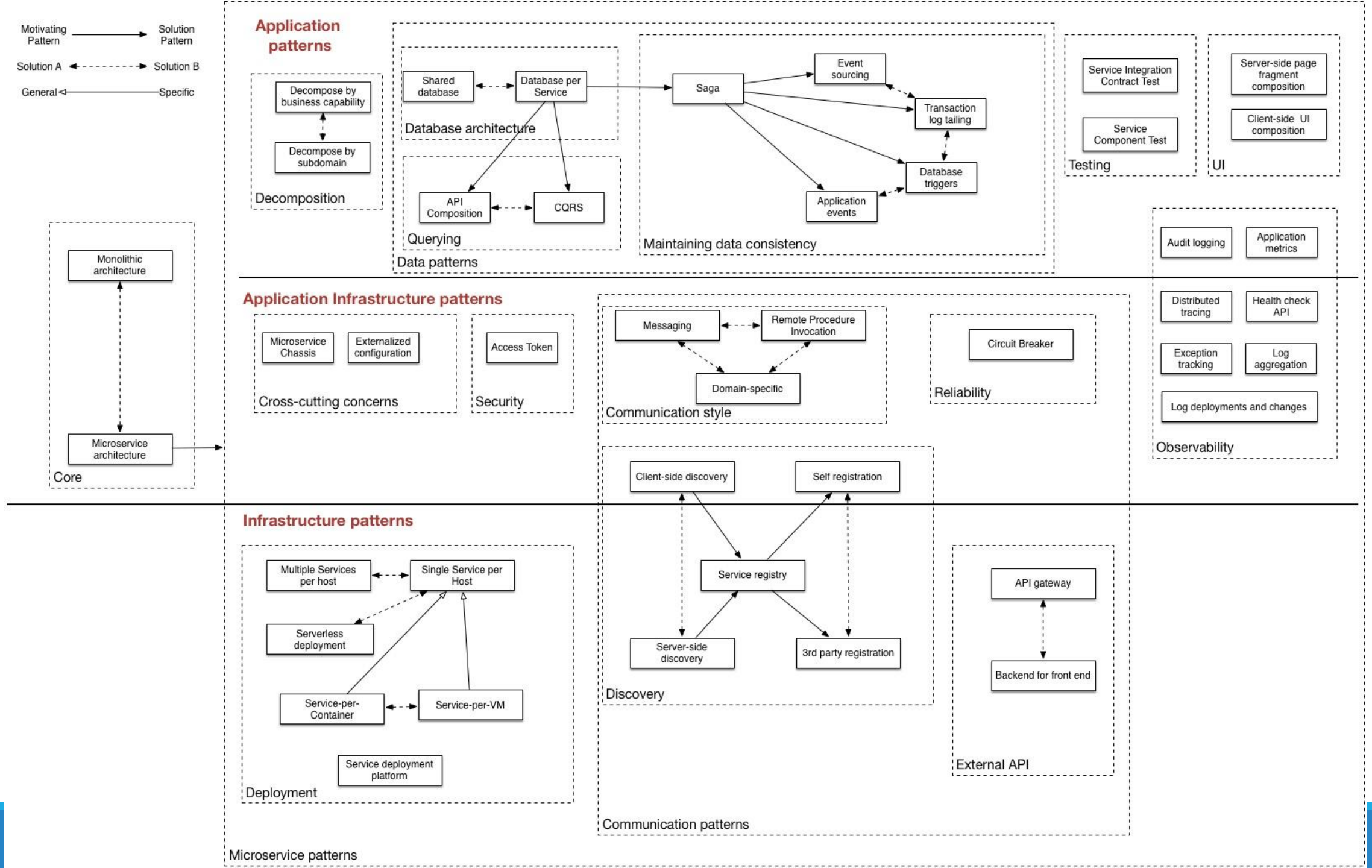
Training on

Micro services Spring boot and Docker

MOHAMMED LUQMAN SHAREEF

Microservices Patterns


Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Application patterns
Decompose by business capability
Decompose by subdomain
Decomposition
Shared database
Database per Service
Database architecture
API Composition
CQRS
Querying
Data patterns
Saga
Event sourcing
Transaction log tailing
Database triggers
Application events
Maintaining data consistency
Service Integration Contract Test
Service Component Test
Testing
Server-side page fragment composition
Client-side UI composition
UI
Audit logging
Application metrics
Distributed tracing
Health check API
Exception tracking
Log aggregation
Log deployments and changes
Observability
Monolithic architecture
Microservice architecture
Core
Application Infrastructure patterns
Microservice Chassis
Externalized configuration
Cross-cutting concerns
Access Token
Security
Messaging
Remote Procedure Invocation
Domain-specific
Communication style
Circuit Breaker
Reliability
Infrastructure patterns
Multiple Services per host
Single Service per Host
Serverless deployment
Service-per-Container
Service-per-VM
Service deployment platform
Deployment
Client-side discovery
Self registration
Service registry
Server-side discovery
3rd party registration
Discovery
API gateway
Backend for front end
External API
Communication patterns
Microservice patterns

Microservices Patterns

Decomposition Patterns

Communication Patterns

Data Management Patterns

Observability Patterns

Resilience Patterns

Advanced Patterns

Deployment Patterns

Decomposition Patterns

How to break down a monolith into microservices, and the patterns used in this process.

Decompose by Business Capability

- Split the system based on distinct business capabilities or domains.
- Example: An e-commerce application might have services such as Order Service, Payment Service, Inventory Service, Shipping Service, etc. Each service represents a specific business function.

Decompose by Subdomain (Bounded Context)

- Based on Domain-Driven Design (DDD), services are built around specific subdomains (bounded contexts).
- Example: In a financial services application, there could be a "Customer" subdomain and a "Loan" subdomain, each with separate services that do not overlap.

Strangler Pattern

- Gradually replace parts of a monolithic application by building new microservices and routing requests to the new services.
- Example: In an existing CRM monolith, you might start by breaking off the "Customer Notification" feature as a separate service, and gradually refactor other features.

Communication Patterns

How microservices communicate with each other.

Synchronous Communication (REST, gRPC)

