Service Discovery, API Gateway & Configuration Management in Microservices

Author: Shreyansh Kumar

1. Introduction

In the world of modern application development, microservices architecture has emerged as a powerful approach to building scalable, resilient, and maintainable systems. Unlike monolithic applications where all functionality exists in a single codebase, microservices break down applications into smaller, independent services that communicate with each other over a network.

However, this distributed nature introduces new challenges. When you have dozens or even hundreds of services running across multiple servers or containers, how do they find each other? How do external clients access these services without needing to know their exact locations? And how do you manage configuration across all these services without having to redeploy them whenever a setting changes?

These are the fundamental challenges that Service Discovery, API Gateway, and Configuration Management solve in a microservices ecosystem. Think of these components as the infrastructure that enables your microservices to work together harmoniously:

Service Discovery allows services to find and communicate with each other without hardcoded locations.

API Gateway provides a single entry point for all client requests, routing them to the appropriate services.

Configuration Management centralizes configuration data, allowing you to change application behavior without rebuilding or redeploying services.

In this document, we'll explore each of these concepts in detail, with practical examples using Spring Cloud technologies, which provide a robust implementation of these patterns. Whether you're just starting with microservices or looking to enhance your existing architecture, understanding these three pillars will help you build more resilient and manageable distributed systems.

2. Service Discovery & Registration

What is Service Discovery and Why It's Needed

In a traditional monolithic application, components call each other using language-level methods or function calls. Everything runs in the same process, so there's no need to "discover" where other components are located. But in a microservices architecture, services run as separate processes, often on different machines across a network.

This introduces a critical challenge: How does Service A know where to find Service B when it needs to make a request? This is where Service Discovery comes in.

Service Discovery is a mechanism that maintains a registry of available services, their instances, and their locations (typically host and port). It allows services to find and communicate with each other without hardcoding network locations.

Why is this needed? Consider these scenarios:

Dynamic environments: In cloud environments, services can be scaled up or down automatically, with new instances being created or destroyed based on demand. IP addresses and ports change frequently.

Resilience: If a service instance fails, requests should automatically be routed to healthy instances.

Load balancing: Requests should be distributed across multiple instances of the same service to optimize performance.

Zero-downtime deployments: New versions of services can be deployed alongside old versions, with traffic gradually shifted from old to new.

Without Service Discovery, managing these scenarios would require manual configuration updates or complex networking rules. Service Discovery automates this process, making your microservices architecture more dynamic and resilient.

Concepts: Eureka Server and Eureka Client

Netflix Eureka, part of the Spring Cloud Netflix suite, is one of the most popular Service Discovery solutions in the Java ecosystem. It follows a client-server model:

Eureka Server: Acts as the service registry. It maintains a registry of all available service instances, their health status, and metadata.

Eureka Client: Any service that registers itself with the Eureka Server or wants to discover other services. Clients can be:

- Service providers that register themselves with the Eureka Server
- Service consumers that query the Eureka Server to find other services

The workflow is straightforward:

1. The Eureka Server starts up and waits for service registrations.
2. Service instances (Eureka Clients) start and register themselves with the Eureka Server, providing their host, port, and other metadata.
3. The Eureka Server maintains a registry of all registered services.
4. Service instances send heartbeats to the Eureka Server to indicate they're still alive.
5. If a service fails to send heartbeats, the Eureka Server removes it from the registry after a timeout.
6. Service consumers query the Eureka Server to find the locations of services they want to call.
7. Service consumers can then make requests directly to the service instances.

Example: Registering User Service and Order Service with Eureka

Let's walk through a concrete example of how two services—a User Service and an Order Service—would use Eureka for service discovery.

