SDE with AI

Event-Driven Architecture

Kafka and ZooKeeper

Roadmap Day 47

Presented by Nikhil Nair

Today Plan

- Define monolith and microservices using our Spring Boot project as the reference.
- Define synchronous and asynchronous communication with concrete API examples.
- Understand Kafka basics needed for producers, consumers, topics, offsets, and groups.
- Demo: local install and startup of ZooKeeper and Kafka, then CLI produce and consume.
- Live coding: Java producer and consumer for topic guvi.events with key:value messages.

Warm-up quiz

Warm Up Quiz

- Answer multiple choice questions. Reply format: 1 / 2 / 3 / 4
- Topics: OpenAPI, Swagger UI, Postman auth, headers, environment variables
- Goal: refresh testing fundamentals before we introduce async systems

Swagger Purpose

What is the main purpose of OpenAPI and Swagger in our backend project?

- 1) Generate UI for database tables
- 2) Document endpoints and contracts for testing
- 3) Speed up MongoDB queries
- 4) Replace Postman completely

Swagger Purpose

What is the main purpose of OpenAPI and Swagger in our backend project?

- 1) Generate UI for database tables
- 2) Document endpoints and contracts for testing
- 3) Speed up MongoDB queries
- 4) Replace Postman completely
- Correct answer: 2
- Why:
- It defines the API contract so humans and tools know what to send and what to expect.
- It reduces guesswork during testing by documenting inputs, outputs, and errors.

Swagger Artifact

When you open Swagger UI, what are you actually reading?

- 1) A live database dump
- 2) The OpenAPI specification for your endpoints
- 3) The Maven dependency tree
- 4) The Spring Security filter chain

Swagger Artifact

When you open Swagger UI, what are you actually reading?

- 1) A live database dump
- 2) The OpenAPI specification for your endpoints
- 3) The Maven dependency tree
- 4) The Spring Security filter chain
- Correct answer: 2
- Why:
- Swagger UI is a viewer that renders the OpenAPI spec into a readable page.
- The spec describes routes, request fields, response fields, and status codes.

401 in Postman

You call GET /api/courses and get 401 in Postman. What is most likely missing?

- 1) Query param size
- 2) Authorization header with credentials
- 3) Content Type header
- 4) Response body schema

401 in Postman

You call GET /api/courses and get 401 in Postman. What is most likely missing?

- 1) Query param size
- 2) Authorization header with credentials
- 3) Content Type header
- 4) Response body schema
- Correct answer: 2
- Why:
- 401 means the server did not receive valid credentials for a protected endpoint.
- Adding valid auth is required before the controller logic runs.

Basic Auth Header

In Basic Auth, what does Postman send in the request?

- 1) A JWT token in the body
- 2) Username and password encoded in the Authorization header
- 3) A session id cookie generated by MongoDB
- 4) A random API key from Swagger UI

Basic Auth Header

In Basic Auth, what does Postman send in the request?

- 1) A JWT token in the body
- 2) Username and password encoded in the Authorization header
- 3) A session id cookie generated by MongoDB
- 4) A random API key from Swagger UI
- Correct answer: 2
- Why:
- Basic Auth is transmitted through the Authorization header, not the JSON body.
- Postman formats it for you when you choose Basic Auth in the Authorization tab.

Environment Variables

Your teammate’s Postman collection works, yours fails. What should you check first?

- 1) Rename the request
- 2) Verify environment variables like url and credentials
- 3) Delete the collection and import again
- 4) Restart MongoDB

Environment Variables

Your teammate’s Postman collection works, yours fails. What should you check first?

- 1) Rename the request
- 2) Verify environment variables like url and credentials
- 3) Delete the collection and import again
- 4) Restart MongoDB
- Correct answer: 2
- Why:
- If base URL or saved credential variables are wrong, every request can fail.
- Fixing environment values is faster than changing requests.

Fast Verification

After a Postman request, what is the quickest reliable proof it worked?

- 1) The response looks long
- 2) Status code matches expectation and key fields are present
- 3) The request name is green
- 4) The console has no logs

