# Complete Event-Driven Architecture - Beginner's Guide

\*\*A Step-by-Step Explanation of Building Production-Grade Cloud Applications\*\*

\*Author: Learning Guide for Beginners\*

\*Date: October 2025\*

\*Technologies: Apache Kafka, PostgreSQL, MinIO (S3), Java, Docker\*

---

## Table of Contents

1. [Introduction - What Did We Build?](#introduction)

2. [The Big Picture - High-Level Architecture](#big-picture)

3. [Component 1: Apache Kafka - The Message Streaming Platform](#kafka)

4. [Component 2: PostgreSQL - The Relational Database](#postgresql)

5. [Component 3: MinIO (S3) - The Blob Storage](#minio)

6. [Component 4: Java Application - The Orchestrator](#java-app)

7. [How Everything Works Together](#integration)

8. [Docker & Containerization](#docker)

9. [Deep Dive: Complete Data Flow](#data-flow)

10. [Real-World Applications](#real-world)

11. [Key Concepts for Beginners](#concepts)

12. [Troubleshooting & Common Issues](#troubleshooting)

---

<a name="introduction"></a>

## 1. Introduction - What Did We Build?

### Overview

You've just built a \*\*complete event-driven architecture\*\* that's used by billion-dollar companies like:

- \*\*Netflix\*\* - Video streaming and content delivery

- \*\*Instagram\*\* - Photo sharing and social media

- \*\*Dropbox\*\* - File storage and synchronization

- \*\*Spotify\*\* - Music streaming and recommendations

- \*\*Uber\*\* - Real-time ride tracking

- \*\*Airbnb\*\* - Property listings and bookings

### What Makes This "Production-Grade"?

1. \*\*Scalable\*\* - Can handle millions of users

2. \*\*Reliable\*\* - If one part fails, others keep working

3. \*\*Fast\*\* - Asynchronous processing means no waiting

4. \*\*Flexible\*\* - Easy to add new features without breaking existing code

5. \*\*Maintainable\*\* - Each component has a clear responsibility

### The Three Main Components

| Component | Purpose | Real-World Equivalent |

|-----------|---------|----------------------|

| \*\*Apache Kafka\*\* | Message streaming & event processing | Post office that delivers messages between services |

| \*\*PostgreSQL\*\* | Structured data storage | Filing cabinet with organized folders |

| \*\*MinIO (S3)\*\* | File/blob storage | Warehouse for storing physical items |

---

<a name="big-picture"></a>

## 2. The Big Picture - High-Level Architecture

### Visual Architecture

```

┌─────────────────────────────────────────────────────────────────┐

│ YOUR APPLICATION │

├─────────────────────────────────────────────────────────────────┤

│ │

│ USER UPLOADS FILE (e.g., "vacation-photo.jpg") │

│ │ │

│ ↓ │

│ ┌──────────────────────────────────────────────────────┐ │

│ │ STEP 1: Java Application Receives File │ │

│ │ (CompleteIntegrationDemo.java) │ │

│ └──────────────────────────────────────────────────────┘ │

│ │ │

│ ↓ │

│ ┌──────────────────────────────────────────────────────┐ │

│ │ STEP 2: Upload to MinIO (S3 Storage) │ │

│ │ - S3Manager.uploadFile() │ │

│ │ - Stores: /uploads/user/vacation-photo.jpg │ │

│ │ - Returns: http://localhost:9000/bucket/uploads/... │ │

│ └──────────────────────────────────────────────────────┘ │

│ │ │

│ ↓ │

│ ┌──────────────────────────────────────────────────────┐ │

│ │ STEP 3: Save Metadata to PostgreSQL │ │

│ │ - FileRepository.insertFileMetadata() │ │

│ │ - Table: files │ │

│ │ - Columns: id, filename, s3\_url, size, user, date │ │

│ └──────────────────────────────────────────────────────┘ │

│ │ │

│ ↓ │

│ ┌──────────────────────────────────────────────────────┐ │

│ │ STEP 4: Publish Event to Kafka │ │

│ │ - FileEventProducer.sendUploadedEvent() │ │

│ │ - Topic: "file-events" │ │

│ │ - Message: {eventType: "UPLOADED", metadata: {...}} │ │

│ └──────────────────────────────────────────────────────┘ │

│ │ │

│ ↓ │

│ ┌──────────────────────────────────────────────────────┐ │

│ │ STEP 5: Kafka Consumer Processes Event │ │

│ │ - FileEventConsumer.start() │ │

│ │ - Triggers: │ │

│ │ • Generate thumbnails │ │

│ │ • Scan for viruses │ │

│ │ • Send notifications │ │

│ │ • Update search index │ │

│ └──────────────────────────────────────────────────────┘ │

│ │

└─────────────────────────────────────────────────────────────────┘

```

### Why This Architecture?

\*\*Traditional Approach (Old Way):\*\*

```

User uploads file → Process EVERYTHING (thumbnails, scan, notify) → Response

(User waits 30 seconds!)

```

\*\*Event-Driven Approach (Modern Way):\*\*

```

User uploads file → Store file → Quick response (1 second!)

↓

Background processing happens asynchronously

```

The user gets immediate feedback, and heavy processing happens in the background!

---

<a name="kafka"></a>

## 3. Component 1: Apache Kafka - The Message Streaming Platform

### What is Kafka?

\*\*Simple Analogy\*\*: Think of Kafka as a \*\*super-efficient post office\*\* that:

- Never loses mail (messages)

- Delivers to multiple people (consumers) simultaneously

- Keeps a record of all deliveries

- Can handle millions of letters per second

### Why Do We Need Kafka?

\*\*Problem Without Kafka:\*\*

```java

// Direct function calls - tightly coupled

uploadFile(file);

generateThumbnail(file); // If this fails, everything stops!

scanVirus(file); // User waits for all of this

sendNotification(file); // Too slow!

```

\*\*Solution With Kafka:\*\*

```java

// Loosely coupled through events

uploadFile(file);

kafka.publish("file-uploaded", fileInfo); // Fire and forget!

// User gets immediate response

// Meanwhile, in the background:

// - Thumbnail service listens for "file-uploaded" events

// - Virus scanner listens for "file-uploaded" events

// - Notification service listens for "file-uploaded" events

```

### Kafka Architecture

```

┌─────────────────────────────────────────────────────┐

│ KAFKA CLUSTER │

├─────────────────────────────────────────────────────┤

│ │

│ TOPICS (Categories of Messages) │

│ ┌──────────────────────────────────────┐ │

│ │ Topic: "file-events" │ │

│ │ Partitions: 3 (for parallel processing)│ │

│ │ Replication: 1 (copies for backup) │ │

│ └──────────────────────────────────────┘ │

│ │

│ PRODUCERS (Send Messages) │

│ • FileEventProducer │

│ • Sends: File uploaded, File deleted events │

│ │

│ CONSUMERS (Receive Messages) │

│ • FileEventConsumer (Group: "file-processors") │

│ • ThumbnailGenerator (Group: "thumbnail-gen") │

│ • VirusScanner (Group: "virus-scanners") │

│ │

│ ZOOKEEPER (Manages Kafka) │

│ • Tracks which consumers are active │

│ • Manages partition assignments │

│ • Monitors broker health │

│ │

└─────────────────────────────────────────────────────┘

```

### Key Kafka Concepts

#### 1. \*\*Topics\*\*

- \*\*What\*\*: A category or feed name to which messages are published

- \*\*Example\*\*: "file-events", "user-signups", "payment-transactions"

- \*\*Real-World\*\*: Like departments in a company (HR, Finance, Engineering)

#### 2. \*\*Partitions\*\*

- \*\*What\*\*: Sub-divisions of topics for parallel processing

- \*\*Example\*\*: Topic "file-events" has 3 partitions

- Partition 0: Handles files from users A-H

- Partition 1: Handles files from users I-P

- Partition 2: Handles files from users Q-Z

- \*\*Benefit\*\*: 3 consumers can process simultaneously = 3x faster!

#### 3. \*\*Producers\*\*

- \*\*What\*\*: Applications that send messages to Kafka topics

- \*\*Our Code\*\*: `FileEventProducer.java`

- \*\*Example\*\*:

```java

// FileEventProducer.java:72

public void sendUploadedEvent(FileMetadata metadata) {

FileEvent event = new FileEvent("UPLOADED", metadata);

String jsonMessage = objectMapper.writeValueAsString(event);

ProducerRecord<String, String> record = new ProducerRecord<>(

topicName, // "file-events"

metadata.getId(), // Partition key (keeps same user's events in order)

jsonMessage // The actual message

);

producer.send(record); // Send to Kafka

}

```

#### 4. \*\*Consumers\*\*

- \*\*What\*\*: Applications that read messages from Kafka topics

- \*\*Our Code\*\*: `FileEventConsumer.java`

- \*\*Example\*\*:

```java

// FileEventConsumer.java:87

consumer.subscribe(Collections.singletonList(topicName));

while (running) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

FileEvent event = objectMapper.readValue(record.value(), FileEvent.class);

// Process the event

eventHandler.accept(event); // Your custom processing logic

}

}

```

#### 5. \*\*Consumer Groups\*\*

- \*\*What\*\*: Multiple consumers working together to process messages

- \*\*Example\*\*:

- Group "thumbnail-generators": 5 consumers generate thumbnails in parallel

- Group "virus-scanners": 3 consumers scan files in parallel

- Group "notification-senders": 2 consumers send emails in parallel

- \*\*Benefit\*\*: Each group processes ALL messages independently

```

Message: "File uploaded: photo.jpg"

↓

├──→ Consumer Group "thumbnail-generators" (5 consumers)

│ All 5 share the work, each processes different files

│

├──→ Consumer Group "virus-scanners" (3 consumers)

│ All 3 share the work, each scans different files

│

└──→ Consumer Group "notification-senders" (2 consumers)

Both process the SAME message independently

```

### Kafka Configuration in Our Project

\*\*File\*\*: `docker-compose.yml:21-51`

```yaml

kafka:

image: confluentinc/cp-kafka:7.5.0

ports:

- "9092:9092" # External access port

- "29092:29092" # Internal (Docker) access port

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS:

PLAINTEXT://kafka:29092, # For Docker containers

PLAINTEXT\_HOST://localhost:9092 # For your Java app

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE: 'true'

KAFKA\_NUM\_PARTITIONS: 3 # Default partitions per topic

```

### How Kafka Guarantees Reliability

1. \*\*Durability\*\*: Messages are written to disk, not just memory

2. \*\*Replication\*\*: Messages are copied to multiple brokers

3. \*\*Acknowledgments\*\*: Producer waits for confirmation

4. \*\*Offset Tracking\*\*: Consumers remember where they left off

5. \*\*Retry Logic\*\*: Failed sends are automatically retried

### Kafka in Our Code

\*\*Producer Configuration\*\* (`FileEventProducer.java:51`):

```java

Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, bootstrapServers);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

props.put(ProducerConfig.ACKS\_CONFIG, "all"); // Wait for all replicas

props.put(ProducerConfig.RETRIES\_CONFIG, 3); // Retry 3 times

props.put(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true"); // Prevent duplicates

```

\*\*Consumer Configuration\*\* (`FileEventConsumer.java:58`):

```java

Properties props = new Properties();

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, bootstrapServers);

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, groupId);

props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class.getName());

props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class.getName());

props.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest"); // Read from beginning

props.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "true"); // Auto-save progress

```

---

<a name="postgresql"></a>

## 4. Component 2: PostgreSQL - The Relational Database

### What is PostgreSQL?

\*\*Simple Analogy\*\*: Think of PostgreSQL as a \*\*highly organized filing cabinet\*\* where:

- Each drawer is a \*\*table\*\*

- Each file is a \*\*row\*\*

- Each label on the file is a \*\*column\*\*

- The filing system ensures everything is organized and easy to find

### Why Do We Need a Database?

\*\*What We Store in PostgreSQL:\*\*

- \*\*Metadata\*\*: Information ABOUT files (not the files themselves)

- \*\*Structured Data\*\*: Data with clear relationships

- \*\*Searchable Information\*\*: Things you need to query quickly

\*\*What We DON'T Store in PostgreSQL:\*\*

- Large files (photos, videos, documents) → Goes to S3/MinIO

- Temporary data → Goes to Redis/Memcached

- Real-time events → Goes to Kafka

### Database Schema

\*\*File\*\*: `init-db.sql:6-18`

```sql

-- Table to store file metadata

CREATE TABLE IF NOT EXISTS files (

id VARCHAR(255) PRIMARY KEY, -- Unique identifier (UUID)

filename VARCHAR(500) NOT NULL, -- Original filename

s3\_key VARCHAR(1000) NOT NULL, -- Path in S3 bucket

s3\_url TEXT NOT NULL, -- Full URL to access file

file\_size BIGINT NOT NULL, -- Size in bytes

content\_type VARCHAR(255), -- MIME type (image/jpeg, video/mp4)

uploaded\_by VARCHAR(255) NOT NULL, -- Username of uploader

uploaded\_at BIGINT NOT NULL, -- Unix timestamp

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, -- Auto-populated

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP -- Auto-updated

);

