

Microservices Interview Questions – All in One Place

◆ **1. Microservices Basics**

1. What are Microservices?
 2. Monolithic vs Microservices – What's the difference?
 3. What are the main benefits of using Microservices?
 4. What are the key challenges in Microservice architecture?
 5. Explain the principle of "Single Responsibility" in Microservices.
-

◆ **2. Architecture & Design Patterns**

6. What is Domain-Driven Design (DDD) and how does it apply to Microservices?
 7. What is Bounded Context in Microservices?
 8. How do Microservices communicate with each other?
 9. What are synchronous and asynchronous communication patterns?
 10. What is the role of REST APIs in Microservices?
-

◆ **3. Spring Boot + Microservices**

11. How do you create Microservices using Spring Boot?
 12. What annotations are required to expose a REST endpoint?
 13. How do you register a Microservice with Eureka Server?
 14. How does Spring Boot simplify Microservices development?
 15. What is the role of application.yml in a Microservice?
-

◆ **4. Service Discovery (Eureka)**

16. What is Service Discovery?
17. What is Eureka Server and Eureka Client?
18. How do you register a Microservice with Eureka?

19. How does a client discover services from Eureka?

20. What are the advantages of using Eureka?

◆ **5. Load Balancing (Ribbon)**

21. What is Client-Side Load Balancing?

22. What is Ribbon and how does it work?

23. How does Ribbon choose which service instance to use?

24. Is Ribbon still supported in Spring Boot 3.x?

25. What can replace Ribbon in newer projects?

◆ **6. API Gateway (Spring Cloud Gateway)**

26. What is an API Gateway?

27. Why do we need API Gateway in Microservices?

28. What is Spring Cloud Gateway?

29. How do you route traffic using Spring Cloud Gateway?

30. How can you apply filters in Spring Cloud Gateway?

◆ **7. Configuration Management (Spring Cloud Config)**

31. What is Spring Cloud Config?

32. How does it help in managing configuration in Microservices?

33. How can you refresh configuration without restarting the service?

34. What is the role of @RefreshScope?

35. What happens if Config Server is down?

◆ **8. Fault Tolerance (Resilience4j)**

36. What is fault tolerance in Microservices?

37. What is a Circuit Breaker?

38. What are the three states of a Circuit Breaker?

-
- 39. How does Resilience4j Circuit Breaker work?
 - 40. What is a Retry mechanism in Resilience4j?
 - 41. How can you configure number of retries and delay?
 - 42. What is Rate Limiting and why is it important?
 - 43. What is Bulkhead and Time Limiter in Resilience4j?
-

- ◆ **9. Security**

- 44. How do you secure Microservices?
 - 45. What is OAuth2 and how does it help in Microservices security?
 - 46. What is JWT (JSON Web Token)?
 - 47. How does authentication and authorization work in Microservices?
 - 48. What is a centralized authentication service?
-

- ◆ **10. Database Strategies**

- 49. Should each Microservice have its own database?
 - 50. What is Database Per Service pattern?
 - 51. How do you manage data consistency across services?
 - 52. What is SAGA pattern?
-

- ◆ **11. Testing & Deployment**

- 53. How do you test Microservices?
 - 54. What is contract testing?
 - 55. How do you deploy Microservices?
 - 56. What is containerization? Why is Docker used?
 - 57. What is the role of Kubernetes in Microservices?
-

- ◆ **12. Advanced**

- 58. What is Event-Driven Architecture?

59. What is a Message Broker? Examples?

60. What is Centralized Logging? Tools?

61. How do you monitor Microservices?

62. What is API Versioning?

Kafka + Redis Roadmap (For Full-Stack/Java Developers)

◆ **Apache Kafka Topics:**

1. What is Kafka and why use it?
 2. Kafka Architecture (Broker, Topic, Partition, etc.)
 3. Producer & Consumer (with Spring Boot)
 4. Kafka Messaging Patterns (Pub-Sub)
 5. Kafka Use Cases (real-time processing)
 6. Kafka Configuration (basic setup)
 7. Kafka Offset & Consumer Groups
 8. Kafka with Spring Boot (hands-on example)
 9. Kafka vs RabbitMQ
 10. Kafka Interview Questions
-

◆ **Redis Topics:**

1. What is Redis?
 2. Redis Data Structures
 3. Why Redis is used (caching, session store)
 4. Redis Installation & CLI Basics
 5. Redis with Spring Boot
 6. Redis Cache Annotations
 7. TTL (Time to Live), Expiry, Eviction
 8. Redis vs Memcached
 9. Redis Use Cases in Microservices
 10. Redis Interview Questions
-

Shall we start with **Kafka – Topic 1: What is Kafka and why is it used?**

You said:

Shall we start with **Kafka – Topic 1: What is Kafka and why is it used?**

yes continue

ChatGPT said:

Great! Let's begin with:

Kafka – Topic 1: What is Kafka and Why Is It Used?

◆ **1. What is Apache Kafka?**

Apache Kafka is a **distributed event streaming platform** used to build **real-time data pipelines** and **streaming applications**.

It can:

- Publish
 - Subscribe
 - Store
 - Process **streams of records** in real-time
-

◆ **2. Why Use Kafka?**

Kafka is used because:

Feature	Benefit
High Throughput	Handles millions of messages/sec
Scalable	Easily add more brokers (horizontal scaling)
Durable	Messages are stored on disk and replicated
Fault-Tolerant	Keeps data even if some nodes fail
Real-time	Enables instant data flow between services

◆ **3. Where is Kafka Used in Real Life?**

Use Case	Example
 Log aggregation	Collect logs from microservices and push to central system
 Activity tracking	Track user clicks on websites or apps
 E-commerce	Order placed → inventory update → shipment → notification
 Real-time notifications	Notify users instantly (e.g., payment success/failure)
 Data streaming to DB	Process and push analytics to databases in real-time

◆ 4. Kafka Real-Life Analogy (Easy)

Think of **Kafka** as a **post office**:

- **Producer** = Sends letter (writes message)
- **Kafka Broker** = Post office (holds messages)
- **Consumer** = Reads the letter (reads message)

Messages are organized into **topics** like "orders", "logs", or "payments".

◆ 5. Core Kafka Terms

Term	Meaning
------	---------

Producer Sends messages to Kafka

Consumer Reads messages from Kafka

Broker Kafka server that stores messages

Topic Named stream (e.g., order_topic, email_topic)

Partition Sub-division of a topic for parallelism

Offset Position of a message in a partition

◆ 6. Benefits of Kafka

- High performance
 - Scalability
 - Durability
 - Fault tolerance
 - Real-time streaming
 - Works with **Spring Boot**, Flink, Spark, Hadoop, etc.
-

◆ 7. Drawbacks

- Complex to set up & manage
 - Not suitable for small/simple applications
 - Requires understanding of distributed systems
 - No message priority or built-in delay
-

◆ 8. Summary

- Kafka is a real-time, distributed messaging platform
- Used by large-scale systems like LinkedIn, Uber, Netflix
- Ideal for loosely-coupled microservices communication

Kafka – Topic 2: Kafka Architecture (Broker, Topic, Partition, etc.)