Fast Verification

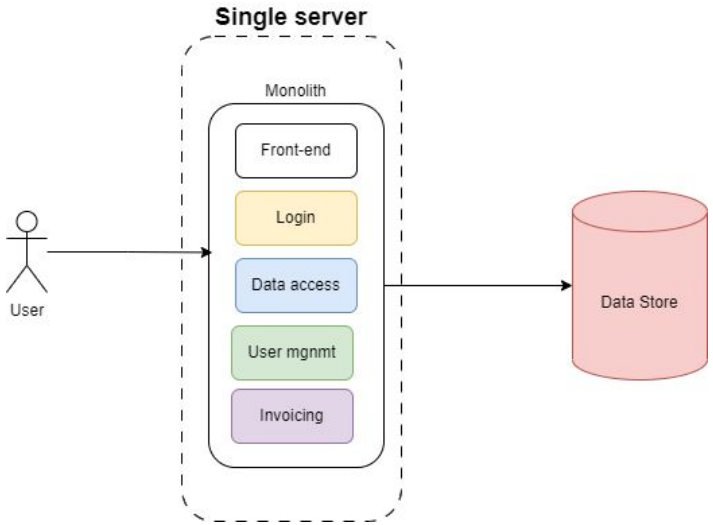
After a Postman request, what is the quickest reliable proof it worked?

- 1) The response looks long
- 2) Status code matches expectation and key fields are present
- 3) The request name is green
- 4) The console has no logs
- Correct answer: 2
- Why:
- Status code confirms the outcome and key fields confirm the response shape.
- “Looks fine” is unreliable without those two checks.

Monolith vs Microservices

What Is a Monolith

- A monolith is a single backend application that contains many features in one codebase and runs as one server process.
- Inside a monolith, features call each other using normal method calls, not network calls.
- A monolith usually uses one shared database for the whole application, so data access is centralized.
- Myth: Monoliths are the *old* way of building software.
    - Not at all - a monolith remains an effective way to build software, given project specific modifications.

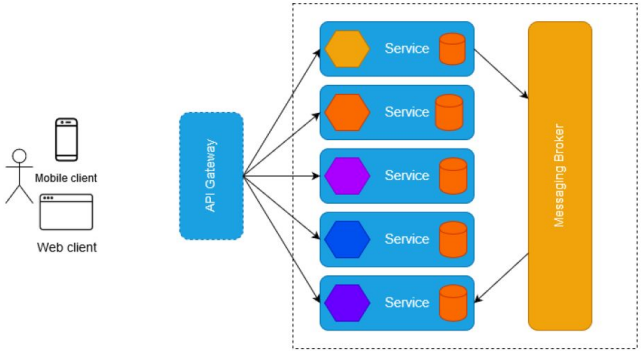


Monolith in Our Code

- One Spring Boot application starts from one main class and runs on one port.
- Multiple features live in the same server: students, courses, enrollments, and authentication.
- These features are organized as controllers, services, and repositories, but still run in the same JVM.
- When EnrollmentService needs Student info, it reads from the same database, not from a remote service.
- Note: This is why our application feels like one system. Everything is inside one running app.

What Are Microservices?

- Microservices split a system into multiple small backend applications, each focused on one capability.
- Each microservice runs as its own server process, has its own deployment, and can fail independently.
- Microservices communicate over the network, so failures and delays are normal and must be handled by design.
- A realistic split for our project: Students Service, Courses Service, Enrollments Service, Auth Service.
- Note: Microservices are not folders. They are separate applications that talk to each other.



When Microservices Help

- Multiple teams can release independently because each team owns one service boundary.
- You can scale only the hot part of the system instead of scaling everything together.
- Failure isolation improves because one service crash does not automatically kill all features.
- Integrations become easier because you can add consumers or services without rewriting the core app.
- Note: Microservices help when your system and organization grow. They are not required for every product.

When Microservices Hurt?

- More network calls means more latency and more failure cases than in-process method calls.
- Debugging becomes harder because one user action can touch many services, logs, and data stores.
- Testing becomes harder because you must test not only services, but also service-to-service failures.
- Data modeling becomes harder because you lose the simple “single database” view.
- Note: Microservices have real costs. Choose them when benefits outweigh overhead.

Synchronous vs Asynchronous

Synchronous Communication

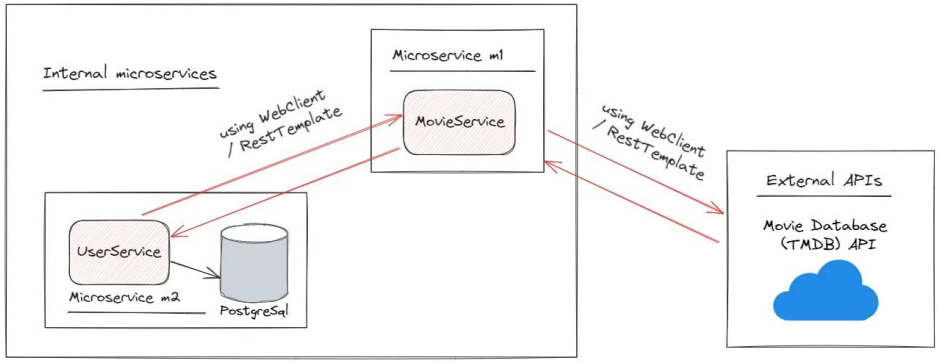
- Synchronous communication means Service A sends a request to Service B and waits for a response.
- The caller cannot finish until the callee returns success or failure, so the user experience depends on the slowest dependency.
- If the dependency is slow or down, the caller may time out or return an error, which can cascade failures.
- Example: Postman calls a protected endpoint and waits for 200 or 401 before it can move on.
- Think of it like a phone call: you ask a question and stay on the call until you hear the answer.

