Apache Kafka

for Java Developers

Serialization Strategies

Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

Agenda

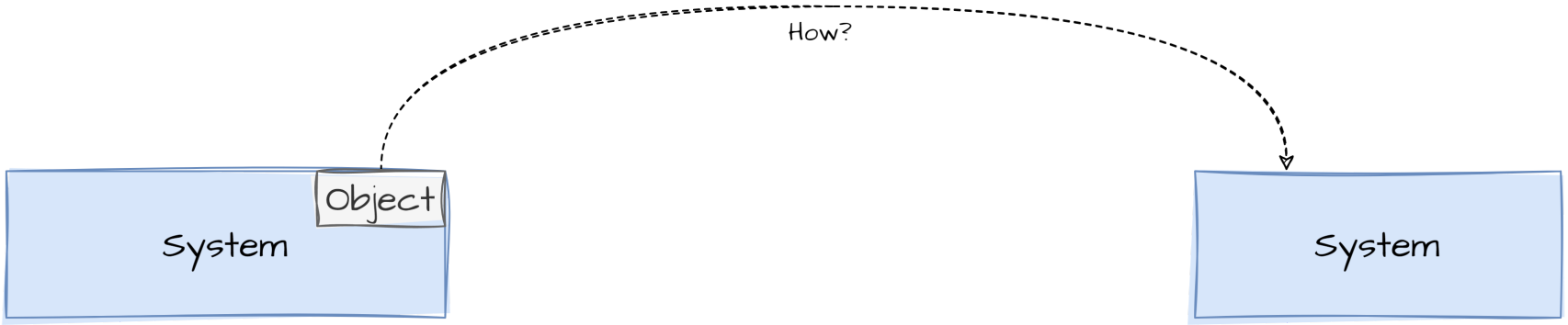
We'll talk about

- Definition, purpose, challenges
- (Basic) Serialization mechanisms and facilities in Kafka
- Common options with pro's and con's
- Solutions for schema management

Data (De)Serialization

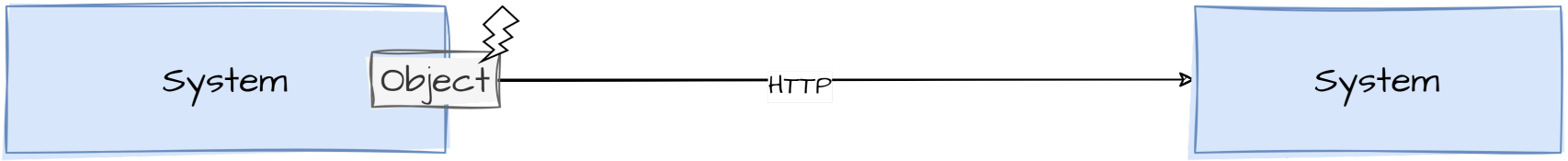
Definition, Purpose, Challenges

Why do we need to serialize data anyways?



- How can systems exchange data?
- We need a way to ...
  - connect systems via network
  - present data in a way that both systems understand

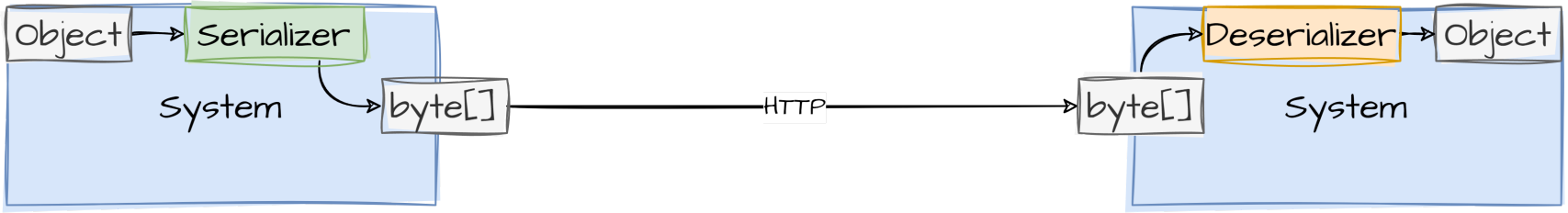
Why do we need to serialize data anyways? (cont.)



