitted to the *DISTRIBUTED* context.

Streaming applications can be submitted externally to the OpenShift cluster containing Cloud Pak for Data. The *streamsx* package must be installed locally and applications are submitted to the *DISTRIBUTED* context. The specific environment variables depend on if the Streams instance is in a integrated or standalone configuration. See *DISTRIBUTED* for details.

5.2.4 IBM Streams 4.2, 4.3

For a distributed cluster running Streams Python 3.7, 3.6 or 3.5 may be used.

Anaconda or Miniconda distributions may be used as the Python runtime, these have the advantage of being pre-built and including a number of standard packages. Ananconda installs may be downloaded at: https://www.continuum.io/downloads .

If building Python from source then it must be built to support embedding of the runtime with shared libraries (`--enable-shared` option to *configure*).

Distributed

For distributed the Streams application environment variable `PYTHONHOME` must be set to the Python install path.

This is set using *streamtool* as:

```
streamtool setproperty --application-ev PYTHONHOME=path_to_python_install
```

The application environment variable may also be set using the Streams console. The *Instance Management* view has an *Application Environment Variables* section. Expanding the details for that section allows modification of the set of environment variables available to Streams applications.

The Python install path must be accessible on every application resource that will execute Python code within a Streams application.

Note: The Python version used to declare and submit the application must compatible with the setting of `PYTHONHOME` in the instance. For example, if `PYTHONHOME` Streams application instance variable points to a Python 3.6 install, then Python 3.5 or 3.6 can be used to declare and submit the application.

Standalone

The environment `PYTHONHOME` must be set to the Python install path.

5.2.5 Bundle Python version compatibility

As of 1.13 Streams application bundles (sab files) invoking Python are binary compatible with a range of Python releases when using Python 3.

The minimum verson supported is the version of Python used during bundle creation.

The maximum version supported is the highest version of Python with a proposed release schedule.

For example if a sab is built with Python 3.6 then it can be submitted to a Streams instance using 3.6 or higher, up to & including 3.9 which is the highest Python release with a proposed release schedule as of 1.13.

Note: Compatability across Python releases is dependent on Python's Stable Application Binary Inteface.

5.3 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

streamsx.build, 111

c

streamsx.topology.composite, 59
streamsx.topology.context, 37

e

streamsx.ec, 71

o

streamsx.spl.op, 77

r

streamsx.rest, 113
streamsx.rest_primitives, 118

s

streamsx.spl.spl, 95
streamsx.topology.schema, 48
streamsx.topology.state, 56

t

streamsx.spl.toolkit, 93
streamsx.spl.types, 89
streamsx.topology, 3
streamsx.topology.tester, 62
streamsx.topology.tester_runtime, 71
streamsx.topology.topology, 6

INDEX

A

activateOption (*streamsx.rest_primitives.View attribute*), 148
ActiveService (*class in streamsx.rest_primitives*), 118
ActiveVersion (*class in streamsx.rest_primitives*), 119
add() (*streamsx.topology.context.JobConfig method*), 44
add_condition() (*streamsx.topology.tester.Tester method*), 63
add_file_dependency() (*streamsx.topology.topology.Topology method*), 12
add_pip_package() (*streamsx.topology.topology.Topology method*), 13
add_toolkit() (*in module streamsx.spl.toolkit*), 93
add_toolkit_dependency() (*in module streamsx.spl.toolkit*), 93
aggregate() (*streamsx.topology.topology.Window method*), 34
aliased_as() (*streamsx.topology.topology.Stream method*), 19
all_ports_ready() (*streamsx.spl.spl.PrimitiveOperator method*), 107
ApplicationBundle (*class in streamsx.rest_primitives*), 120
ApplicationConfiguration (*class in streamsx.rest_primitives*), 120
applicationName (*streamsx.rest_primitives.Job attribute*), 131
applicationResource (*streamsx.rest_primitives.ResourceAllocation attribute*), 142
architecture (*streamsx.rest_primitives.ActiveVersion attribute*), 119
architecture (*streamsx.rest_primitives.Installation attribute*), 124
as_dict() (*streamsx.topology.schema.StreamSchema method*), 53
as_json() (*streamsx.topology.topology.Stream method*), 19
as_overlays() (*streamsx.topology.context.JobConfig method*), 44
as_string() (*streamsx.topology.topology.Stream method*), 19
as_tuple() (*streamsx.topology.schema.StreamSchema method*), 54
attribute() (*streamsx.spl.op.Invoke method*), 80
attribute() (*streamsx.spl.op.Map method*), 84
attribute() (*streamsx.spl.op.Sink method*), 86
attribute() (*streamsx.spl.op.Source method*), 82
autonomous() (*streamsx.topology.topology.Stream method*), 20