Asynchronous Communication

- Asynchronous communication means Service A sends a message or event and does not wait for downstream work to finish.
- The message is stored in a broker so consumers can process later, retry safely, and scale independently.
- The main request returns after saving core data, while extra work happens in the background.
- Example: enrollment API saves enrollment and publishes EnrollmentCreated, notification consumer sends email later.
- Think of it like a courier drop: you hand off the parcel and do not wait for delivery.

Use Cases for Asynchronous

- Async is useful when the main request should finish quickly and extra work can happen after core data is saved.
- Notifications: when an enrollment is completed, send an email or SMS.
- Audit logs: record important events without slowing down the API response.
- Background jobs: exports, reports, heavy computations, and sync jobs run after the response.
- Fan out: one event triggers multiple consumers such as audit, analytics, and notifications.



Sync vs Async Tradeoffs

- Sync is easier to reason about but tightly couples services and makes failures spread faster.
- Async reduces coupling and supports fan out, but introduces eventual consistency because results may arrive later.
    - Eventual consistency means that changes don’t instantly appear everywhere, but given enough time, all copies will reflect the same value. Eg: when you send a bank transfer: your account shows the deduction right away, but the recipient’s balance may take a little while to reflect the change.
- Async requires handling retries and duplicates because the same message can be delivered more than once.
- Rule of thumb: sync when the user needs an immediate answer, async when work can complete after the main transaction.
- Note: Today we learn mechanics with Kafka. Reliability patterns come in the next sessions.

Scenario Check

- Enrollment created, then send email
- Answer: async, because email can happen after enrollment is saved
- Payment status check during checkout
- Answer: sync, because user needs an immediate yes or no
- Generate a report and send a link later
- Answer: async, because it is heavy work and can complete later

Event Driven Architecture

Event Driven Architecture

- Event-driven architecture means systems publish events that represent facts, instead of calling each other for every step.
- An event is something that happened: EnrollmentCreated, CourseUpdated, PasswordChanged.
- In such an architecture, *producers* publish events without knowing *consumers*, which reduces *coupling* and makes the system easier to extend.
- Consumers subscribe and react when ready, supporting retries and scaling under load.
- Note: we'll learn about Kafka, which stores, delivers, and lets us replay these events reliably.

Event Flow Example

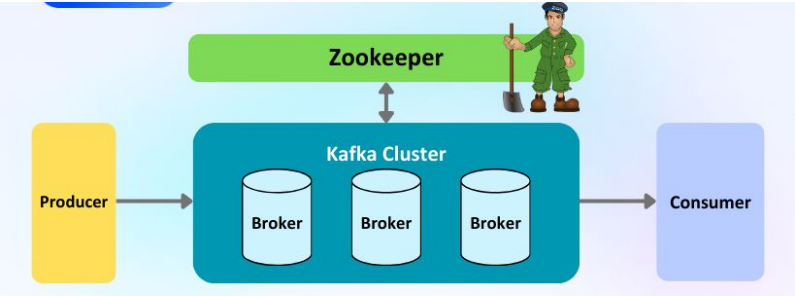
- Consider a scenario: sending a notification after successfully enrolling a student.
- How does this work in an event driven scenario?
    - Step 1: EnrollmentController saves enrollment using EnrollmentService and EnrollmentRepository.
    - Step 2: After saving, publish EnrollmentCreated to Kafka on a topic.
    - Step 3: Notification service consumes and sends email without blocking the enrollment API.
    - Step 4: Audit service consumes the same event and writes an audit log.
- Note: The API stays fast and new consumers can be added without changing the producer.

Producer and Consumer Roles

- Roles & Responsibilities
    - **Producer**: publish the right event to the right topic with enough data for consumers.
    - **Consumer**: process events safely, retry on failure, and avoid incorrect duplicates.
- Key benefit
    - Add a new consumer like analytics without changing the producer code.
- Note: Keep this mental model when we integrate Kafka into Spring Boot later.