- A connection alone is not enough
- Objects are in-memory structures that can't be transferred directly
- Even if you **could** magically share memory over the wire
  - different in-memory representations
  - security nightmare

Why do we need to serialize data anyways? (cont.)

The solution: a common *language* that can be transferred via network



Definition

Serialization

is the process of converting a data object into a series of bytes that saves the state of the object in an easily transmittable form

Deserialization

is the process of reconstructing a data structure or object from a series of bytes or a string in order to instantiate the object for consumption.

Purpose

- Enables efficient, ordered, and reliable data streaming across distributed systems
- Maintains the integrity and consistency of data across systems
- Acts as a communication contract between 1 to n systems
- Allows data to be
  - monitored easily while *in-flight*
  - mocked for testing / development purposes
  - saved and analyzed / debugged later on

Some common challenges

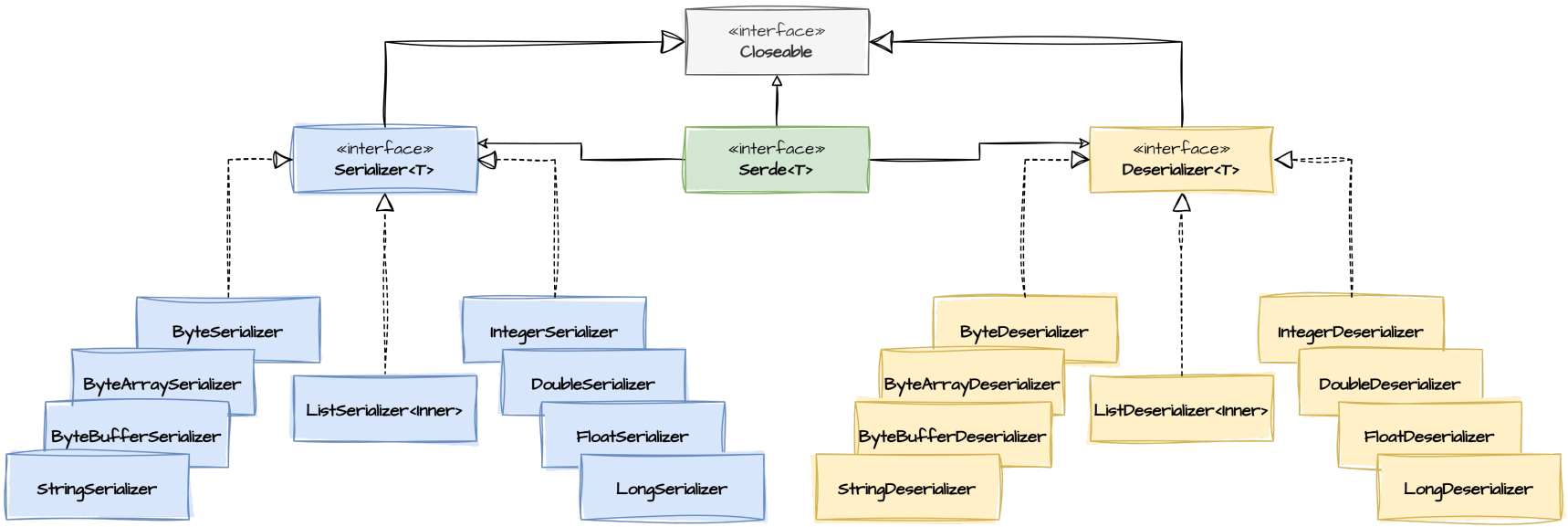
- Structures change / evolve
  - How can we keep that in sync on a multitude of systems?
- Transferring large datasets
  - Is the *language* and medium up to the task?
  - Is compression feasible?
- Computational overhead
  - (De)Serialization costs CPU time
  - **Especially** when size matters