B

batch() (*streamsx.topology.topology.Stream method*), 20
Binary (*streamsx.topology.schema.CommonSchema attribute*), 55
BROADCAST (*streamsx.topology.topology.Routing attribute*), 11
bufferCapacitySeconds (*streamsx.rest_primitives.View attribute*), 148
bufferCapacityTuples (*streamsx.rest_primitives.View attribute*), 148
bufferCapacityUnits (*streamsx.rest_primitives.View attribute*), 148
Buffered (*streamsx.topology.topology.SubscribeConnection attribute*), 11
build() (*in module streamsx.topology.context*), 47
BUILD_ARCHIVE (*streamsx.topology.context.ContextTypes attribute*), 38
build_version (*streamsx.rest_primitives.ActiveVersion attribute*), 119
BuildService (*class in streamsx.build*), 111
buildVersion (*streamsx.rest_primitives.Installation attribute*), 124
BUNDLE (*streamsx.topology.context.ContextTypes*

attribute), 38

C

cancel() (*streamsx.rest_primitives.Job method*), 131
cancel_job() (*streamsx.rest_primitives.StreamingAnalyticsService method*), 145
cancel_job_button() (*streamsx.topology.context.SubmissionResult method*), 46
category() (*streamsx.spl.op.Invoke property*), 81
category() (*streamsx.spl.op.Map property*), 85
category() (*streamsx.spl.op.Sink property*), 87
category() (*streamsx.spl.op.Source property*), 83
category() (*streamsx.topology.topology.Sink property*), 36
category() (*streamsx.topology.topology.Stream property*), 21
channel() (*in module streamsx.ec*), 75
checkpoint_period() (*streamsx.topology.topology.Topology property*), 14
collectionTime (*streamsx.rest_primitives.ViewItem attribute*), 147
colocate() (*streamsx.spl.op.Invoke method*), 81
colocate() (*streamsx.spl.op.Map method*), 85
colocate() (*streamsx.spl.op.Sink method*), 87
colocate() (*streamsx.spl.op.Source method*), 83
colocate() (*streamsx.topology.topology.Sink method*), 36
colocate() (*streamsx.topology.topology.Stream method*), 21
comment() (*streamsx.topology.context.JobConfig property*), 45
CommonSchema (*class in streamsx.topology.schema*), 55
complete() (*streamsx.topology.topology.PendingStream method*), 33
Composite (*class in streamsx.topology.composite*), 59
Condition (*class in streamsx.topology.tester_runtime*), 71
ConfigParams (*class in streamsx.topology.context*), 42
ConsistentRegionConfig (*class in streamsx.topology.state*), 57
ConsistentRegionConfig.Trigger (*class in streamsx.topology.state*), 58
contents() (*streamsx.topology.tester.Tester method*), 64
ContextTypes (*class in streamsx.topology.context*), 38
count() (*streamsx.spl.types.Timestamp method*), 90
Counter (*streamsx.ec.MetricKind attribute*), 76
create_application_configuration() (*streamsx.rest_primitives.Instance method*), 126
create_submission_parameter() (*streamsx.topology.topology.Topology method*), 14
creationTime (*streamsx.rest_primitives.ApplicationConfiguration attribute*), 121
creationTime (*streamsx.rest_primitives.Domain attribute*), 121
creationTime (*streamsx.rest_primitives.Instance attribute*), 125
creationuser (*streamsx.rest_primitives.Domain attribute*), 121
creationuser (*streamsx.rest_primitives.Instance attribute*), 125
CustomMetric (*class in streamsx.ec*), 76

D

data (*streamsx.rest_primitives.ViewItem attribute*), 147
datetime() (*streamsx.spl.types.Timestamp method*), 91
definition_format_properties (*streamsx.rest_primitives.ResourceTag attribute*), 144
delete() (*streamsx.rest_primitives.ApplicationConfiguration method*), 121
dependencies() (*streamsx.rest_primitives.Toolkit property*), 147
description (*streamsx.rest_primitives.ApplicationConfiguration attribute*), 120
description (*streamsx.rest_primitives.Metric attribute*), 134
description (*streamsx.rest_primitives.ResourceTag attribute*), 144
description (*streamsx.rest_primitives.View attribute*), 148
Direct (*streamsx.topology.topology.SubscribeConnection attribute*), 11
display() (*streamsx.rest_primitives.View method*), 148
display() (*streamsx.topology.topology.View method*), 32
displayName (*streamsx.rest_primitives.Resource attribute*), 144
DISTRIBUTED (*streamsx.topology.context.ContextTypes attribute*), 39
Domain (*class in streamsx.rest_primitives*), 121
domain_id() (*in module streamsx.ec*), 74

E

edition_name (*streamsx.rest_primitives.ActiveVersion attribute*), 119
editionName (*streamsx.rest_primitives.Installation attribute*), 124

end_low_latency() (*streamsx.topology.topology.Stream method*), 21
end_parallel() (*streamsx.topology.topology.Stream method*), 21
eventual_result() (*streamsx.topology.tester.Tester method*), 64
exclude_packages (*streamsx.topology.topology.Topology attribute*), 12
ExportedStream (*class in streamsx.rest_primitives*), 122
Expression (*class in streamsx.spl.op*), 88
expression() (*streamsx.spl.op.Expression static method*), 88
expression() (*streamsx.spl.op.Invoke method*), 81
expression() (*streamsx.spl.op.Map method*), 85
expression() (*streamsx.spl.op.Sink method*), 87
expression() (*streamsx.spl.op.Source method*), 83
extend() (*streamsx.topology.schema.CommonSchema method*), 56
extend() (*streamsx.topology.schema.StreamSchema method*), 54
extracting() (*in module streamsx.spl.spl*), 110