◆ 1. Kafka Architecture Overview

Apache Kafka is built around a **publish-subscribe** model, where:

- **Producers** write data (events)
- **Consumers** read data
- Kafka stores this data across **brokers** and **partitions**

Here's a diagram-style breakdown (imagine visually):

css

CopyEdit

[Producer] ---> [Kafka Broker (Topic: orders - Partition 0, 1, 2)] ---> [Consumer Group]

◆ 2. Key Components

1. Producer

- Sends messages to **topics**
- Decides which **partition** to send the data to (default: round-robin or key-based)

Example: OrderService sends messages to topic order_topic

2. Broker

- A **Kafka server** that stores topics and messages
- Cluster typically contains multiple brokers (e.g., Broker 1, Broker 2, ...)

Example: A cluster with 3 brokers can share the load

3. Topic

- Logical category to which messages are sent
- Like a "channel" or "subject"

Example:

- order_topic → order messages

- payment_topic → payment messages
-

4. Partition

- Topics are split into **partitions** for scalability
- Each partition is **ordered**, and messages inside get a unique **offset**

Example:

order_topic → Partition 0, 1, 2

Each partition stores a **subset** of the topic's messages

- More partitions = better parallelism
-

5. Offset

- Unique ID for each message in a partition
- Used to track **read progress** by the consumer

Example:

Offset 0, 1, 2 in Partition 0

Consumer remembers last read offset = 2

6. Consumer

- Subscribes to one or more topics
- Reads messages **based on offset**

Example:

InventoryService subscribes to order_topic

7. Consumer Group

- Set of consumers working **together**
- Kafka ensures **each partition** is read by only **one consumer in the group**

- Helps with **load balancing**
-

◆ 3. Architecture Example (Real-Life)

Scenario: E-commerce System

- OrderService → produces to order_topic
- BillingService → reads from order_topic
- ShippingService → also reads from order_topic in a different consumer group

Kafka allows:

- Parallel processing
 - Independent scaling
 - Loose coupling
-

◆ 4. Summary Table

Component	Description
Producer	Sends messages to Kafka topics
Broker	Kafka server, stores data
Topic	Named category of messages
Partition	Subset of topic messages (for parallelism)
Offset	Message position in partition
Consumer	Reads messages from Kafka
Consumer Group	Load-sharing group of consumers

◆ 5. Real-Life Analogy

Think of Kafka like a newspaper company:

- Reporters (Producers) write news (messages)
- News is sorted into categories (Topics)
- Sections (Partitions) organize articles
- Readers (Consumers) subscribe to topics
- Each reader gets articles without duplication

Kafka – Topic 3: Kafka Producer & Consumer with Spring Boot

◆ 1. What Are Kafka Producers and Consumers?

Producer

- A component that **sends data** (events/messages) to Kafka topics.
- You define the topic and send the payload (JSON/String/Object).

Consumer

- A component that **reads data** from Kafka topics.
 - Listens continuously and processes incoming messages.
-

◆ 2. Real-Life Analogy

 Imagine a messaging app:

- You (producer) send a message to a group (topic)
 - Group members (consumers) receive and read it
-

◆ 3. When to Use Producer/Consumer in Spring Boot?

- Microservices communication (Order → Inventory → Billing)
 - Real-time event processing
 - Notification system
 - Payment transaction status tracking
-

◆ 4. Spring Boot Kafka Project – Setup

Dependencies

In your pom.xml:

xml

CopyEdit

<dependency>

```
<groupId>org.springframework.kafka</groupId>  
<artifactId>spring-kafka</artifactId>  
</dependency>
```

application.properties

```
properties  
CopyEdit  
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id=demo_group  
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  
spring.kafka.producer.key-  
serializer=org.apache.kafka.common.serialization.StringSerializer  
spring.kafka.producer.value-  
serializer=org.apache.kafka.common.serialization.StringSerializer
```

5. Create Kafka Producer in Spring Boot

◆ **KafkaProducer.java**

```
java  
CopyEdit  
@Service  
public class KafkaProducer {  
  
    @Autowired  
    private KafkaTemplate<String, String> kafkaTemplate;
```

```
private static final String TOPIC = "order_topic";  
  
public void sendMessage(String message) {  
    kafkaTemplate.send(TOPIC, message);  
    System.out.println("Message sent: " + message);  
}  
}
```

◆ Controller to Trigger Producer

```
java  
CopyEdit  
@RestController  
@RequestMapping("/publish")  
public class KafkaController {  
  
    @Autowired  
    private KafkaProducer producer;  
  
    @PostMapping  
    public ResponseEntity<String> sendMessage(@RequestBody String message) {  
        producer.sendMessage(message);  
        return ResponseEntity.ok("Message published to Kafka topic");  
    }  
}
```

💻 6. Create Kafka Consumer in Spring Boot

◆ KafkaConsumer.java

```
java  
CopyEdit
```

```
@Service  
public class KafkaConsumer {  
  
    @KafkaListener(topics = "order_topic", groupId = "demo_group")  
    public void consume(String message) {  
        System.out.println("Consumed message: " + message);  
    }  
}
```

7. Flow Diagram (Text Format)

rust

CopyEdit

[POST /publish] --> KafkaProducer --> [Kafka Topic: order_topic] --> KafkaConsumer --> Console

8. Benefits

- **Asynchronous communication**
 - **Loose coupling between microservices**
 - **Reliable and scalable event delivery**
 - Easy to integrate with Spring Boot
-

9. Drawbacks

- Requires Kafka server to run locally
 - Additional complexity for small apps
 - Error handling & retries must be explicitly managed
-

10. Testing

1. Run Kafka server (localhost:9092)

2. Start Spring Boot app
 3. POST message to <http://localhost:8080/publish> using Postman
 4. See message printed by KafkaConsumer
-

Summary

Part	Component
------	-----------

Send	KafkaProducer
------	---------------

Receive	KafkaConsumer
---------	---------------

Topic	order_topic
-------	-------------

Spring Tool KafkaTemplate + @KafkaListener

Kafka – Topic 4: Kafka Messaging Patterns (Publish-Subscribe & More)

◆ 1. What Is a Messaging Pattern?

A **messaging pattern** is a design strategy that defines how producers and consumers communicate using messages.

Kafka mainly supports two patterns:

1. **Point-to-Point (Queue model)**
2. **Publish-Subscribe (Pub/Sub model)**

Let's understand both.

◆ 2. Point-to-Point Pattern (with Consumer Group)

- **One message → one consumer**
- Multiple consumers in a **group** but only **one** processes the message from a partition

Example:

- order_topic has 3 partitions
- Consumer Group: order_group has 3 consumers
- Each consumer gets a **different partition's** messages

Use Case: Microservices (Order → Inventory → Billing)

◆ 3. Publish-Subscribe Pattern (Multiple Consumer Groups)

- **One message → delivered to all consumer groups**
- Each group processes the message **independently**

Example:

- Producer sends message to payment_topic
- BillingService, NotificationService, and AnalyticsService (all separate consumer groups) receive **same** message

Use Case: Broadcasting events to many services

◆ 4. Kafka's Design for Both Patterns

Feature	How Kafka Handles It
Multiple consumers	Add them to the same group or different groups
Load balancing	Kafka assigns partitions across consumers
Broadcast messages	Use multiple consumer groups
Guaranteed delivery	Consumers track offset ; can reprocess if needed

💡 5. Real-Life Analogy

🎙 Think of a **radio station**:

- The DJ (producer) speaks on a channel (topic)
- Many radios (consumer groups) are tuned in
- Each radio group hears the same thing → **Pub/Sub model**

⌚ But if only one person in a team gets a task, it's **Point-to-Point**

📌 6. Summary Table

Pattern	Key Idea	Kafka Way
Point-to-Point	One message → One consumer	Consumers in same group
Publish-Subscribe	One message → All interested consumers	Consumers in different groups

✅ 7. When to Use What?

