Data Types and Serialization

Every Kafka Streams application must provide SerDes (Serializer/Deserializer) for the data types of record keys and record values (e.g. `java.lang.String` or Avro objects) to materialize the data when necessary. Operations that require such SerDes information include: `stream()`, `table()`, `to()`, `through()`, `groupByKey()`, `groupBy()`.

You can provide SerDes by using either of these methods:

- By setting default SerDes via a `Properties` instance.
- By specifying explicit SerDes when calling the appropriate API methods, thus overriding the defaults.

You can configure Java streams applications to deserialize and ingest data in multiple ways, including Kafka console producers, JDBC source connectors, and Java client producers. For full code examples, see connect-streams-pipeline.

Configuring SerDes

SerDes specified in the Streams configuration via the `Properties` config are used as the default in your Kafka Streams application.

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsConfig;

Properties settings = new Properties();
// Default serde for keys of data records (here: built-in serde for String type)
settings.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
// Default serde for values of data records (here: built-in serde for Long type)
settings.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass().getName());
```

Note

If a `Serde` is specified via `Properties`, the `Serde` class cannot have generic types, i.e., a class `MySerde<T extends Number> implements Serde<T>` cannot be used. This implies that you cannot use any `Serde` that is created via `Serdes.serdeFrom(Serializer<T>, Deserializer<T>)`. Only fully typed `Serde` classes like `MySerde implements Serde<MyCustomType>` are supported due to Java type erasure.

Overriding default SerDes

You can also specify SerDes explicitly by passing them to the appropriate API methods, which overrides the default serde settings:

```
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;

final Serde<String> stringSerde = Serdes.String();
final Serde<Long> longSerde = Serdes.Long();

// The stream userCountByRegion has type `String` for record keys (for region)
// and type `Long` for record values (for user counts).
KStream<String, Long> userCountByRegion = ...;
userCountByRegion.to("RegionCountsTopic", Produced.with(stringSerde, longSerde));
```

If you want to override serdes selectively, i.e., keep the defaults for some fields, then don't specify the serde whenever you want to leverage the default settings:

```
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;

// Use the default serializer for record keys (here: region as String) by not specifying the key serde,
// but override the default serializer for record values (here: userCount as Long).
final Serde<Long> longSerde = Serdes.Long();
KStream<String, Long> userCountByRegion = ...;
userCountByRegion.to("RegionCountsTopic", Produced.valueSerde(Serdes.Long()));
```

Note

If some of your incoming records are corrupted or ill-formatted, they will cause the deserializer class to report an error. Since 4.0.0 release we have introduced an `org.apache.kafka.streams.errors.DeserializationExceptionHandler` interface which allows you to customize how to handle such records. The customized implementation of the interface can be specified via the `StreamsConfig`. For more details, please feel free to read the Failure and exception handling FAQ

Available SerDes

Primitive and basic types

Apache Kafka® includes several built-in serde implementations for Java primitives and basic types such as `byte[]` in its `kafka-clients` Maven artifact:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.3.0-ccs</version>
</dependency>
```

This artifact provides the following serde implementations under the packageorg.apache.kafka.common.serialization, which you can leverage when e.g., defining default serializers in your Streams configuration.

Data type	Serde
byte[]	`Serdes.ByteArray()`, `Serdes.Bytes()` (see tip below)
ByteBuffer	`Serdes.ByteBuffer()`
Double	`Serdes.Double()`
Integer	`Serdes.Integer()`
Long	`Serdes.Long()`
String	`Serdes.String()`
UUID	`Serdes.UUID()`

Tip

Bytes is a wrapper for Java's `byte[]` (byte array) that supports proper equality and ordering semantics. You may want to consider using `Bytes` instead of `byte[]` in your applications.

You would use the built-in SerDes as follows, using the example of the String serde:

```
// When configuring the default SerDes of StreamConfig
Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,   Serdes.String().getClass().getName());
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

// When you want to override SerDes explicitly/selectively
final Serde<String> stringSerde = Serdes.String();
StreamsBuilder builder = new StreamsBuilder();
builder.stream("my-avro-topic", Consumed.with(keyGenericAvroSerde, valueGenericAvroSerde));
```

Avro

Confluent provides schema-registry compatible Avro serdes for data in generic Avro and in specific Avro format:

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-streams-avro-serde</artifactId>
    <version>5.3.0</version>
</dependency>
```

Both the generic and the specific Avro serde require you to configure the endpoint of Confluent Schema Registry via the `schema.registry.url` setting:

- When you define the generic or specific Avro serde as a default serde via `StreamsConfig`, then you must also set the Schema Registry endpoint in `StreamsConfig`.
- When you instantiate the generic or specific Avro serde directly (e.g. `new GenericAvroSerde()`), you must call `Serde#configure()` on the serde instance to set the Schema Registry endpoint before using the serde instance. Additionally, you must tell `Serde#configure()` via a boolean parameter whether the serde instance is used for serializing/deserializing record *keys* (`true`) or record *values* (`false`).

Usage example for Confluent `GenericAvroSerde`:

