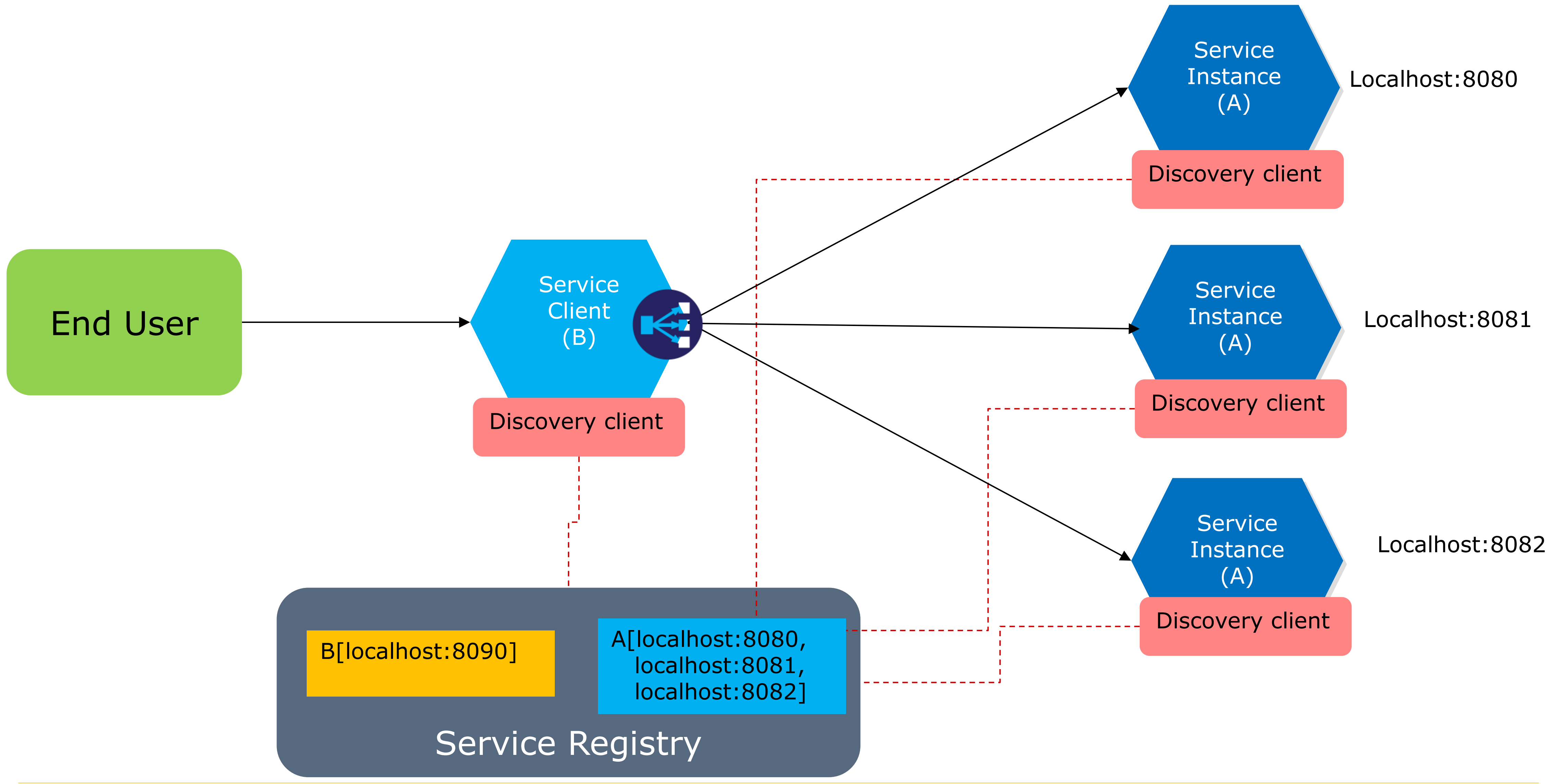
# Service Discovery and Load Balancing



# Client Side Load Balancing

Spring Cloud Loadbalancer

## Introduction to Client-Side Load Balancing

- **Definition:**

  Client-side load balancing distributes network or application traffic across multiple servers.

- **Use case:**

  Ensures even distribution of requests among available services in a microservice architecture.