F

fetch_tuples() (*streamsx.rest_primitives.View method*), 149
fetch_tuples() (*streamsx.topology.topology.View method*), 33
filter (*class in streamsx.spl.spl*), 106
filter() (*streamsx.topology.topology.Stream method*), 21
flat_map() (*streamsx.topology.topology.Stream method*), 22
float32() (*in module streamsx.spl.types*), 92
float64() (*in module streamsx.spl.types*), 93
for_each (*class in streamsx.spl.spl*), 107
for_each() (*streamsx.topology.topology.Stream method*), 22
FORCE_REMOTE_BUILD (*streamsx.topology.context.ConfigParams attribute*), 42
ForEach (*class in streamsx.topology.composite*), 61
from_datetime() (*streamsx.spl.types.Timestamp static method*), 91
from_overlays() (*streamsx.topology.context.JobConfig static method*), 45
from_time() (*streamsx.spl.types.Timestamp static method*), 91
full_product_version (*streamsx.rest_primitives.ActiveVersion attribute*), 119
fullProductVersion (*streamsx.rest_primitives.Installation attribute*), 124

G

Gauge (*streamsx.ec.MetricKind attribute*), 76
get_active_services() (*streamsx.rest_primitives.Domain method*), 122
get_active_services() (*streamsx.rest_primitives.Instance method*), 126
get_application_configuration() (*in module streamsx.ec*), 74
get_application_configurations() (*streamsx.rest_primitives.Instance method*), 126
get_application_directory() (*in module streamsx.ec*), 74
get_connections() (*streamsx.rest_primitives.OperatorInputPort method*), 135
get_connections() (*streamsx.rest_primitives.OperatorOutputPort method*), 136
get_domain() (*streamsx.rest.StreamingAnalyticsConnection method*), 116
get_domain() (*streamsx.rest.StreamsConnection method*), 115
get_domain() (*streamsx.rest_primitives.Instance method*), 126
get_domain() (*streamsx.rest_primitives.Job method*), 132
get_domain() (*streamsx.rest_primitives.View method*), 149
get_domains() (*streamsx.rest.StreamingAnalyticsConnection method*), 116
get_domains() (*streamsx.rest.StreamsConnection method*), 115
get_exported_streams() (*streamsx.rest_primitives.Instance method*), 126
get_host() (*streamsx.rest_primitives.Operator method*), 137
get_host() (*streamsx.rest_primitives.PE method*), 140
get_hosts() (*streamsx.rest_primitives.Domain method*), 122
get_hosts() (*streamsx.rest_primitives.Instance method*), 126
get_hosts() (*streamsx.rest_primitives.Job method*), 132
get_imported_streams() (*streamsx.rest_primitives.Instance method*), 126
get_input_ports()

(*streamsx.rest_primitives.Operator method*), 138
get_installations() (*streamsx.rest.StreamingAnalyticsConnection method*), 116
get_installations() (*streamsx.rest.StreamsConnection method*), 115
get_instance() (*streamsx.rest.StreamingAnalyticsConnection method*), 117
get_instance() (*streamsx.rest.StreamsConnection method*), 115
get_instance() (*streamsx.rest_primitives.Job method*), 132
get_instance() (*streamsx.rest_primitives.View method*), 149
get_instance_status() (*streamsx.rest_primitives.StreamingAnalyticsService method*), 145
get_instances() (*streamsx.rest.StreamingAnalyticsConnection method*), 117
get_instances() (*streamsx.rest.StreamsConnection method*), 116
get_instances() (*streamsx.rest_primitives.Domain method*), 122
get_job() (*streamsx.rest_primitives.Instance method*), 126
get_job() (*streamsx.rest_primitives.Operator method*), 138
get_job() (*streamsx.rest_primitives.PE method*), 141
get_job() (*streamsx.rest_primitives.View method*), 149
get_job_group() (*streamsx.rest_primitives.Job method*), 132
get_job_groups() (*streamsx.rest_primitives.Instance method*), 127
get_jobs() (*streamsx.rest_primitives.Instance method*), 127
get_jobs() (*streamsx.rest_primitives.ResourceAllocation method*), 143
get_metrics() (*streamsx.rest_primitives.Operator method*), 138
get_metrics() (*streamsx.rest_primitives.OperatorInputPort method*), 135
get_metrics() (*streamsx.rest_primitives.OperatorOutputPort method*), 136
get_metrics() (*streamsx.rest_primitives.PE method*), 141
get_metrics() (*streamsx.rest_primitives.Resource method*), 144
get_operator_connections() (*streamsx.rest_primitives.Instance method*), 127
get_operator_connections() (*streamsx.rest_primitives.Job method*), 132
get_operator_output_port() (*streamsx.rest_primitives.ExportedStream method*), 123
get_operators() (*streamsx.rest_primitives.Instance method*), 127
get_operators() (*streamsx.rest_primitives.Job method*), 132
get_output_ports() (*streamsx.rest_primitives.Operator method*), 138
get_pe() (*streamsx.rest_primitives.Operator method*), 138
get_pe_connections() (*streamsx.rest_primitives.Instance method*), 128
get_pe_connections() (*streamsx.rest_primitives.Job method*), 132
get_pes() (*streamsx.rest_primitives.Instance method*), 128
get_pes() (*streamsx.rest_primitives.Job method*), 132
get_pes() (*streamsx.rest_primitives.ResourceAllocation method*), 143
get_published_topics() (*streamsx.rest_primitives.Instance method*), 128
get_resource() (*streamsx.rest_primitives.PE method*), 141
get_resource() (*streamsx.rest_primitives.ResourceAllocation method*), 143
get_resource() (*streamsx.rest_primitives.RestResource method*), 145
get_resource_allocation() (*streamsx.rest_primitives.PE method*), 141
get_resource_allocations() (*streamsx.rest_primitives.Domain method*), 122
get_resource_allocations() (*streamsx.rest_primitives.Instance method*), 128
get_resource_allocations() (*streamsx.rest_primitives.Job method*), 133
get_resources() (*streamsx.build.BuildService method*), 112
get_resources() (*streamsx.rest.StreamingAnalyticsConnection method*), 117
get_resources() (*streamsx.rest.StreamsConnection method*), 116
get_resources() (*streamsx.rest_primitives.Domain method*), 122
get_streaming_analytics() (*streamsx.rest.StreamingAnalyticsConnection method*), 117
get_streams_version()

