



Java Technologies

Lecture 9

Building Cloud-Native Applications

Fall, 2025

# Agenda

- "Distributed Systems Hell", Spring Cloud
- Observability: Metrics, Logs, Traces
- Service Discovery, Service Registry
- API Gateway, Load Balancing
- Managing Configuration
- Storing Secrets, Vaults
- Messaging, Spring Cloud Stream

# "Distributed Systems Hell"

- Microservices are great: flexibility, technology diversity, independent deployment, scalability. But...
- How to manage configurations for dozens of services?
- How to prevent a failure in one service from cascading to others?
- How do services find and talk to each other?
- How to provide a unified entry point for clients?
- How to debug and monitor thousands of requests that flow through multiple services?

---

We need a set of tools for building common patterns in distributed systems.

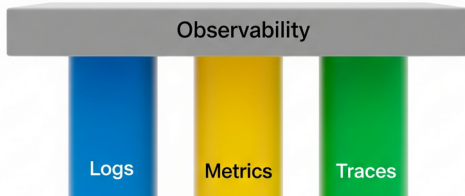


## Spring Cloud is a toolbox for microservices

- Observability
- Service discovery & Registration
- Routing & API gateway
- Load balancing & Resilience
- Distributed configuration & Secrets
- Event-driven messaging / streaming
- Cloud provider integration
- Security

# Observability

- We are in the context of a distributed system with a **large number of communicating nodes (microservices)**.
  - How do we know if the services function properly?
  - How do we measure the performance of the services?
  - If something went wrong, how to trace the source of error?
- **Monitoring**: Collecting and analyzing **predefined** outputs to check the health and performance of a system.
- **Observability**: The ability to **understand the internal state** of a system by analyzing its outputs, even for unknown issues.



# Metrics

- Metrics are **numerical measurements** that provide a high-level, time-series view of a service's performance.
- The Four Golden Signals (The Foundation)
  - ① **Latency**: The time it takes to serve a request.
  - ② **Traffic**: How much demand is being placed on your system.
  - ③ **Errors**: The rate of requests that fail.
  - ④ **Saturation**: How "full" your service is.
- What to measure?
  - **Infrastructure Metrics**: CPU, Memory, Disk I/O, Network I/O, Platform specific values.
  - **Application Metrics** (Business Logic): Login attempts, Orders processed per second, DB connection pool size.
  - **Runtime Metrics**: JVM Memory, GC Cycles, Thread Count

# SLI, SLO, SLA

- **SLI – Service Level Indicator:** A quantitative measure (metric) of some aspect of a service's performance or reliability. It is the actual performance of a service.  
"In a 30-day month, the service was down for 30 minutes."
- **SLO – Service Level Objective:** A target value or range for an SLI. It defines what level of service you aim to achieve.  
"Our service should have at least 99.99% uptime per month."
- **SLA – Service Level Agreement:** A formal contract between a service provider and a customer specifying guaranteed levels of service and consequences if they aren't met.  
"Commitment: 99.9% uptime per month."  
"Penalty if missed: If uptime falls below 99.9%, the customer gets 1 day of service credit per 0.1% below target."

# Micrometer Library

- Vendor-neutral application observability facade.
- Works with Prometheus, Grafana, Datadog, JMX, and more.



# Spring Boot Actuator + Micrometer

- Collect application metrics (memory, CPU, requests).
- Collect custom metrics (counters, timers, gauges).
- Export metrics to external monitoring systems.
- Observe real-time performance and troubleshoot issues.

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-core</artifactId>
</dependency>

<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

```
management.endpoints.web.exposure.include=metrics,prometheus
management.endpoint.prometheus.enabled=true
```

# Counters & Timers

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    @Timed(value = "hello.timer")
    @Counted(value = "hello.requests")
    public String hello() throws InterruptedException {
        Thread.sleep(500); // simulate some work
        return "Hello, World!";
    }
} // @Timed and @Counted are from io.micrometer
```

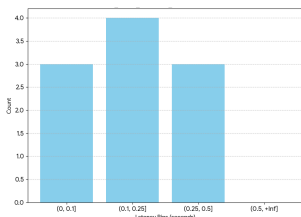
```
/actuator/metrics/hello.requests
/actuator/metrics/hello.timer
/actuator/prometheus
```

```
hello_requests_total      5
hello_timer_seconds_count 5
hello_timer_seconds_sum   1.5
hello_timer_seconds_max   0.35
```

# Histograms & Percentiles