```

\*\*What Each Column Means:\*\*

| Column | Type | Purpose | Example |

|--------|------|---------|---------|

| `id` | VARCHAR(255) | Unique identifier | "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6" |

| `filename` | VARCHAR(500) | Original name | "vacation-photo.jpg" |

| `s3\_key` | VARCHAR(1000) | S3 storage path | "uploads/alice/vacation-photo.jpg" |

| `s3\_url` | TEXT | Complete URL | "http://localhost:9000/bucket/uploads/alice/vacation-photo.jpg" |

| `file\_size` | BIGINT | Size in bytes | 2621440 (2.5 MB) |

| `content\_type` | VARCHAR(255) | File type | "image/jpeg" |

| `uploaded\_by` | VARCHAR(255) | User | "alice" |

| `uploaded\_at` | BIGINT | Unix timestamp | 1729090624308 |

| `created\_at` | TIMESTAMP | Row creation time | "2025-10-16 16:37:04" |

| `updated\_at` | TIMESTAMP | Last update time | "2025-10-16 16:37:04" |

### Database Indexes

\*\*File\*\*: `init-db.sql:21-24`

```sql

-- Indexes for faster queries

CREATE INDEX IF NOT EXISTS idx\_files\_uploaded\_by ON files(uploaded\_by);

CREATE INDEX IF NOT EXISTS idx\_files\_created\_at ON files(created\_at DESC);

CREATE INDEX IF NOT EXISTS idx\_files\_content\_type ON files(content\_type);

```

\*\*Why Indexes?\*\*

Without Index:

```sql

SELECT \* FROM files WHERE uploaded\_by = 'alice';

-- Database scans ALL 1,000,000 rows → Takes 5 seconds!

```

With Index:

```sql

SELECT \* FROM files WHERE uploaded\_by = 'alice';

-- Database uses index → Finds 10 rows instantly → Takes 0.01 seconds!

```

\*\*Analogy\*\*: Like the index at the back of a textbook - instead of reading every page, you jump directly to the relevant page!

### Connection Pooling with HikariCP

\*\*What is Connection Pooling?\*\*

\*\*Without Pooling\*\* (Slow):

```

Request 1: Create DB connection → Use it → Close it

Request 2: Create DB connection → Use it → Close it

Request 3: Create DB connection → Use it → Close it

(Each connection creation takes 100ms!)

```

\*\*With Pooling\*\* (Fast):

```

Startup: Create 10 connections in advance (1 second total)

Request 1: Borrow connection 1 → Use it → Return it

Request 2: Borrow connection 2 → Use it → Return it

Request 3: Borrow connection 1 → Use it → Return it

(Each request takes 0.1ms!)

```

\*\*Our Configuration\*\* (`DatabaseManager.java:30-50`):

```java

HikariConfig config = new HikariConfig();

config.setJdbcUrl("jdbc:postgresql://localhost:5432/mydb");

config.setUsername("admin");

config.setPassword("admin123");

// Pool settings

config.setMaximumPoolSize(10); // Max 10 connections

config.setMinimumIdle(2); // Always keep 2 ready

config.setConnectionTimeout(30000); // Wait max 30s for connection

config.setIdleTimeout(600000); // Close idle connections after 10 min

config.setMaxLifetime(1800000); // Renew connections every 30 min

// Performance optimizations

config.addDataSourceProperty("cachePrepStmts", "true");

config.addDataSourceProperty("prepStmtCacheSize", "250");

config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");

dataSource = new HikariDataSource(config);

```

### CRUD Operations

\*\*File Repository\*\* (`FileRepository.java`)

#### Create (Insert)

```java

// FileRepository.java:48-76

public void insertFileMetadata(FileMetadata metadata) throws SQLException {

String sql = """

INSERT INTO files (id, filename, s3\_key, s3\_url, file\_size,

content\_type, uploaded\_by, uploaded\_at)

VALUES (?, ?, ?, ?, ?, ?, ?, ?)

""";

try (Connection conn = databaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, metadata.getId());

stmt.setString(2, metadata.getFilename());

stmt.setString(3, metadata.getS3Key());

stmt.setString(4, metadata.getS3Url());

stmt.setLong(5, metadata.getFileSize());

stmt.setString(6, metadata.getContentType());

stmt.setString(7, metadata.getUploadedBy());

stmt.setLong(8, metadata.getUploadedAt());

stmt.executeUpdate();

logger.info("Inserted file metadata: {}", metadata.getFilename());

}

}

```

#### Read (Query)

```java

// FileRepository.java:83-108

public FileMetadata getFileById(String id) throws SQLException {

String sql = "SELECT \* FROM files WHERE id = ?";

try (Connection conn = databaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, id);

ResultSet rs = stmt.executeQuery();

if (rs.next()) {

return new FileMetadata(

rs.getString("id"),

rs.getString("filename"),

rs.getString("s3\_key"),

rs.getString("s3\_url"),

rs.getLong("file\_size"),

rs.getString("content\_type"),

rs.getString("uploaded\_by"),

rs.getLong("uploaded\_at")

);

}

return null;

}

}

```

#### Update

```java

// FileRepository.java:115-132

public void updateFileMetadata(String id, FileMetadata metadata) throws SQLException {

String sql = """

UPDATE files

SET filename = ?, s3\_key = ?, s3\_url = ?, file\_size = ?,

content\_type = ?, updated\_at = CURRENT\_TIMESTAMP

WHERE id = ?

""";

try (Connection conn = databaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, metadata.getFilename());

stmt.setString(2, metadata.getS3Key());

stmt.setString(3, metadata.getS3Url());

stmt.setLong(4, metadata.getFileSize());

stmt.setString(5, metadata.getContentType());

stmt.setString(6, id);

stmt.executeUpdate();

}

}

```

#### Delete

```java

// FileRepository.java:139-153

public void deleteFileMetadata(String id) throws SQLException {

String sql = "DELETE FROM files WHERE id = ?";

try (Connection conn = databaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, id);

int rowsAffected = stmt.executeUpdate();

if (rowsAffected > 0) {

logger.info("Deleted file metadata: {}", id);

}

}

}

```

### Transactions

\*\*What is a Transaction?\*\*

A transaction ensures that \*\*all operations succeed together, or all fail together\*\*.

\*\*Example Without Transaction:\*\*

```java

// BAD: If step 2 fails, step 1 is already done!

updateUserBalance(userId, -100); // Deduct $100

updateMerchantBalance(merchantId, +100); // This fails!

// User lost $100, merchant didn't get it! 💸

```

\*\*Example With Transaction:\*\*

```java

// GOOD: Both succeed or both fail

try (Connection conn = dataSource.getConnection()) {

conn.setAutoCommit(false); // Start transaction

updateUserBalance(conn, userId, -100);

updateMerchantBalance(conn, merchantId, +100);

conn.commit(); // Both succeeded! ✅

} catch (Exception e) {

conn.rollback(); // Undo everything! ↩️

}

```

### PostgreSQL Configuration

\*\*File\*\*: `docker-compose.yml:89-104`

```yaml

postgres:

image: postgres:16-alpine

container\_name: postgres-db

environment:

POSTGRES\_USER: admin # Username

POSTGRES\_PASSWORD: admin123 # Password

POSTGRES\_DB: mydb # Database name

ports:

- "5432:5432" # Port mapping

volumes:

- postgres-data:/var/lib/postgresql/data # Persistent storage

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U admin -d mydb"]

interval: 10s

timeout: 5s

retries: 5

```

### Common SQL Queries

\*\*Query 1: Get all files uploaded by a user\*\*

```sql

SELECT \* FROM files

WHERE uploaded\_by = 'alice'

ORDER BY created\_at DESC

LIMIT 10;

```

\*\*Query 2: Get total storage used by user\*\*

```sql

SELECT uploaded\_by, SUM(file\_size) as total\_bytes

FROM files

GROUP BY uploaded\_by;

```

\*\*Query 3: Find large files\*\*

```sql

SELECT filename, file\_size, uploaded\_by

FROM files

WHERE file\_size > 10485760 -- 10 MB

ORDER BY file\_size DESC;

```

\*\*Query 4: Get file statistics by type\*\*

```sql

SELECT

content\_type,

COUNT(\*) as file\_count,

SUM(file\_size) as total\_size,

AVG(file\_size) as avg\_size

FROM files

GROUP BY content\_type;

```

---

<a name="minio"></a>

## 5. Component 3: MinIO (S3) - The Blob Storage

### What is MinIO?

\*\*Simple Analogy\*\*: Think of MinIO as a \*\*massive warehouse\*\* where:

- Each item (file) has a unique barcode (S3 key)

- Items are organized in sections (buckets)

- You can retrieve any item instantly using its barcode

- The warehouse is virtually unlimited in size

\*\*MinIO vs AWS S3:\*\*

- \*\*AWS S3\*\*: Cloud storage service by Amazon

- \*\*MinIO\*\*: Open-source, self-hosted, S3-compatible storage

- \*\*API\*\*: MinIO uses the EXACT same API as AWS S3

- \*\*Benefit\*\*: Develop locally with MinIO, deploy to AWS S3 with zero code changes!

### Why Separate File Storage from Database?

\*\*Storing Files in Database\*\* (Bad Idea ❌):

```

Database size: 100 GB

- User data: 1 GB

- Files: 99 GB (images, videos, documents)

Problems:

❌ Slow queries (database scans huge files)

❌ Expensive (database storage costs 10x more)

❌ Backups take forever

❌ Can't scale independently

```

\*\*Storing Files in S3/MinIO\*\* (Good Idea ✅):

```

Database: 1 GB (just metadata)

S3 Storage: 99 GB (actual files)

Benefits:

✅ Fast queries (database only scans metadata)

✅ Cheap (S3 costs 10x less than database storage)

✅ Quick backups

✅ Scales independently

```

### S3 Architecture Concepts

#### 1. \*\*Buckets\*\*

- \*\*What\*\*: Top-level containers for storing objects

- \*\*Example\*\*: "user-uploads", "profile-pictures", "video-content"

- \*\*Analogy\*\*: Like a hard drive or USB stick

- \*\*Our Bucket\*\*: "complete-integration-bucket"

#### 2. \*\*Objects (Files)\*\*

- \*\*What\*\*: The actual files you store

- \*\*Components\*\*:

- \*\*Key\*\*: Unique identifier/path (e.g., "uploads/alice/photo.jpg")

- \*\*Data\*\*: The file content

- \*\*Metadata\*\*: Additional info (content-type, size, etc.)