- Microservices communicate using synchronous APIs (HTTP, REST, or gRPC). This approach is simple but introduces tight coupling and potential performance bottlenecks.
- Example: A Product Service calls the Inventory Service to check stock availability via a REST API.

Asynchronous Communication (Message Broker, Event-Driven Architecture)

- Services communicate asynchronously using messaging queues or publish/subscribe systems (e.g., RabbitMQ, Apache Kafka).
- Example: A Payment Service publishes a "payment completed" event. The Order Service listens to this event and processes the order fulfillment.

Data Management Patterns

Patterns to manage data in distributed systems where services have their own databases.

Database per Service Pattern

- Each service has its own database, ensuring loose coupling.
- Example: The Order Service uses a SQL database, while the User Service might use NoSQL (MongoDB) for faster lookups.

Shared Database

- Multiple microservices share the same database. (Generally less favored in a true microservice architecture)
- Example: (With caution) A small e-commerce app might initially share a database between product catalog and inventory services.

Saga Pattern

- A pattern to manage distributed transactions across microservices, ensuring eventual consistency.
- Example: In an e-commerce scenario, an Order Service calls the Payment Service, then the Shipping Service. If any step fails, compensating actions are taken (e.g., cancel payment).

CQRS (Command Query Responsibility Segregation) Pattern

- Separate read and write operations using different models, improving performance and scalability.
- Example: A reporting service that reads data from a read-optimized database while another service writes to a different write-optimized data store.

Observability Patterns

How to ensure visibility, monitoring, and troubleshooting in a distributed system.

Log Aggregation / Centralized Logging

- Collect logs from different microservices into a centralized system for monitoring and debugging.
- Example: Use tools like ELK (Elasticsearch, Logstash, Kibana) stack to aggregate and search logs across services like Order, Payment, and Inventory services.

Distributed Tracing

- Track a request as it traverses multiple services to identify bottlenecks and failures.
- Example: Using Jaeger or Zipkin, trace an API request from the API Gateway through the User Service to the Order Service.

Health Check API

- Services expose a health check API for monitoring their availability.
- Example: Each service (e.g., Order Service) exposes an HTTP endpoint (/health) that returns the service's health status. If unhealthy, the orchestrator or load balancer can take appropriate actions.