```
@Timed(  
    value = "work.timer",  
    description = "Time spent processing work",  
    percentiles = {0.5, 0.95}, // enable p50, p95  
    histogram = true           // enable histogram buckets  
)  
@GetMapping("/work")  
public String doWork() throws InterruptedException {  
    Thread.sleep((long) (Math.random() * 500));  
    return "done";  
}
```

```
work_timer_seconds_bucket{le="0.1"} 3  
work_timer_seconds_bucket{le="0.25"} 7  
work_timer_seconds_bucket{le="0.5"} 10  
work_timer_seconds_bucket{le="+Inf"} 10  
work_timer_seconds_bucket 10  
work_timer_seconds_sum 3.1  
  
work_timer_seconds{quantile="0.5"} 0.23  
work_timer_seconds{quantile="0.95"} 0.48
```



# Gauges

A **gauge** is a metric that represents a value at a specific point in time. Unlike counters (which only go up), gauges can go up or down.

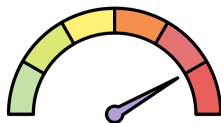
```
@Component
public class TaskQueue {

    private final AtomicInteger queueSize = new AtomicInteger(0);

    // Annotated getter exposes the value as a Gauge
    @Gauge(value = "task.queue.size",
           description = "Number of tasks in the queue")
    public int getQueueSize() {
        return queueSize.get();
    }

    public void addTask() {
        queueSize.incrementAndGet();
    }

    public void completeTask() {
        queueSize.decrementAndGet();
    }
}
```



- Timestamped, unstructured or structured text records of discrete events that happened at a specific time.
- Help trace application flow, errors, and state changes.
- **Best Practices:**
  - Use structured logging (JSON), for filtering and querying
  - Add context (a unique trace ID of a request), to aggregate data.
  - Define log levels (trace > debug > info > warn > error > off).
  - Logs should be centralized.
- **SL4J + Logback** is the default logging setup in Spring Boot.
- **SLF4J** (Simple Logging Facade for Java) is an abstraction (facade) over different logging frameworks.
  - Decouples code from logging implementation
  - Unified logging API
- **Logback** is a popular, high-performance logging framework

# Logging in Spring Boot

- spring-boot-starter includes SLF4J + Logback.

```
@Service
public class HelloService {
    private static final Logger log =
        LoggerFactory.getLogger(HelloService.class);

    public String sayHello(String name) {
        log.info("sayHello called with name={}", name);
        // .debug, warn, error
        return "Hello, " + name + "!";
    }
} // Logger, LoggerFactory are from org.slf4j package
```

- Out of the box, the messages are written only to the console.

```
INFO c.e.d.HelloService : sayHello called with name=Alice
```

- Logging can be **configured** in application.properties

```
logging.level.com.example.demo=INFO # level per package
logging.file.name=myapp.log # archiving/rotation
```

# Structured Logging

- To centralize and filter or query logs from different microservices easily, **the log data must be structured**.
- **Logstash** is an open-source, server-side data processing pipeline that collects data from multiple sources, transforms it through various plugins, and then sends it to a desired destination.
- Logback can integrate with Logstash using **logstash-logback-encoder**, in order to generate JSON output.
- The text of the log entries now will be in JSON format:

```
{
  "timestamp": "2025-09-01T12:34:56.789Z",
  "level": "INFO",
  "logger": "com.example.demo.HelloService",
  "message": "sayHello called with name=Alice",
  "requestId": "d4f9a2b8-7c91-4b8a-9fd6-2a51c7e6d5f1" 
}
```

# Correlated Log Entries

- To correlate a log entry with an actual request received by a microservice, that entry must include a **unique request ID** (also called correlation ID, or trace ID).
- How to add the correlation ID in the generated JSON?
- **MDC (Mapped Diagnostic Context)** lets you attach key-value pairs to the logging context of the current thread.
- A simple **web filter** can be used to intercept all HTTP requests and generate/propagate a unique correlation ID.