#### 3. \*\*Keys (File Paths)\*\*

- \*\*What\*\*: The "address" of an object in a bucket

- \*\*Format\*\*: Like a file system path

- \*\*Examples\*\*:

- `uploads/alice/vacation-photo.jpg`

- `documents/2025/report.pdf`

- `videos/tutorial-01.mp4`

#### 4. \*\*URLs\*\*

- \*\*What\*\*: Web address to access the file

- \*\*Format\*\*: `http://endpoint/bucket-name/key`

- \*\*Example\*\*: `http://localhost:9000/complete-integration-bucket/uploads/alice/photo.jpg`

### MinIO Configuration

\*\*File\*\*: `docker-compose.yml:120-136`

```yaml

minio:

image: minio/minio:latest

container\_name: minio-s3

command: server /data --console-address ":9001"

environment:

MINIO\_ROOT\_USER: minioadmin # Access key

MINIO\_ROOT\_PASSWORD: minioadmin123 # Secret key

ports:

- "9000:9000" # API port (S3 operations)

- "9001:9001" # Web console port

volumes:

- minio-data:/data # Persistent storage

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]

interval: 10s

timeout: 5s

retries: 5

```

### S3Manager - Java Implementation

\*\*File\*\*: `S3Manager.java`

#### Initialization

```java

// S3Manager.java:33-55

public S3Manager(String endpoint, String accessKey, String secretKey) {

// Configure AWS SDK for MinIO

AwsBasicCredentials credentials = AwsBasicCredentials.create(accessKey, secretKey);

this.s3Client = S3Client.builder()

.endpointOverride(URI.create(endpoint)) // Point to MinIO instead of AWS

.region(Region.US\_EAST\_1)

.credentialsProvider(StaticCredentialsProvider.create(credentials))

.serviceConfiguration(S3Configuration.builder()

.pathStyleAccessEnabled(true) // Required for MinIO

.build())

.build();

}

```

#### Create Bucket

```java

// S3Manager.java:62-77

public void createBucketIfNotExists(String bucketName) {

try {

// Check if bucket exists

s3Client.headBucket(HeadBucketRequest.builder()

.bucket(bucketName)

.build());

logger.info("Bucket {} already exists", bucketName);

} catch (NoSuchBucketException e) {

// Bucket doesn't exist, create it

s3Client.createBucket(CreateBucketRequest.builder()

.bucket(bucketName)

.build());

logger.info("Created bucket: {}", bucketName);

}

}

```

#### Upload File

```java

// S3Manager.java:90-119

public String uploadFile(String bucketName, String key, byte[] fileData, String contentType) {

try {

// Upload file to S3

PutObjectRequest request = PutObjectRequest.builder()

.bucket(bucketName)

.key(key)

.contentType(contentType)

.contentLength((long) fileData.length)

.build();

s3Client.putObject(request, RequestBody.fromBytes(fileData));

// Generate URL

String url = String.format("%s/%s/%s",

s3Client.serviceClientConfiguration().endpointOverride().get(),

bucketName,

key);

logger.info("Uploaded file to S3: {} ({})", key, formatFileSize(fileData.length));

return url;

} catch (Exception e) {

logger.error("Failed to upload file to S3: {}", key, e);

throw new RuntimeException("S3 upload failed", e);

}

}

```

\*\*What Happens When You Upload:\*\*

1. File is divided into chunks (for large files)

2. Each chunk is uploaded separately

3. MinIO reassembles chunks on the server

4. Returns a URL to access the file

5. File is immediately available worldwide

#### Download File

```java

// S3Manager.java:128-147

public byte[] downloadFile(String bucketName, String key) {

try {

GetObjectRequest request = GetObjectRequest.builder()

.bucket(bucketName)

.key(key)

.build();

ResponseInputStream<GetObjectResponse> response = s3Client.getObject(request);

byte[] fileData = response.readAllBytes();

logger.info("Downloaded file from S3: {} ({})", key, formatFileSize(fileData.length));

return fileData;

} catch (NoSuchKeyException e) {

logger.error("File not found in S3: {}", key);

throw new RuntimeException("File not found", e);

}

}

```

#### Delete File

```java

// S3Manager.java:154-169

public void deleteFile(String bucketName, String key) {

try {

DeleteObjectRequest request = DeleteObjectRequest.builder()

.bucket(bucketName)

.key(key)

.build();

s3Client.deleteObject(request);

logger.info("Deleted file from S3: {}", key);

} catch (Exception e) {

logger.error("Failed to delete file from S3: {}", key, e);

throw new RuntimeException("S3 delete failed", e);

}

}

```

#### List Files

```java

// S3Manager.java:176-201

public List<String> listFiles(String bucketName, String prefix) {

List<String> fileKeys = new ArrayList<>();

try {

ListObjectsV2Request request = ListObjectsV2Request.builder()

.bucket(bucketName)

.prefix(prefix)

.build();

ListObjectsV2Response response = s3Client.listObjectsV2(request);

for (S3Object object : response.contents()) {

fileKeys.add(object.key());

}

logger.info("Listed {} files with prefix: {}", fileKeys.size(), prefix);

return fileKeys;

} catch (Exception e) {

logger.error("Failed to list files from S3", e);

throw new RuntimeException("S3 list failed", e);

}

}

```

### File Organization Strategy

\*\*Good File Organization:\*\*

```

bucket: complete-integration-bucket

├── uploads/

│ ├── alice/

│ │ ├── vacation-photo-1.jpg

│ │ ├── vacation-photo-2.jpg

│ │ └── document.pdf

│ ├── bob/

│ │ ├── profile-pic.png

│ │ └── resume.pdf

│ └── carol/

│ └── presentation.pptx

├── thumbnails/

│ ├── alice/

│ │ ├── vacation-photo-1\_thumb.jpg

│ │ └── vacation-photo-2\_thumb.jpg

│ └── bob/

│ └── profile-pic\_thumb.png

└── processed/

├── alice/

│ └── vacation-photo-1\_filtered.jpg

└── bob/

└── profile-pic\_cropped.png

```

\*\*Benefits:\*\*

- Easy to find user's files

- Can set permissions per folder

- Simple to backup specific users

- Clear separation of file types

### S3 Best Practices

#### 1. \*\*Unique Keys\*\*

```java

// BAD: Filename collisions

String key = "uploads/" + filename; // Multiple users upload "photo.jpg"

// GOOD: Use UUID or timestamp

String key = "uploads/" + userId + "/" + UUID.randomUUID() + "\_" + filename;

```

#### 2. \*\*Content-Type\*\*

```java

// Set proper content-type for browsers to handle correctly

String contentType = "image/jpeg"; // Browser displays image

String contentType = "video/mp4"; // Browser plays video

String contentType = "application/pdf"; // Browser shows PDF

```

#### 3. \*\*File Size Limits\*\*

```java

// Check file size before upload

long MAX\_FILE\_SIZE = 100 \* 1024 \* 1024; // 100 MB

if (fileData.length > MAX\_FILE\_SIZE) {

throw new IllegalArgumentException("File too large!");

}

```

#### 4. \*\*Error Handling\*\*

```java

try {

s3Manager.uploadFile(bucket, key, data, contentType);

} catch (S3Exception e) {

if (e.statusCode() == 403) {

// Permission denied

} else if (e.statusCode() == 404) {

// Bucket not found

} else if (e.statusCode() == 503) {

// Service unavailable, retry

}

}

```

### S3 vs Database: When to Use What?

| Data Type | Store In | Reason |

|-----------|----------|--------|

| Images | S3 | Large, binary, served directly to users |

| Videos | S3 | Very large, streaming required |

| Documents (PDF, DOCX) | S3 | Large, downloadable |

| User profiles | Database | Small, structured, frequently queried |

| Comments | Database | Small, structured, searchable |

| Thumbnails | S3 | Images, but smaller versions |

| Log files | S3 | Large, infrequently accessed |

| Session data | Redis/Cache | Temporary, needs fast access |

| Configurations | Database | Small, structured, version controlled |

---

<a name="java-app"></a>

## 6. Component 4: Java Application - The Orchestrator

### Application Architecture

```

┌─────────────────────────────────────────────────────┐

│ JAVA APPLICATION LAYER │

├─────────────────────────────────────────────────────┤

│ │

│ CONFIG LAYER │

│ • KafkaConfig.java - Loads configurations │

│ • application.properties - App settings │

│ • kafka.properties - Kafka settings │

│ │

│ MODEL LAYER (Data Structures) │

│ • Message.java - Kafka message model │

│ • FileMetadata.java - File information │

│ • FileEvent.java - Event payload │

│ │

│ STORAGE LAYER (S3) │

│ • S3Manager.java - MinIO operations │

│ - uploadFile(), downloadFile(), deleteFile() │

│ │

│ DATABASE LAYER (PostgreSQL) │

│ • DatabaseManager.java - Connection pooling │

│ • FileRepository.java - File metadata CRUD │

│ • MessageRepository.java - Message CRUD │

│ │

│ KAFKA LAYER (Messaging) │

│ • KafkaProducer.java - Send messages │

│ • KafkaConsumer.java - Receive messages │

│ • FileEventProducer.java - Send file events │

│ • FileEventConsumer.java - Receive file events │

│ │

│ APPLICATION LAYER (Main Programs) │

│ • CompleteIntegrationDemo.java - Full demo │

│ • S3IntegrationDemo.java - S3 demo │

│ • KafkaDatabaseIntegration.java - Kafka+DB demo │

│ • DatabaseDemo.java - DB demo │

│ • KafkaApplication.java - Basic Kafka demo │

│ │

└─────────────────────────────────────────────────────┘

```

### Complete Integration Flow

\*\*File\*\*: `CompleteIntegrationDemo.java`

#### Main Method

```java

// CompleteIntegrationDemo.java:56-78

public static void main(String[] args) {

logger.info("=".repeat(70));

logger.info(" COMPLETE S3 + KAFKA + DATABASE INTEGRATION DEMO");

logger.info("=".repeat(70));

CompleteIntegrationDemo demo = new CompleteIntegrationDemo();

try {

// Initialize all components

demo.initializeComponents();

// Start consumer in background

demo.startConsumer();

// Upload sample files

demo.uploadSampleFiles();

// Wait for processing

Thread.sleep(5000);

// Show statistics

demo.displayStatistics();

} finally {

demo.cleanup();

}

}

```

#### Component Initialization

```java

// CompleteIntegrationDemo.java:85-115

private void initializeComponents() {

logger.info("\n" + "=".repeat(70));

logger.info(" INITIALIZING COMPONENTS");

logger.info("=".repeat(70));

// 1. Initialize S3 (MinIO)

logger.info("\n[1/4] Initializing S3 Storage (MinIO)...");

s3Manager = new S3Manager(

"http://localhost:9000",

"minioadmin",

"minioadmin123"

);

s3Manager.createBucketIfNotExists(BUCKET\_NAME);

// 2. Initialize Database

logger.info("[2/4] Initializing PostgreSQL Database...");

databaseManager = new DatabaseManager(

"jdbc:postgresql://localhost:5432/mydb",

"admin",

"admin123"

);

fileRepository = new FileRepository(databaseManager);

// 3. Initialize Kafka Producer

logger.info("[3/4] Initializing Kafka Producer...");

eventProducer = new FileEventProducer(

"localhost:9092",

"file-events"

);

// 4. Initialize Kafka Consumer

logger.info("[4/4] Initializing Kafka Consumer...");

eventConsumer = new FileEventConsumer(

"localhost:9092",

"complete-integration-group",

"file-events"

);

logger.info("\n✅ All components initialized successfully!\n");

}

```

#### File Upload Process

```java

// CompleteIntegrationDemo.java:156-214

private void uploadFile(String filename, String contentType) {

try {

logger.info("\n" + "─".repeat(70));

logger.info("📤 UPLOADING: {}", filename);

logger.info("─".repeat(70));

// STEP 1: Create sample file data

byte[] fileData = ("Sample content for " + filename).getBytes();

String fileId = UUID.randomUUID().toString();

String s3Key = "uploads/" + DEMO\_USER + "/" + filename;

logger.info(" [1/4] Created file: {} ({} bytes)", filename, fileData.length);

// STEP 2: Upload to S3

logger.info(" [2/4] Uploading to S3...");

String s3Url = s3Manager.uploadFile(BUCKET\_NAME, s3Key, fileData, contentType);

logger.info(" ✅ S3 URL: {}", s3Url);

// STEP 3: Create metadata object

FileMetadata metadata = new FileMetadata(

fileId,

filename,

s3Key,

s3Url,

(long) fileData.length,

contentType,

DEMO\_USER,

System.currentTimeMillis()

);

// STEP 4: Save metadata to database

logger.info(" [3/4] Saving metadata to PostgreSQL...");

fileRepository.insertFileMetadata(metadata);

logger.info(" ✅ Saved to database with ID: {}", fileId);

// STEP 5: Send event to Kafka

logger.info(" [4/4] Publishing event to Kafka...");

eventProducer.sendUploadedEvent(metadata);

logger.info(" ✅ Event published to topic: file-events");

logger.info("\n✅ File upload complete: {}", filename);

filesUploaded.incrementAndGet();

} catch (Exception e) {

logger.error("❌ Failed to upload file: {}", filename, e);

}

}

```

#### Consumer Event Handler

```java

// CompleteIntegrationDemo.java:125-147

private void startConsumer() {

logger.info("\n" + "=".repeat(70));

logger.info(" STARTING EVENT CONSUMER");

logger.info("=".repeat(70));

// Define event handler

Consumer<FileEvent> eventHandler = event -> {

try {

logger.info("\n" + "▬".repeat(70));

logger.info("📥 RECEIVED EVENT: {}", event.getEventType());

logger.info("▬".repeat(70));

FileMetadata metadata = event.getFileMetadata();

logger.info(" • Filename: {}", metadata.getFilename());

logger.info(" • Size: {}", formatFileSize(metadata.getFileSize()));

logger.info(" • User: {}", metadata.getUploadedBy());

logger.info(" • S3 URL: {}", metadata.getS3Url());

// Simulate processing

logger.info("\n 🔄 Processing tasks:");

logger.info(" [1] Generating thumbnails...");

logger.info(" [2] Scanning for viruses...");

logger.info(" [3] Updating search index...");

logger.info(" [4] Sending notifications...");

Thread.sleep(100); // Simulate work

logger.info("\n ✅ Processing complete!");

eventsProcessed.incrementAndGet();

} catch (Exception e) {

logger.error("❌ Error processing event", e);

}

};

// Start consumer in background thread

new Thread(() -> eventConsumer.start(eventHandler)).start();

logger.info("✅ Consumer started and listening for events...\n");

}