(*streamsx.topology.tester.Tester static method*), 64
get_toolkit() (*streamsx.build.BuildService method*), 112
get_toolkits() (*streamsx.build.BuildService method*), 112
get_view_items() (*streamsx.rest_primitives.View method*), 149
get_views() (*streamsx.rest_primitives.Instance method*), 128
get_views() (*streamsx.rest_primitives.Job method*), 133

H

HASH_PARTITIONED (*streamsx.topology.topology.Routing attribute*), 11
health (*streamsx.rest_primitives.Instance attribute*), 125
health (*streamsx.rest_primitives.Job attribute*), 131
health (*streamsx.rest_primitives.PE attribute*), 139
Host (*class in streamsx.rest_primitives*), 123

I

id (*streamsx.rest_primitives.Domain attribute*), 121
id (*streamsx.rest_primitives.Instance attribute*), 125
id (*streamsx.rest_primitives.Job attribute*), 131
id (*streamsx.rest_primitives.OperatorConnection attribute*), 134
id (*streamsx.rest_primitives.PE attribute*), 139
id (*streamsx.rest_primitives.PEConnection attribute*), 138
id (*streamsx.rest_primitives.Resource attribute*), 143
id (*streamsx.rest_primitives.Toolkit attribute*), 146
id (*streamsx.rest_primitives.View attribute*), 147
ignore() (*in module streamsx.spl.spl*), 110
ImportedStream (*class in streamsx.rest_primitives*), 124
include_packages (*streamsx.topology.topology.Topology attribute*), 12
index() (*streamsx.spl.types.Timestamp method*), 91
indexWithinJob (*streamsx.rest_primitives.Operator attribute*), 137
indexWithinJob (*streamsx.rest_primitives.PE attribute*), 139
indexWithinOperator (*streamsx.rest_primitives.OperatorInputPort attribute*), 135
indexWithinOperator (*streamsx.rest_primitives.OperatorOutputPort attribute*), 136
input_port (*class in streamsx.spl.spl*), 108
Installation (*class in streamsx.rest_primitives*), 124
Instance (*class in streamsx.rest_primitives*), 125
instance_id() (*in module streamsx.ec*), 74
int16() (*in module streamsx.spl.types*), 92
int32() (*in module streamsx.spl.types*), 92
int64() (*in module streamsx.spl.types*), 92
int8() (*in module streamsx.spl.types*), 92
Invoke (*class in streamsx.spl.op*), 80
ipAddress (*streamsx.rest_primitives.Host attribute*), 123
ipAddress (*streamsx.rest_primitives.Resource attribute*), 144
is_active() (*in module streamsx.ec*), 74
is_common() (*in module streamsx.topology.schema*), 50
is_complete() (*streamsx.topology.topology.PendingStream method*), 34
is_standalone() (*in module streamsx.ec*), 74
isolate() (*streamsx.topology.topology.Stream method*), 23

J

Job (*class in streamsx.rest_primitives*), 130
job() (*streamsx.topology.context.SubmissionResult property*), 46
JOB_CONFIG (*streamsx.topology.context.ConfigParams attribute*), 42
job_id() (*in module streamsx.ec*), 74
JobConfig (*class in streamsx.topology.context*), 43
jobGroup (*streamsx.rest_primitives.Job attribute*), 131
Json (*streamsx.topology.schema.CommonSchema attribute*), 55