```
@Component
public class RequestIdFilter implements Filter {
    public void doFilter( ... ) {
        ...
        MDC.put("requestId", UUID.randomUUID().toString());
        chain.doFilter(servletRequest, servletResponse);
        MDC.remove("requestId");
    }
} // MDC is similar to a ThreadLocal map
```

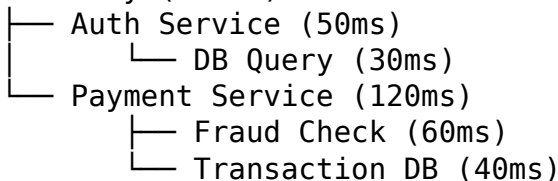
# Centralized Logging

- Using a simple logging approach results in log files **scattered across** multiple services, containers, and machines.
- **Logging centralization** is the practice of collecting and consolidating log data into a **single, central location**.
  - 1 **Collection:** Log agents ("shippers") are installed on each server. They collect log data as it is generated.
  - 2 **Aggregation & Storage:** The collected logs are streamed in real-time to a central log management platform.
  - 3 **Analysis & Visualization:** A user interface provides search, filtering and visualization capabilities (dashboards, graphs).
- **ELK/EFK Stack** is the most well-known open-source stack:
  - Elasticsearch (storage and search);
  - Logstash/Fluentd (processing pipeline);
  - Kibana (visualization).

# Distributed Tracing

- In a microservices system, a single user request may trigger a chain of calls across many independent services.

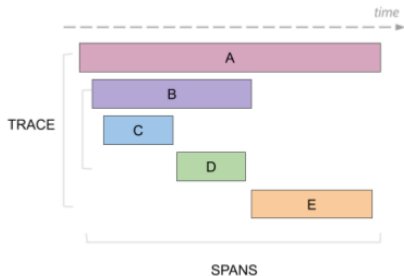
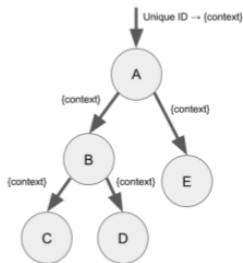
API Gateway (200ms)



- **Tracing** captures this request path end-to-end.
  - Where time is spent (latency breakdown per service).
  - Which services were involved in processing a request.
  - Where failures occur (errors, bottlenecks, retries).
- **Why it matters:** Debugging, root-cause analysis, improves observability when combined with metrics & logs.

# Traces & Spans

- **Trace:** A record of the journey of a request across services.
- **Span:** A single operation within a service.
- **Trace ID:** A unique ID shared by all spans in one request.
- **Span ID & Parent ID:** Used to link spans together.
- A trace in microservices forms a **directed tree**.



# How to Implement Distributed Tracing

- 1 **Choose a backend** to collect traces and visualize the flow of requests: [Jaeger](#), Zipkin, Grafana Tempo.
- 2 **Instrument the services**: [Open Telemetry](#) uses bytecode manipulation to automatically instrument the code. This provides **out-of-the-box tracing** for common operations.

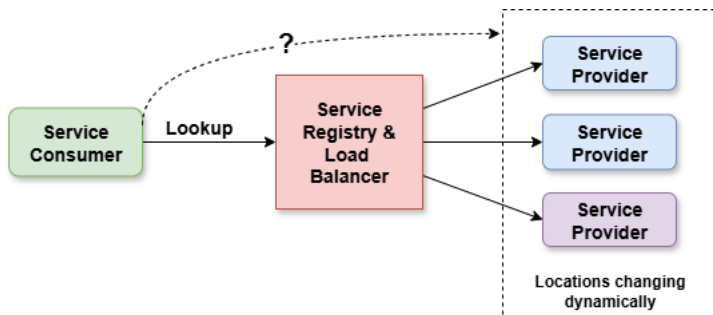
```
java -javaagent:opentelemetry-javaagent.jar ...
```

- 3 **Ensure context propagation**: The OTel instrumentation handle this automatically for HTTP calls. It injects headers into outgoing requests and extract them from incoming requests.
- 4 **Query and analyze**: Use Jaeger's UI.

```
http://localhost:16686  
Select your services → See the traces
```

# Service Discovery

- Microservices are often deployed across multiple hosts, containers, or clusters, and their locations change frequently.
- **Service Discovery** is a mechanism that allows microservices to **find and communicate with each other dynamically**, without requiring hard-coded network locations.



# Setting up a Service Registry

- Create a Spring Boot Application

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>  
    spring-cloud-starter-netflix-eureka-server  
  </artifactId>  
</dependency>
```

- Make the app an **Eureka Server**

```
@SpringBootApplication  
@EnableEurekaServer
```

- Configure application.properties

```
server.port=8761 # The default Eureka port  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false
```

- Alternatives to Eureka: Consul, Kubernetes DNS

# Registering a Microservice with the Server

- Each microservice app must register with the server.

```
spring-cloud-starter-netflix-eureka-client
```