First, we need to set up a Eureka Server. In a Spring Boot application, this is remarkably simple:

```
// Eureka Server application
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }
}
```

In the `application.properties` or `application.yml` file:

```
server:
  port: 8761

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

This configuration tells the Eureka Server not to register itself as a client or fetch the registry, as it is the registry itself.

Now, let's set up our User Service as a Eureka Client:

```
// User Service application
@SpringBootApplication
@EnableEurekaClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
```

```
    }
}
```

In the `application.properties` or `application.yml` file:

```yaml
spring:
  application:
    name: user-service

server:
  port: 8081

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Similarly, for the Order Service:

```java
// Order Service application
@SpringBootApplication
@EnableEurekaClient
public class OrderServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }
}
```

In the `application.properties` or `application.yml` file:

```yaml
spring:
  application:
    name: order-service

server:
  port: 8082

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Now, when the User Service needs to call the Order Service, it can do so without knowing the exact location:

```java
@Service
public class UserOrderService {

    @Autowired
    private RestTemplate restTemplate;

    public OrderDto getOrderForUser(String userId) {
        // The service name "order-service" is used instead of a hardcoded URL
```

```
        return restTemplate.getForObject("http://order-service/orders/user/" +
        userId, OrderDto.class);
    }
}
```

To make this work, we need to configure the `RestTemplate` to use Eureka for service discovery:

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

The `@LoadBalanced` annotation tells Spring to use Netflix Ribbon (a client-side load balancer) in conjunction with Eureka to resolve the service name to an actual host and port, and to balance requests across multiple instances if available.

Tools: Spring Cloud Netflix Eureka

Spring Cloud Netflix Eureka is a battle-tested service discovery solution that integrates seamlessly with Spring Boot applications. It provides:

High availability: Eureka servers can be clustered for redundancy.

Self-preservation mode: If a network partition occurs, Eureka servers will stop expiring instances to prevent mass de-registrations.

Health checks: Services can implement health endpoints that Eureka uses to determine if they're healthy.

Metadata: Services can register with metadata that other services can use to make routing decisions.

Zone awareness: Eureka can be configured to prefer services in the same zone, reducing latency and network costs.

While Eureka is popular in the Spring ecosystem, other service discovery options include:

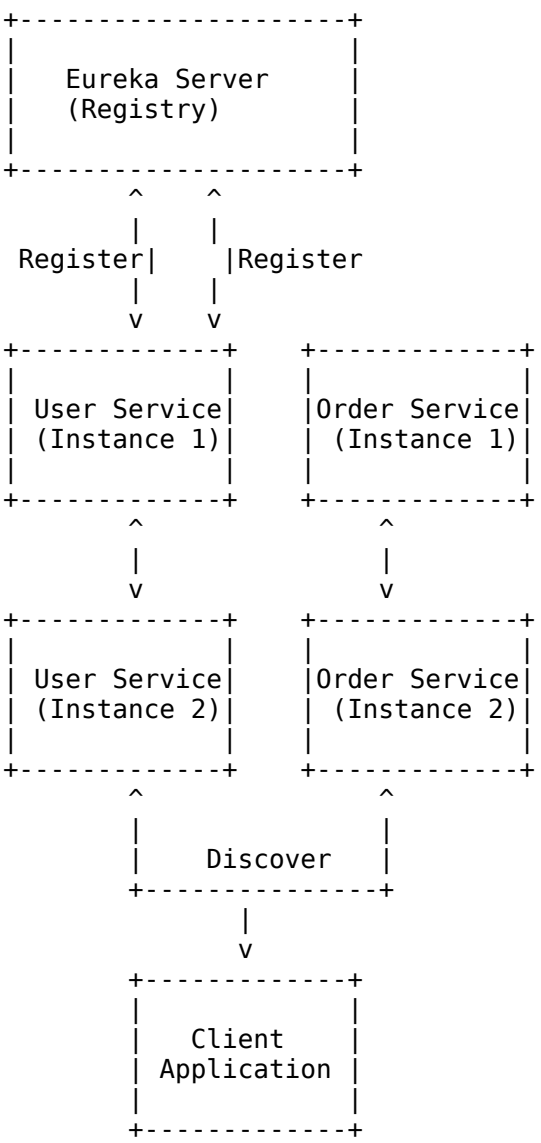
Consul: A service mesh solution from HashiCorp that provides service discovery, configuration, and segmentation.

etcd: A distributed key-value store that can be used for service discovery.

ZooKeeper: A centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services.

Kubernetes Service Discovery: If you're running on Kubernetes, you can use its built-in service discovery mechanisms.

Diagram: Eureka Architecture



In this diagram:

1. Multiple instances of User Service and Order Service register themselves with the Eureka Server.
2. Each service sends heartbeats to the Eureka Server to indicate it's still alive.
3. When the User Service needs to call the Order Service, it queries the Eureka Server to discover available Order Service instances.
4. The User Service can then make requests directly to the Order Service instances, typically with client-side load balancing.
5. Client applications can also discover and communicate with services through the registry.

Service Discovery is the foundation of a dynamic microservices architecture. It enables services to find each other without hardcoded locations, making your system more resilient to changes in the environment. With Eureka, implementing Service Discovery in Spring-based microservices becomes straightforward, allowing you to focus on building the business functionality of your services.

3. API Gateway

What is an API Gateway and Its Purpose

An API Gateway serves as the single entry point for all client requests to a microservices-based application. It sits between clients and services, routing requests to the appropriate microservice, aggregating responses, and handling cross-cutting concerns.

Think of an API Gateway as the receptionist at a large office building. When visitors (client requests) arrive, the receptionist (API Gateway) directs them to the right department (microservice), handles common tasks like checking visitor IDs (authentication), and ensures the building isn't overcrowded (rate limiting).

