SDE with AI

Kafka and Spring Boot

Asynchronous Communication

Roadmap Day 48

Presented by Nikhil Nair

Today plan

- Warm-up quiz
- Core concepts for today
- Demo 1: Standalone Java producer and consumer
- Demo 2: Spring Boot integration and course stats projection
- Monitoring and real-world use cases

Warm-up quiz

Warm-up instructions

- Answer each question with 1, 2, 3, or 4.
- Pick the best option.

Warmup 1

What is an event in event-driven architecture?

- A. A fact that something already happened.
- B. A direct function call from one service to another.
- C. A database table that stores records.
- D. A UI screen that shows data.

Warmup 1 Answer

What is an event in event-driven architecture?

- A. A fact that something already happened.
- B. A direct function call from one service to another.
- C. A database table that stores records.
- D. A UI screen that shows data.

Correct answer: A

Why:

- Events represent facts like “Enrollment created”.
- They are not direct calls or storage tables.

Warmup 2

What happens in synchronous communication?

- A. The caller sends a message and always continues immediately.
- B. The caller waits for the response before continuing.
- C. The caller stores data and never contacts other services.
- D. The caller and callee always run in the same process.

Warmup 2 Answer

What happens in synchronous communication?

- A. The caller sends a message and always continues immediately.
- B. The caller waits for the response before continuing.
- C. The caller stores data and never contacts other services.
- D. The caller and callee always run in the same process.

Correct answer: B

Why:

- In sync calls, the caller blocks until it gets a response.
- Async messaging does not require waiting for the consumer.

Warmup 3

What is the role of a producer in Kafka?

- A. It reads messages from a topic.
- B. It tracks offsets for consumers.
- C. It writes messages to a topic.
- D. It manages user authentication.

Warmup 3 Answer

What is the role of a producer in Kafka?

- A. It reads messages from a topic.
- B. It tracks offsets for consumers.
- C. It writes messages to a topic.
- D. It manages user authentication.

Correct answer: C

Why:

- Producers publish events to Kafka topics.
- Consumers read and process those events.

Warmup 4

Why do teams use Kafka between microservices?

- A. To force services to deploy together.
- B. To guarantee exactly-once delivery always.
- C. To remove the need for databases.
- D. To publish events without waiting for other services to be online.

Warmup 4 Answer

Why do teams use Kafka between microservices?

- A. To force services to deploy together.
- B. To guarantee exactly-once delivery always.
- C. To remove the need for databases.
- D. To publish events without waiting for other services to be online.

Correct answer: D

Why:

- Producers can publish even if consumers are temporarily down.
- This reduces runtime coupling between services.

Warmup 5 Answer

In ZooKeeper-mode Kafka, ZooKeeper is mainly used for what?

- A. Coordinating brokers and storing cluster metadata.
- B. Storing business data like enrollments.
- C. Encrypting Kafka messages by default.
- D. Generating REST API endpoints.

Warmup 5 Answer

In ZooKeeper-mode Kafka, ZooKeeper is mainly used for what?

- A. Coordinating brokers and storing cluster metadata.
- B. Storing business data like enrollments.
- C. Encrypting Kafka messages by default.
- D. Generating REST API endpoints.

Correct answer: A

Why:

- ZooKeeper supports coordination and metadata in this Kafka mode.
- It is not where application data is stored.

Warmup 6 Answer

If the consumer is down, but producers keep sending messages, what happens?

- A. Messages are lost immediately.
- B. Kafka deletes the topic automatically.
- C. Messages stay in the topic and the consumer can read later.
- D. Producers stop sending automatically.

Warmup 6 Answer

If the consumer is down, but producers keep sending messages, what happens?

- A. Messages are lost immediately.
- B. Kafka deletes the topic automatically.
- C. Messages stay in the topic and the consumer can read later.
- D. Producers stop sending automatically.

Correct answer: C

Why:

- Kafka retains messages for a configured time.
- Consumers can catch up later if retention still includes those messages.

Let’s build!

What are we building today?

- Our app already stores enrollments in MongoDB as the source of truth.
- Today we publish an event to Kafka when an enrollment changes.
- A consumer builds a separate read model in MongoDB called `course_stats`.
- This makes dashboard reads fast without scanning the enrollments collection.
- We will use one Kafka topic in both demos: `guvi.events`.

What is an event?

- An event is a fact about something that already happened.
- Example: “Enrollment was created” is a fact other parts of the system can react to.
- Producers publish events without knowing who will consume them.
- Consumers subscribe and do their own work independently.
- Intuition: Events reduce direct dependencies between services.