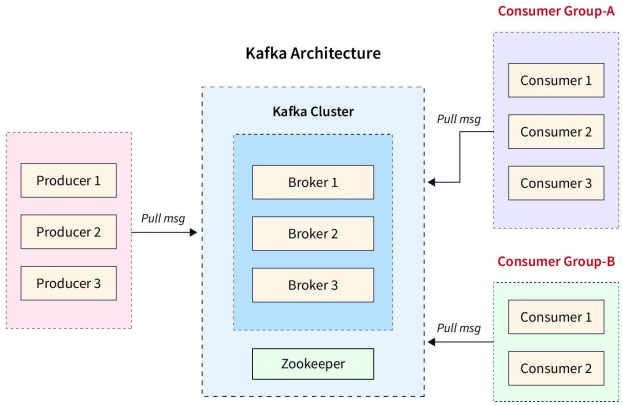
Kafka and ZooKeeper

- Kafka is a broker that stores events and delivers them to consumers using topics.
- ZooKeeper is required for today's local setup mode and must be started before Kafka.
- We will prove Kafka works using CLI tools, then write Java code that produces and consumes messages.



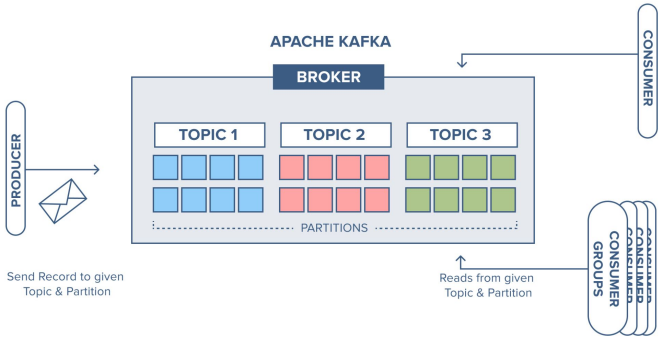
Broker and Topic

- **Broker**: the Kafka server that stores events and serves producers and consumers.
- **Topic**: a named stream of events, like guvi.events, that producers write to and consumers read from.
- Think of a topic as a shared channel name for one kind of event flow.
- Note: In our demo, producer and consumer meet through the topic, not by calling each other directly.



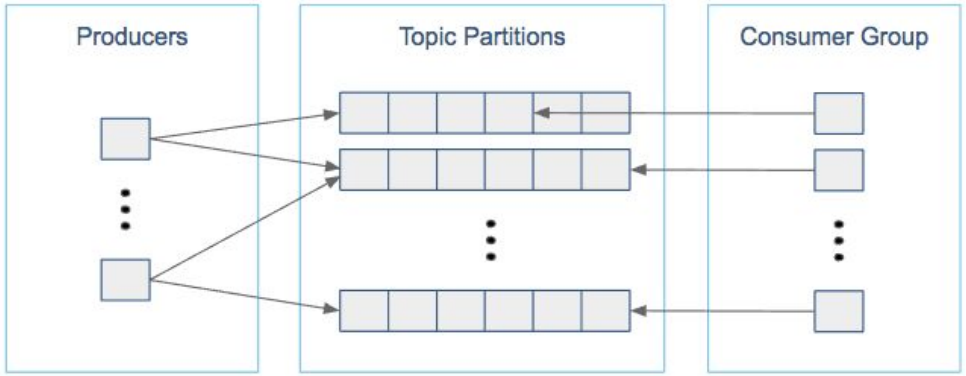
Partition, Offset, Consumer Group

- **Partition**: a topic is split into partitions so Kafka can scale and process events in parallel.
- **Offset**: each event in a partition has a position number; consumers use offsets to track progress and replay.
- **Consumer group**: a set of consumers that share work so messages are balanced across instances.
    - Different groups means fan out: each group gets its own copy of the events.
    - Microservice mapping: notification and audit are different groups reading the same topic.
- Note: These three ideas decide scaling, replay, and fan out in microservice systems.



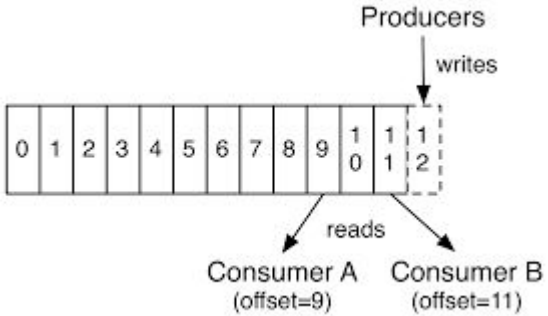
Partitions (contd.)