Resilience Patterns

How to design microservices that can handle failures and remain operational.

Bulkhead Pattern

- Isolate services to limit the impact of a failure on the system.
- Example: The Order Service has its own isolated resources (e.g., database connection pool) separate from the Payment Service, preventing resource exhaustion if one service crashes.

Retry Pattern

- Automatically retry failed requests with exponential backoff to recover from transient failures.
- Example: If a Payment Service request fails due to a network glitch, the Order Service retries after a brief delay, improving chances of success without overloading the system.

Rate Limiting

- Limit the number of requests a service can handle in a specific time window, protecting the service from overload.
- Example: A Notification Service might limit SMS notifications to 100 per second to prevent overwhelming the downstream SMS gateway provider.

Advanced Patterns

Circuit Breaker Pattern

- Prevents cascading failures by stopping calls to a service that is not responding or is failing consistently.
- Example: If the Inventory Service is down, the Order Service trips the circuit breaker to avoid waiting for the service and returns a fallback response.

API Gateway Pattern

- A centralized entry point for all client requests, providing cross-cutting concerns like authentication, rate limiting, and logging.
- Example: The API Gateway authenticates requests before routing them to the respective services like Order, User, or Payment.

Service Discovery Pattern

- Enable services to dynamically discover and communicate with each other.
- Tools: Consul, Eureka, etcd.

Deployment Patterns

Multiple service instances per host

Service instance per host

Service instance per VM

Service instance per Container

Serverless deployment

Service deployment platform

Spring Cloud

Cloud Native Applications

Cloud-native is a way of approaching the development and deployment of applications in such a way that takes account of the characteristics and nature of the cloud - resulting in processes and workflows that fully take advantage of the platform.

Cloud-native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model.

Cloud-native applications are specifically designed to thrive in the cloud environment. They are built with cloud technologies like microservices, containers, and serverless functions to take full advantage of cloud scalability, flexibility, and resilience.

Characteristics of Cloud-Native Applications

Microservices:

- Broken down into small, independent services that can be developed, deployed, and scaled independently.

Containers:

- Packaged with their dependencies into containers, making them portable and easy to deploy across different cloud environments.

Dynamically orchestrated:

- Use orchestration tools (like Kubernetes) to automate deployment, scaling, and management of containers.

Resilient:

- Designed to handle failures gracefully and continue operating even if some components are down.

Observable:

- Provide insights into their performance and health through monitoring and logging.

Benefits of Cloud-Native Applications

Increased agility and speed:

- Faster development cycles and continuous delivery.

Improved scalability and efficiency:

- Scale resources up or down automatically based on demand.

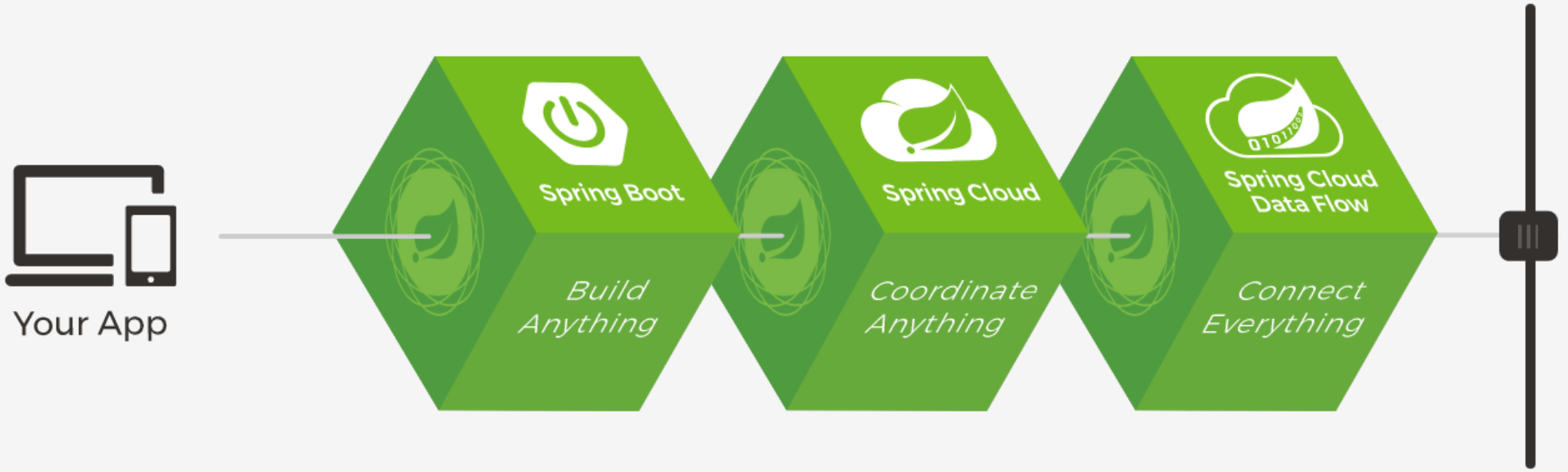
Enhanced resilience and fault tolerance:

