

Getting Started

```
public class User {  
    String name;  
    int age;  
}
```

```
Producer<byte[]> producer =  
    client.newProducer()  
        .topic(topic)  
        .create();  
User user = new User("Tom", 28);  
// serialize the `user` by yourself;  
byte[] message = ...;  
producer.send(message);
```

Parse Exceptions

- The field you're looking for doesn't exist any more
- The type of field has changed
 - e.g. what used to be a `String` is now an `Integer`

“I am starting to think that `schemaless` just means your schema is scattered randomly throughout your code. It is almost impossible to troubleshoot anything non-trivial because there are endless assumptions, but few explicit requirements”

–Robert Kluin (@robertkluin) May 29, 2018

Introducing Schema

Schema

- Define how to serialize and deserialize data
- Define how to evolve your data format
- Handle backward compatibility

Schema Info

- `type`: the type of schema.
- `schema`: the schema data. Schema type implementation specific.
- `properties`: the properties associated with the schema. Application specific data.

```
```json
{
 "type": "JSON",
 "schema": "{
 \"type\": \"record\",
 \"name\": \"User\",
 \"namespace\": \"com.foo\",
 \"fields\": [
 {
 \"name\": \"file1\",
 \"type\": [\"null\", \"string\"],
 \"default\": null
 },
 {
 \"name\": \"file2\",
 \"type\": \"string\",
 \"default\": null
 },
 {
 \"name\": \"file3\",
 \"type\": [\"null\", \"string\"],
 \"default\": \"dfdf\"
 }
]
 }",
 "properties": {}
}
```
```

Schema Types

- Primitive Types
- Complex Types

Primitive Types

- **`BOOLEAN`**: a binary value
- **`INT8`**: 8-bit signed integer
- **`INT16`**: 16-bit signed integer
- **`INT32`**: 32-bit signed integer
- **`INT64`**: 64-bit signed integer
- **`FLOAT`**: single precision (32-bit) IEEE 754 floating-point number
- **`DOUBLE`**: double precision (64-bit) IEEE 754 floating-point number
- **`BYTES`**: sequence of 8-bit unsigned bytes
- **`STRING`**: unicode character sequence
- **`TIMESTAMP`** (**`DATE`**, **`TIME`**) : A logic type represents a specific instant in time, with millisecond precision. It stores the number of milliseconds since ``January 1, 1970, 00:00:00 GMT`` as a ``INT64`` value.

Primitive Types - Example

```
```java
// Create producer with String schema and send messages
Producer<String> producer = client.newProducer(Schema.STRING).create();
producer.newMessage().value("Hello Pulsar!").send();

// Create consumer with String schema and receive messages
Consumer<String> consumer = client.newConsumer(Schema.STRING).create();
consumer.receive();
```
```

Complex Types

- Key-Value
- Struct

Struct Types

- Supported Types: AVRO / JSON / PROTOBUF
- Schema Definition - AVRO
- Two approaches
 - Static - The struct is predefined. POJO, or Avro/Protobuf generated classes
 - Generic - The struct is unknown or not predefined.

Static Schema

```
```java
// Create producer with Struct schema and send messages
Producer<User> producer = client.newProducer(Schema.AVRO(User.class)).create();
producer.newMessage()
 .value(User.builder()
 .userName("pulsar-user")
 .userId(1L)
 .build())
 .send();

// Create consumer with Struct schema and receive messages
Consumer<User> consumer = client.newConsumer(Schema.AVRO(User.class)).create();
consumer.receive();
```
```

Generic Schema

- ``GenericSchemaBuilder``: Build a generic schema
- ``GenericRecordBuilder``: Build a generic record

Generic Schema - Example

```
```` java
RecordSchemaBuilder recordSchemaBuilder = SchemaBuilder.record("schemaName");
recordSchemaBuilder
 .field("intField")
 .type(SchemaType.INT32);
SchemaInfo schemaInfo = recordSchemaBuilder.build(SchemaType.AVR0);

Producer<GenericRecord> producer = client.newProducer(Schema.generic(schemaInfo)).create();

producer.newMessage().value(schema.newRecordBuilder()
 .set("intField", 32)
 .build()).send();
````
```

Auto Schema

- AUTO_PRODUCE
 - Producers validate bytes according to the schema in the topic
- AUTO_CONSUME
 - Consumers deserialize messages into `GenericRecord`
 - Schema is unknown in advance

How does Schema work

```
`` `java
Producer<User> producer = client.newProducer(Schema.AVRO(User.class)).create();
`` `
```

1. `Schema.AVRO(User.class)` => Generates SchemaInfo
2. `newProducer` => connect to broker and send the schema info
3. Broker receives the schema info
 1. If a topic doesn't have a schema, creates the schema
 2. If a topic already have a schema, broker verifies if the schema is compatible with existing schemas.
 1. If it is compatible and is a new schema, generates a new version of schema
 2. If it is not compatible, fail the producer

Schema Evolution

Schema Compatibility Check

- Schema Compatibility Checker
 - One checker per schema type
 - Configured by ``schemaRegistryCompatibilityCheckers``
 - Only AVRO and JSON supports schema evolution for now
 - All other schema types don't allow schema evolution