Use Case	Pattern to Use
Event Notification	Publish-Subscribe
One service hands over a task	Point-to-Point
Logging/Analytics	Publish-Subscribe

Use Case	Pattern to Use
Inventory updates	Point-to-Point

! 8. Drawbacks to Consider

Concern	Explanation
Duplication of messages	Can happen in Pub/Sub if multiple groups process
Ordering guarantee	Only within a single partition
Consumer failure	May require manual offset commit or retry

✓ Summary

Kafka supports **both messaging patterns**:

- Use **consumer groups** for **Point-to-Point**
- Use **multiple groups** for **Publish-Subscribe**

It gives you the flexibility to design systems for both **scalability** and **independent processing**.

Kafka – Topic 5: Topics, Partitions, and Offsets Deep Dive

◆ 1. What Is a Kafka Topic?

A **Kafka Topic** is a **logical channel** or **category** to which messages are published and from which messages are read.

 Think of a topic like a folder where related messages (data) are stored.

◆ Example:

- order_topic → stores all order messages
 - payment_topic → stores payment-related events
-

◆ 2. Partitions in Kafka

◆ What is a Partition?

A **partition** is a **sub-division** of a topic.

Kafka stores messages in these partitions to:

- Scale horizontally
- Ensure parallel processing
- Maintain **ordering** within a partition

 A topic can have multiple partitions, and each partition holds messages in **append-only** order.

◆ Example:

yaml

CopyEdit

Topic: order_topic

Partitions:

- Partition 0: [msg1, msg2, msg3]
 - Partition 1: [msg4, msg5]
 - Partition 2: [msg6, msg7, msg8]
-

◆ 3. Kafka Offsets

- Each message in a partition is assigned a **unique offset** (like an index).
- Consumers use these offsets to know **which message to read next**.
- Offsets are **maintained per partition, per consumer group**.

◆ Example:

sql

CopyEdit

Partition 0:

[Offset 0] "order-placed"

[Offset 1] "order-confirmed"

[Offset 2] "order-shipped"

💡 Consumers read messages **by offset**, not by ID or timestamp.

◆ 4. Topic Configuration: Partitions & Replication

When creating a topic:

shell

CopyEdit

kafka-topics.sh --create \

--topic order_topic \

--partitions 3 \

--replication-factor 2 \

--bootstrap-server localhost:9092

- --partitions 3: Allows **3-way parallel** processing
 - --replication-factor 2: Keeps a **backup** copy of each partition for fault tolerance
-

◆ 5. Benefits of Partitioning

Benefit	Explanation
Scalability	More partitions → More consumers → Higher throughput
Ordering	Kafka guarantees order within a partition
Fault Tolerance	Combined with replication, ensures durability

◆ 6. How Consumers Use Offsets

- Consumers **track** offsets to know **which message was last processed**
 - You can:
 - Let Kafka auto-manage (default)
 - Manually commit offsets (for more control)
-

🧠 Real-Life Analogy

📁 Think of a **topic as a book**, partitions as **chapters**, and offsets as **line numbers** in each chapter.
A reader (consumer) remembers the last line (offset) they read from each chapter.

⚠ 7. Drawbacks

Drawback	Description
Partition rebalancing	Adding/removing partitions can affect order
Cross-partition ordering	No guaranteed ordering between partitions
Manual offset management	Adds complexity if not using auto-commit

🔍 Summary Table

Concept Role in Kafka

Topic	Logical stream of data
Partition	Unit of parallelism & ordering

Concept Role in Kafka

Offset Position of each message in a partition

Consumer Reads messages by offset from partition(s)

Replication Provides high availability and fault tolerance

✓ Example: E-Commerce Application

Component Kafka Usage

OrderService Sends events to order_topic

order_topic Has 3 partitions for parallel order processing

BillingService Reads from all partitions via consumer group

Offset Tracking Ensures no duplication or message loss

Kafka – Topic 6: Kafka Fault Tolerance & Message Durability

◆ 1. What Is Fault Tolerance?

Fault tolerance is Kafka's ability to **keep working even when something fails**, like:

- A **broker** goes down
- A **consumer** crashes
- A **producer** loses connection

Kafka ensures **no data loss** and **minimal disruption**.

◆ 2. Key Components Supporting Fault Tolerance

◆ 1. Replication

- Each Kafka **partition** is replicated across **multiple brokers**
- One broker is **leader**, others are **followers**
- If leader fails, Kafka **automatically switches** to another replica

Example:

order_topic with 3 partitions and replication factor 2:

sql

CopyEdit

Partition 0: Leader → Broker 1, Follower → Broker 2

If Broker 1 fails → Broker 2 becomes the leader

◆ 2. Acknowledgment Settings (acks)

acks setting	What it means	Reliability
--------------	---------------	-------------

acks = 0 Producer doesn't wait for any response Fastest, but risky

acks = 1 Waits for **leader** only Balanced

acks = all Waits for **all replicas** Safest

 Use acks=all to **guarantee delivery even if a broker fails**.

◆ 3. ISR (In-Sync Replica Set)

- Only replicas that are **fully synced** with the leader are in ISR.
- Kafka **only acknowledges writes** once **all ISR replicas** have data (when `acks=all` is used).

 This ensures data won't be lost even if the leader dies suddenly.

◆ 3. Message Durability

Kafka **persists all messages** to disk before acknowledging them.

 Data is stored in logs on disk → even a broker crash **won't lose messages**.

Also:

- You can configure **retention time** (e.g., 7 days) or **retention size** (e.g., 1 GB)
 - Messages are retained **even after they're consumed**, until retention policy deletes them
-

Real-Life Analogy

Imagine writing messages in a notebook and photocopying each page for backup. Even if your notebook is stolen (broker dies), your **photocopies (replicas)** are safe.

◆ 4. Consumer Fault Tolerance

Kafka tracks **consumer offsets**, which:

- Can be auto-committed (less reliable)
- Or manually committed (safer)

If a consumer crashes:

- It can **resume from last committed offset** (not from the beginning)
-

5. Kafka Guarantees

Guarantee Type	Kafka Support
Message durability	<input checked="" type="checkbox"/> (writes to disk + replicas)
Leader failure recovery	<input checked="" type="checkbox"/> (automatic)
Message reprocessing	<input checked="" type="checkbox"/> (via offset reset)

⚠ 6. Drawbacks / Things to Manage

Concern	Notes
ISR lag	If followers are slow → they fall out of ISR
Leader election delay	Can briefly impact availability
Consumer duplication	If offset isn't committed properly, messages may be reprocessed

📌 Summary

Feature	Kafka Behavior
Replication	Maintains multiple copies of data
acks=all	Ensures data written to all in-sync replicas
Disk persistence	Messages are durable and stored on disk
Offset management	Enables recovery from consumer crash

✓ Real-Life Example Use Case

E-commerce:

Component	Kafka Behavior
OrderService	Sends order to Kafka
Replication	Ensures order isn't lost even if one broker crashes
NotificationSvc	Picks up order message and sends confirmation

Component Kafka Behavior

Crash Recovery If consumer crashes, it resumes from last offset

Kafka – Topic 7: Delivery Semantics in Kafka (At-most-once, At-least-once, Exactly-once)

◆ 1. What Are Kafka Delivery Semantics?