- Minimize downtime and disruptions.

Reduced costs:

- Optimize resource utilization and pay only for what you use.

Cloud computing provides the infrastructure, and cloud-native applications are the modern way to build and run software in that environment, maximizing its potential.



Spring Boot

BUILD ANYTHING

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production ready applications.

Spring Cloud

COORDINATE ANYTHING

Built directly on Spring Boot's innovative approach to enterprise Java, Spring Cloud simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.

Spring Cloud Data Flow

CONNECT ANYTHING

Connect the Enterprise to the Internet of Anything-mobile devices, sensors, wearables, automobiles, and more. Spring Cloud Data Flow provides a unified service for creating composable data microservices that address streaming and ETL-based data processing patterns.

What is Spring Cloud?

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems. e.g.

- configuration management,
- service discovery,
- circuit breakers,
- intelligent routing,
- micro-proxy,
- global locks,
- leadership election,
- distributed sessions,
- cluster state

Spring Cloud

Spring Cloud provides solutions to cloud enable your microservices.

Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns.

It leverages and builds on top of some of the Cloud solutions open-sourced by Netflix for Service discovery, Configuration Management, Load balancing and Fault Tolerance and Tracing and Monitoring etc.

Spring cloud follow the spring boot model of providing useful defaults with ability to easily configure them up.

Spring Cloud takes a very declarative approach, and often you get a lot of features with just a classpath change and/or an annotation

Some of the Spring Cloud Projects

Spring Cloud Config
- Centralized external configuration management backed by a git repository.
- The configuration resources map directly to Spring Environment but could be used by non-Spring applications if desired.

Spring Cloud Netflix
- Integration with various Netflix OSS components (Eureka, Hystrix, Zuul, etc.).

Spring Cloud Bus
- An event bus for linking services and service instances together with distributed messaging.

Spring Cloud Cluster
- Leadership election and common stateful patterns with an abstraction and implementation for Zookeeper, Redis, Hazelcast, Consul.

Spring Cloud Security
- Provides support for load-balanced OAuth2 rest client and authentication header relays in a Zuul proxy.

Spring Cloud AWS
- Easy integration with hosted Amazon Web Services.

Netflix OSS

Netflix's Open Source Software (Netflix OSS) is a set of frameworks and libraries that Netflix wrote to solve some interesting distributed-systems problems at scale.

Netflix OSS is aimed at making the services provided by AWS more reliable;

Patterns for service discovery, load balancing, fault-tolerance, etc are incredibly important concepts for scalable distributed systems and Netflix brings nice solutions for these.

Netflix decided to contribute to the broader open-source community with these libraries and frameworks.