L

last() (*streamsx.topology.topology.Stream method*), 23
lastModifiedTime (*streamsx.rest_primitives.ApplicationConfiguration attribute*), 121
lastTimeRetrieved (*streamsx.rest_primitives.Metric attribute*), 134
launchCount (*streamsx.rest_primitives.PE attribute*), 139
leader (*streamsx.rest_primitives.ActiveService attribute*), 119
local_channel() (*in module streamsx.ec*), 75
local_check() (*streamsx.topology.tester.Tester method*), 64
local_max_channels() (*in module streamsx.ec*), 75
logicalOperatorName (*streamsx.rest_primitives.View attribute*), 148
low_latency() (*streamsx.topology.topology.Stream method*), 24

M

machine_id (*streamsx.spl.types.Timestamp attribute*), 90
machine_id() (*streamsx.spl.types.Timestamp property*), 91
main_composite() (*in module streamsx.spl.op*), 88
Map (*class in streamsx.spl.op*), 84
map (*class in streamsx.spl.spl*), 105
Map (*class in streamsx.topology.composite*), 60
map() (*streamsx.topology.topology.Stream method*), 24
max_channels() (*in module streamsx.ec*), 75
maximumTupleRate (*streamsx.rest_primitives.View attribute*), 148
Metric (*class in streamsx.rest_primitives*), 134
MetricKind (*class in streamsx.ec*), 76
metricKind (*streamsx.rest_primitives.Metric attribute*), 134
metricType (*streamsx.rest_primitives.Metric attribute*), 134
minimum_os_base_version (*streamsx.rest_primitives.ActiveVersion attribute*), 120
minimum_os_patch_version (*streamsx.rest_primitives.ActiveVersion attribute*), 120
minimum_streams_version() (*streamsx.topology.tester.Tester static method*), 65
minimumOSBaseVersion (*streamsx.rest_primitives.Installation attribute*), 125
minimumOSPatchVersion (*streamsx.rest_primitives.Installation attribute*), 125

N

name (*streamsx.rest_primitives.ApplicationConfiguration attribute*), 120
name (*streamsx.rest_primitives.Host attribute*), 123
name (*streamsx.rest_primitives.Job attribute*), 131
name (*streamsx.rest_primitives.Metric attribute*), 134
name (*streamsx.rest_primitives.Operator attribute*), 137
name (*streamsx.rest_primitives.OperatorInputPort attribute*), 135
name (*streamsx.rest_primitives.OperatorOutputPort attribute*), 136
name (*streamsx.rest_primitives.ResourceTag attribute*), 144
name (*streamsx.rest_primitives.RestResource attribute*), 145
name (*streamsx.rest_primitives.Toolkit attribute*), 146
name (*streamsx.rest_primitives.Toolkit.Dependency attribute*), 146
name (*streamsx.rest_primitives.View attribute*), 147
name() (*streamsx.topology.topology.Stream property*), 25
name() (*streamsx.topology.topology.Topology property*), 15
name_to_runtime_id (*streamsx.topology.topology.Topology attribute*), 12
namespace() (*streamsx.topology.topology.Topology property*), 15
nanoseconds (*streamsx.spl.types.Timestamp attribute*), 90
nanoseconds() (*streamsx.spl.types.Timestamp property*), 91
now() (*streamsx.spl.types.Timestamp static method*), 91
null() (*in module streamsx.spl.types*), 93

O

of_definition() (*streamsx.rest.StreamingAnalyticsConnection static method*), 117
of_endpoint() (*streamsx.build.BuildService static method*), 112
of_endpoint() (*streamsx.rest_primitives.Instance static method*), 129
of_service() (*streamsx.rest_primitives.Instance static method*), 129
Operator (*class in streamsx.rest_primitives*), 137
OPERATOR_DRIVEN (*streamsx.topology.state.ConsistentRegionConfig.Tri attribute*), 58
operator_driven() (*streamsx.topology.state.ConsistentRegionConfig static method*), 58
OperatorConnection (*class in streamsx.rest_primitives*), 134
OperatorInputPort (*class in streamsx.rest_primitives*), 135
operatorKind (*streamsx.rest_primitives.Operator attribute*), 137
OperatorOutputPort (*class in streamsx.rest_primitives*), 136
optionalConnections (*streamsx.rest_primitives.PE attribute*), 139
output() (*streamsx.spl.op.Invoke method*), 81
output() (*streamsx.spl.op.Map method*), 85
output() (*streamsx.spl.op.Sink method*), 87
output() (*streamsx.spl.op.Source method*), 83
owner (*streamsx.rest_primitives.Instance attribute*), 125