Kafka's **delivery semantics** define how reliably a message is delivered from **producer** → **broker** → **consumer**.

There are **3 types**:

Type	Guarantee
------	-----------

At-most-once	Message is delivered 0 or 1 time (might be lost)
--------------	---

At-least-once	Message is delivered 1 or more times (possible duplication)
---------------	--

Exactly-once	Message is delivered only once (no loss, no duplication)
--------------	---

◆ 2. At-most-once Delivery

! Messages may be **lost**, but **never duplicated**

◆ **How?**

- Producer sends the message and **does not retry** if failure occurs
- Consumer may **read but crash before processing**

◆ **Used when:**

- Occasional data loss is okay
- Performance is more important than reliability

◆ **Real-Life Example:**

- Logging metrics that don't require perfect accuracy
-

◆ 3. At-least-once Delivery

Messages are **never lost**, but **may be duplicated**

◆ **How?**

- Producer **retries sending** until broker confirms

- Consumer **processes** but may **crash before offset is committed**, so on restart
→ **reprocesses same message**

◆ **Handling Duplicates:**

- Use **idempotent operations** in consumer (process once even if received multiple times)

◆ **Real-Life Example:**

- Payment system sending confirmation messages → better to duplicate than lose
-

◆ **4. Exactly-once Delivery**

Messages are **never lost** and **never duplicated**

◆ **How?**

- Kafka supports **idempotent producers**
- Kafka and consumer coordinate using **transactional APIs**
- Offsets are committed **as part of the transaction**

◆ **Requirements:**

- Enable idempotence on producer:

java

CopyEdit

```
Properties props = new Properties();
props.put("enable.idempotence", "true");
```

- **Use Kafka Transactions:**

java

CopyEdit

```
producer.initTransactions();
producer.beginTransaction();
producer.send(...);
producer.commitTransaction();
```

◆ **Real-Life Example:**

- Banking transaction system – **no loss or duplication allowed**
-

◆ 5. Comparison Table

Feature	At-most-once	At-least-once	Exactly-once
Message loss	<input checked="" type="checkbox"/> Possible	<input type="checkbox"/> Not possible	<input type="checkbox"/> Not possible
Duplication	<input type="checkbox"/> None	<input checked="" type="checkbox"/> Possible	<input type="checkbox"/> None
Complexity	★ Low	★ ★ Medium	★ ★ ★ High
Performance	★ ★ ★ Fast	★ ★ Moderate	★ Slowest
Use case	Logging	Email/Notification	Finance, Transactions

🧠 Real-Life Analogy

Delivery Type Analogy

At-most-once Postal service that **won't retry** if no one is home

At-least-once Delivery guy **comes back** if you're not home

Exactly-once Delivery with **signature required** and a system to avoid duplicate

Summary

Concept Notes

At-most-once Fastest, but data may be lost

At-least-once Reliable, but requires duplicate handling

Exactly-once Most reliable, but complex and slower

💡 Kafka Defaults:

- By default, Kafka provides **at-least-once** semantics
- You must configure it explicitly for **exactly-once**

Kafka – Topic 8: Consumer Groups & Load Balancing

◆ 1. What Is a Kafka Consumer Group?

A **Consumer Group** is a group of **one or more consumers** working together to **consume data from a Kafka topic**.

- Each consumer in the group **gets a subset of the partitions**.
- Kafka ensures **each partition is consumed by only one consumer** in a group.

 **Why?** It enables **parallel processing** and **scalability** without overlapping.

◆ Example

You have a topic `order_topic` with 3 partitions:

- Group A has 3 consumers: C1, C2, C3
- Kafka will assign:
 - Partition 0 → C1
 - Partition 1 → C2
 - Partition 2 → C3

◆ 2. How Load Balancing Works

Kafka does **partition-based load balancing**:

Case	What Happens
Consumers < Partitions	Some consumers handle multiple partitions
Consumers = Partitions	One-to-one assignment
Consumers > Partitions	Some consumers stay idle

◆ Rebalancing

When:

- A new consumer joins/leaves the group

- A partition is added/removed

Kafka triggers **rebalancing**, reassigning partitions among consumers.

◆ 3. Offsets in Consumer Groups

Each consumer group maintains its own **offsets**:

- Offsets are stored in Kafka (`__consumer_offsets` topic)
- Let's **different consumer groups consume the same topic independently**

 Example:

- order-consumer-group: tracks order service
- analytics-group: reads same topic for analytics

They don't interfere with each other.

◆ 4. Benefits of Consumer Groups

Benefit	Description
Scalability	Add more consumers to handle more load
Fault Tolerance	If one consumer fails, Kafka redistributes partitions to others
Parallelism	Different consumers can process different partitions in parallel
Flexibility	Different apps can read same topic with different group IDs

◆ 5. Real-Life Example

Use case: E-commerce app

- **Topic:** `order_placed`
- **Groups:**
 - `billing-group`: generates bills
 - `notification-group`: sends email
 - `analytics-group`: tracks trends

Each group gets the **same data** but processes **separately**.

◆ 6. Drawbacks / Things to Handle

Challenge	Description
Rebalancing delays	During rebalance, consumers pause and may cause processing gaps
Offset commit failure	Risk of message reprocessing if offset isn't committed properly
Idle consumers	If consumers > partitions → some stay idle

◆ 7. How to Create a Consumer Group (Java)

java

CopyEdit

```
Properties props = new Properties();
props.put("group.id", "order-service-group");
props.put("enable.auto.commit", "true");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props,
keyDeserializer, valueDeserializer);
```

✓ Summary

Feature	Kafka Behavior
Group ID	Uniquely identifies a consumer group
Partition assignment	Each partition → only one consumer in group
Offset tracking	Maintained per group for independent processing
Scalability	More consumers = more parallel processing (up to number of partitions)

Kafka – Topic 9: Kafka Streams & Real-Time Processing

◆ 1. What is Kafka Streams?

Kafka Streams is a **Java library** provided by Apache Kafka to build **real-time applications** that process **data streams** directly from Kafka topics.

It lets you process data **inside your Java app** — no need for separate stream processing frameworks like Spark or Flink.

◆ 2. Why Kafka Streams?