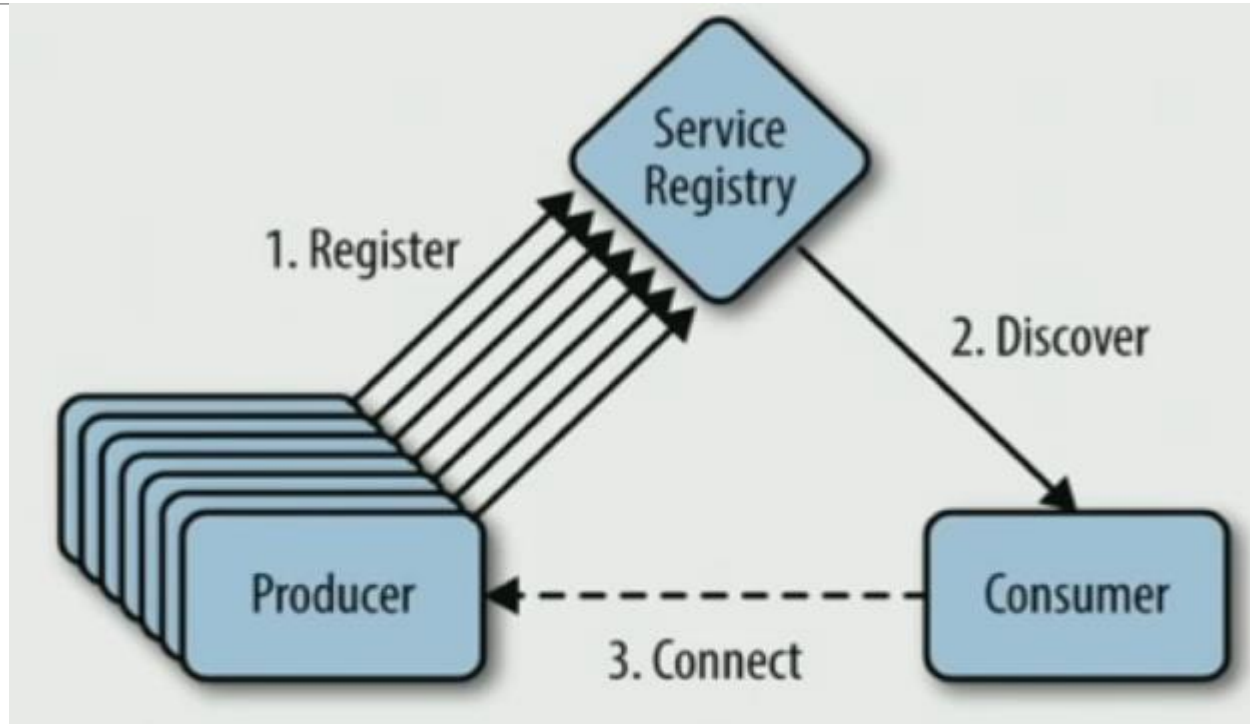
Service Discovery

Cloud Native Applications are dynamic in nature.

URLs may not remain fixed.

Service discovery allows micro services to easily discover the routes to the service it needs.

- Eureka (Netflix)
- Zookeeper
- Consulv



Service Discovery

A service registry is a phone book for your microservices.

Each service registers itself with the service registry and tells the registry where it lives (host, port, node name) and perhaps other service-specific metadata.

Service Discovery Server **Netflix Eureka** allows microservices to register themselves at runtime as they appear in the system landscape.

Service Discovery with Eureka

What is Netflix Eureka

Eureka is a REST based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

Eureka also comes with a Java-based client component, the Eureka Client, which makes interactions with the service much easier.

At Netflix, a much more sophisticated load balancer wraps Eureka to provide weighted load balancing based on several factors like traffic, resource usage, error conditions etc to provide superior resiliency.

Why Eureka?

In AWS cloud, because of its inherent nature, servers come and go.

Unlike the traditional load balancers which work with servers with well known IP addresses and host names, in AWS, load balancing requires much more sophistication in registering and de-registering servers with load balancer on the fly.

Since AWS does not yet provide a middle tier load balancer, Eureka fills a big gap in the area of mid-tier load balancing.

Hands on

Load Balancing

Why Client Side load balancing?

No more single point of failure

Avoid bottleneck

Auto Discovery

One less hop

Spring Cloud Config

Centralized Configuration

API Gateway

Routing and Filtering

A common challenge when building microservices is providing a unified interface to the consumers of your system.

The fact that your services are split into small composable apps shouldn’t be visible to users or result in substantial development effort.

To solve this problem, Netflix created and open-sourced its Zuul proxy server.

Zuul is an edge service that proxies requests to multiple backing services. It provides a unified “front door” to your system, which allows a browser, mobile app, or other user interface to consume services from multiple hosts .