Serialization Mechanisms in Kafka

Kafka and (De)Serialization

- Kafka ...
  - acts as a transport medium for data between *producer* & *consumer*
  - doesn't care about the data contract (unless you use a schema registry)
  - is agnostic regarding the structure of the data
- But: Kafka provides ...
  - mechanisms how applications can handle these issues
  - a set of default implementations for primitive data types

Kafka (De)Serializer interfaces and built-in support



The Serializer interface

```
public interface Serializer<T> extends Closeable {

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

The Deserializer interface

```
public interface Deserializer<T> extends Closeable {

    default void configure(Map<String, ?> configs, boolean isKey) {
        // intentionally left blank
    }

    default T deserialize(String topic, Headers headers, byte[] data) {
        return deserialize(topic, data);
    }

    default T deserialize(String topic, Headers headers, ByteBuffer data) {
        return deserialize(topic, headers, Utils.toNullableArray(data));
    }

    @Override
    default void close() {
        // intentionally left blank
    }
}
```

Configuring the (De)Serializer

```
public class ConfigurationExample {

  public void basicProducerConfiguration() {
    Map<String, Object> config = Map.of(
      ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(),
      ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName()
    );
    // ...
  }

  public void basicConsumerConfiguration() {
    Map<String, Object> config = Map.of(
      ConsumerConfig.KEY_DESERIALIZER_CONFIG, StringDeserializer.class.getName(),
      ConsumerConfig.VALUE_DESERIALIZER_CONFIG, StringDeserializer.class.getName()
    );
  }