- Make the app an **Eureka Client** (optional)

```
@SpringBootApplication  
@EnableEurekaClient
```

- Configure application.properties

```
spring.application.name=user-service  
# This is the unique service ID for discovery!  
  
server.port=0 # Let Spring choose a random port (common in cloud)  
               # or use a fixed port.  
  
eureka.client.service-url.defaultZone=  
                                     http://localhost:8761/eureka
```

# Calling a Microservice via Server Registry

- Use RestTemplate or WebClient with **load balancing**

```
@Configuration
public class AppConfig {
    @Bean @LoadBalanced // ←Client-side load balancing.
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

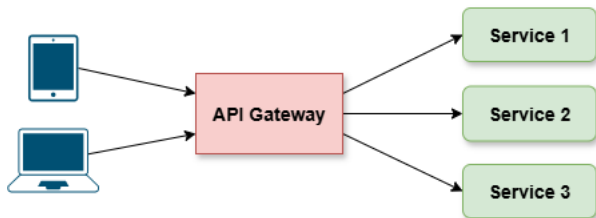
- Make calls using the service name instead of the hardcoded URL

```
@Service
public class OrderService {
    @Autowired private RestTemplate restTemplate;

    public User getUser(String userId) {
        var response = restTemplate.getForEntity(
            "http://user-service/users/" + userId, User.class);
        return response.getBody();
    }
} // "user-service" is resolved by Eureka
```

# The API Gateway Pattern

- An API Gateway is a **single entry point** that sits between web/mobile clients and the backend (micro)services.



- It handles things like:
  - **Routing**: Forwarding client requests to the correct microservice.
  - **Load balancing**: Distributing requests across instances.
  - **Security**: Authentication, authorization, and rate limiting.
  - **Cross-cutting concerns**: Logging, monitoring, retries, caching.

# Spring Cloud Gateway

- Create a "Gateway" Spring Boot application.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

- Configure application.yml (the gateway port is the default 8080).

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081
          predicates:
            - Path=/users/**
          filters:
            - AddRequestHeader=X-Request-Source, Gateway
# Client calls http://localhost:8080/users/1
# The gateway sends all /users/** requests
# to http://localhost:8081, and adds a request header.
```

# Gateway Predicates & Filters

- **Predicates** are conditions that decide whether a request should match a route. If all predicates for a route are satisfied, the request will be routed to the destination (uri).

```
Path=/users/**  
Method=GET,POST  
Header=X-Request-Source, Gateway  
Query=version, v1
```

- **Filters** are actions applied to requests (pre-filters) and responses (post-filters) once a route has been matched.  
Examples: AddRequestHeader, SetRequestHeader, AddResponseHeader, RewritePath, RequestRateLimiter, CircuitBreaker

```
AddRequestHeader=X-Request-Source, Gateway  
RewritePath=/api/(?<segment>.*), /${segment}
```

# Distributing Network Traffic

- **Client-Side Load Balancing**

- There is no centralized load balancer.
- The client is responsible for choosing the service.
- Uses a service discovery mechanism (like Eureka or Consul).

- **Server-Side Load Balancing**

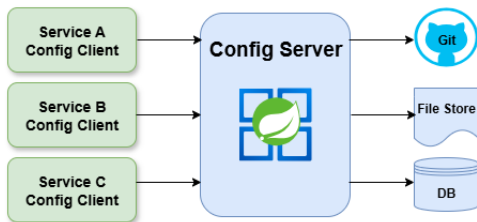
- Uses a hardware or software dedicated component.
- All incoming client requests first hit the load balancer.
- NGINX, API Gateway

- **Load Balancing Algorithms**

- Random
- (Weighted) Round Robin, with Health Checks
- (Weighted) Least Connections
- Least Response Time / Bandwidth / Load
- IP Hash (sticky session)

# Managing Configurations in Distributed Systems

- What happens if we have to deploy services on multiple machines and we have to change their configurations?
- **Spring Cloud Config** solves the "configuration management problem" in the microservice architectures.
- A **Config Server** is a centralized service that manages and distributes configuration settings.
- A **Config Client** is a service or application that fetches its configuration settings from a Config Server instead of local files.