- Process data **as it arrives** in **real-time**
 - Simplifies building **event-driven microservices**
 - Built-in support for **fault tolerance, scaling, and stateful processing**
 - No separate cluster required — runs inside your app
-

◆ 3. Core Concepts

Term	Meaning
Stream	Continuous flow of records (like an infinite list of messages)
Table	A snapshot of the latest value for a key (like a database table)
KStream	Represents a record-by-record stream of data
KTable	Represents a changelog stream (key → latest value)
Topology	Defines how data is processed from source to destination
Processor API	Gives full control over low-level stream processing

◆ 4. How Kafka Streams Works (Flow)

python

CopyEdit

Kafka Topic → KStream → Processing Logic (filter/map/groupBy) → Output Topic or DB

◆ Example Flow

1. Consume order data from Kafka topic orders
 2. Filter only “paid” orders
 3. Group by customer ID
 4. Write to another topic paid-orders-summary
-

◆ 5. Kafka Streams Code Example

java

CopyEdit

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "order-processor");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

```
StreamsBuilder builder = new StreamsBuilder();
```

```
KStream<String, String> orders = builder.stream("orders");
```

```
KStream<String, String> paidOrders = orders.filter(
    (key, value) -> value.contains("PAID")
);
```

```
paidOrders.to("paid-orders");
```

```
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

◆ 6. Features of Kafka Streams

Feature	Description
Lightweight	Runs in a simple JVM application
Fault-Tolerant	Handles failures with internal state stores and changelogs
Exactly-once	Supports exactly-once semantics with transactional processing
Stateful processing	Supports aggregation, joins, and windowing
Integration	Works with any Kafka setup; no cluster needed

◆ 7. Real-Life Example

Use case: E-commerce fraud detection

- App reads orders
 - Applies logic to detect unusual patterns
 - Flags suspicious transactions and pushes alerts to another Kafka topic
-

◆ 8. Benefits

Benefit	Description
Real-time	Instant data processing as events occur
Simple to use	Embedded in Java apps, no extra servers
Powerful API	Supports filters, maps, joins, and aggregations
Built-in scaling	Use multiple app instances for horizontal scaling

◆ 9. Drawbacks

Limitation	Notes
Java-only	Only supports Java (unlike Kafka Connect or Flink)
Limited to Kafka	Works only with Kafka topics
In-memory state limits	For large state, needs tuning or external store

Summary

Topic	Description
Kafka Streams	Java library to build real-time apps on Kafka
KStream vs KTable Stream:	All events
Use Cases	Fraud detection, real-time analytics, etc.
Runs in App	No Kafka cluster needed — just your app

Kafka – Topic 10: Kafka Connect (Integrating Kafka with External Systems)

◆ 1. What is Kafka Connect?

Kafka Connect is a **tool** in the Kafka ecosystem used to **stream data between Kafka and other systems** (like databases, file systems, cloud services).

Think of it as a **connector-based framework** that handles ingestion or export of data with **minimal code**.

◆ 2. Why Kafka Connect?

Without Kafka Connect, you'd need to write custom producers/consumers to push/pull data. Kafka Connect simplifies it using **ready-made plugins** called **Connectors**.

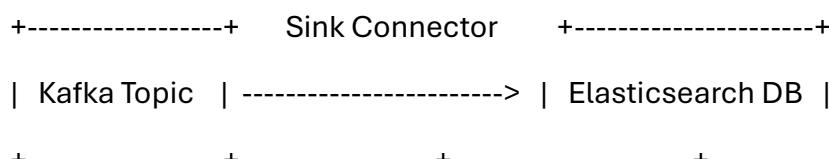
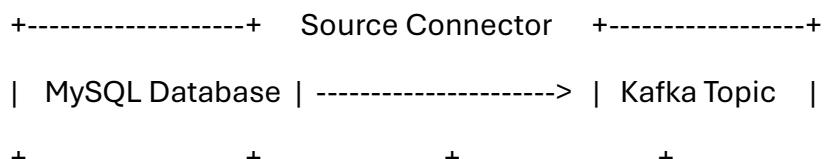
◆ 3. Types of Connectors

Type	Description	Example
Source	Pulls data from external system → Kafka topic	MySQL, MongoDB, FTP, REST API
Sink	Pushes data from Kafka topic → external system	PostgreSQL, Elasticsearch

◆ 4. How It Works (Architecture)

pgsql

CopyEdit



- You define connectors using **JSON configuration files**
 - Kafka Connect takes care of **streaming, offset tracking, retries, etc.**
-

◆ 5. Features

Feature	Description
Scalable	Can run in distributed mode or standalone for small tasks
Fault-tolerant	Tracks offsets and resumes on failure
Pluggable	Use pre-built or custom connectors
Schema support	Works well with Kafka + Schema Registry

◆ 6. Real-Life Use Cases

Scenario	Connector Type	Description
Stream data from MySQL to Kafka	Source	Sync orders in real-time
Save Kafka logs into Elasticsearch	Sink	For real-time log analytics
Send Kafka events to Google BigQuery	Sink	For business reporting
Fetch Salesforce leads into Kafka	Source	CRM + real-time lead processing

◆ 7. Example – MySQL to Kafka (Source Connector)

Source JSON Config:

json

CopyEdit

{

 "name": "mysql-source",

 "config": {

```

"connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
"connection.url": "jdbc:mysql://localhost:3306/mydb",
"connection.user": "root",
"connection.password": "pass",
"topic.prefix": "mysql-",
"mode": "incrementing",
"incrementing.column.name": "id",
"table.whitelist": "orders"
}