The primary purposes of an API Gateway include:

Routing: Directing incoming requests to the appropriate microservice based on the request path, method, headers, or other criteria.

Request aggregation: Combining results from multiple microservices into a single response, reducing the number of round trips between the client and the backend.

Protocol translation: Converting between different protocols (e.g., HTTP to gRPC) or API styles (e.g., REST to GraphQL).

Encapsulation of internal structure: Hiding the complexity of the microservices architecture from clients, allowing services to be reorganized without affecting clients.

Cross-cutting concerns: Handling aspects that apply to many or all services, such as authentication, authorization, rate limiting, caching, and monitoring.

In a microservices architecture without an API Gateway, clients would need to know the locations of all services they interact with, handle different protocols or API styles, and implement cross-cutting concerns themselves. This would make the client code more complex and tightly coupled to the backend structure.

With an API Gateway, clients have a single point of contact, simplifying their implementation and allowing the backend architecture to evolve independently.

Features: Routing, Filtering, Authentication, Rate Limiting

Modern API Gateways offer a rich set of features that make them essential components in a microservices architecture:

Routing: The core function of an API Gateway is to route requests to the appropriate service. This can be based on:

- URL path (e.g., `/users/*` goes to the User Service)
- HTTP method (e.g., GET vs. POST)
- Headers (e.g., content type, accept language)
- Query parameters
- Request body content
- Combination of the above

Filtering: API Gateways can modify requests and responses as they pass through. Common filtering operations include:

- Adding or removing headers
- Transforming request or response bodies
- Logging request details for audit purposes
- Validating request parameters or payload
- Error handling and response normalization

Authentication and Authorization: API Gateways often handle security concerns:

- Verifying user identity (authentication) through tokens, certificates, or credentials
- Checking if users have permission to access specific resources (authorization)
- Integrating with identity providers like OAuth, OpenID Connect, or LDAP
- Generating and validating JWT tokens
- Implementing single sign-on (SSO) across services

Rate Limiting: Protecting backend services from being overwhelmed:

- Limiting the number of requests per client in a given time window
- Implementing different rate limits for different clients or endpoints
- Queuing or throttling excess requests
- Returning appropriate status codes (e.g., 429 Too Many Requests) when limits are exceeded

Circuit Breaking: Preventing cascading failures:

- Detecting when a service is failing or responding slowly
- Temporarily stopping requests to failing services
- Providing fallback responses when services are unavailable
- Gradually resuming traffic when services recover

Load Balancing: Distributing traffic across service instances:

- Round-robin distribution
- Weighted distribution based on instance capacity

- Sticky sessions for stateful interactions
- Health checking to avoid routing to unhealthy instances

Caching: Improving performance and reducing backend load:

- Storing responses for frequently requested resources
- Configurable time-to-live (TTL) for cached items
- Cache invalidation strategies
- Support for conditional requests (If-Modified-Since, ETag)

Analytics and Monitoring: Providing visibility into API usage:

- Request/response timing
- Error rates and types
- Traffic patterns and trends
- Service-level agreement (SLA) compliance

API Versioning: Managing changes to APIs over time:

- Supporting multiple API versions simultaneously
- Routing based on version information in URL, header, or content type
- Deprecation notices for older versions

These features make API Gateways powerful tools for managing the complexity of microservices architectures while providing a consistent, secure, and performant experience for clients.

Example: Using Spring Cloud Gateway as a Single Entry Point for Multiple Services

Spring Cloud Gateway is a modern API Gateway built on Spring WebFlux, providing a reactive, non-blocking approach to handling requests. Let's walk through an example of how to set up Spring Cloud Gateway to route requests to our User Service and Order Service.

First, we need to create a new Spring Boot application for our API Gateway:

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

In the `application.yml` file, we configure the routes:

```
spring:
  application:
```

```
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/users/**
          filters:
            - AddRequestHeader=X-Gateway-Source, api-gateway

        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/orders/**
          filters:
            - AddRequestHeader=X-Gateway-Source, api-gateway

server:
  port: 8080

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

In this configuration:

- We define two routes, one for the User Service and one for the Order Service.
- The `uri` property uses the `lb://` prefix to indicate that we want to use client-side load balancing with the service name registered in Eureka.
- The `predicates` section defines when a route should be taken. In this case, we're using path-based routing.
- The `filters` section allows us to modify the request or response. Here, we're adding a custom header to each request.

Now, when a client sends a request to `http://api-gateway:8080/users/123`, the API Gateway will:

1. Match the request against the defined routes and find that it matches the `user-service` route.
2. Add the `X-Gateway-Source` header to the request.
3. Use Eureka to find available instances of the `user-service`.
4. Forward the request to one of those instances, using client-side load balancing.

Let's add some more advanced features to our gateway:

```
spring:
  application:
    name: api-gateway
  cloud:
```

```yaml
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/users/**
          filters:
            - AddRequestHeader=X-Gateway-Source, api-gateway
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20

        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/orders/**
          filters:
            - AddRequestHeader=X-Gateway-Source, api-gateway
            - name: CircuitBreaker
              args:
                name: orderServiceCircuitBreaker
                fallbackUri: forward:/fallback/orders

      default-filters:
        - name: Retry
          args:
            retries: 3
            statuses: BAD_GATEWAY
        - name: RequestSize
          args:
            maxSize: 5MB
```

In this enhanced configuration:

- We've added rate limiting to the User Service route, allowing 10 requests per second with a burst capacity of 20.
- We've added a circuit breaker to the Order Service route, which will redirect to a fallback endpoint if the service is unavailable.
- We've defined default filters that apply to all routes, including retry logic for failed requests and a maximum request size.

To handle authentication, we can add a custom filter:

```java
@Component
public class AuthenticationFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
        chain) {
        ServerHttpRequest request = exchange.getRequest();
```

```
        // Check for authentication token
        List<String> authHeader = request.getHeaders().get("Authorization");

        if (authHeader == null || authHeader.isEmpty() ||
        !isValidToken(authHeader.get(0))) {
            // Return unauthorized response
            ServerHttpResponse response = exchange.getResponse();
            response.setStatusCode(HttpStatus.UNAUTHORIZED);
            return response.setComplete();
        }

        // Continue the filter chain
        return chain.filter(exchange);
    }

    private boolean isValidToken(String token) {
        // Token validation logic here
        return token.startsWith("Bearer ") && token.length() > 10;
    }
}
```

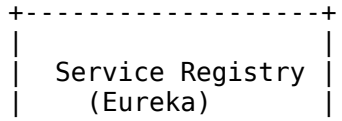
This filter will check for the presence and validity of an Authorization header on every request, returning a 401 Unauthorized response if the token is missing or invalid.