```

### Project Structure Explained

```

src/main/java/com/example/kafka/

│

├── config/

│ └── KafkaConfig.java

│ • Loads kafka.properties

│ • Provides configuration to producers/consumers

│ • Validates settings

│

├── model/

│ ├── Message.java

│ │ • Simple message with id, content, timestamp

│ │ • Used in basic Kafka examples

│ ├── FileMetadata.java

│ │ • File information (id, name, size, user, etc.)

│ │ • Stored in PostgreSQL

│ │ • Sent in Kafka events

│ └── FileEvent.java

│ • Event wrapper (eventType, fileMetadata, timestamp)

│ • JSON serialized for Kafka

│

├── producer/

│ └── KafkaProducer.java

│ • Generic Kafka producer

│ • Sends Message objects to topics

│ • Used in basic examples

│

├── consumer/

│ └── KafkaConsumer.java

│ • Generic Kafka consumer

│ • Receives Message objects from topics

│ • Used in basic examples

│

├── db/

│ ├── DatabaseManager.java

│ │ • HikariCP connection pool setup

│ │ • Manages database connections

│ │ • Health checks and cleanup

│ ├── FileRepository.java

│ │ • CRUD operations for files table

│ │ • Insert, select, update, delete file metadata

│ └── MessageRepository.java

│ • CRUD operations for messages table

│ • Used in basic examples

│

├── storage/

│ └── S3Manager.java

│ • MinIO/S3 client wrapper

│ • Upload, download, delete, list files

│ • Bucket management

│

├── kafka/

│ ├── FileEventProducer.java

│ │ • Specialized producer for file events

│ │ • sendUploadedEvent(), sendDeletedEvent()

│ │ • JSON serialization

│ └── FileEventConsumer.java

│ • Specialized consumer for file events

│ • Deserializes FileEvent objects

│ • Calls custom event handlers

│

└── [Demo Applications]

├── CompleteIntegrationDemo.java

│ • Full S3 + Kafka + DB integration

│ • Shows complete workflow

├── S3IntegrationDemo.java

│ • S3 + DB integration

│ • File upload and metadata storage

├── KafkaDatabaseIntegration.java

│ • Kafka + DB integration

│ • Messages stored in database

├── DatabaseDemo.java

│ • Database operations demo

│ • Connection pooling example

└── KafkaApplication.java

• Basic Kafka producer/consumer demo

• Interactive mode

```

### Maven Dependencies

\*\*File\*\*: `pom.xml`

```xml

<dependencies>

<!-- Kafka Client -->

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>3.6.0</version>

</dependency>

<!-- Logging -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.9</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.4.11</version>

</dependency>

<!-- JSON Processing -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.16.0</version>

</dependency>

<!-- PostgreSQL Driver -->

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.7.1</version>

</dependency>

<!-- Connection Pooling -->

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

<version>5.1.0</version>

</dependency>

<!-- AWS S3 SDK (works with MinIO) -->

<dependency>

<groupId>software.amazon.awssdk</groupId>

<artifactId>s3</artifactId>

<version>2.21.0</version>

</dependency>

</dependencies>

```

---

<a name="integration"></a>

## 7. How Everything Works Together

### The Complete Workflow

```

USER ACTION: Upload file "vacation-photo.jpg"

│

├─▶ STEP 1: Java Application Receives File

│ │ File: CompleteIntegrationDemo.java:156

│ │ Method: uploadFile()

│ │ Input: filename, contentType, fileData

│ │

│ ├─▶ STEP 2: Upload to S3 (MinIO)

│ │ │ File: S3Manager.java:90

│ │ │ Method: uploadFile()

│ │ │ Actions:

│ │ │ • Creates S3 key: "uploads/alice/vacation-photo.jpg"

│ │ │ • Uploads file data to MinIO

│ │ │ • Returns URL: "http://localhost:9000/bucket/uploads/alice/vacation-photo.jpg"

│ │ │ Result: File stored in blob storage ✅

│ │ │

│ │ └─▶ STEP 3: Save Metadata to PostgreSQL

│ │ │ File: FileRepository.java:48

│ │ │ Method: insertFileMetadata()

│ │ │ Actions:

│ │ │ • Creates FileMetadata object

│ │ │ • Executes INSERT SQL statement

│ │ │ • Saves: id, filename, s3\_url, size, user, timestamp

│ │ │ Result: Metadata stored in database ✅

│ │ │

│ │ └─▶ STEP 4: Publish Event to Kafka

│ │ │ File: FileEventProducer.java:72

│ │ │ Method: sendUploadedEvent()

│ │ │ Actions:

│ │ │ • Creates FileEvent object

│ │ │ • Converts to JSON

│ │ │ • Sends to Kafka topic "file-events"

│ │ │ • Uses file ID as partition key

│ │ │ Result: Event published to Kafka ✅

│ │ │

│ │ └─▶ STEP 5: Consumer Receives Event

│ │ │ File: FileEventConsumer.java:87

│ │ │ Method: start()

│ │ │ Actions:

│ │ │ • Polls Kafka for new messages

│ │ │ • Deserializes JSON to FileEvent

│ │ │ • Calls custom event handler

│ │ │ • Processes: thumbnails, virus scan, notifications

│ │ │ Result: Event processed ✅

│ │

│ └─▶ RESPONSE: Return success to user (1 second)

│ │ User sees: "File uploaded successfully!"

│ │ Background: Processing continues asynchronously

│ │

│ └─▶ BACKGROUND PROCESSING (runs independently)

│ • Thumbnail generation (10 seconds)

│ • Virus scanning (15 seconds)

│ • Search indexing (5 seconds)

│ • Email notification (2 seconds)

│ Total: 32 seconds (but user doesn't wait!)

```

### Component Interaction Diagram

```

┌─────────────────────────────────────────────────────────────────┐

│ USER REQUEST │

│ "Upload vacation.jpg" │

└──────────────────────────┬──────────────────────────────────────┘

│

▼

┌─────────────────────────────────────────────────────────────────┐

│ JAVA APPLICATION │

│ (CompleteIntegrationDemo.java) │

└──┬────────────────┬────────────────┬─────────────────┬──────────┘

│ │ │ │

│ uploadFile() │ insert() │ publish() │ process()

▼ ▼ ▼ ▼

┌─────────┐ ┌─────────────┐ ┌──────────┐ ┌──────────────┐

│ │ │ │ │ │ │ │

│ MinIO │ │ PostgreSQL │ │ Kafka │ │ Consumer │

│ (S3) │ │ Database │ │ Broker │ │ Thread │

│ │ │ │ │ │ │ │

│ Stores │ │ Stores │ │ Routes │ │ Processes │

│ File │ │ Metadata │ │ Events │ │ Event │

│ │ │ │ │ │ │ │

│ Port: │ │ Port: │ │ Port: │ │ (Background)│

│ 9000 │ │ 5432 │ │ 9092 │ │ │

└─────────┘ └─────────────┘ └──────────┘ └──────────────┘

│ │ │ │

│ │ │ │

▼ ▼ ▼ ▼

┌─────────┐ ┌─────────────┐ ┌──────────┐ ┌──────────────┐

│ Bucket: │ │ Table: │ │ Topic: │ │ Actions: │

│complete-│ │ files │ │file-events│ │ • Thumbnails │

│integra- │ │ │ │ │ │ • Virus scan │

│tion- │ │ Columns: │ │Partitions│ │ • Notify │

│ bucket │ │ • id │ │ 0, 1, 2│ │ • Index │

│ │ │ • filename │ │ │ │ │

│Files: │ │ • s3\_url │ │ Messages:│ │ Async │

│ /uploads│ │ • size │ │ JSON │ │ Processing │

│ /alice │ │ • user │ │ Events │ │ │

│ /\*.jpg│ │ • timestamp │ │ │ │ │

└─────────┘ └─────────────┘ └──────────┘ └──────────────┘

```

### Data Flow Timeline

```

Time | Component | Action

-------|-------------------|------------------------------------------

0.0s | User | Clicks "Upload" button

0.1s | Java App | Receives file (10 MB)

0.2s | S3Manager | Starts upload to MinIO

1.0s | MinIO | File stored, returns URL

1.1s | FileRepository | Saves metadata to PostgreSQL

1.2s | PostgreSQL | INSERT complete

1.3s | FileEventProducer | Creates JSON event

1.4s | Kafka Broker | Receives event, stores in partition 1

1.5s | Java App | Returns success to user ✅

| |

| | [User sees "Upload successful!"]

| |

2.0s | Consumer Thread | Polls Kafka, receives event

2.1s | Event Handler | Starts processing

2.2s | Thumbnail Service | Generates 3 sizes (200x200, 500x500, 1000x1000)

12.0s | Thumbnail Service | Complete ✅

12.1s | Virus Scanner | Scans file for malware

27.0s | Virus Scanner | Complete ✅ (Clean)

27.1s | Search Indexer | Extracts metadata, updates search DB

32.0s | Search Indexer | Complete ✅

32.1s | Notification | Sends email: "Your file is ready!"

34.0s | Notification | Complete ✅

| |

| TOTAL USER WAIT | 1.5 seconds

| TOTAL PROCESSING | 34.0 seconds (async)

```

### Error Handling & Resilience

\*\*Scenario 1: S3 Upload Fails\*\*

```java

try {

s3Url = s3Manager.uploadFile(bucket, key, data, contentType);

} catch (S3Exception e) {

logger.error("S3 upload failed: {}", e.getMessage());

// Don't save to database or send Kafka event

return "Upload failed - please try again";

}

```

\*\*Result\*\*: User sees error, database and Kafka stay clean

\*\*Scenario 2: Database Insert Fails\*\*

```java

try {

fileRepository.insertFileMetadata(metadata);

} catch (SQLException e) {

logger.error("Database insert failed: {}", e.getMessage());

// Rollback: Delete from S3

s3Manager.deleteFile(bucket, key);

return "Upload failed - please try again";

}

```

\*\*Result\*\*: S3 file deleted, user sees error, Kafka not notified

\*\*Scenario 3: Kafka Publish Fails\*\*

```java

try {

eventProducer.sendUploadedEvent(metadata);

} catch (Exception e) {

logger.error("Kafka publish failed: {}", e.getMessage());

// File is uploaded and in database

// Event will be retried later or processing triggered manually

}

```

\*\*Result\*\*: File uploaded and saved, but processing delayed

\*\*Scenario 4: Consumer Processing Fails\*\*

```java

// FileEventConsumer.java

consumer.subscribe(Collections.singletonList(topicName));

while (running) {

try {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

try {

FileEvent event = deserialize(record.value());

eventHandler.accept(event); // Process event

} catch (Exception e) {

logger.error("Failed to process event: {}", record.key(), e);

// Event offset not committed - will retry on next poll

}

}

consumer.commitSync(); // Commit only if all successful

} catch (Exception e) {

logger.error("Consumer error", e);

// Will reconnect and resume from last committed offset

}

}

```

\*\*Result\*\*: Failed events are retried automatically

---

<a name="docker"></a>

## 8. Docker & Containerization

### What is Docker?

\*\*Simple Analogy\*\*: Docker is like \*\*shipping containers for software\*\*:

- Just as shipping containers standardize how goods are transported

- Docker containers standardize how software runs

- Works the same on your laptop, your teammate's computer, and production servers

\*\*Without Docker:\*\*

```

Developer 1: "Works on my machine!" (Windows)

Developer 2: "Doesn't work on mine!" (Mac)

Server: "Crashes in production!" (Linux)

Problem: Different environments, different results!

```

\*\*With Docker:\*\*

```

Developer 1: Runs Docker container ✅

Developer 2: Runs same Docker container ✅

Server: Runs same Docker container ✅

Solution: Same environment everywhere!