}

```

→ This will push new orders from MySQL to mysql-orders Kafka topic.

◆ 8. Benefits

Benefit	Why It Matters
----------------	-----------------------

No code needed Just write config files

Fault-tolerant Built-in offset tracking, retries, error handling

Reusability Hundreds of open-source connectors available

Distributed Can run as a scalable Kafka service

◆ 9. Drawbacks

Limitation	Notes
-------------------	--------------

Initial setup can be complex Requires understanding of config files and Kafka

Limited transformation logic Use Kafka Streams or SMTs (Single Message Transforms)

Debugging sometimes tricky Especially in distributed mode

◆ 10. Kafka Connect vs Kafka Streams

Feature	Kafka Streams	Kafka Connect
Language	Java API	JSON-based config
Purpose	Real-time data processing	Data integration
Works with code	Yes (Java code required)	No (Config files used)

Summary

Concept	Meaning
Kafka Connect Tool for integrating Kafka with external systems	
Connectors	Plugins that pull (source) or push (sink) data
Uses	Database sync, log export, cloud integration
Code needed?	Minimal → only configuration

Kafka Interview Questions – Complete List for Job Prep

◆ **1. What is Apache Kafka?**

- Explain its architecture (Producer → Broker → Consumer)
 - Why it's used (high-throughput, fault-tolerant messaging)
-

◆ **2. What are Kafka Topics and Partitions?**

- Topic: category/feed name for messages
 - Partition: split of a topic for parallelism
 - Each partition has its own offset
-

◆ **3. What is a Kafka Producer?**

- Sends data (events) to Kafka topics
 - Can choose partition manually or let Kafka decide (round robin or hash key)
-

◆ **4. What is a Kafka Consumer?**

- Subscribes to one or more topics
 - Processes messages and tracks offsets
 - Part of a **Consumer Group**
-

◆ **5. What is a Consumer Group?**

- A group of consumers working together to process data
 - Kafka ensures each partition is consumed by **only one consumer in the group**
-

◆ **6. What are Kafka Brokers and Clusters?**

- **Broker:** A Kafka server
- **Cluster:** Multiple brokers working together to share load

◆ **7. What is a Kafka Offset?**

- A unique ID for each message within a partition
 - Used by consumers to track what they've read
-

◆ **8. How does Kafka ensure fault tolerance?**

- Data replicated across **multiple brokers**
 - Each partition has:
 - **Leader** (handles read/write)
 - **Followers** (replicas for failover)
-

◆ **9. What is a Kafka Topic Replication Factor?**

- Number of copies of data
 - Example: RF = 3 → 3 copies across brokers
-

◆ **10. What is the difference between Kafka and traditional messaging systems?**

- Kafka is distributed, persistent, and **high-throughput**
 - Unlike RabbitMQ or JMS, Kafka **retains messages** for a configured time
-

◆ **11. What is Kafka Retention?**

- Messages are stored for **X days/hours/GB**, even after consumption
 - Controlled using log.retention.hours or log.retention.bytes
-

◆ **12. Explain the role of Zookeeper in Kafka.**

- Stores Kafka metadata (brokers, partitions)
- Coordinates leader elections
- Note: Kafka 3.x+ can run without Zookeeper (KRaft mode)

◆ 13. What is Kafka Streams?

- Java library for **real-time processing**
 - Transforms, filters, and aggregates data from Kafka topics
 - Runs **inside your app** (no cluster needed)
-

◆ 14. Difference between KStream and KTable?

KStream	KTable
All events (stream)	Latest value per key (table)
Immutable events	Mutable updates

◆ 15. What is Kafka Connect?

- Tool to **integrate Kafka with external systems** like MySQL, Elasticsearch
 - Uses **source/sink connectors**
-

◆ 16. What are Kafka Source and Sink Connectors?

- Source: External system → Kafka
 - Sink: Kafka → External system
 - Example: MySQL → Kafka → Elasticsearch
-

◆ 17. What is an ISR (In-Sync Replica)?

- Set of replicas that are fully caught up with the leader
 - Used for **reliable failover**
-

◆ 18. What is exactly-once semantics (EOS) in Kafka?

- Guarantees that a message is **processed only once**, even in failure cases
- Requires use of:

- Idempotent producers
 - Transactions
-

◆ **19. How does Kafka handle message ordering?**

- Messages are ordered **within a partition**
 - Kafka does **not guarantee order across partitions**
-

◆ **20. How can Kafka scale horizontally?**

- Add more **brokers** → More partitions
 - Add more **consumers** → Load distributed per partition
-

◆ **21. Can a topic have multiple partitions? Why?**

Yes:

- Improves **throughput** via parallelism
 - Ensures better **scalability**
-

◆ **22. What are Kafka Internal Topics?**

- Used by Kafka itself (e.g., for offset tracking, transactions)
 - Usually prefixed with __
-

◆ **23. What is a Kafka Dead Letter Queue (DLQ)?**

- A separate topic where failed or unprocessable messages are stored
 - Helps in debugging and reprocessing
-

◆ **24. How does Kafka provide durability?**

- Messages are written to **disk**
- **Replicated** across brokers

- Consumers can read even after a crash
-

◆ **25. What are some real-life Kafka use cases?**

- Fraud detection in banking
 - Real-time clickstream analysis (e-commerce)
 - Log aggregation (DevOps)
 - Order processing systems (Swiggy, Zomato)
-

 **These questions cover both:**

- Theoretical concepts
- Real-world scenarios and practical use

Redis – Roadmap for Interviews and Projects

Here's what we'll cover (only the **required** topics for full stack/backend dev):

Sr Topic	Purpose
1 What is Redis?	Basics, uses, and features
2 Data Types in Redis	Strings, Hashes, Lists, Sets, Sorted Sets
3 Redis Use Cases	Where and why to use Redis in real apps
4 Redis as Cache	Caching in Spring Boot, TTL, eviction policies
5 Redis Pub/Sub	Real-time messaging and event-based systems
6 Redis Persistence	Snapshot (RDB) and AOF explained
7 Spring Boot + Redis Integration	How to integrate and use Redis with Java apps
8 Redis Interview Questions (Final Page)	All important questions for job interviews

Let's begin with **Topic 1**.

Redis – Topic 1: What is Redis?

◆ What is Redis?

Redis (REmote DIctionary Server) is an **in-memory data store** used as:

- **Database**
- **Cache**
- **Message broker**

It stores **key-value pairs** and supports multiple data structures.

 Redis is super fast because it keeps everything in memory (RAM).

◆ Why Use Redis?

Feature	Description
In-Memory Storage	Very low latency → nanoseconds response time
Key-Value Store	Stores data in simple key-value format
Thread-safe	Single-threaded = no concurrency bugs
Fast Operations	Ideal for caching, leaderboard, session data
Supports Persistence	Data saved to disk (if needed) using RDB or AOF
Rich Data Types	Strings, Lists, Sets, Hashes, Sorted Sets, etc.

◆ How Redis Works (In Simple Words)

- Redis stores everything in **RAM** → Access speed is extremely high.
- You interact with Redis using commands like:

sql

CopyEdit

SET user:101 "Viraj"

GET user:101

◆ Real-Life Use Cases

Use Case	Description
Caching	API results, DB queries, token data
Session Store	Store login/session tokens (JWT, cookies)
Leaderboards	Gaming scores (Sorted Sets)
Real-time Chat	Use Pub/Sub feature
Analytics	Count views, likes (incr/decr on keys)

◆ Tools to Work with Redis

Tool	Use
redis-cli	Command-line interface
RedisInsight	GUI-based Redis viewer
Jedis / Lettuce	Java Redis clients (used with Spring Boot)

◆ Benefits of Redis

Benefit	Why It Matters
Super Fast	In-memory = sub-ms speed
Easy to Use	Simple commands like GET, SET, INCR
Highly Available	Supports replication and clustering
Works with All Stacks	Can be used with Java, Node, Python, etc.

◆ Drawbacks of Redis

Limitation	Why it matters
Volatile (RAM-based)	If no persistence → data can be lost
Expensive (Memory Cost)	RAM is costlier than disk
Single-threaded	Limited CPU core utilization

◆ Summary

Term	Meaning
Redis	In-memory key-value database
Speed	Extremely fast
Use Cases	Caching, sessions, counters, etc.
Tools	redis-cli, RedisInsight, Jedis

Redis – Topic 2: Data Types in Redis (With Simple Examples & Use Cases)

Redis supports several powerful **data types**, not just strings.

Each data type is optimized for specific use cases (e.g., caching, ranking, real-time updates, etc.)

◆ 1. String (Most Common)

► Description:

- Simple key-value pair.
- Can store text, numbers, JSON, even serialized objects.

► Commands:

bash

CopyEdit

SET name "Viraj"

GET name

INCR views

DECR likes

► Use Cases:

- Storing tokens, user info, session data.
 - Counter (page views, likes).
-

◆ 2. List

► Description:

- Ordered collection of string elements.
- Like a **LinkedList**.

► Commands:

bash

CopyEdit

LPUSH tasks "Learn Redis"

LPUSH tasks "Learn Spring"

LRANGE tasks 0 -1

► **Use Cases:**

- Recent activities
 - Chat messages
 - Task queues (FIFO/LIFO)
-

◆ **3. Set**

► **Description:**

- Unordered collection of unique strings.

► **Commands:**

bash

CopyEdit

SADD skills "Java"

SADD skills "Spring"

SADD skills "Java" # Duplicate won't be added

SMEMBERS skills

► **Use Cases:**

- Unique tags
 - User roles
 - Mutual friends
-

◆ **4. Sorted Set (ZSet)**

► **Description:**

- Like Set, but with a **score** → Sorted automatically.
- Each member is associated with a numeric score.

► **Commands:**

bash

CopyEdit

ZADD leaderboard 100 "Viraj"

ZADD leaderboard 200 "Neha"

ZRANGE leaderboard 0 -1 WITHSCORES

► **Use Cases:**

- Leaderboards (gaming)
 - Priority queues
 - Top 10 products, scores, etc.
-

◆ **5. Hash**

► **Description:**

- Like a Java Map (key-value inside a key).
- Stores multiple fields under one Redis key.

► **Commands:**

bash

CopyEdit

HSET user:101 name "Viraj" age "22"

HGET user:101 name

HGETALL user:101

► **Use Cases:**

- Storing user profile, product info, employee details.
-

◆ **Comparison Table**

Type	Ordered?	Duplicates?	Best Use
String	No	N/A	Token, JSON, counters
List	Yes	Yes	Logs, messages, notifications

Type	Ordered?	Duplicates?	Best Use
Set	No	No	Tags, user groups, likes
SortedSet	Yes	No	Leaderboards, scores
Hash	N/A	Field keys	Objects like users, products

◆ **Summary for Interviews**

Data Type Interview Key Point

String	Basic key-value, used in 90% of use cases
List	Ordered data – good for queues
Set	Unique values – good for tags, followers
Sorted Set	Auto-sorted scores – good for rankings
Hash	Ideal for structured objects (like Java Map)

Redis – Topic 3: Redis Use Cases in Real Applications

Redis is widely used in production systems — not just for caching but for messaging, analytics, leaderboards, and more.

◆ 1. Caching Layer (Most Popular Use Case)

◆ How it Works:

- Redis stores **frequently accessed data** (like DB results or API responses) in memory.
- This reduces DB calls and speeds up the app.

◆ Example:

java

CopyEdit