P

parallel() (*streamsx.topology.topology.Stream method*), 26
params() (*streamsx.spl.op.Invoke property*), 81
params() (*streamsx.spl.op.Map property*), 85
params() (*streamsx.spl.op.Sink property*), 87
params() (*streamsx.spl.op.Source property*), 83

partition() (*streamsx.topology.topology.Window method*), 35
path (*streamsx.rest_primitives.Toolkit attribute*), 146
PE (*class in streamsx.rest_primitives*), 139
pe_id() (*in module streamsx.ec*), 74
PEConnection (*class in streamsx.rest_primitives*), 138
PendingStream (*class in streamsx.topology.topology*), 33
pendingTracingLevel (*streamsx.rest_primitives.PE attribute*), 140
PERIODIC (*streamsx.topology.state.ConsistentRegionConfig.Trigger attribute*), 58
periodic() (*streamsx.topology.state.ConsistentRegionConfig static method*), 59
populate() (*streamsx.topology.composite.ForEach method*), 61
populate() (*streamsx.topology.composite.Map method*), 61
populate() (*streamsx.topology.composite.Source method*), 60
primitive_operator (*class in streamsx.spl.spl*), 109
PrimitiveOperator (*class in streamsx.spl.spl*), 107
print() (*streamsx.topology.topology.Stream method*), 27
processId (*streamsx.rest_primitives.ActiveService attribute*), 119
processId (*streamsx.rest_primitives.PE attribute*), 140
processorCount (*streamsx.rest_primitives.Host attribute*), 123
product_name (*streamsx.rest_primitives.ActiveVersion attribute*), 120
product_version (*streamsx.rest_primitives.ActiveVersion attribute*), 120
productName (*streamsx.rest_primitives.Installation attribute*), 125
productVersion (*streamsx.rest_primitives.Installation attribute*), 125
properties (*streamsx.rest_primitives.ApplicationConfiguration attribute*), 120
properties_definition (*streamsx.rest_primitives.ResourceTag attribute*), 144
publish() (*streamsx.topology.topology.Stream method*), 27
PublishedTopic (*class in streamsx.rest_primitives*), 142
Python (*streamsx.topology.schema.CommonSchema attribute*), 56

R

raw_overlay() (*streamsx.topology.context.JobConfig property*), 45
refresh() (*streamsx.rest_primitives.ActiveService method*), 119
refresh() (*streamsx.rest_primitives.ApplicationBundle method*), 120
refresh() (*streamsx.rest_primitives.ApplicationConfiguration method*), 121
refresh() (*streamsx.rest_primitives.Domain method*), 122
refresh() (*streamsx.rest_primitives.ExportedStream method*), 123
refresh() (*streamsx.rest_primitives.Host method*), 124
refresh() (*streamsx.rest_primitives.ImportedStream method*), 124
refresh() (*streamsx.rest_primitives.Installation method*), 125
refresh() (*streamsx.rest_primitives.Instance method*), 130
refresh() (*streamsx.rest_primitives.Job method*), 133
refresh() (*streamsx.rest_primitives.Metric method*), 134
refresh() (*streamsx.rest_primitives.Operator method*), 138
refresh() (*streamsx.rest_primitives.OperatorConnection method*), 135
refresh() (*streamsx.rest_primitives.OperatorInputPort method*), 136
refresh() (*streamsx.rest_primitives.OperatorOutputPort method*), 137
refresh() (*streamsx.rest_primitives.PE method*), 141
refresh() (*streamsx.rest_primitives.PEConnection method*), 139
refresh() (*streamsx.rest_primitives.Resource method*), 144
refresh() (*streamsx.rest_primitives.ResourceAllocation method*), 143
refresh() (*streamsx.rest_primitives.RestResource method*), 145
refresh() (*streamsx.rest_primitives.Toolkit method*), 147
refresh() (*streamsx.rest_primitives.View method*), 150
refresh() (*streamsx.rest_primitives.ViewItem method*), 147
relocatable (*streamsx.rest_primitives.PE attribute*), 140
require_streams_version() (*streamsx.topology.tester.Tester static method*), 66
required (*streamsx.rest_primitives.OperatorConnection attribute*), 135
required (*streamsx.rest_primitives.PEConnection attribute*), 139
requiredConnections