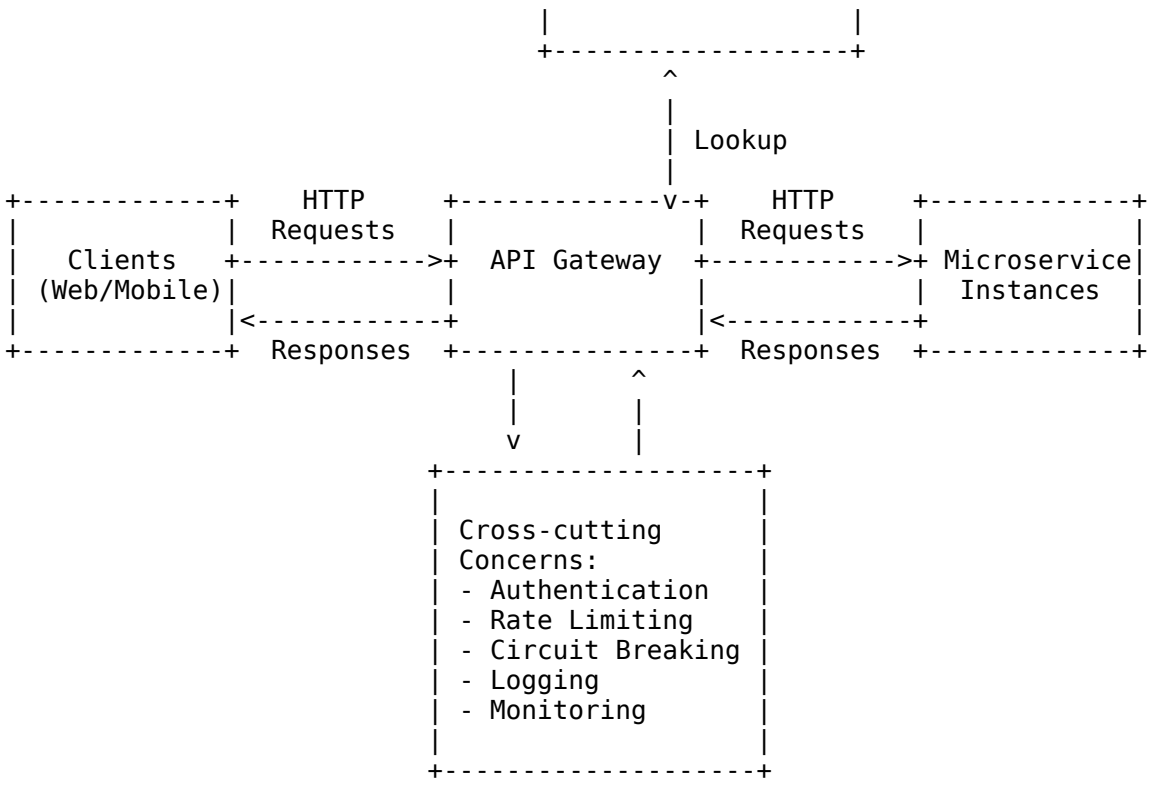
For the fallback endpoint mentioned in the circuit breaker configuration, we can create a controller:

```
@RestController
@RequestMapping("/fallback")
public class FallbackController {

    @GetMapping("/orders")
    public ResponseEntity<Map<String, String>> orderServiceFallback() {
        Map<String, String> response = new HashMap<>();
        response.put("message", "Order Service is currently unavailable. Please
        try again later.");
        response.put("timestamp", new Date().toString());

        return
        ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE).body(response);
    }
}
```

This controller provides a graceful response when the Order Service is unavailable, rather than letting the request fail with an error.

Diagram: API Gateway Handling Requests





In this diagram:

1. Clients (web browsers, mobile apps, or other services) send HTTP requests to the API Gateway.
2. The API Gateway handles cross-cutting concerns like authentication, rate limiting, and circuit breaking.
3. The API Gateway looks up service locations in the Service Registry (Eureka).
4. The API Gateway routes requests to the appropriate microservice instances.
5. Responses flow back through the API Gateway to the clients.

This centralized approach simplifies client interactions with the microservices ecosystem while providing consistent handling of cross-cutting concerns.

Pros and Cons of API Gateway

Like any architectural pattern, API Gateways come with advantages and trade-offs:

Pros:

Simplified client code: Clients interact with a single endpoint, reducing complexity in client applications.

Encapsulation of internal structure: The internal organization of microservices can change without affecting clients.

Centralized cross-cutting concerns: Authentication, monitoring, and other cross-cutting concerns can be implemented once at the gateway level.

Protocol translation: The gateway can handle different protocols internally and externally, allowing services to use the most appropriate protocol for their needs.

Reduced round trips: The gateway can aggregate responses from multiple services, reducing the number of network round trips for clients.

Improved security: With a single entry point, security policies can be enforced consistently.

Better monitoring and analytics: A gateway provides a central point for collecting metrics and logs about all API traffic.

Cons:

Single point of failure: If not properly designed for high availability, the gateway can become a single point of failure.

Potential performance bottleneck: All traffic passes through the gateway, which can become a bottleneck if not scaled appropriately.

Increased complexity: Adding another component to the architecture increases overall system complexity.