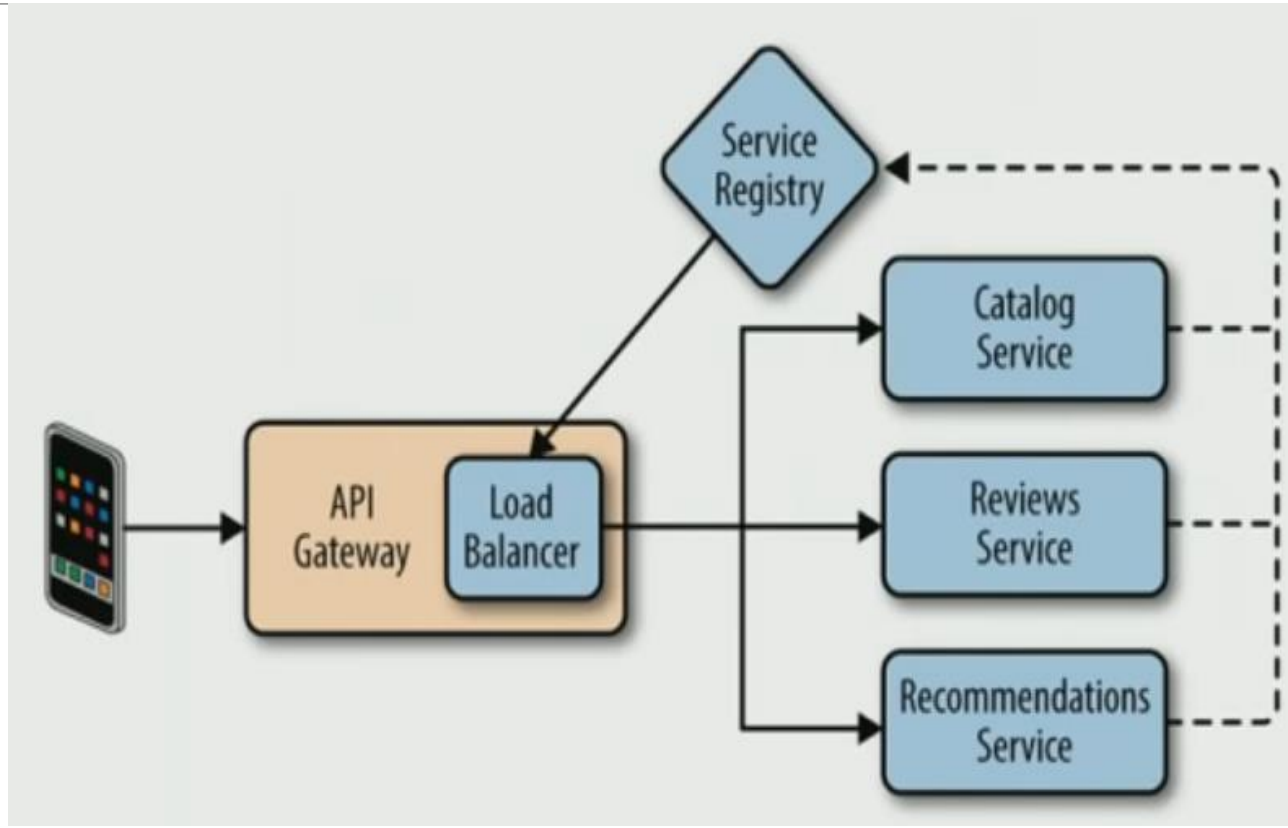
You can integrate Zuul with other Netflix projects like Hystrix for fault tolerance and Eureka for service discovery, or use it to manage routing rules, filters, and load balancing across your system.

API Gateway

API Gateway allows you to route API requests (both internal and external) to the correct service.

Zuul (Netflix)

Spring Cloud Gateway



API Gateway - Zuul

API Gateway, aka **Edge Service**, provides a unified interface for a set of microservices so that clients no need to know about all the details of microservices internals.

Pros:

- Provides easier interface to clients
- Can be used to prevent exposing the internal microservices structure to clients
- Allows to refactor microservices without forcing the clients to refactor consuming logic
- Can centralize cross-cutting concerns like security, monitoring, rate limiting etc

Cons:

- It could become a single point of failure if proper measures are not taken to make it highly available
- Knowledge of various microservice API may creep into API Gateway