Topic and message key

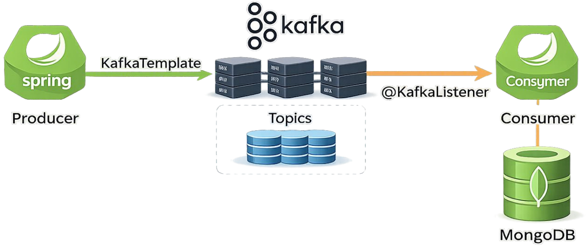
- A topic is a named stream of events stored durably by Kafka.
- A message key helps decide where an event is placed within the topic.
- Example: key = `courseId` keeps events for one course in order.
- For our live coding, we’ll name our topic `guvi.events` and our key is `courseId`.

Producer and consumer roles

- A producer writes events to a Kafka topic.
- A consumer reads events from a Kafka topic and runs business logic.
- Producers and consumers can be separate apps or separate modules.
- Example: Enrollment publishes. Stats builder consumes.

Spring Kafka in one picture

- Spring Boot produces messages using KafkaTemplate.
- Spring Boot consumes messages using @KafkaListener.
- Kafka runs separately and stores events for later consumption.
- MongoDB stores both source-of-truth data and the derived read model.



Spring Kafka config we must set

- Set `spring.kafka.bootstrap-servers` to your Kafka broker address.
- Producer must set a key serializer and a value serializer.
- Consumer must set a key deserializer and a value deserializer.
- Consumer must set a `group-id` to identify the consumer group.

Our event contract

- We publish one JSON event per enrollment action.
- Fields: eventId, type, enrollmentId, studentId, courseId, status, occurredAt.
- Example types: ENROLLMENT_CREATED, ENROLLMENT_STATUS_UPDATED, ENROLLMENT_DELETED.
- The message key is courseId to keep course events ordered.

Dealing with Duplicates

- Consumers can see the same event more than once when retries happen.
- This is normal in real distributed systems.
- We handle it using an `eventId` idempotency check before updating stats.
- Intuition: Make consumers safe to repeat.

Demo 1

Demo 1 steps

DEMO

- Confirm Kafka and ZooKeeper are running.
- Create or verify topic `guvi.events`.
- Start the Java consumer and keep it running.
- Run the Java producer and send events with multiple `courseId` values.
- Observe key, partition, and per-course counts.

```
cd $KAFKA_HOME
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic guvi.events

# Run consumer
mvn -q exec:java -Dexec.mainClass="com.guvi.playground.ConsumerApp"

# Run producer
mvn -q exec:java -Dexec.mainClass="com.guvi.playground.ProducerApp"
```

Demo 1 expected output

DEMO

- Consumer prints key and partition for each message.
- Messages with the same `courseId` appear in a consistent order.
- A running count per `courseId` increases as events arrive.

```
key=SPRING | partition=1 | count[SPRING]=4
key=JAVA   | partition=0 | count[JAVA]=3
key=SPRING | partition=1 | count[SPRING]=5
```

Demo 2

Demo 2 steps 1

DEMO

- Add Spring Kafka dependency to the project.
- Add Kafka config in `application.properties`.
- Create an `EnrollmentEvent` DTO and a producer service.
- Publish an event after create and after status update in `EnrollmentService`.

```
app.kafka.topic=guvi.events
spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.consumer.group-id=course-stats-builder-v1
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

Demo 2 steps 2

DEMO

- Create a consumer using `@KafkaListener` for topic `guvi.events`.
- Update the `course_stats` projection in MongoDB for each event.
- Add idempotency using `eventId` so duplicates do not double-count.
- Add a simple read endpoint for `course_stats`.

```
@KafkaListener(topics = "${app.kafka.topic}", groupId = "${spring.kafka.consumer.group-id}")
public void onMessage(ConsumerRecord<String, String> record) {
    // parse JSON event
    // update course_stats
}
```

Course stats data model

- `course_stats` is a derived read model, not the source of truth.
- Fields: `courseId`, `totalEnrollments`, `activeEnrollments`, `cancelledEnrollments`, `lastEventAt`.
- It is updated only by consuming Kafka events.
- Reads are fast because we do not aggregate enrollments on every request.

Demo 2 verification

DEMO

- Create or update enrollments using Postman.
- Confirm the producer publishes events by checking consumer logs.
- Confirm MongoDB contains `course_stats` documents that update.
- Call the stats endpoint and confirm values match the actions.

```
POST /api/enrollments
{
  "studentId": "<id>",
  "courseId": "<id>",
  "status": "ACTIVE"
}