```
// Check cache first
```

```
User user = redisTemplate.get("user:101");  
if (user == null) {  
    user = db.findUserById(101);  
    redisTemplate.set("user:101", user);  
}
```

◆ Common Cache Items:

- Product lists
- Homepage data
- User sessions
- JWT Tokens

◆ Benefit:

- Faster performance
 - Lower DB load
-

◆ 2. Session Management

◆ Description:

- Store **user session data** like login tokens or preferences.

◆ Example:

bash

CopyEdit

```
SETEX session:101 1800 "JWT_TOKEN_STRING" # expires in 30 mins
```

◆ Real Use:

- OAuth session tokens
 - Cart info in e-commerce sites
-

◆ 3. Real-Time Chat & Notifications

◆ Use:

- Use Redis **Pub/Sub** to broadcast chat messages or notifications in real time.

◆ Example:

bash

CopyEdit

```
PUBLISH chat-room-1 "Viraj: Hello!"
```

```
SUBSCRIBE chat-room-1
```

◆ 4. Analytics & Counters

◆ Use:

- Count page views, clicks, likes using atomic commands like INCR.

◆ Example:

bash

CopyEdit

```
INCR page:view:product123
```

◆ **Real Use:**

- Product view counters
 - Daily visits
 - YouTube-like view count
-

◆ **5. Rate Limiting (API Security)**

◆ **Use:**

- Prevent user from hitting API too many times per second.

◆ **Example:**

bash

CopyEdit

INCR api:limit:userid123

EXPIRE api:limit:userid123 60 # resets every 60 seconds

◆ **Real Use:**

- Login attempts
 - Spam control
-

◆ **6. Leaderboard / Ranking System**

◆ **Use:**

- Use **Sorted Sets** to store scores with player names.

◆ **Example:**

bash

CopyEdit

ZADD game:scores 350 "Viraj"

ZADD game:scores 550 "Neha"

ZRANGE game:scores 0 -1 WITHSCORES

◆ 7. Queue / Task Management

◆ Use:

- Redis List works as a message queue for background processing.

◆ Example:

bash

CopyEdit

LPUSH task:email "sendEmailToUser"

RPOP task:email

◆ 8. Distributed Locking

◆ Use:

- Prevent multiple processes from updating the same resource.

◆ Example:

bash

CopyEdit

SET lock:product123 "LOCKED" NX PX 30000 # Lock for 30 seconds

◆ Summary Table of Real-Life Use Cases

Use Case	Redis Feature Used	Real-Life Example
Caching	String / TTL	Cache product list
Session Store	String + EXPIRE	User login sessions
Chat System	Pub/Sub	Real-time messaging
View Counters	String + INCR	YouTube views
Rate Limiting	String + INCR + TTL	API call limits
Leaderboard	Sorted Set	Top 10 players
Queues	List	Email background jobs

Use Case	Redis Feature Used	Real-Life Example
-----------------	---------------------------	--------------------------

Distributed Lock String + NX + PX		Locking stock update
-----------------------------------	--	----------------------

 **Real Company Usage Examples**

Company	Redis Use Case
----------------	-----------------------

Twitter	Caching trending topics
---------	-------------------------

Instagram	Session storage and counters
-----------	------------------------------

GitHub	Job queue processing for background jobs
--------	--

Netflix	Distributed caching for movie data
---------	------------------------------------

StackOverflow	Cache top questions and tags
---------------	------------------------------

Redis – Topic 4: Using Redis as a Cache in Spring Boot (with TTL & Eviction)

Redis is often used with **Spring Boot** to cache expensive operations like DB queries, API calls, etc.

This helps **improve performance** and reduce **database load**.

What is Caching?

Caching stores temporary data so that future requests are served faster without hitting the database.

Why Use Redis for Caching?

Feature	Redis Support
---------	---------------

In-memory = Very Fast	
-----------------------	---

Key-based storage	
-------------------	--

Expiry (TTL)	
--------------	---

Auto eviction	
---------------	---

Spring Boot support	 Easy with @Cacheable
---------------------	--

Step-by-Step: Redis Cache Integration in Spring Boot

Step 1: Add Dependencies (Maven)

xml

CopyEdit

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-redis</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
<groupId>org.apache.commons</groupId>
<artifactId>commons-pool2</artifactId>
</dependency>
```

Step 2: Configure Redis in application.properties

properties

CopyEdit

```
# Redis Configuration
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

Optional pool settings

```
spring.redis.jedis.pool.max-active=10
spring.redis.jedis.pool.max-idle=5
```

Step 3: Enable Caching

java

CopyEdit

```
@SpringBootApplication
@EnableCaching
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Step 4: Use @Cacheable Annotation

```

java
CopyEdit
@Service
public class ProductService {

    @Cacheable(value = "product", key = "#id")
    public Product getProductById(Long id) {
        simulateSlowService(); // simulates DB delay
        return productRepo.findById(id).orElse(null);
    }

    private void simulateSlowService() {
        try {
            Thread.sleep(3000); // 3 seconds delay
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Other Annotations:

- `@CachePut` – Updates the cache
 - `@CacheEvict` – Removes from the cache
-

TTL (Time to Live) in Redis

By default, `@Cacheable` does not expire values. To add TTL:

Option 1: Use RedisTemplate (Manual TTL)

java

CopyEdit