(*streamsx.rest_primitives.PE attribute*), 140
requiredProductVersion (*streamsx.rest_primitives.Toolkit attribute*), 146
reserved (*streamsx.rest_primitives.ResourceTag attribute*), 144
resets() (*streamsx.topology.tester.Tester method*), 66
Resource (*class in streamsx.rest_primitives*), 143
resource (*streamsx.rest_primitives.RestResource attribute*), 145
resource_tags() (*streamsx.spl.op.Invoke property*), 82
resource_tags() (*streamsx.spl.op.Map property*), 85
resource_tags() (*streamsx.spl.op.Sink property*), 88
resource_tags() (*streamsx.spl.op.Source property*), 83
resource_tags() (*streamsx.topology.topology.Sink property*), 37
resource_tags() (*streamsx.topology.topology.Stream property*), 28
resource_url() (*streamsx.build.BuildService property*), 113
resource_url() (*streamsx.rest.StreamingAnalyticsConnection property*), 117
resource_url() (*streamsx.rest.StreamsConnection property*), 116
ResourceAllocation (*class in streamsx.rest_primitives*), 142
ResourceTag (*class in streamsx.rest_primitives*), 144
resourceType (*streamsx.rest_primitives.ActiveService attribute*), 119
resourceType (*streamsx.rest_primitives.Domain attribute*), 121
resourceType (*streamsx.rest_primitives.ExportedStream attribute*), 122
resourceType (*streamsx.rest_primitives.Host attribute*), 123
resourceType (*streamsx.rest_primitives.ImportedStream attribute*), 124
resourceType (*streamsx.rest_primitives.Installation attribute*), 124
resourceType (*streamsx.rest_primitives.Instance attribute*), 125
resourceType (*streamsx.rest_primitives.Job attribute*), 131
resourceType (*streamsx.rest_primitives.Metric attribute*), 134
resourceType (*streamsx.rest_primitives.Operator attribute*), 137
resourceType (*streamsx.rest_primitives.OperatorConnection attribute*), 135
resourceType (*streamsx.rest_primitives.OperatorInputPort attribute*), 135
resourceType (*streamsx.rest_primitives.OperatorOutputPort attribute*), 136
resourceType (*streamsx.rest_primitives.PE attribute*), 139
resourceType (*streamsx.rest_primitives.PEConnection attribute*), 139
resourceType (*streamsx.rest_primitives.ResourceAllocation attribute*), 142
resourceType (*streamsx.rest_primitives.Toolkit attribute*), 146
resourceType (*streamsx.rest_primitives.View attribute*), 148
resourceType (*streamsx.rest_primitives.ViewItem attribute*), 147
restartable (*streamsx.rest_primitives.PE attribute*), 140
RestResource (*class in streamsx.rest_primitives*), 145
restrictedTags (*streamsx.rest_primitives.Host attribute*), 123
result (*streamsx.topology.tester.Tester attribute*), 69
retrieve_console_log() (*streamsx.rest_primitives.PE method*), 141
retrieve_log_trace() (*streamsx.rest_primitives.Job method*), 133
retrieve_trace() (*streamsx.rest_primitives.PE method*), 142
ROUND_ROBIN (*streamsx.topology.topology.Routing attribute*), 11
Routing (*class in streamsx.topology.topology*), 11
rstring() (*in module streamsx.spl.types*), 93
run() (*in module streamsx.topology.context*), 48
run_for() (*streamsx.topology.tester.Tester method*), 66
runtime_id() (*streamsx.topology.topology.Stream property*), 28

S

SC_OPTIONS (*streamsx.topology.context.ConfigParams attribute*), 42
schedulerStatus (*streamsx.rest_primitives.ResourceAllocation attribute*), 142
schema (*streamsx.rest_primitives.PublishedTopic attribute*), 142
schema() (*streamsx.topology.schema.CommonSchema method*), 56
schema() (*streamsx.topology.schema.StreamSchema method*), 54
seconds (*streamsx.spl.types.Timestamp attribute*), 90
seconds() (*streamsx.spl.types.Timestamp property*), 91
SERVICE_DEFINITION (*streamsx.topology.context.ConfigParams attribute*), 42

SERVICE_NAME (*streamsx.topology.context.ConfigParams attribute*), 43
services (*streamsx.rest_primitives.Host attribute*), 123
session (*streamsx.rest.StreamsConnection attribute*), 115
set_consistent() (*streamsx.topology.topology.Stream method*), 29
set_parallel() (*streamsx.topology.topology.Stream method*), 29
setup_distributed() (*streamsx.topology.tester.Tester static method*), 66
setup_standalone() (*streamsx.topology.tester.Tester static method*), 68
setup_streaming_analytics() (*streamsx.topology.tester.Tester static method*), 68
shutdown() (*in module streamsx.ec*), 74
Sink (*class in streamsx.spl.op*), 86
Sink (*class in streamsx.topology.topology*), 36
Source (*class in streamsx.spl.op*), 82
source (*class in streamsx.spl.spl*), 104
Source (*class in streamsx.topology.composite*), 60
source() (*streamsx.topology.topology.Topology method*), 15
split() (*streamsx.topology.topology.Stream method*), 30
SSL_VERIFY (*streamsx.topology.context.ConfigParams attribute*), 43
STANDALONE (*streamsx.topology.context.ContextTypes attribute*), 41
start_data_fetch() (*streamsx.rest_primitives.View method*), 150
start_data_fetch() (*streamsx.topology.topology.View method*), 33
start_instance() (*streamsx.rest_primitives.StreamingAnalyticsService method*), 145
startedBy (*streamsx.rest_primitives.Job attribute*), 131
startTime (*streamsx.rest_primitives.ActiveService attribute*), 119
startTime (*streamsx.rest_primitives.Instance attribute*), 125
status (*streamsx.rest_primitives.ActiveService attribute*), 119
status (*streamsx.rest_primitives.Domain attribute*), 121
status (*streamsx.rest_primitives.Host attribute*), 123
status (*streamsx.rest_primitives.Instance attribute*), 125
status (*streamsx.rest_primitives.Job attribute*), 131
status (*streamsx.rest_primitives.PE attribute*), 140
status (*streamsx.rest_primitives.PEConnection attribute*), 139
status (*streamsx.rest_primitives.Resource attribute*), 144
status (*streamsx.rest_primitives.ResourceAllocation attribute*), 143
statusReason (*streamsx.rest_primitives.PE attribute*), 140
stop_data_fetch() (*streamsx.rest_primitives.View method*), 150
stop_data_fetch() (*streamsx.topology.topology.View method*), 33
stop_instance() (*streamsx.rest_primitives.StreamingAnalyticsService method*), 145
Stream (*class in streamsx.topology.topology*), 18
stream() (*streamsx.spl.op.Map property*), 86
stream() (*streamsx.spl.op.Source property*), 84
STREAMING_ANALYTICS_SERVICE (*streamsx.topology.context.ContextTypes attribute*), 41
StreamingAnalyticsConnection (*class in streamsx.rest*), 116
StreamingAnalyticsService (*class in streamsx.rest_primitives*), 145
streamName (*streamsx.rest_primitives.OperatorOutputPort attribute*), 136
streams() (*streamsx.topology.topology.Topology property*), 17
STREAMS_CONNECTION (*streamsx.topology.context.ConfigParams attribute*), 43
streams_connection (*streamsx.topology.tester.Tester attribute*), 69
StreamSchema (*class in streamsx.topology.schema*), 50
StreamsConnection (*class in streamsx.rest*), 114
streamsx.build (*module*), 111
streamsx.ec (*module*), 71
streamsx.rest (*module*), 113
streamsx.rest_primitives (*module*), 118
streamsx.spl.op (*module*), 77
streamsx.spl.spl (*module*), 95
streamsx.spl.toolkit (*module*), 93
streamsx.spl.types (*module*), 89
streamsx.topology (*module*), 3
streamsx.topology.composite (*module*), 59
streamsx.topology.context (*module*), 37
streamsx.topology.schema (*module*), 48
streamsx.topology.state (*module*), 56
streamsx.topology.tester (*module*), 62