```

### Docker Compose

\*\*What\*\*: Tool to run multiple Docker containers together

\*\*File\*\*: `docker-compose.yml`

```yaml

version: '3.8'

services:

# Service 1: Zookeeper (Kafka's coordinator)

zookeeper:

image: confluentinc/cp-zookeeper:7.5.0

ports:

- "2181:2181"

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

volumes:

- zookeeper-data:/var/lib/zookeeper/data

healthcheck:

test: ["CMD", "nc", "-z", "localhost", "2181"]

# Service 2: Kafka (Message broker)

kafka:

image: confluentinc/cp-kafka:7.5.0

depends\_on:

- zookeeper

ports:

- "9092:9092"

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092

volumes:

- kafka-data:/var/lib/kafka/data

# Service 3: PostgreSQL (Database)

postgres:

image: postgres:16-alpine

ports:

- "5432:5432"

environment:

POSTGRES\_USER: admin

POSTGRES\_PASSWORD: admin123

POSTGRES\_DB: mydb

volumes:

- postgres-data:/var/lib/postgresql/data

# Service 4: MinIO (S3-compatible storage)

minio:

image: minio/minio:latest

command: server /data --console-address ":9001"

ports:

- "9000:9000" # API

- "9001:9001" # Web Console

environment:

MINIO\_ROOT\_USER: minioadmin

MINIO\_ROOT\_PASSWORD: minioadmin123

volumes:

- minio-data:/data

# Service 5: pgAdmin (Database UI)

pgadmin:

image: dpage/pgadmin4:latest

ports:

- "5050:80"

environment:

PGADMIN\_DEFAULT\_EMAIL: admin@admin.com

PGADMIN\_DEFAULT\_PASSWORD: admin123

# Service 6: Kafka UI (Kafka management)

kafka-ui:

image: provectuslabs/kafka-ui:latest

ports:

- "8080:8080"

environment:

KAFKA\_CLUSTERS\_0\_NAME: local

KAFKA\_CLUSTERS\_0\_BOOTSTRAPSERVERS: kafka:29092

volumes:

zookeeper-data:

kafka-data:

postgres-data:

minio-data:

pgadmin-data:

```

### Docker Commands

\*\*Start all services:\*\*

```bash

docker-compose up -d

# -d = detached mode (runs in background)

```

\*\*Stop all services:\*\*

```bash

docker-compose down

```

\*\*Stop and remove all data:\*\*

```bash

docker-compose down -v

# -v = removes volumes (deletes all data)

```

\*\*View logs:\*\*

```bash

docker-compose logs kafka

docker-compose logs postgres

docker-compose logs -f # -f = follow (live updates)

```

\*\*Check status:\*\*

```bash

docker-compose ps

```

\*\*Restart a service:\*\*

```bash

docker-compose restart kafka

```

### Docker Networking

\*\*Internal Network\*\* (`docker-compose.yml` creates network: `kafka-network`):

```

Container: kafka-broker

Internal hostname: kafka

Internal IP: 172.18.0.3

Accessible from other containers: kafka:29092

Container: postgres-db

Internal hostname: postgres

Internal IP: 172.18.0.4

Accessible from other containers: postgres:5432

Container: minio-s3

Internal hostname: minio

Internal IP: 172.18.0.5

Accessible from other containers: minio:9000

```

\*\*Port Mapping\*\*:

```

Host Machine Docker Container

localhost:9092 → kafka:29092

localhost:5432 → postgres:5432

localhost:9000 → minio:9000

localhost:8080 → kafka-ui:8080

localhost:5050 → pgadmin:80

```

### Volumes & Data Persistence

\*\*Without Volumes:\*\*

```

1. Start container

2. Write data

3. Stop container

4. Data is LOST! ❌

```

\*\*With Volumes:\*\*

```

1. Start container (mounts volume)

2. Write data (saved to volume)

3. Stop container

4. Data is PRESERVED! ✅

5. Start container again (data still there)

```

\*\*Our Volumes:\*\*

- `postgres-data` → Database files

- `kafka-data` → Kafka logs and events

- `zookeeper-data` → Zookeeper state

- `minio-data` → Uploaded files

---

<a name="data-flow"></a>

## 9. Deep Dive: Complete Data Flow

### Detailed Step-by-Step Execution

#### Step 1: Application Startup

\*\*File\*\*: `CompleteIntegrationDemo.java:56`

```java

public static void main(String[] args) {

CompleteIntegrationDemo demo = new CompleteIntegrationDemo();

demo.initializeComponents();

// ...

}

```

\*\*What Happens:\*\*

1. JVM starts

2. Loads all classes

3. Initializes logging (Logback)

4. Calls `initializeComponents()`

---

#### Step 2: Initialize S3 Manager

\*\*File\*\*: `S3Manager.java:33`

```java

public S3Manager(String endpoint, String accessKey, String secretKey) {

AwsBasicCredentials credentials = AwsBasicCredentials.create(accessKey, secretKey);

this.s3Client = S3Client.builder()

.endpointOverride(URI.create("http://localhost:9000"))

.credentialsProvider(StaticCredentialsProvider.create(credentials))

.build();

}

```

\*\*Network Request:\*\*

```

Java App → MinIO

└─▶ GET http://localhost:9000/ (health check)

◀── Response: 200 OK

```

\*\*Result\*\*: S3 client ready to upload files

---

#### Step 3: Initialize Database Connection Pool

\*\*File\*\*: `DatabaseManager.java:30`

```java

HikariConfig config = new HikariConfig();

config.setJdbcUrl("jdbc:postgresql://localhost:5432/mydb");

config.setUsername("admin");

config.setPassword("admin123");

config.setMaximumPoolSize(10);

dataSource = new HikariDataSource(config);

```

\*\*Network Request:\*\*

```

Java App → PostgreSQL

└─▶ TCP connection to localhost:5432

◀── Response: Connection accepted

└─▶ AUTH: username=admin, password=\*\*\*

◀── Response: AUTH\_OK

└─▶ SELECT 1 (connection test)

◀── Response: 1 row

```

\*\*Result\*\*: Connection pool with 10 connections ready

---

#### Step 4: Initialize Kafka Producer

\*\*File\*\*: `FileEventProducer.java:51`

```java

Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);

producer = new KafkaProducer<>(props);

```

\*\*Network Request:\*\*

```

Java App → Kafka

└─▶ TCP connection to localhost:9092

◀── Response: Connection accepted

└─▶ API\_VERSIONS request

◀── Response: [list of supported APIs]

└─▶ METADATA request (list topics)

◀── Response: {topics: ["file-events"], brokers: [{id:1, host:"kafka"}]}

```

\*\*Result\*\*: Producer ready to send messages

---

#### Step 5: Initialize Kafka Consumer

\*\*File\*\*: `FileEventConsumer.java:58`

```java

Properties props = new Properties();

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, "complete-integration-group");

consumer = new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("file-events"));

```

\*\*Network Request:\*\*

```

Java App → Kafka

└─▶ JOIN\_GROUP request (group: complete-integration-group)

◀── Response: {memberId: "consumer-1", leader: true}

└─▶ SYNC\_GROUP request

◀── Response: {assignment: [file-events-0, file-events-1, file-events-2]}

└─▶ FETCH\_OFFSET request (get last read position)

◀── Response: {file-events-0: offset 0, file-events-1: offset 0, file-events-2: offset 0}

```

\*\*Result\*\*: Consumer subscribed to topic, ready to receive messages

---

#### Step 6: Upload File to S3

\*\*File\*\*: `S3Manager.java:90`

```java

public String uploadFile(String bucketName, String key, byte[] fileData, String contentType) {

PutObjectRequest request = PutObjectRequest.builder()

.bucket("complete-integration-bucket")

.key("uploads/alice/vacation-photo.jpg")

.contentType("image/jpeg")

.contentLength(2500000L)

.build();

s3Client.putObject(request, RequestBody.fromBytes(fileData));

}

```

\*\*Network Request:\*\*

```

Java App → MinIO

└─▶ PUT http://localhost:9000/complete-integration-bucket/uploads/alice/vacation-photo.jpg

Headers:

Content-Type: image/jpeg

Content-Length: 2500000

Authorization: AWS4-HMAC-SHA256 Credential=...

Body: [2.5 MB of image data]

◀── Response: 200 OK

Headers:

ETag: "abc123def456"

Location: /complete-integration-bucket/uploads/alice/vacation-photo.jpg

```

\*\*MinIO Internal Processing:\*\*

1. Receives HTTP PUT request

2. Validates credentials

3. Writes file to disk: `/data/complete-integration-bucket/uploads/alice/vacation-photo.jpg`

4. Updates bucket index

5. Calculates ETag (MD5 hash)

6. Returns success response

\*\*Result\*\*: File stored in MinIO at `/data/complete-integration-bucket/uploads/alice/vacation-photo.jpg`

---

#### Step 7: Save Metadata to PostgreSQL

\*\*File\*\*: `FileRepository.java:48`

```java

public void insertFileMetadata(FileMetadata metadata) throws SQLException {

String sql = """

INSERT INTO files (id, filename, s3\_key, s3\_url, file\_size,

content\_type, uploaded\_by, uploaded\_at)

VALUES (?, ?, ?, ?, ?, ?, ?, ?)

""";

try (Connection conn = databaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6");

stmt.setString(2, "vacation-photo.jpg");

stmt.setString(3, "uploads/alice/vacation-photo.jpg");

stmt.setString(4, "http://localhost:9000/complete-integration-bucket/uploads/alice/vacation-photo.jpg");

stmt.setLong(5, 2500000L);

stmt.setString(6, "image/jpeg");

stmt.setString(7, "alice");

stmt.setLong(8, 1729090624308L);

stmt.executeUpdate();

}

}

```

\*\*Network Request:\*\*

```

Java App → PostgreSQL

└─▶ TCP packet to localhost:5432

PostgreSQL Wire Protocol:

Message Type: P (Parse)

SQL: INSERT INTO files (id, filename, s3\_key, s3\_url, file\_size,

content\_type, uploaded\_by, uploaded\_at)

VALUES ($1, $2, $3, $4, $5, $6, $7, $8)

Message Type: B (Bind)

Parameters:

$1: "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6"

$2: "vacation-photo.jpg"

$3: "uploads/alice/vacation-photo.jpg"

$4: "http://localhost:9000/complete-integration-bucket/uploads/alice/vacation-photo.jpg"

$5: 2500000

$6: "image/jpeg"

$7: "alice"

$8: 1729090624308

Message Type: E (Execute)

◀── Response:

Message Type: C (CommandComplete)

Tag: INSERT 0 1

```

\*\*PostgreSQL Internal Processing:\*\*

1. Parses SQL statement

2. Validates syntax

3. Checks constraints (NOT NULL, PRIMARY KEY)

4. Inserts row into `files` table

5. Updates indexes (`idx\_files\_uploaded\_by`, `idx\_files\_created\_at`)

6. Writes to Write-Ahead Log (WAL)

7. Returns success

\*\*Database State:\*\*

```sql

SELECT \* FROM files WHERE id = 'a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6';

-- Result:

id | a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6

filename | vacation-photo.jpg

s3\_key | uploads/alice/vacation-photo.jpg

s3\_url | http://localhost:9000/complete-integration-bucket/uploads/alice/vacation-photo.jpg

file\_size| 2500000

content\_type | image/jpeg

uploaded\_by | alice

uploaded\_at | 1729090624308

created\_at | 2025-10-16 16:37:04.308

updated\_at | 2025-10-16 16:37:04.308

```

---

#### Step 8: Publish Event to Kafka

\*\*File\*\*: `FileEventProducer.java:72`

```java

public void sendUploadedEvent(FileMetadata metadata) throws Exception {

FileEvent event = new FileEvent("UPLOADED", metadata, System.currentTimeMillis());

String jsonMessage = objectMapper.writeValueAsString(event);

ProducerRecord<String, String> record = new ProducerRecord<>(

"file-events",

metadata.getId(), // Partition key

jsonMessage

);

producer.send(record).get(); // Wait for acknowledgment

}

```

\*\*JSON Message Created:\*\*

```json

{

"eventType": "UPLOADED",

"fileMetadata": {

"id": "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6",

"filename": "vacation-photo.jpg",

"s3Key": "uploads/alice/vacation-photo.jpg",

"s3Url": "http://localhost:9000/complete-integration-bucket/uploads/alice/vacation-photo.jpg",

"fileSize": 2500000,

"contentType": "image/jpeg",

"uploadedBy": "alice",

"uploadedAt": 1729090624308

},

"timestamp": 1729090624308

}

```

\*\*Network Request:\*\*

```

Java App → Kafka

└─▶ PRODUCE request

Topic: file-events

Partition: 1 (calculated from key hash)

Key: "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6"

Value: [JSON message above]

Compression: none

Acks: all (wait for all replicas)

◀── Response (after 50ms):

Status: SUCCESS

Partition: 1

Offset: 42

Timestamp: 1729090624358