Potential for gateway bloat: Over time, there's a risk of adding too much logic to the gateway, making it difficult to maintain.

Additional network hop: The gateway adds an extra network hop to each request, which can increase latency.

Development and operational overhead: The gateway requires development, testing, deployment, and monitoring resources.

To mitigate these drawbacks, consider these best practices:

Deploy multiple gateway instances: Use load balancing across multiple gateway instances to avoid a single point of failure.

Keep the gateway focused: Resist the temptation to add business logic to the gateway; keep it focused on routing, filtering, and cross-cutting concerns.

Monitor gateway performance: Set up comprehensive monitoring to detect and address performance issues early.

Use caching judiciously: Implement caching at the gateway level for frequently accessed, relatively static data.

Consider a microgateway architecture: For very large systems, consider using multiple specialized gateways for different parts of the application.

Automate gateway configuration: Use infrastructure as code and CI/CD pipelines to manage gateway configuration, reducing operational overhead.

An API Gateway is a powerful tool in a microservices architecture, but like any tool, it should be used thoughtfully. When implemented well, it can significantly simplify client interactions with your microservices ecosystem while providing consistent handling of cross-cutting concerns.

4. Configuration Management

Importance of Centralized Configuration

In a microservices architecture, configuration management becomes significantly more complex than in monolithic applications. With potentially hundreds of services, each potentially having multiple instances across different environments (development, testing, staging, production), managing configuration becomes a critical challenge.

Centralized Configuration Management provides a solution to this challenge by externalizing configuration from application code and providing a central place to manage it. This approach offers several important benefits:

Environment-specific configuration: Different environments (development, testing, staging, production) often require different configuration values. Centralized configuration makes it easy to manage these variations.

Runtime configuration changes: With centralized configuration, you can change application behavior without rebuilding or redeploying services. Changes can be applied dynamically while the application is running.

Configuration consistency: When multiple instances of the same service are running, centralized configuration ensures they all use the same configuration values.

Configuration versioning: Configuration changes can be versioned, allowing you to track changes over time and roll back if necessary.

Sensitive information protection: Centralized configuration can integrate with secret management systems to handle sensitive information like passwords and API keys securely.

Reduced configuration duplication: Without centralized configuration, the same configuration values might be duplicated across multiple services. Centralization reduces this duplication and the associated maintenance burden.

Simplified operations: Operations teams can manage configuration changes without requiring developer intervention or application redeployments.

Consider a real-world analogy: Think of centralized configuration as the control room of a large factory. From this central location, operators can adjust settings for all machines on the factory floor without having to physically visit each machine. If a setting needs to be changed, it can be done once from the control room, ensuring consistency across all machines and minimizing downtime.

Concepts: Spring Cloud Config Server and Config Clients

Spring Cloud Config provides a powerful implementation of centralized configuration management for Spring-based microservices. It follows a client-server model:

Config Server: A dedicated service that serves configuration properties to client applications. It can pull configuration from various backend sources, most commonly Git repositories.

Config Clients: Microservices that retrieve their configuration from the Config Server during startup and, optionally, during runtime.

The basic workflow is as follows:

1. The Config Server starts up and connects to its backend configuration store (e.g., a Git repository).
2. Config Clients start up and make requests to the Config Server for their configuration.
3. The Config Server serves the appropriate configuration based on the client's application name, profile, and label (e.g., Git branch).
4. Config Clients apply the configuration and start their normal operation.
5. When configuration changes, the Config Server can notify clients to refresh their configuration.

Spring Cloud Config supports several important concepts:

Application name: Each microservice has a unique name (e.g., `user-service`, `order-service`). The Config Server uses this name to determine which configuration files to serve.

Profiles: Different environments or deployment scenarios are represented as profiles (e.g., `dev`, `test`, `prod`). Profiles allow for environment-specific configuration.

Labels: In Git-backed configurations, labels typically correspond to branches or tags, allowing for version-specific configuration.

Property sources: Configuration can come from multiple sources, which are prioritized. For example, profile-specific properties override default properties.

Encryption and decryption: Sensitive properties can be encrypted in the configuration store and decrypted by the Config Server before being served to clients.

Actuator endpoints: Spring Boot Actuator provides endpoints for refreshing configuration, viewing current property sources, and more.