```
@Autowired  
private RedisTemplate<String, Object> redisTemplate;  
  
public void saveWithTTL(String key, Object value, long ttlInSeconds) {  
    redisTemplate.opsForValue().set(key, value, ttlInSeconds, TimeUnit.SECONDS);  
}
```

◆ Eviction Policies (Auto Cache Removal)

Redis evicts keys when memory is full.

Policy Behavior

noeviction Returns error when memory is full

allkeys-lru Removes least recently used keys (recommended)

volatile-ttl Removes keys with expiry first

Set policy in redis.conf:

conf

CopyEdit

maxmemory-policy allkeys-lru

◆ Real Example: Product Caching

java

CopyEdit

```
@GetMapping("/products/{id}")  
public Product getProduct(@PathVariable Long id) {  
    return productService.getProductById(id);  
}
```

First call:

- Hits DB → Stores result in Redis (3s delay)

Second call:

- Hits Redis → Response in milliseconds
-

◆ Benefits

- ✓ Faster API responses
 - ✓ Reduced load on DB
 - ✓ Supports TTL and eviction
 - ✓ Easy to implement with annotations
-

◆ Drawbacks

- ✗ If cache is too large → memory may fill
 - ✗ Redis crash = lost cache
 - ✗ Must sync data if DB is updated outside of app
-

◆ Real-Life Use

System What's Cached

Amazon Product details

Flipkart Categories, offers

Netflix User viewing history

Zomato Restaurant list

Redis – Topic 5: Redis vs In-Memory Caching (EhCache vs Caffeine vs Redis)

As a Java full-stack developer, it's important to understand **when to use Redis vs local caches** like EhCache or Caffeine.

What is In-Memory Caching?

In-memory caches store data in the **same JVM memory** as your application.

Common in-memory caching tools:

- **EhCache** (older, XML-configured)
 - **Caffeine** (modern, high-performance)
-

Redis vs In-Memory Caches – Quick Comparison

Feature	Redis	EhCache / Caffeine
Location	External (runs as a server)	Inside application JVM
Storage	Shared memory (accessible by multiple services)	App's memory (per instance)
Speed	Very fast	Super fast
Scalability	Scales horizontally (clustered)	Limited to one JVM
Persistence	Optional (RDB, AOF)	No persistence
TTL / Expiry	Yes	Yes
Eviction Policies	Yes (customizable)	Yes (LRU, LFU, etc.)
Best For	Distributed, multi-node apps	Single-node microservices

1. Use Redis When:

- You have **multiple microservices** that share the same cache
- You want **centralized control**
- You need **large cache size**

- You want to cache **login sessions, tokens, or DB queries** across the cluster
 - You want optional **data persistence**
-

◆ 2. Use EhCache or Caffeine When:

- You want a **quick, lightweight cache**
 - You are building a **monolith or single microservice**
 - No need to share cache with others
 - Simpler config without external dependencies
-

◆ Example Use Case: Product Catalog

Caffeine Example

java

CopyEdit

```
Cache<Long, Product> productCache = Caffeine.newBuilder()  
    .expireAfterWrite(10, TimeUnit.MINUTES)  
    .maximumSize(100)  
    .build();
```

```
Product product = productCache.get(id, k -> db.getProductById(k));
```

Redis Alternative

java

CopyEdit

```
@Cacheable(value = "product", key = "#id")  
  
public Product getProductById(Long id) {  
    return productRepository.findById(id).orElse(null);  
}
```

◆ Real-Life Scenarios

Scenario	Recommended Cache
Shopping site with 5 services	Redis
Small REST API with login	Caffeine / EhCache
Shared login tokens	Redis
Local development / prototype	Caffeine

◆ Performance Comparison

Operation	Redis (External)	Caffeine (In-JVM)
Read Latency	~1ms	<1ms
Write Latency	~1ms	<1ms
Startup Simplicity	Needs setup	Super simple

◆ Interview Notes Summary

Redis	In-Memory (Caffeine / EhCache)
Shared across services	Local to service
Needs separate Redis server	Built into app
Great for microservices	Great for monoliths or dev use
Supports persistence	No persistence
Slightly slower (network call)	Extremely fast (same memory)

✓ Final Advice

- Use **Redis** for:
 - Scalable, distributed, and persistent cache
 - APIs shared across multiple services
 - Rate limiting, JWT token storage, login sessions

- Use **Caffeine** for:
 - Local, high-speed, no-config caching
 - Rapid prototyping or internal caching
 - When Redis is too heavyweight

Redis – Topic 6: Interview Questions & Answers

These are **must-know** Redis interview questions with clear, easy-to-remember answers. At the end, you'll find a **summary table** for last-minute revision.

◆ 1. What is Redis?

Answer:

Redis is an **in-memory data structure store** used as a **database, cache, and message broker**. It supports strings, lists, sets, hashes, sorted sets, etc.

- ⚡ Very fast because it stores everything in RAM.
-

◆ 2. Why is Redis used in Spring Boot projects?

Answer:

Redis is used for:

- **Caching** database queries
- **Storing login sessions/tokens**
- **Rate limiting**
- **Pub/Sub messaging**

Helps reduce DB load and improves app speed.

◆ 3. Difference between Redis and in-memory caches (EhCache / Caffeine)?

Feature	Redis	Caffeine / EhCache
Scope	Shared (external)	Local (JVM only)
Scalability	High	Low
Persistence	Optional	No

Use Redis when sharing cache between services.

◆ 4. What is TTL in Redis?

Answer:

TTL = Time To Live

Redis allows keys to expire automatically after a set time.

shell

CopyEdit

SET key "value" EX 10 → expires after 10 sec

◆ **5. What are Redis eviction policies?**

Answer:

When memory is full, Redis uses eviction strategies:

- noevasion: throws error
- allkeys-lru: removes least recently used key
- volatile-ttl: removes expiring keys first

Set in redis.conf.

◆ **6. What is the use of @Cacheable in Spring?**

Answer:

It caches the result of a method call.

java

CopyEdit

@Cacheable(value = "product", key = "#id")

public Product getProductById(Long id) { ... }

Next time, Redis returns the result directly (no DB hit).

◆ **7. What is @CacheEvict and @CachePut?**

- @CacheEvict: removes the key from cache
 - @CachePut: updates the cache with the new result
-

◆ **8. How to configure Redis in Spring Boot?**

- Add dependency:

xml

CopyEdit

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- In application.properties:

properties

CopyEdit

```
spring.redis.host=localhost
spring.redis.port=6379
```

◆ 9. Can Redis store complex Java objects?

Answer:

Yes. You can use:

- RedisTemplate with custom serializers (like Jackson)
 - Or Spring's default caching with configuration
-

◆ 10. Is Redis a NoSQL DB?

Answer:

Yes. Redis is a **NoSQL, key-value store** but supports various data types like lists, sets, hashes, and more.

Summary Table for Last-Minute Review

Question	Key Point
What is Redis?	In-memory key-value store
Why Redis in Spring Boot?	Caching, token store, rate limiting

Question	Key Point
TTL?	Time-to-live for auto key expiry
Eviction Policy?	Handles memory limits (LRU, TTL etc.)
Redis vs Caffeine	Redis = shared, Caffeine = local
@Cacheable	Caches method result
@CachePut / @CacheEvict	Updates or removes cache
Redis data types	String, List, Set, Hash, SortedSet
Java Object storage	Yes, via RedisTemplate + serializers
NoSQL DB?	Yes