```

\*\*Kafka Internal Processing:\*\*

1. Receives PRODUCE request

2. Calculates partition from key hash: `hash("a1b2c3d4...") % 3 = 1`

3. Appends message to partition 1 log file

4. Writes to disk

5. Replicates to other brokers (if configured)

6. Returns acknowledgment with offset

\*\*Kafka Log File\*\* (`/var/lib/kafka/data/file-events-1/00000000000000000042.log`):

```

Offset: 42

Timestamp: 1729090624358

Key Length: 36

Key: a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6

Value Length: 423

Value: {"eventType":"UPLOADED","fileMetadata":{...}}

```

---

#### Step 9: Consumer Polls Kafka

\*\*File\*\*: `FileEventConsumer.java:87`

```java

while (running) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

FileEvent event = objectMapper.readValue(record.value(), FileEvent.class);

eventHandler.accept(event);

}

consumer.commitSync();

}

```

\*\*Network Request:\*\*

```

Java App → Kafka

└─▶ FETCH request

Topics: [file-events]

Partitions: [0, 1, 2]

CurrentOffsets: [0, 42, 0] // Partition 1 has new message at offset 42

MaxWaitMs: 100

MinBytes: 1

MaxBytes: 52428800 (50 MB)

◀── Response (immediately, since message is available):

Partition: 1

HighWaterMark: 43

Messages: [

{

Offset: 42

Key: "a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6"

Value: "{\"eventType\":\"UPLOADED\",\"fileMetadata\":{...}}"

Timestamp: 1729090624358

}

]

```

\*\*Java Processing:\*\*

1. Receives message from Kafka

2. Deserializes JSON to `FileEvent` object

3. Calls `eventHandler.accept(event)`

4. Event handler processes the event

---

#### Step 10: Process Event

\*\*File\*\*: `CompleteIntegrationDemo.java:127`

```java

Consumer<FileEvent> eventHandler = event -> {

FileMetadata metadata = event.getFileMetadata();

logger.info("📥 RECEIVED EVENT: {}", event.getEventType());

logger.info(" • Filename: {}", metadata.getFilename());

logger.info(" • Size: {}", formatFileSize(metadata.getFileSize()));

logger.info(" • User: {}", metadata.getUploadedBy());

// Simulate processing tasks

logger.info("\n 🔄 Processing tasks:");

logger.info(" [1] Generating thumbnails...");

generateThumbnails(metadata);

logger.info(" [2] Scanning for viruses...");

scanForViruses(metadata);

logger.info(" [3] Updating search index...");

updateSearchIndex(metadata);

logger.info(" [4] Sending notifications...");

sendNotification(metadata);

logger.info("\n ✅ Processing complete!");

eventsProcessed.incrementAndGet();

};

```

\*\*Real-World Processing Examples:\*\*

\*\*1. Generate Thumbnails:\*\*

```java

private void generateThumbnails(FileMetadata metadata) {

if (metadata.getContentType().startsWith("image/")) {

// Download original image from S3

byte[] imageData = s3Manager.downloadFile(bucket, metadata.getS3Key());

// Generate 3 sizes

byte[] thumb200 = resizeImage(imageData, 200, 200);

byte[] thumb500 = resizeImage(imageData, 500, 500);

byte[] thumb1000 = resizeImage(imageData, 1000, 1000);

// Upload thumbnails back to S3

s3Manager.uploadFile(bucket, metadata.getS3Key() + "\_thumb\_200", thumb200, "image/jpeg");

s3Manager.uploadFile(bucket, metadata.getS3Key() + "\_thumb\_500", thumb500, "image/jpeg");

s3Manager.uploadFile(bucket, metadata.getS3Key() + "\_thumb\_1000", thumb1000, "image/jpeg");

}

}

```

\*\*2. Scan for Viruses:\*\*

```java

private void scanForViruses(FileMetadata metadata) {

// Download file from S3

byte[] fileData = s3Manager.downloadFile(bucket, metadata.getS3Key());

// Call virus scanning API (e.g., ClamAV)

boolean isClean = virusScanner.scan(fileData);

if (!isClean) {

// Quarantine file

s3Manager.deleteFile(bucket, metadata.getS3Key());

fileRepository.deleteFileMetadata(metadata.getId());

// Notify user

emailService.send(metadata.getUploadedBy(),

"Your file was removed due to security concerns");

}

}

```

\*\*3. Update Search Index:\*\*

```java

private void updateSearchIndex(FileMetadata metadata) {

// Create search document

SearchDocument doc = new SearchDocument();

doc.setId(metadata.getId());

doc.setTitle(metadata.getFilename());

doc.setContent(extractText(metadata)); // OCR for images, parse PDF, etc.

doc.setUser(metadata.getUploadedBy());

doc.setTimestamp(metadata.getUploadedAt());

// Index in Elasticsearch

elasticsearchClient.index("files", doc);

}

```

\*\*4. Send Notification:\*\*

```java

private void sendNotification(FileMetadata metadata) {

// Send email

emailService.send(

metadata.getUploadedBy(),

"File uploaded successfully",

"Your file '" + metadata.getFilename() + "' is ready!"

);

// Push notification

pushService.send(

metadata.getUploadedBy(),

"File uploaded",

metadata.getFilename()

);

// Update user's notification feed

notificationRepository.create(

metadata.getUploadedBy(),

"FILE\_UPLOADED",

metadata.getId()

);

}

```

---

#### Step 11: Commit Offset

\*\*File\*\*: `FileEventConsumer.java:102`

```java

consumer.commitSync();

```

\*\*Network Request:\*\*

```

Java App → Kafka

└─▶ OFFSET\_COMMIT request

Group: complete-integration-group

Offsets: {

file-events-0: 0,

file-events-1: 43, // We processed message at offset 42

file-events-2: 0

}

◀── Response:

Status: SUCCESS

```

\*\*What This Means:\*\*

- If consumer crashes and restarts, it will resume from offset 43

- Message at offset 42 won't be processed again

- Ensures "exactly-once" processing (with proper configuration)

---

### Complete Timeline Summary

```

Time | Component | Action | Duration

--------|---------------------|----------------------------------|----------

0.000s | User | Clicks "Upload" | -

0.001s | Browser | Sends HTTP POST to Java app | -

0.002s | Java App | Receives file (10 MB) | -

0.003s | S3Manager | Starts upload to MinIO | -

0.500s | MinIO | File uploaded ✅ | 497ms

0.501s | FileRepository | Starts DB insert | -

0.520s | PostgreSQL | Row inserted ✅ | 19ms

0.521s | FileEventProducer | Creates JSON event | -

0.522s | Kafka | Event stored ✅ | 1ms

0.523s | Java App | Returns HTTP 200 to user | -

0.524s | User | Sees "Upload successful!" ✅ | Total: 524ms

| | |

1.000s | Consumer | Polls Kafka | -

1.001s | Kafka | Returns event | 1ms

1.002s | Event Handler | Starts processing | -

2.000s | Thumbnail Gen | Generates 3 thumbnails | 998ms

10.000s | Virus Scanner | Scans file | 8000ms

12.000s | Search Indexer | Updates Elasticsearch | 2000ms

13.000s | Notification | Sends email & push | 1000ms

13.001s | Consumer | Commits offset ✅ | -

| | |

TOTAL | User perceived time | 524ms |

TOTAL | Background processing | 12.5 seconds (asynchronous) |

```

---

<a name="real-world"></a>

## 10. Real-World Applications

### Case Study 1: Instagram Photo Upload

\*\*Architecture:\*\*

```

User uploads photo → Instagram API

↓

[1] Upload to S3

↓

[2] Save to MySQL (metadata)

↓

[3] Publish Kafka event: "photo.uploaded"

↓

┌───────────┼───────────┐

↓ ↓ ↓

[Thumbnail] [AI Filter] [Feed Update]

Service Service Service

↓ ↓ ↓

Generate Apply ML Update

3 sizes models followers'

feeds

```

\*\*Our Implementation Maps To:\*\*

- `S3Manager.uploadFile()` → Instagram's photo storage

- `FileRepository.insertFileMetadata()` → Instagram's post metadata

- `FileEventProducer.sendUploadedEvent()` → Instagram's event system

- `FileEventConsumer` → Instagram's background processors

\*\*Scale:\*\*

- 95 million photos uploaded per day

- 1,100 photos per second

- Each photo generates ~20 events

- 22,000 events per second processed

---

### Case Study 2: Netflix Video Upload

\*\*Architecture:\*\*

```

Content provider uploads video → Netflix API

↓

[1] Upload to S3 (original)

↓

[2] Save to Cassandra (metadata)

↓

[3] Publish Kafka event: "video.uploaded"

↓

┌─────────────────┼─────────────────┐

↓ ↓ ↓

[Transcoder] [Thumbnail] [Subtitle]

Service Generator Extractor

↓ ↓ ↓

Generate Create 5 Extract

1080p, 720p, thumbnails audio for

480p, 360p, at different subtitles

240p versions timestamps

```

\*\*Transcoding Pipeline:\*\*

```java

// Similar to our event handler

Consumer<VideoEvent> transcodingHandler = event -> {

VideoMetadata video = event.getVideoMetadata();

// Download original from S3

byte[] originalVideo = s3Manager.downloadFile(bucket, video.getS3Key());

// Transcode to multiple qualities

byte[] video1080p = transcode(originalVideo, "1080p");

byte[] video720p = transcode(originalVideo, "720p");

byte[] video480p = transcode(originalVideo, "480p");

byte[] video360p = transcode(originalVideo, "360p");

byte[] video240p = transcode(originalVideo, "240p");

// Upload all versions back to S3

s3Manager.uploadFile(bucket, video.getS3Key() + "\_1080p", video1080p, "video/mp4");

s3Manager.uploadFile(bucket, video.getS3Key() + "\_720p", video720p, "video/mp4");

// ... and so on

// Publish event: "video.transcoded"

eventProducer.sendTranscodedEvent(video);

};

```

\*\*Scale:\*\*

- Netflix processes thousands of hours of new content per month

- Each video generates 5 quality versions + multiple thumbnails

- Transcoding can take hours for a 2-hour movie

- All done asynchronously while content provider sees immediate confirmation

---

### Case Study 3: Dropbox File Sync

\*\*Architecture:\*\*

```

User uploads file to Dropbox → Dropbox API

↓

[1] Upload to S3 (chunked)

↓

[2] Save to MySQL (file metadata)

↓

[3] Publish Kafka event: "file.changed"

↓

┌───────────────┼───────────────┐

↓ ↓ ↓

[Sync Service] [Preview Gen] [Version History]

↓ ↓ ↓

Notify all Generate Create

devices with preview for snapshot for

this folder supported rollback

file types

```

\*\*File Chunking\*\* (How Dropbox uploads large files):

```java

public void uploadLargeFile(String filename, byte[] fileData) {

int CHUNK\_SIZE = 4 \* 1024 \* 1024; // 4 MB chunks

int numChunks = (int) Math.ceil((double) fileData.length / CHUNK\_SIZE);

String uploadSessionId = UUID.randomUUID().toString();

for (int i = 0; i < numChunks; i++) {

int start = i \* CHUNK\_SIZE;

int end = Math.min(start + CHUNK\_SIZE, fileData.length);

byte[] chunk = Arrays.copyOfRange(fileData, start, end);

// Upload chunk

s3Manager.uploadFilePart(

bucket,

uploadSessionId,

i, // Part number

chunk

);

logger.info("Uploaded chunk {}/{}", i + 1, numChunks);

}

// Complete multipart upload

String s3Key = s3Manager.completeMultipartUpload(uploadSessionId, filename);

// Save metadata and publish event

saveMetadataAndPublishEvent(s3Key, filename, fileData.length);

}

```

---

### Case Study 4: Uber Ride Tracking

\*\*Architecture (Simplified):\*\*

```

Driver's phone sends GPS update → Uber API

↓

[1] Save to Redis (current location)

↓

[2] Save to PostgreSQL (location history)

↓

[3] Publish Kafka event: "driver.location.updated"

↓

┌──────────────────┼──────────────────┐

↓ ↓ ↓

[Rider App] [Analytics] [Heatmap]

Update Track driver Update

rider's map patterns city heatmap

```

\*\*Real-time Location Updates:\*\*

```java

Consumer<LocationEvent> locationHandler = event -> {

DriverLocation location = event.getLocation();

// Update Redis (fast, in-memory)

redisClient.set("driver:" + location.getDriverId(), location);

// Save to PostgreSQL (persistent history)

locationRepository.insertLocation(location);

// Find nearby riders looking for rides

List<Rider> nearbyRiders = findRidersNearLocation(location);

// Notify each rider

for (Rider rider : nearbyRiders) {

pushNotification(rider, "Driver nearby!");

}

// Update city heatmap

heatmapService.updateCell(location.getLat(), location.getLon());

};