}
```

The Kafka Clients facilities are pretty bare metal

- What's with complex objects?
- What happens when a (De)Serializer fails?
- What about schema management / validation?

With only the Kafka Client library you have to solve these issues on your own, e.g. with a custom (De)Serializer.

Advanced Use-Cases (cont.)

- Using JSON Objects is very common
- Easy to integrate with e.g. Jackson / Gson
- Spring Kafka has more convenience
  - Provides a `JsonSerializer` / `JsonDeserializer`
  - Provides a `ErrorHandlingDeserializer` that can handle faulty data
  - But is very opinionated (as usual) and might cause problems with other clients

We will build our own `JsonSerializer` and `JsonDeserializer` as part of the lab

Data Serialization Options

Schema Management in Kafka SerDes

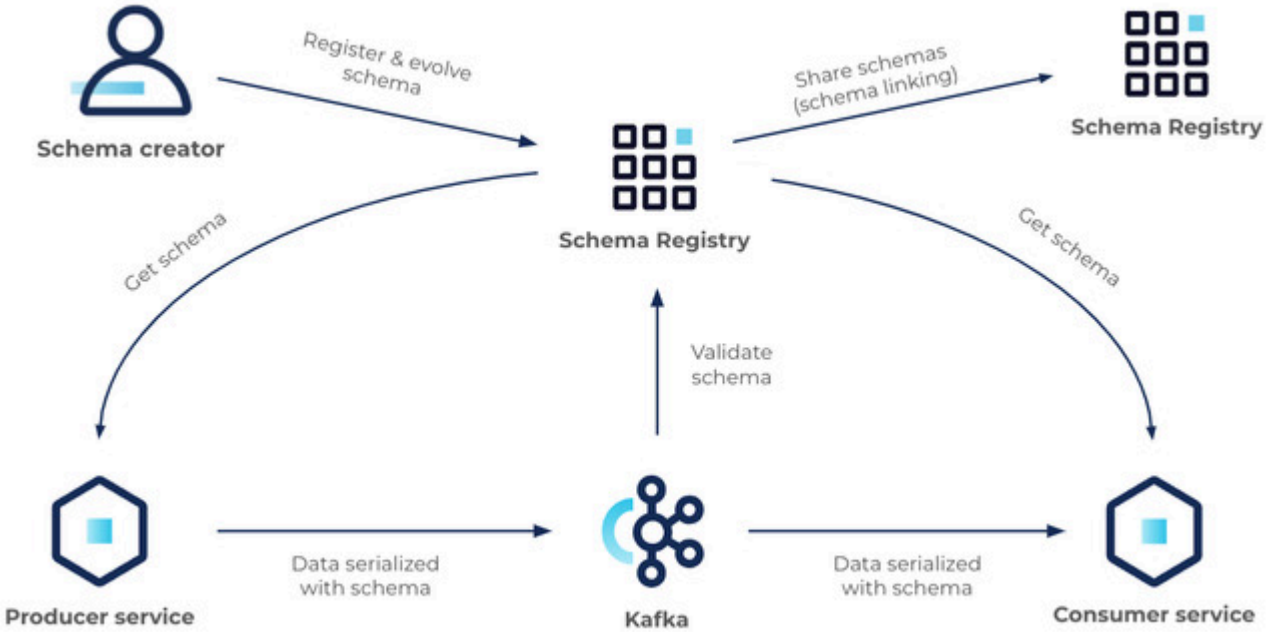
- Ensures compatibility across different versions of data producers and consumers.
- **Key Options**
    1. Confluent Schema Registry
    2. Apache Avro without a Registry
    3. Protobuf with or without a Registry
    4. JSON Schema

> *Choosing the right schema management tool depends on needs for compatibility, performance, and ease of use.*

Confluent Schema Registry

- A centralized service that provides runtime schema enforcement.
- **Pros**
  - Centralizes schema management.
  - Supports schema evolution with full compatibility checks.
  - Reduces payload size as schema is not included in each message.
- **Cons**
  - Introduces a single point of failure in the architecture.
  - Dependency on external service increases system complexity.
  - Primarily supports Avro, limited support for other formats.

Confluent Schema Registry (cont.)



Apache Avro without a Registry

- Using Avro for serialization without a centralized schema registry
- Typically means embedding the schema within each message payload
- **Pros**
  - Schema is self-contained within each message, ensuring the consumer can always deserialize data
  - Removes the dependency on an external schema registry service
  - Flexible and simple to implement in small-scale systems

Apache Avro without a Registry (cont.)

- **Cons**
  - Increases message size as schema is included in every message
  - No centralized schema management, which can lead to inconsistencies
  - Lacks automatic compatibility checks, increasing the risk of runtime errors

Protobuf with or without a Registry

- Binary serialization format by Google
- **Pros**
  - Highly efficient binary format reduces message size and improves performance
  - Strongly typed, which helps in catching errors during development
  - Schema registry integration is optional but recommended for large scale deployments
- **Cons**
  - Steeper learning curve due to its binary nature and tooling
  - Less human-readable than JSON or Avro
  - Managing schemas without a registry can be challenging in large environments

JSON Schema

- Defines the structure of JSON data for validation and documentation
- **Pros**
  - Human-readable and easy to understand, making it popular for web applications
  - Flexible and easy to integrate with modern web technologies
  - Does not require a centralized registry, simplifying deployment
- **Cons**
  - Less efficient in terms of payload size compared to binary formats like Protobuf