GET /api/course-stats/<courseId>
```

Monitoring Kafka

Monitoring with Kafka CLI

DEMO

- Use `kafka-topics` to list and describe topics.
- Use `kafka-console-consumer` to peek at messages quickly.
- Use `kafka-consumer-groups` to inspect consumer group state when needed.

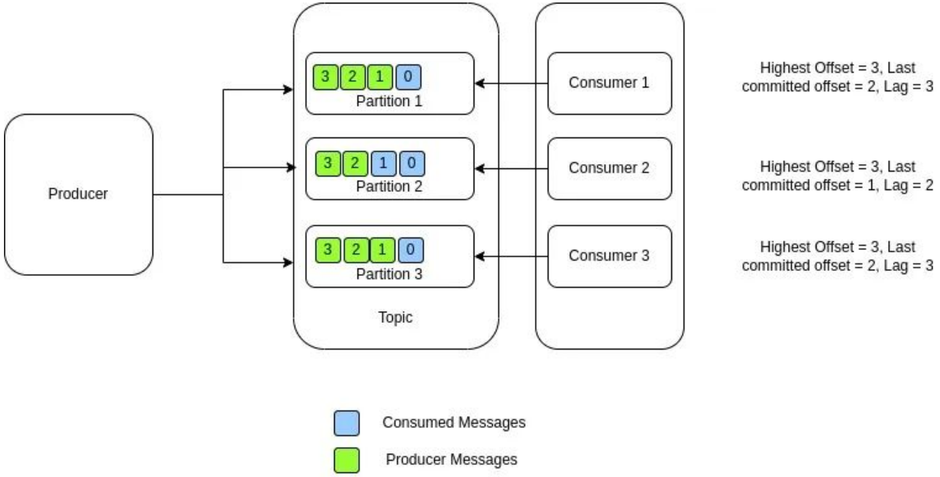
```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic guvi.events

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic guvi.events --from-beginning

bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group
course-stats-builder-v1
```

What is consumer lag?

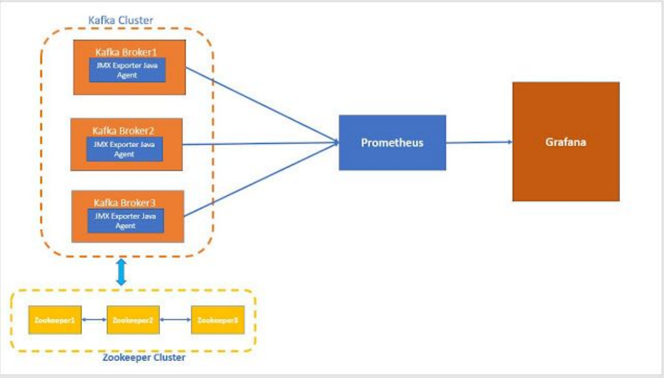
- Consumer lag is how far a consumer is behind the latest events.
- High lag means consumers are slower than producers.
- Lag is a common signal that the system is falling behind under load.
- Example: Enrollment spikes can temporarily increase lag.



Producer
3 2 1 0
Partition 1
3 2 1 0
Partition 2
3 2 1 0
Partition 3
Topic
Consumer 1
Consumer 2
Consumer 3
Highest Offset = 3, Last committed offset = 2, Lag = 3
Highest Offset = 3, Last committed offset = 1, Lag = 2
Highest Offset = 3, Last committed offset = 2, Lag = 3
Consumed Messages
Producer Messages

Metrics pipeline for Kafka

- Kafka exposes health + performance stats via JMX (e.g., requests/sec, bytes in/out, error rate).
- A JMX Exporter acts like a translator: it reads JMX metrics and exposes them in a standard HTTP /metrics format that monitoring tools can collect
- Prometheus is one such tool
    - it regularly “scrapes” that /metrics endpoint and stores the numbers over time.
- Grafana turns those metrics into dashboards + alerts (e.g., “consumer lag is growing”).



Dashboards that matter

- Producer throughput shows how many events are being published.
- Consumer throughput shows how many events are being processed.
- Consumer lag shows backlog growth or recovery.
- Error rate shows failing consumers or bad messages.

Real World Use Cases

Decoupling microservices with events

- Producers do not call consumers directly at runtime.
- Consumers can be added later without changing producers.
- Example: Enrollment publishes once; notifications and analytics consume later.
- Intuition: Async events reduce tight coupling and improve resilience.

Fan out without tight coupling

- One event can be consumed by multiple independent services.
- Each consumer can scale and deploy independently.
- Example: Enrollment events can update stats, send emails, and trigger audits.

Projections and fast reads

- A projection is a derived view built from events.
- We built `course_stats` as a projection for fast dashboard reads.
- This is useful when reads need aggregation but writes must stay simple.
- Intuition: Write events once, build many read views.

Case study: LinkedIn

- Problem: Many systems needed the same stream of user and system activity.
- Approach: Publish activity events once and let multiple consumers process them.
- Result: Teams can add new consumers without changing the producers.

Case study: Netflix

- Problem: High-volume operational and user events needed reliable pipelines.
- Approach: Use event streaming as a durable buffer between producers and processors.
- Result: Monitoring and analytics pipelines can scale independently.

Case study: Uber

- Problem: Many real-time services must react to rapidly changing events.
- Approach: Use event streams to avoid long synchronous call chains.
- Result: Systems stay responsive under spikes by decoupling work.

Wrap up

- We published enrollment events to Kafka on `guvi.events`.
- We consumed events to build `course_stats` as a MongoDB projection.
- This is a core microservices pattern for async workflows and fast reads.