streamsx.topology.tester_runtime (*module*), 71
streamsx.topology.topology (*module*), 6
String (*streamsx.topology.schema.CommonSchema attribute*), 56
style() (*streamsx.topology.schema.StreamSchema property*), 54
submission_parameters() (*streamsx.topology.context.JobConfig property*), 45
submission_result (*streamsx.topology.tester.Tester attribute*), 69
SubmissionResult (*class in streamsx.topology.context*), 46
submit() (*in module streamsx.topology.context*), 47
submit() (*streamsx.spl.spl.PrimitiveOperator method*), 108
submit_job() (*streamsx.rest_primitives.ApplicationBundle method*), 120
submit_job() (*streamsx.rest_primitives.Instance method*), 130
submit_job() (*streamsx.rest_primitives.StreamingAnalyticsService method*), 145
submitTime (*streamsx.rest_primitives.Job attribute*), 131
subscribe() (*streamsx.topology.topology.Topology method*), 18
SubscribeConnection (*class in streamsx.topology.topology*), 11

T

tag (*streamsx.rest_primitives.Host attribute*), 123
tags (*streamsx.rest_primitives.Resource attribute*), 144
target_pe_count() (*streamsx.topology.context.JobConfig property*), 46
test() (*streamsx.topology.tester.Tester method*), 68
Tester (*class in streamsx.topology.tester*), 63
Time (*streamsx.ec.MetricKind attribute*), 76
time() (*streamsx.spl.types.Timestamp method*), 91
Timestamp (*class in streamsx.spl.types*), 90
Toolkit (*class in streamsx.rest_primitives*), 146
TOOLKIT (*streamsx.topology.context.ContextTypes attribute*), 41
Toolkit.Dependency (*class in streamsx.rest_primitives*), 146
topic (*streamsx.rest_primitives.PublishedTopic attribute*), 142
Topology (*class in streamsx.topology.topology*), 12
tracing() (*streamsx.topology.context.JobConfig property*), 46
tracingLevel (*streamsx.rest_primitives.PE attribute*), 140
trigger() (*streamsx.topology.topology.Window method*), 35
tuple_check() (*streamsx.topology.tester.Tester method*), 69
tuple_count() (*streamsx.topology.tester.Tester method*), 70
type (*streamsx.rest_primitives.ActiveService attribute*), 119
type_checking (*streamsx.topology.topology.Topology attribute*), 12

U

uint16() (*in module streamsx.spl.types*), 92
uint32() (*in module streamsx.spl.types*), 92
uint64() (*in module streamsx.spl.types*), 92
uint8() (*in module streamsx.spl.types*), 92
union() (*streamsx.topology.topology.Stream method*), 31
update() (*streamsx.rest_primitives.ApplicationConfiguration method*), 121
update_operators() (*streamsx.rest_primitives.Job method*), 133
upload_bundle() (*streamsx.rest_primitives.Instance method*), 130
upload_toolkit() (*streamsx.build.BuildService method*), 113

V

value (*streamsx.rest_primitives.Metric attribute*), 134
value() (*streamsx.ec.CustomMetric property*), 77
VCAP_SERVICES (*streamsx.topology.context.ConfigParams attribute*), 43
version (*streamsx.rest_primitives.Toolkit attribute*), 146
version (*streamsx.rest_primitives.Toolkit.Dependency attribute*), 147
View (*class in streamsx.rest_primitives*), 147
View (*class in streamsx.topology.topology*), 32
view() (*streamsx.topology.topology.Stream method*), 31
ViewItem (*class in streamsx.rest_primitives*), 147

W

Window (*class in streamsx.topology.topology*), 34

X

XML (*streamsx.topology.schema.CommonSchema attribute*), 56