- **Topics** are where producers publish and consumers subscribe.
- **Partitions** allow parallel reading because different consumers can read different partitions.
- Ordering is guaranteed only within a partition, not across the whole topic.
- For today, we use one partition so behavior is predictable.



Offsets (contd.)

- Offsets enable replay and progress tracking: consumers store the last offset they processed.
- From beginning means start from the earliest offset and replay the history.
- Restarting a consumer uses offsets to continue where it left off, depending on group and config.
- Note: Offsets are why Kafka is useful for event history, not just real-time delivery.



ZooKeeper Role

- ZooKeeper stores coordination information used by Kafka in this mode.
- Kafka relies on ZooKeeper being available, which is why ZooKeeper must start first and stay running.
- If ZooKeeper stops, Kafka may fail to operate correctly or fail on restarts.
- Note: Treat ZooKeeper as a required dependency today.

Installation

Local Setup Goal

- Demo: start ZooKeeper and Kafka locally in ZooKeeper mode.
- Demo: create topic guvi.events.
- Demo: prove it works using console producer and console consumer.
- Evidence: capture one proof (topic list output or consumer output).

Install Checklist

- Java installed and java -version works.
- Kafka distribution installed or extracted on your machine.
- Two terminals ready: one for ZooKeeper and one for Kafka.
- Ports free: 2181 for ZooKeeper, 9092 for Kafka.

Start ZooKeeper

DEMO

- Demo: start ZooKeeper and keep this terminal running for the entire class.
- Success signal: process stays running and does not exit with an error.

```
# Terminal 1
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Start Kafka

DEMO

- Demo: start Kafka broker after ZooKeeper is running.
- Success signal: broker stays running and CLI commands can connect.

```
# Terminal 2
bin/kafka-server-start.sh config/server.properties
```

Create Topic

DEMO

- Demo: create topic guvi.events with one partition and replication factor one.
- Demo: list topics and confirm guvi.events exists.

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 \
+  --create --topic guvi.events --partitions 1 --replication-factor 1

# Verify
bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

Console Producer

DEMO

- Demo: start producer for guvi.events and type three key:value messages.
- Tip: use your name as the key so you can identify your messages.

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9092 \
+  --topic guvi.events \
+  --property "parse.key=true" --property "key.separator=:"

# Type messages like
nikhil:hello-1
```

Console Consumer

DEMO

- Demo: start consumer and read messages from the beginning.
- Expected: you see key and value printed for the messages you produced.

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
+  --topic guvi.events --from-beginning \
+  --property print.key=true --property key.separator=" | "
```

Setup Checklist

- Topic exists: kafka-topics --list includes guvi.events.
- Producer sends messages without error.
- Consumer prints messages, including earlier messages with from-beginning.
- Note: If you have this proof, you are ready for Java code.

Common Fixes

- Port in use: stop old ZooKeeper or Kafka processes and retry.
- Broker not available: Kafka may still be starting. Wait 10 seconds and retry.
- Consumer shows nothing: confirm topic name, use from-beginning, and send new messages.
- Topic create fails: Kafka not fully started. Retry after a short wait.

Producer & Consumer in Java

Java Producer and Consumer

- Demo: live code a Java producer that publishes key:value messages to guvi.events.
- Demo: live code a Java consumer that prints key, value, partition, and offset.
- This proves real applications can publish and consume events using Kafka clients.
- Note: We keep this separate from Spring Boot today and integrate tomorrow.

Java Config Values

- bootstrap.servers: localhost:9092
- topic: guvi.events
- group.id: demo-group
- serializers: String for key and String for value
- Note: These four values are the minimum to build a working producer and consumer.

Producer Steps

- Create producer properties with bootstrap server and String serializers.
- Create KafkaProducer and send five messages with a key and a value.
- Print publish metadata so learners can see partition and offset returned by Kafka.
- Close the producer cleanly so messages flush before exit.

Consumer Steps

- Create consumer properties with bootstrap server, group id, and String deserializers.
- Subscribe to guvi.events and poll in a loop.
- Print key, value, partition, and offset for each record.
- Note: The consumer output is our final proof that the pipeline works.

Run Order

- Start ZooKeeper first and keep it running.
- Start Kafka second and keep it running.
- Start the Java consumer so it is ready to read messages.
- Run the Java producer to publish messages into the topic.
- Note: When you see producer success and consumer output, you have full end-to-end proof.

That Is a Wrap

- You learned when async communication makes sense in microservices and what tradeoffs it introduces.
- You ran ZooKeeper and Kafka locally, created guvi.events, and verified CLI produce and consume.
- You built a Java producer and consumer that published and consumed real messages from Kafka.
- Next: integrate Kafka with Spring Boot and discuss practical patterns for reliability.