# How to Use Spring Cloud Config

- **Set up the Config Server:** `spring-cloud-config-server`
  - `@EnableConfigServer` on the main Spring Boot class.
  - Prepare a Git repository to store config files (most common).

```
spring.cloud.config.server.git.uri=https://github.com/my-repo.git
```

- Organize configuration files in the repository.

```
{application}-{profile}.yml or {application}-{profile}.properties
```

- **Set up the Config Clients:** `spring-cloud-starter-config`
  - Point to the Config Server.

```
# In bootstrap.yml or bootstrap.properties  
spring.config.import=optional:configserver:http://localhost:8888
```

- Make sure the client has an application name.

```
# In application.yml or application.properties  
spring.application.name=my-service
```

- Access the properties from the Spring environment

# Storing Secrets Securely

- How to manage sensitive information in a secure and centralized manner, rather than hardcoding it into application code, config files, or environment variables on individual servers?
- **The Secrets:** API Keys, database passwords, SSL certificates, OAuth tokens or JWTs, SSH keys.
- **The Problem:** If a secret is hardcoded or stored in a simple file → security risks, secret sprawl, lack of centralization.
- **The Solution:** **Secrets Management System** or **Vault**.
  - Centralized storage
  - Fine grained access control policies (RBAC)
  - Dynamic secrets (temporary database passwords)
  - Complete audit trail (who accessed which secret)
  - Rotation on a regular schedule, and revocation

# Spring Boot + HashiCorp Vault

- **HashiCorp Vault**, AWS Secrets Manager, Google Cloud Secret Manager, Azure Key Vault, Kubernetes Secrets.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-vault-config</artifactId>
</dependency>
```

- Configure the Vault Client bootstrap.yml

```
spring:
  cloud:
    vault:
      uri: http://localhost:8200
      token: s.xxxxxx
      kv:
        enabled: true
        backend: secret
# token = authentication token your app uses to talk to Vault.
# kv = Key-Value (KV) secrets engine
# backend = the Vault path where your secrets live
```

# How a Vault Client Works

- Assume that a service, named `user-service` has the username and the password of a DB stored in the Vault.

```
vault kv put secret/user-service/db username=alice password=pwd
```

- Access them in `application.yml` for default configurations:

```
spring:  
  datasource:  
    username: ${db.username}  
    password: ${db.password}
```

- Or inject them in the source code:

```
@Value("${db.username}")  
private String dbUser;  
  
@Value("${db.password}")  
private String dbPassword;
```

# Messaging in Microservices Architectures

- Messaging is a method of **asynchronous** communication between services or applications (agents).
- Instead of calling each other directly, services send messages to a **broker**, which routes and delivers them to consumers.
- It enables **decoupled, reliable, and scalable** communication.
- **It is paramount for microservices architectures.**



# Spring Cloud Stream

- A framework for building scalable **event-driven microservices** that are connected by a shared messaging system.
- Abstracts away the complexities of the underlying message brokers, such as Apache Kafka or RabbitMQ.
- Key concepts:
  - **Binders**: The components that bridge your application to the external messaging system.
  - **Channels**: The communication endpoints within your app: input (for consuming) and output (for producing).
  - **Messages**: Payload and headers.
  - **Functional Programming Model**: The message producers, consumers, and processors are Supplier, Consumer, or Function beans → reduces boilerplate code.
- Similar to **Eclipse MicroProfile Reactive Messaging**

# Producer – Broker – Consumer Pipeline

- Add the dependency for Kafka or RabbitMQ.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- Configure application.yml

```
spring:
  cloud:
    stream:
      bindings:
        output-out-0: # producer channel
          destination: messages
        input-in-0: # consumer channel
          destination: messages
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

# Producer & Consumer Components

```
@Component
public class MessageProducer {

    @Bean
    public Supplier<String> output() {
        return () -> "Hello from functional producer!";
    }
} // auto-generated binding name: output-out-0
```

```
@Component
public class MessageConsumer {

    @Bean
    public Consumer<String> input() {
        return message ->
            System.out.println("Received: " + message);
    }
} // auto-generated binding name: input-in-0
```

Use StreamBridge for the old-style manual send/receive.

# Producer, Processor, Consumer

```
# Service A is a Producer  
# Supplier<String> produce  
  
spring.cloud.stream.bindings.produce-out-0.destination=topic-A
```

```
# Service B is a Processor; it receives from A and sends to B  
# Function<String, String> process
```

```
spring:  
  cloud:  
    stream:  
      bindings:  
        process-in-0:  
          destination: topic-A    # Input from Service A  
        process-out-0:  
          destination: topic-B    # Output to Service C
```

```
# Service C is a Consumer  
# Consumer<String> consume  
  
spring.cloud.stream.bindings.consume-in-0.destination=topic-B
```