```

---

### Case Study 5: E-commerce Order Processing

\*\*Architecture:\*\*

```

User places order → E-commerce API

↓

[1] Save to PostgreSQL (order)

↓

[2] Publish Kafka event: "order.placed"

↓

┌─────────────────┼─────────────────┐

↓ ↓ ↓

[Payment] [Inventory] [Notification]

Service Service Service

↓ ↓ ↓

Charge Decrease Send email

credit card stock count confirmation

↓ ↓ ↓

Publish Publish -

"paid" "reserved"

↓ ↓

└─────────────────┼─────────────────┐

↓

[Shipping Service]

↓

Create shipping label

↓

Publish "shipped"

↓

[Tracking Service]

↓

Start tracking

```

\*\*Order Processing Flow:\*\*

```java

// Order placement

public void placeOrder(Order order) {

// Save order to database

orderRepository.insertOrder(order);

// Publish event

kafkaProducer.send("order-events", new OrderEvent("PLACED", order));

}

// Payment processor

Consumer<OrderEvent> paymentHandler = event -> {

if (event.getType().equals("PLACED")) {

Order order = event.getOrder();

// Charge credit card

boolean success = paymentGateway.charge(

order.getPaymentMethod(),

order.getTotalAmount()

);

if (success) {

// Update order status

orderRepository.updateStatus(order.getId(), "PAID");

// Publish payment success event

kafkaProducer.send("order-events",

new OrderEvent("PAID", order));

} else {

// Handle payment failure

orderRepository.updateStatus(order.getId(), "PAYMENT\_FAILED");

kafkaProducer.send("order-events",

new OrderEvent("PAYMENT\_FAILED", order));

}

}

};

// Inventory manager

Consumer<OrderEvent> inventoryHandler = event -> {

if (event.getType().equals("PAID")) {

Order order = event.getOrder();

// Decrease stock for each item

for (OrderItem item : order.getItems()) {

inventoryRepository.decreaseStock(

item.getProductId(),

item.getQuantity()

);

}

// Publish inventory reserved event

kafkaProducer.send("order-events",

new OrderEvent("INVENTORY\_RESERVED", order));

}

};

// Shipping service

Consumer<OrderEvent> shippingHandler = event -> {

if (event.getType().equals("INVENTORY\_RESERVED")) {

Order order = event.getOrder();

// Create shipping label

ShippingLabel label = shippingService.createLabel(order);

// Update order with tracking number

orderRepository.updateTrackingNumber(

order.getId(),

label.getTrackingNumber()

);

// Publish shipped event

kafkaProducer.send("order-events",

new OrderEvent("SHIPPED", order));

}

};

// Notification service

Consumer<OrderEvent> notificationHandler = event -> {

Order order = event.getOrder();

switch (event.getType()) {

case "PLACED":

emailService.send(order.getCustomerEmail(),

"Order received",

"We've received your order #" + order.getId());

break;

case "PAID":

emailService.send(order.getCustomerEmail(),

"Payment confirmed",

"Your payment has been processed");

break;

case "SHIPPED":

emailService.send(order.getCustomerEmail(),

"Order shipped",

"Track your order: " + order.getTrackingNumber());

break;

case "PAYMENT\_FAILED":

emailService.send(order.getCustomerEmail(),

"Payment failed",

"Please update your payment method");

break;

}

};

```

\*\*Why Event-Driven Architecture Wins Here:\*\*

- \*\*Decoupling\*\*: Payment failure doesn't crash inventory system

- \*\*Scalability\*\*: Each service scales independently

- \*\*Resilience\*\*: If email service is down, order still processes

- \*\*Auditability\*\*: Complete event log of order lifecycle

- \*\*Flexibility\*\*: Easy to add new services (fraud detection, recommendations, etc.)

---

<a name="concepts"></a>

## 11. Key Concepts for Beginners

### 1. Event-Driven Architecture

\*\*Traditional (Synchronous):\*\*

```java

// Caller waits for each step to complete

void uploadFile(File file) {

s3.upload(file); // Wait 3 seconds

database.save(metadata); // Wait 0.5 seconds

thumbnail.generate(file); // Wait 10 seconds

virus.scan(file); // Wait 15 seconds

email.send(notification); // Wait 2 seconds

// Total wait: 30.5 seconds 😴

}

```

\*\*Event-Driven (Asynchronous):\*\*

```java

// Caller returns immediately

void uploadFile(File file) {

s3.upload(file);

database.save(metadata);

kafka.publish("file.uploaded", metadata);

// Returns immediately! ⚡ (3.5 seconds)

}

// Background services process independently

void onFileUploaded(FileEvent event) {

thumbnail.generate(event.file); // Service 1

}

void onFileUploaded(FileEvent event) {

virus.scan(event.file); // Service 2

}

void onFileUploaded(FileEvent event) {

email.send(notification); // Service 3

}

```

\*\*Benefits:\*\*

- ✅ Fast response to user

- ✅ Services don't block each other

- ✅ Easy to add new services

- ✅ Handles failures gracefully

---

### 2. Microservices Architecture

\*\*Monolith (All in one):\*\*

```

┌──────────────────────────────────────┐

│ SINGLE APPLICATION │

│ │

│ • File Upload │

│ • Thumbnail Generation │

│ • Virus Scanning │

│ • Email Notifications │

│ • User Management │

│ • Payment Processing │

│ │

│ Problem: One bug crashes everything │

└──────────────────────────────────────┘

```

\*\*Microservices (Separate services):\*\*

```

┌─────────────┐ ┌─────────────┐ ┌─────────────┐

│ Upload │ │ Thumbnail │ │ Virus │

│ Service │ │ Service │ │ Scanner │

│ │ │ │ │ │

│ Java │ │ Python │ │ Go │

│ Port 8080 │ │ Port 8081 │ │ Port 8082 │

└─────────────┘ └─────────────┘ └─────────────┘

│ │ │

└────────────────┼────────────────┘

│

┌─────────┐

│ Kafka │

│ (Events)│

└─────────┘

│

┌────────────────┼────────────────┐

↓ ↓ ↓

┌─────────────┐ ┌─────────────┐ ┌─────────────┐

│ Email │ │ Search │ │ Analytics │

│ Service │ │ Indexer │ │ Service │

│ │ │ │ │ │

│ Node.js │ │ Python │ │ Scala │

│ Port 8083 │ │ Port 8084 │ │ Port 8085 │

└─────────────┘ └─────────────┘ └─────────────┘

```

\*\*Benefits:\*\*

- ✅ Independent deployment

- ✅ Technology flexibility

- ✅ Team autonomy

- ✅ Fault isolation

- ✅ Independent scaling

---

### 3. CAP Theorem

\*\*CAP Theorem\*\*: In a distributed system, you can only guarantee 2 out of 3:

1. \*\*Consistency\*\*: All nodes see the same data at the same time

2. \*\*Availability\*\*: Every request receives a response

3. \*\*Partition Tolerance\*\*: System continues despite network failures

\*\*Examples:\*\*

\*\*PostgreSQL\*\* (CP - Consistency + Partition Tolerance):

```

Scenario: Network partition between database replicas

Choice: Reject writes to maintain consistency

Result: Some requests fail, but data stays consistent

Use case: Banking (consistency critical)

```

\*\*Cassandra\*\* (AP - Availability + Partition Tolerance):

```

Scenario: Network partition between database nodes

Choice: Accept writes on all nodes

Result: All requests succeed, but data may be inconsistent temporarily

Use case: Social media (availability critical)

```

\*\*Our Architecture:\*\*

- \*\*PostgreSQL\*\*: CP (consistency for metadata)

- \*\*S3/MinIO\*\*: AP (availability for file storage)

- \*\*Kafka\*\*: AP (availability for event delivery)

---

### 4. ACID vs BASE

\*\*ACID\*\* (Traditional databases):

- \*\*A\*\*tomicity: All or nothing

- \*\*C\*\*onsistency: Data always valid

- \*\*I\*\*solation: Transactions don't interfere

- \*\*D\*\*urability: Committed data persists

\*\*Example:\*\*

```java

// Bank transfer - ACID

beginTransaction();

try {

account1.subtract(100); // Must succeed

account2.add(100); // Must succeed

commit(); // Both succeed or both fail

} catch (Exception e) {

rollback(); // Undo everything

}

```

\*\*BASE\*\* (Modern distributed systems):

- \*\*B\*\*asically \*\*A\*\*vailable: System is always available

- \*\*S\*\*oft state: State may change without input

- \*\*E\*\*ventual consistency: Data becomes consistent eventually

\*\*Example:\*\*

```java

// File upload - BASE

s3.upload(file); // Available immediately

kafka.publish("file.uploaded"); // Eventually processed

// Thumbnails generated eventually

// Search index updated eventually

// User notified eventually

```

---

### 5. Connection Pooling

\*\*Without Pooling\*\* (Slow):

```java

// Every request creates a new connection

void handleRequest() {

Connection conn = DriverManager.getConnection(url, user, pass);

// Connection creation takes 100ms!

// Do work (10ms)

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM files");

conn.close();

// Total: 110ms per request

}

```

\*\*With Pooling\*\* (Fast):

```java

// Connections created once at startup

HikariCP pool = new HikariCP(10); // 10 connections ready

void handleRequest() {

Connection conn = pool.getConnection(); // Instant! (0.1ms)

// Do work (10ms)

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM files");

conn.close(); // Returns to pool, doesn't actually close

// Total: 10ms per request (11x faster!)

}

```

\*\*Benefits:\*\*

- ✅ 10-100x faster

- ✅ Less resource usage

- ✅ Automatic connection management

- ✅ Connection health checks

---

### 6. Serialization & Deserialization

\*\*Serialization\*\*: Converting objects to bytes/string

\*\*Deserialization\*\*: Converting bytes/string back to objects

\*\*Example with JSON:\*\*

```java

// Java Object

FileMetadata metadata = new FileMetadata(

"123",

"photo.jpg",

"uploads/photo.jpg",

"http://s3.com/photo.jpg",

2500000L,

"image/jpeg",

"alice",

1729090624308L

);

// Serialization (Object → JSON String)

ObjectMapper mapper = new ObjectMapper();

String json = mapper.writeValueAsString(metadata);

// Result: {"id":"123","filename":"photo.jpg",...}

// Send JSON over network to Kafka

// Deserialization (JSON String → Object)

FileMetadata received = mapper.readValue(json, FileMetadata.class);

// Result: FileMetadata object with same values

```

\*\*Why JSON?\*\*

- ✅ Human-readable

- ✅ Language-independent

- ✅ Widely supported

- ✅ Easy to debug

\*\*Alternatives:\*\*

- \*\*Protocol Buffers\*\*: Faster, smaller, but binary

- \*\*Avro\*\*: Schema evolution support

- \*\*MessagePack\*\*: Compact binary format

---

### 7. Idempotency

\*\*Idempotent\*\*: Same operation can be repeated without different outcome

\*\*Non-Idempotent\*\* (Bad):

```java

void processPayment(Order order) {

// If this runs twice, charges customer twice! ❌

creditCard.charge(order.getTotalAmount());

}

```

\*\*Idempotent\*\* (Good):

```java

void processPayment(Order order) {

// Check if already processed

if (paymentRepository.exists(order.getId())) {

return; // Already processed, skip

}

// Process payment

creditCard.charge(order.getTotalAmount());

// Mark as processed

paymentRepository.markProcessed(order.getId());

}

```

\*\*In Kafka:\*\*

```java

// Enable idempotence

props.put(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true");

// Kafka prevents duplicate sends automatically

// If network fails and retries, Kafka detects duplicate

```

---

### 8. Eventual Consistency

\*\*Strong Consistency\*\* (Traditional):

```

User uploads file

↓

Write to database (wait for replication to all nodes)

↓

Read from database (always sees latest data)

```

\*\*Eventual Consistency\*\* (Distributed):

```

User uploads file

↓

Write to database (write to one node, return immediately)

↓

Background: Replicate to other nodes

↓

Eventually: All nodes have the same data

```

\*\*Example in Our System:\*\*

```

Time 0:00 - User uploads file

Time 0:01 - File saved to S3, metadata saved to PostgreSQL

Time 0:02 - Kafka event published

Time 0:03 - Consumer starts processing

Time 0:05 - Thumbnails generated

Time 0:10 - Search index updated

Time 0:15 - All systems consistent ✅

```

\*\*During 0:01-0:15:\*\*

- File is uploaded ✅

- Metadata in database ✅

- But thumbnails not ready yet ⏳

- And search doesn't show it yet ⏳

\*\*Eventually (at 0:15):\*\*

- Everything is consistent ✅

---

### 9. Backpressure

\*\*Problem\*\*: Producer sends messages faster than consumer can process

\*\*Without Backpressure\*\* (Bad):

```

Producer: 1000 messages/second

Consumer: 100 messages/second

Result: Queue grows infinitely, memory overflow! 💥

```

\*\*With Backpressure\*\* (Good):

```

Consumer tells producer: "Slow down, I can only handle 100/sec"

Producer throttles to match consumer's capacity

Result: Stable system ✅

```

\*\*In Kafka:\*\*

```java

// Consumer automatically handles backpressure

consumer.poll(Duration.ofMillis(100)); // Fetch as much as can process

// Kafka won't send more until consumer commits offset

```

---

### 10. Schema Evolution

\*\*Problem\*\*: How to update data structure without breaking existing code?