Compatibility Check Strategy

| Strategy | Changes Allowed | Check against what schemas | Upgrade first |
|---------------------|---|----------------------------|---------------|
| ALWAYS_INCOMPATIBLE | All changes are disabled | All previous versions | None |
| ALWAYS_COMPATIBLE | All changes are allowed | Latest version | Depends |
| BACKWARD | <ul style="list-style-type: none">Delete fieldsAdd optional fields | Latest version | Consumers |
| FORWARD | <ul style="list-style-type: none">Add fieldsDelete optional fields | Latest version | Producers |
| FULL | Modify optional fields | Latest version | Any Order |

Order of upgrading clients

- **BACKWARD:** Upgrade all consumers before start producing new events
- **FORWARD:**
 - Upgrade all producers to new schema
 - Make sure the data produced using old schemas are not available to consumers anymore
 - Then upgrade producers and consumers independently
- **FULL:** Upgrade producers and consumers independently
- **ALWAYS_COMPATIBLE:** Be cautious about when to upgrade clients

Managing Schemas

Schema Restful API

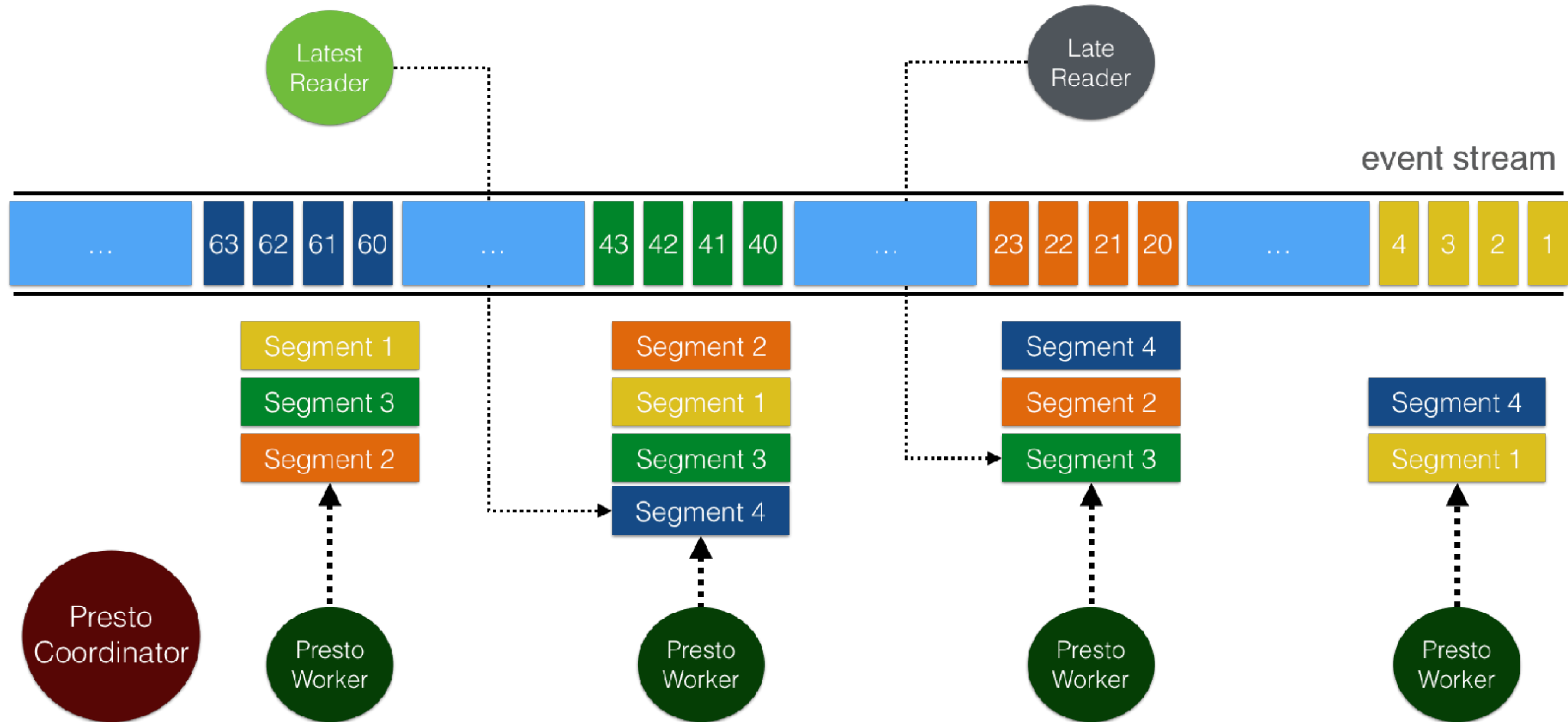
- Upload Schema
- Get Schema : latest or by version
- Delete Schema

```
```shell
$ pulsar-admin schemas upload -f <schema-definition-file> <topic-name>
```
```

Query Streams using Schema

- Presto
- Hive
- Flink SQL
- Spark SQL

Pulsar Presto SQL



Pulsar Presto SQL

```
presto> select * from pulsar."public/default".generator_test;
```

| firstname | middlename | lastname | email |
|-----------|------------|-----------|------------------------------|
| Genesis | Katherine | Wiley | genesis.wiley@gmail.com |
| Brayden | | Stanton | brayden.stanton@yahoo.com |
| Benjamin | Julian | Velasquez | benjamin.velasquez@yahoo.com |
| Michael | Thomas | Donovan | donovan@mail.com |
| Brooklyn | Avery | Roach | brooklynroach@yahoo.com |
| Skylar | | Bradshaw | skylarbradshaw@yahoo.com |