  - Lacks built-in support for schema versioning, evolution and has weak schema enforcement

Overview

	Serialization libraries			Popular formats	
	Avro	Thrift	Protobuf	JSON	XML
Binary representation	+	+	+	BSON	EXI
Generic data types	+	+/-	-	+	+
Schema-based	+	+	+	-	+
Supports schema evolution	+	+	+	-	-
Specific encoding	+	+	+	-	+
Browser support	+	+	-	++	+
Date types	+	-	+	-	+

Summary

Summary

Data serialization is a complex topic that will have a major impact on your system landscape in the long run!

- Choosing the right **serialization strategy** is very important as the wrong choice can be hard to fix later on. As usual: **there is no silver bullet solution**
- Lots of choices, all have their benefits and drawbacks
- A perfect fit might not exist, but there are some key questions you can as! yourself to make a good decision

Key questions

- *What are my performance requirements?*
  - Do I need to optimize for high throughput or low latency?
  - How does the serialization format impact the performance of my system?
- *What is the nature of the data being serialized?*
  - Is the data highly structured or schema-less?
  - Does the data format need to support complex types and hierarchies?

Key questions (cont.)

- *How important is schema evolution to my application?*
  - Will the data structure change over time?
  - Do I need backward and forward compatibility between different versions of the schema?
- *What are the system integration requirements?*
  - Which programming languages and frameworks are being used?
  - Do these technologies have native support or robust libraries for the serialization format I'm considering?

Key questions (cont.)

- *What are the operational considerations?*
  - Do I have the resources to manage a schema registry?
  - What are the implications of adding a schema registry in terms of setup, maintenance, and overhead?
- *How does the choice of serialization impact data security and compliance?*
  - Does the serialization format or schema registry offer features that enhance data security, such as encryption or access control?
  - Are there compliance requirements for data storage or transmission that could influence the choice of serialization?

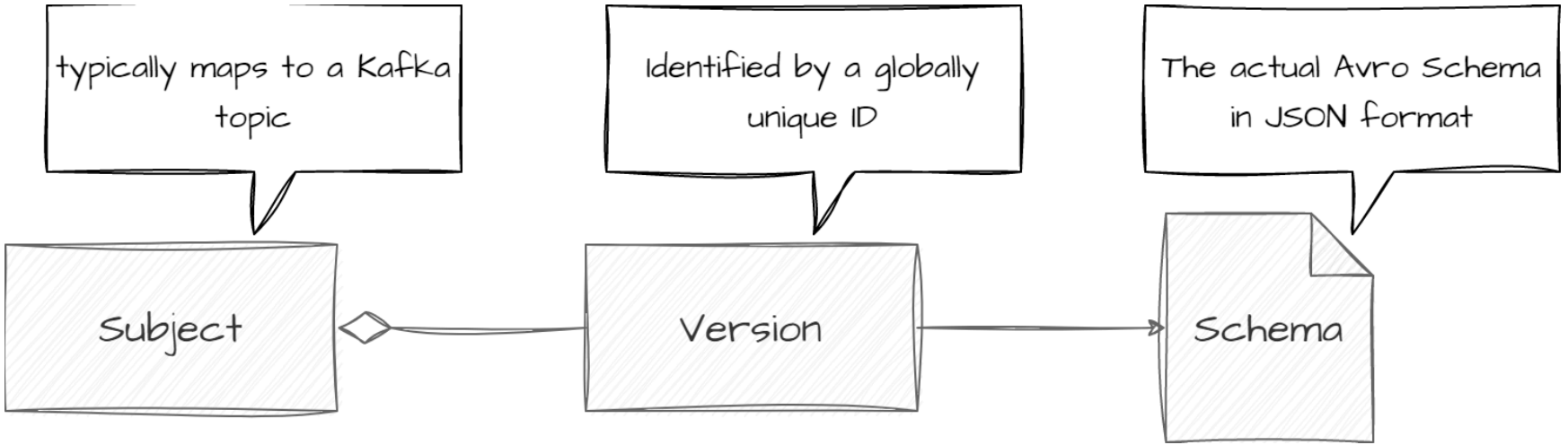
Questions?

assignment is available at

bit.ly/kafka-workshop-serialization-strategies

Confluent Schema Registry

The data model of Confluent's Schema Registry is quite simple.



What is a subject?

A subject

- is typically associated with a topic
  - `{topic-name}-key`
  - `{topic-name}-value`
- Binding between subject and topic *is not strict*
  - the ID of a version is unique across all subjects
  - possible to use the same schema for multiple topics

Confluent Schema Registry also provides a REST API with endpoints to manage schemas.

- **`GET /subjects`**: Get a list of all subjects.
- **`GET /subjects/{subject}/versions`**: Fetch all versions of the schema registered under the specified subject.
- **`GET /subjects/{subject}/versions/{version}`**: Fetch a specific version of the schema registered under the specified subject.
- **`POST /subjects/{subject}/versions`**: Register a new version of the schema under the specified subject.
- **`DELETE /subjects/{subject}/versions/{version}`**: Delete a specific version of the schema registered under the subject.