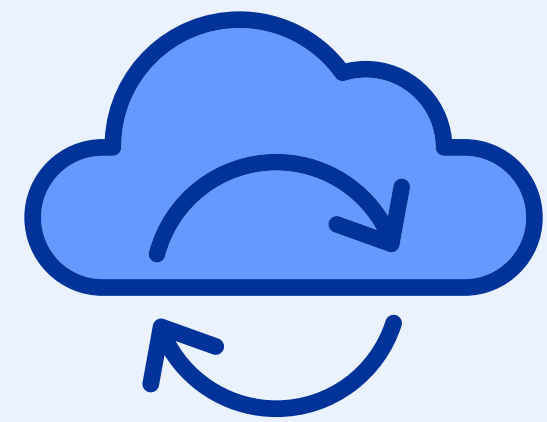
codewithpal @palmuruganc

Java Async Programming - Complete Practical Guide



https://www.linkedin.com/in/palmuruganc

What is Async Programming?

- Asynchronous programming lets tasks run without blocking the main thread.

- Ideal when your application must wait (I/O, network calls, DB, file ops) but you don't want to stop other requests.

- Example: A request triggers file upload + email sending → async allows parallel execution so your API stays fast.

- Async ≠ Multithreading always — async is about non-blocking behavior, not just creating threads.

Why Async Matters?

- Improves throughput without adding more servers.
- Reduces request latency by parallelizing waiting operations.
- Helps build scalable microservices where many operations depend on external calls.
- Works best in:
  - WebFlux (non-blocking apps)
  - Microservices
  - Event-driven systems
  - High-traffic backends

@Async in Spring

What it does:

Runs a method in a separate thread managed by Spring's TaskExecutor.

Good for:

- Sending emails
- Audit logging
- File uploads
- Database archive tasks
- Webhook triggers

When NOT to use:

- Heavy CPU logic
- High-volume real-time tasks
- Scenarios where you need guaranteed execution (use MQ)

@Async: Real Prod Use Case

Scenario:

User signs up → you need to:

1. Save Details
2. Send welcome email
3. Push analytics event
4. Log activity

Only step (1) must block.

Steps 2–4 → perfect for @Async.

Benefit:

Signup API stays <100 ms.

Background jobs run parallelly automatically.

CompletableFuture

- Java's most powerful async abstraction.
- Lets you build async pipelines like:
  - Run task async
  - Combine two async results
  - Handle errors
  - Run tasks in parallel

Useful for:

- Calling multiple microservices in parallel
- Aggregating API responses
- Fan-out + fan-in logic
- Parallel DB queries
- CPU/IO split operations

CF Real Use Case

Scenario: Product Detail Page (like Amazon).

For 1 product, you need:

- Product info API
- Price service
- Inventory service
- Reviews service

If each takes 200 ms sequentially → **800 ms total.**

With CompletableFuture: All run parallel → returns in **200–250 ms.**

Pattern:

allOf() → wait for all.

anyOf() → return fastest data.

thenCombine() → merge results.

Message Brokers – True Async

Message broker (Kafka, RabbitMQ, SQS, Redis Streams) provides:

- Guaranteed delivery
- Retry & dead-letter queues
- Horizontal scaling
- Decoupled services
- Event-driven architecture

Use when you need:

- Reliability
- High throughput
- Background processing
- Multi-service workflows
- Event sourcing

Message Broker Use Case

Scenario: Order Placed → triggers 5 tasks

1. Payment processing
2. Inventory update
3. Notification
4. Invoice generation
5. Analytics event

If API waits for all → response becomes **5–8 seconds**.

With message broker:

- API only publishes OrderCreated event
- All subscribers process independently
- Real async guaranteed
- Each consumer scalable separately

This is the most scalable async pattern.

Reactive Programming

Reactive = non-blocking + event-driven.

Uses **Mono** and **Flux** to handle async streams.

Good for:

- Chat applications
- Live dashboards
- Data streaming
- High-concurrency APIs
- IO heavy microservices

Not good for:

- Heavy CPU tasks
- Apps requiring JDBC blocking queries

Reactive Use Case

Scenario:
A trading application showing real-time price updates for thousands of users.

WebFlux + Flux streams = push data async to all connected users **without blocking threads.**

Traditional servlet model would require 1000x more threads.

When @Async is Best

Use **@Async** when:

- Fire-and-forget tasks
- Short background work
- Low volume events
- Email / Logging / Notifications
- You control the thread pool

Avoid when reliability is required.

When CompletableFuture is Best

Use **CompletableFuture** when:

- You need parallel execution inside one service
- You want to combine multiple API/DB calls
- You need async pipelines
- Low latency API aggregation

Not ideal for inter-service guarantees.

When Message Broker is Best

Use **Kafka/RabbitMQ/SQS** when:

- Workload is large
- Tasks must never be lost
- Services must be decoupled
- Horizontal scaling required
- Long-running tasks involved
- Need event-driven workflow

This is real enterprise-level async.

codewithpal

@palmuruganc

Follow Me

"Stay ahead with cutting-edge technical tips and best practices—follow me on LinkedIn today!"

@palmuruganc

https://medium.com/@palmurugan.c