\*\*Example:\*\*

```java

// Version 1

class FileMetadata {

String id;

String filename;

}

// Deployed to production, messages sent

// Version 2 (need to add new field)

class FileMetadata {

String id;

String filename;

String contentType; // NEW FIELD

}

// Problem: Old messages don't have contentType!

```

\*\*Solution 1: Optional Fields\*\*

```java

class FileMetadata {

String id;

String filename;

String contentType = "application/octet-stream"; // Default value

}

```

\*\*Solution 2: Schema Registry\*\* (Avro, Protobuf):

```java

// Schema Registry tracks versions

// V1: {id, filename}

// V2: {id, filename, contentType}

// Consumer knows how to read both versions

```

---

<a name="troubleshooting"></a>

## 12. Troubleshooting & Common Issues

### Issue 1: Kafka Connection Refused

\*\*Error:\*\*

```

org.apache.kafka.common.errors.TimeoutException:

Failed to update metadata after 60000 ms.

```

\*\*Cause\*\*: Kafka not running or wrong host/port

\*\*Solution:\*\*

```bash

# Check if Kafka is running

docker-compose ps

# Should see:

# kafka-broker Up (healthy)

# If not running:

docker-compose up -d kafka

# Check logs:

docker-compose logs kafka

```

---

### Issue 2: PostgreSQL Connection Failed

\*\*Error:\*\*

```

org.postgresql.util.PSQLException:

Connection refused. Check that the hostname and port are correct.

```

\*\*Cause\*\*: PostgreSQL not running or wrong credentials

\*\*Solution:\*\*

```bash

# Check if PostgreSQL is running

docker-compose ps postgres

# Test connection:

docker-compose exec postgres psql -U admin -d mydb

# If password error, check docker-compose.yml:

postgres:

environment:

POSTGRES\_USER: admin # Must match code

POSTGRES\_PASSWORD: admin123 # Must match code

```

---

### Issue 3: MinIO Access Denied

\*\*Error:\*\*

```

software.amazon.awssdk.services.s3.model.S3Exception:

Access Denied (Service: S3, Status Code: 403)

```

\*\*Cause\*\*: Wrong access key/secret key

\*\*Solution:\*\*

```java

// Check credentials in code

S3Manager s3Manager = new S3Manager(

"http://localhost:9000",

"minioadmin", // Must match docker-compose.yml

"minioadmin123" // Must match docker-compose.yml

);

```

```yaml

# Check docker-compose.yml

minio:

environment:

MINIO\_ROOT\_USER: minioadmin

MINIO\_ROOT\_PASSWORD: minioadmin123

```

---

### Issue 4: Out of Memory

\*\*Error:\*\*

```

java.lang.OutOfMemoryError: Java heap space

```

\*\*Cause\*\*: Uploading too large files or too many at once

\*\*Solution 1: Increase JVM memory\*\*

```bash

# Add to run command

java -Xmx2g -Xms512m -jar app.jar

# -Xmx2g = max 2 GB

# -Xms512m = start with 512 MB

```

\*\*Solution 2: Stream large files\*\*

```java

// BAD: Loads entire file into memory

byte[] fileData = Files.readAllBytes(path); // 1 GB file = 1 GB memory!

// GOOD: Stream file in chunks

try (InputStream is = Files.newInputStream(path)) {

s3Client.putObject(request, RequestBody.fromInputStream(is, fileSize));

}

```

---

### Issue 5: Kafka Lag (Consumer Too Slow)

\*\*Symptom\*\*: Events processed with significant delay

\*\*Diagnosis:\*\*

```bash

# Check consumer lag

docker-compose exec kafka kafka-consumer-groups \

--bootstrap-server localhost:9092 \

--describe \

--group complete-integration-group

# Output shows:

# GROUP TOPIC PARTITION CURRENT-OFFSET LAG

# complete-integration-group file-events 0 1000 500

# complete-integration-group file-events 1 1000 500

# complete-integration-group file-events 2 1000 500

# Total lag: 1500 messages behind!

```

\*\*Solution 1: Add more consumers\*\*

```java

// Run multiple instances

// They'll automatically share the work

// Max consumers = number of partitions (3 in our case)

// Terminal 1

java -jar app.jar

// Terminal 2

java -jar app.jar

// Terminal 3

java -jar app.jar

```

\*\*Solution 2: Optimize processing\*\*

```java

// BAD: Slow processing

Consumer<FileEvent> handler = event -> {

// Synchronous processing (slow)

generateThumbnails(event); // 10 seconds

scanVirus(event); // 15 seconds

sendEmail(event); // 2 seconds

// Total: 27 seconds per event!

};

// GOOD: Parallel processing

Consumer<FileEvent> handler = event -> {

// Async processing (fast)

CompletableFuture.runAsync(() -> generateThumbnails(event));

CompletableFuture.runAsync(() -> scanVirus(event));

CompletableFuture.runAsync(() -> sendEmail(event));

// Total: Max 15 seconds per event (3 tasks in parallel)

};

```

---

### Issue 6: Database Deadlock

\*\*Error:\*\*

```

org.postgresql.util.PSQLException:

ERROR: deadlock detected

```

\*\*Cause\*\*: Two transactions waiting for each other

\*\*Example:\*\*

```

Transaction 1: Lock file A, waiting for file B

Transaction 2: Lock file B, waiting for file A

Result: Deadlock! 💀

```

\*\*Solution: Consistent lock ordering\*\*

```java

// BAD: Random order

void transferFiles(String fileId1, String fileId2) {

lockFile(fileId1);

lockFile(fileId2);

// ...

}

// GOOD: Always lock in same order

void transferFiles(String fileId1, String fileId2) {

String first = fileId1.compareTo(fileId2) < 0 ? fileId1 : fileId2;

String second = fileId1.compareTo(fileId2) < 0 ? fileId2 : fileId1;

lockFile(first);

lockFile(second);

// ...

}

```

---

### Issue 7: S3 Upload Timeout

\*\*Error:\*\*

```

software.amazon.awssdk.core.exception.SdkClientException:

Unable to execute HTTP request: Read timed out

```

\*\*Cause\*\*: Large file upload taking too long

\*\*Solution 1: Increase timeout\*\*

```java

S3Client s3Client = S3Client.builder()

.overrideConfiguration(config -> config

.apiCallTimeout(Duration.ofMinutes(10)) // 10 min timeout

.apiCallAttemptTimeout(Duration.ofMinutes(5)))

.build();

```

\*\*Solution 2: Multipart upload for large files\*\*

```java

public void uploadLargeFile(String bucket, String key, File file) {

CreateMultipartUploadRequest createRequest = CreateMultipartUploadRequest.builder()

.bucket(bucket)

.key(key)

.build();

String uploadId = s3Client.createMultipartUpload(createRequest).uploadId();

// Upload in 5 MB chunks

int partSize = 5 \* 1024 \* 1024;

int partNumber = 1;

List<CompletedPart> completedParts = new ArrayList<>();

try (FileInputStream fis = new FileInputStream(file)) {

byte[] buffer = new byte[partSize];

int bytesRead;

while ((bytesRead = fis.read(buffer)) > 0) {

UploadPartRequest uploadRequest = UploadPartRequest.builder()

.bucket(bucket)

.key(key)

.uploadId(uploadId)

.partNumber(partNumber)

.build();

UploadPartResponse response = s3Client.uploadPart(uploadRequest,

RequestBody.fromBytes(Arrays.copyOf(buffer, bytesRead)));

completedParts.add(CompletedPart.builder()

.partNumber(partNumber)

.eTag(response.eTag())

.build());

partNumber++;

}

}

// Complete upload

CompleteMultipartUploadRequest completeRequest = CompleteMultipartUploadRequest.builder()

.bucket(bucket)

.key(key)

.uploadId(uploadId)

.multipartUpload(CompletedMultipartUpload.builder()

.parts(completedParts)

.build())

.build();

s3Client.completeMultipartUpload(completeRequest);

}

```

---

### Issue 8: Consumer Rebalancing Loop

\*\*Symptom\*\*: Consumer constantly disconnecting and reconnecting

\*\*Logs:\*\*

```

[INFO] Revoking previously assigned partitions [file-events-0, file-events-1]

[INFO] Setting newly assigned partitions [file-events-0, file-events-1]

[INFO] Revoking previously assigned partitions [file-events-0, file-events-1]

[INFO] Setting newly assigned partitions [file-events-0, file-events-1]

// Repeats forever...

```

\*\*Cause\*\*: Consumer processing takes longer than session timeout

\*\*Solution: Increase timeouts\*\*

```java

Properties props = new Properties();

props.put(ConsumerConfig.SESSION\_TIMEOUT\_MS\_CONFIG, "30000"); // 30 seconds

props.put(ConsumerConfig.MAX\_POLL\_INTERVAL\_MS\_CONFIG, "300000"); // 5 minutes

props.put(ConsumerConfig.MAX\_POLL\_RECORDS\_CONFIG, "10"); // Process fewer records

consumer = new KafkaConsumer<>(props);

```

---

### Debugging Checklist

\*\*When something doesn't work:\*\*

```

☐ 1. Check all services are running

docker-compose ps

(All should show "Up" or "Up (healthy)")

☐ 2. Check service logs

docker-compose logs kafka

docker-compose logs postgres

docker-compose logs minio

☐ 3. Check network connectivity

# From your Java app, can you reach services?

telnet localhost 9092 # Kafka

telnet localhost 5432 # PostgreSQL

telnet localhost 9000 # MinIO

☐ 4. Check credentials

# Are usernames/passwords correct?

# Check docker-compose.yml matches your code

☐ 5. Check disk space

df -h

# Ensure you have enough space for data

☐ 6. Check Java version

java -version

# Should be Java 11 or higher

☐ 7. Check Maven dependencies

mvn dependency:tree

# Ensure all dependencies are downloaded

☐ 8. Clean and rebuild

mvn clean compile

docker-compose down -v

docker-compose up -d

```

---

## Conclusion

Congratulations! You now understand:

1. ✅ \*\*Apache Kafka\*\* - Message streaming and event processing

2. ✅ \*\*PostgreSQL\*\* - Relational database and metadata storage

3. ✅ \*\*MinIO (S3)\*\* - Blob storage for large files

4. ✅ \*\*Java Application\*\* - Orchestrating all components

5. ✅ \*\*Docker\*\* - Containerization and deployment

6. ✅ \*\*Event-Driven Architecture\*\* - Building scalable systems

7. ✅ \*\*Real-World Applications\*\* - How Netflix, Instagram, Dropbox work

8. ✅ \*\*Troubleshooting\*\* - Solving common issues

### Next Steps

1. \*\*Extend the project\*\*:

- Add user authentication

- Implement file sharing

- Add thumbnail generation

- Create a REST API

- Build a web frontend

2. \*\*Learn more\*\*:

- Kubernetes (container orchestration)

- Redis (caching)

- Elasticsearch (search)

- GraphQL (API design)

- React (frontend)

3. \*\*Deploy to production\*\*:

- Use AWS S3 instead of MinIO

- Use AWS MSK instead of local Kafka

- Use AWS RDS instead of local PostgreSQL

- Add monitoring (Prometheus, Grafana)

- Add logging (ELK stack)

### Resources

- \*\*Kafka\*\*: https://kafka.apache.org/documentation/

- \*\*PostgreSQL\*\*: https://www.postgresql.org/docs/

- \*\*AWS S3\*\*: https://docs.aws.amazon.com/s3/

- \*\*Docker\*\*: https://docs.docker.com/

- \*\*Java\*\*: https://docs.oracle.com/en/java/

---

\*\*Thank you for learning with this guide!\*\* 🎉

\*If you have questions or find issues, please refer to the README.md and other documentation files in the project.\*