```
// Generic Avro serde example
import io.confluent.kafka.streams.serdes.avro.GenericAvroSerde;

// When configuring the default serdes of StreamConfig
final Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, GenericAvroSerde.class);
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, GenericAvroSerde.class);
streamsConfiguration.put("schema.registry.url", "http://my-schema-registry:8081");

// When you want to override serdes explicitly/selectively
final Map<String, String> serdeConfig = Collections.singletonMap("schema.registry.url",
                                                                 "http://my-schema-registry:8081");
final Serde<GenericRecord> keyGenericAvroSerde = new GenericAvroSerde();
keyGenericAvroSerde.configure(serdeConfig, true); // `true` for record keys
final Serde<GenericRecord> valueGenericAvroSerde = new GenericAvroSerde();
valueGenericAvroSerde.configure(serdeConfig, false); // `false` for record values

StreamsBuilder builder = new StreamsBuilder();
KStream<GenericRecord, GenericRecord> textLines =
  builder.stream(keyGenericAvroSerde, valueGenericAvroSerde, "my-avro-topic");
```

Usage example for Confluent `SpecificAvroSerde`:

```
// Specific Avro serde example
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

// When configuring the default serdes of StreamConfig
final Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, SpecificAvroSerde.class);
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.class);
streamsConfiguration.put("schema.registry.url", "http://my-schema-registry:8081");

// When you want to override serdes explicitly/selectively
final Map<String, String> serdeConfig = Collections.singletonMap("schema.registry.url",
                                                                 "http://my-schema-registry:8081");
// `Foo` and `Bar` are Java classes generated from Avro schemas
final Serde<Foo> keySpecificAvroSerde = new SpecificAvroSerde<>();
keySpecificAvroSerde.configure(serdeConfig, true); // `true` for record keys
final Serde<Bar> valueSpecificAvroSerde = new SpecificAvroSerde<>();
valueSpecificAvroSerde.configure(serdeConfig, false); // `false` for record values

StreamsBuilder builder = new StreamsBuilder();
KStream<Foo, Bar> textLines = builder.stream("my-avro-topic", Consumed.with(keySpecificAvroSerde, valueSpecificAvroSer
de));
```

When you create source streams, you specify input serdes by using the Streams DSL. When you construct the processor topology by using the lower-level Processor API, you can specify the serde class, like the Confluent `GenericAvroSerde` and `SpecificAvroSerde` classes.

```
TopologyBuilder builder = new TopologyBuilder();
builder.addSource("Source", keyGenericAvroSerde.deserializer(), valueGenericAvroSerde.deserializer(), inputTopic);
```

The following end-to-end demos showcase using the Confluent Avro serdes:

- **Java:**
  - GenericAvroIntegrationTest
  - SpecificAvroIntegrationTest
- **Scala:**
  - GenericAvroScalaIntegrationTest
  - SpecificAvroScalaIntegrationTest

JSON

The Kafka Streams code examples also include a basic serde implementation for JSON:

- PageViewTypedDemo

As shown in the example file, you can use JSONSerdes inner classes `Serdes.serdeFrom(<serializerInstance>, <deserializerInstance>)` to construct JSON compatible serializers and deserializers.

Further serdes

The Confluent examples repository demonstrates how to implement templated serdes:

- `PriorityQueue<T>` serde: PriorityQueueSerde

Implementing custom SerDes

If you need to implement custom SerDes, your best starting point is to take a look at the source code references of existing SerDes (see previous section). Typically, your workflow will be similar to:

1. Write a *serializer* for your data type `T` by implementing org.apache.kafka.common.serialization.Serializer.

2. Write a *deserializer* for `T` by implementing org.apache.kafka.common.serialization.Deserializer.
3. Write a *serde* for `T` by implementing org.apache.kafka.common.serialization.Serde, which you either do manually (see existing SerDes in the previous section) or by leveraging helper functions in Serdes such as `Serdes.serdeFrom(Serializer<T>, Deserializer<T>)`. Note that you will need to implement your own class (that has no generic types) if you want to use your custom serde in the configuration provided to `KafkaStreams`. If your serde class has generic types or you use `Serdes.serdeFrom(Serializer<T>, Deserializer<T>)`, you can pass your serde only via methods calls (for example `builder.stream("topicName", Consumed.with(...))`.

Please report any inaccuracies on this page or suggest an edit.

5 Votes

Last updated on Sep 10, 2019.