Example: Storing Application Settings in a Git Repo and Serving Them via Config Server

Let's walk through a concrete example of setting up Spring Cloud Config Server and Clients.

First, we'll create a Git repository to store our configuration. The repository structure might look like this:

```
/
├── application.yml           # Common configuration for all services
├── user-service.yml          # Default configuration for user-service
├── user-service-dev.yml      # Development configuration for user-service
├── user-service-prod.yml     # Production configuration for user-service
├── order-service.yml         # Default configuration for order-service
├── order-service-dev.yml     # Development configuration for order-service
└── order-service-prod.yml    # Production configuration for order-service
```

The `application.yml` file contains configuration that applies to all services:

```yaml
# application.yml
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

management:
  endpoints:
    web:
      exposure:
        include: health,info,refresh

logging:
  level:
    org.springframework: INFO
```

The service-specific files contain configuration for individual services:

```yaml
# user-service.yml
server:
  port: 8081

spring:
  datasource:
    url: jdbc:h2:mem:userdb
    driver-class-name: org.h2.Driver

user-service:
  greeting: "Welcome to the User Service"
```

```
  feature-flags:
    enable-notifications: false

# user-service-dev.yml
spring:
  h2:
    console:
      enabled: true

user-service:
  feature-flags:
    enable-notifications: true

# user-service-prod.yml
spring:
  datasource:
    url: jdbc:mysql://prod-db-server:3306/userdb
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}

  h2:
    console:
      enabled: false

user-service:
  greeting: "Welcome to the User Service - Production Environment"
```

Now, let's set up the Config Server. We create a new Spring Boot application:

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

In the `application.yml` file for the Config Server:

```
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-username/your-config-repo
          default-label: main
          search-paths: /*
```

```
          clone-on-start: true

# For local development, you can use a filesystem backend instead:
# spring:
#   cloud:
#     config:
#       server:
#         native:
#           search-locations: file:///path/to/local/config-repo
```

Next, we configure our User Service to be a Config Client:

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

In the `bootstrap.yml` file (which is loaded before `application.yml`):

```
spring:
  application:
    name: user-service
  profiles:
    active: dev
  cloud:
    config:
      uri: http://localhost:8888
      fail-fast: true
      retry:
        initial-interval: 1000
        max-interval: 2000
        max-attempts: 6
```

Now, we can inject and use the configuration in our service:

```
@RestController
@RefreshScope  // This annotation allows this bean to be refreshed when
        configuration changes
public class UserController {

    @Value("${user-service.greeting}")
    private String greeting;

    @Value("${user-service.feature-flags.enable-notifications}")
    private boolean notificationsEnabled;

    @GetMapping("/greeting")
    public String getGreeting() {
        return greeting;
```

```
    }

    @GetMapping("/features")
    public Map<String, Boolean> getFeatureFlags() {
        Map<String, Boolean> features = new HashMap<>();
        features.put("notifications", notificationsEnabled);
        return features;
    }
}
```

The `@RefreshScope` annotation is particularly powerful. When the configuration changes, you can trigger a refresh without restarting the application by sending a POST request to the `/actuator/refresh` endpoint:

```
curl -X POST http://localhost:8081/actuator/refresh
```

This will cause Spring to reload the configuration and recreate beans annotated with `@RefreshScope`, applying the new configuration values.

For more complex configuration, you can use `@ConfigurationProperties`:

```
@Component
@ConfigurationProperties(prefix = "user-service")
@RefreshScope
public class UserServiceProperties {

    private String greeting;
    private FeatureFlags featureFlags = new FeatureFlags();

    // Getters and setters

    public static class FeatureFlags {
        private boolean enableNotifications;

        // Getters and setters
    }
}
```

Then inject and use this class:

```
@RestController
public class UserController {

    private final UserServiceProperties properties;

    public UserController(UserServiceProperties properties) {
        this.properties = properties;
    }

    @GetMapping("/greeting")
    public String getGreeting() {
        return properties.getGreeting();
```

```
    }

    @GetMapping("/features")
    public Map<String, Boolean> getFeatureFlags() {
        Map<String, Boolean> features = new HashMap<>();
        features.put("notifications",
        properties.getFeatureFlags().isEnableNotifications());
        return features;
    }
}
```

Tools: Spring Cloud Config

Spring Cloud Config is a robust solution