- **Benefits:**

  Better resource utilization, fault tolerance, and improved scalability.

## Client-Side Load Balancing vs. Server-Side Load Balancing

- **Server-Side Load Balancing**:
  - Managed by the server; uses external load balancers like NGINX or HAProxy.
- **Client-Side Load Balancing**:
  - Each client is aware of the available instances and chooses which one to call.
- Example of frameworks:
  - Ribbon (deprecated).
  - Spring Cloud LoadBalancer.

## Spring Cloud LoadBalancer Overview

- Replaced Netflix Ribbon in Spring Cloud.
- Integrated with Spring Boot and Spring Cloud microservices.
- Automatically distributes requests among instances of a service.

## How Spring Cloud LoadBalancer Works

- The client retrieves a list of service instances from a **Service Discovery** mechanism (like Eureka).
- Balancing logic runs on the client to decide which instance to invoke.
- LoadBalancerClient manages the request distribution.

## Configuration of Spring Cloud LoadBalancer

- Add Spring Cloud LoadBalancer dependency:
- Ensure Service Discovery integration (Eureka, Consul, etc.).
- Enable client-side load balancing by annotating the RestTemplate or WebClient.

## Client-Side Load Balancing in Spring Boot

- Example using RestTemplate:

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

- Example using WebClient:

```
@Bean
@LoadBalanced
public WebClient.Builder webClientBuilder() {
   return WebClient.builder();
}
```

## Service Discovery Integration

- Common integrations:
  - **Eureka**: Popular for service registry and discovery.
  - **Consul**: A more robust alternative for service discovery.

- The LoadBalancer gets available instances from these registries.

## Load Balancing Strategies

- **Round Robin**: Distributes requests evenly in a circular order.
- **Random**: Selects an instance at random.
- **Weighted**: Requests are distributed based on pre-defined weights of instances.

## Customizing Load Balancing Behavior

- Custom rule implementation for specific needs:

```
@Bean
public ReactorLoadBalancer<ServiceInstance> customLoadBalancer(Environment environment) {

          String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);

           return new RandomLoadBalancer(name);
}
```

## Monitoring & Troubleshooting

- Use tools like Spring Boot Actuator to monitor service instances.
- Common issues:
  - Misconfiguration of Service Discovery.
  - Inconsistent request distribution.
  - Health-check failures of instances.

## Best Practices

- Use retry mechanisms to handle failed requests.
- Combine client-side load balancing with circuit breakers (e.g., Resilience4j).
- Ensure health checks are configured for all service instances.

## Service Discovery

Eureka Server

## The Need for Service Discovery

- Problem: As microservices scale, services need to find and communicate with each other dynamically.
- Manual service registration vs. automated service discovery
- Why hard-coding service locations doesn’t scale

## What is Service Discovery?

- Definition: A mechanism where services register themselves with a central registry and clients discover the service dynamically.
- Importance in Microservices architecture
- Types of Service Discovery: Client-Side and Server-Side

## Client-Side Service Discovery

- Definition and example
- How clients (microservices) look up services directly from the registry
- Diagram: Service Client interacting directly with Service Registry
- Tools: Netflix Eureka, Consul, Zookeeper

## Server-Side Service Discovery

- Definition and example
- Service registry is used by a load balancer or gateway
- Diagram: Clients interacting with a load balancer that queries the registry
- Tools: Kubernetes (Kube-DNS), AWS ELB, Nginx with Consul

## Introducing Netflix Eureka

- What is Netflix Eureka?
- Role in the Spring ecosystem
- Features: Service Registration, Heartbeats, Service Discovery

## Setting up Eureka Server in Spring Boot

- Add dependencies: spring-cloud-starter-netflix-eureka-server
- Configurations for Eureka server:

```yaml
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

## Setting up Eureka Client in Spring Boot

- Add dependencies: spring-cloud-starter-netflix-eureka-client
- Service registration configuration:

```yaml
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

- Registering multiple services in Eureka

## Service Discovery in Action

- How services register themselves on the Eureka Server
- How clients discover other services via Eureka
- Live demo: Register two microservices and use Eureka to discover them

## Fault Tolerance in Service Discovery

- Circuit breaker pattern with Spring Cloud and Resilience4j
- Handling service unavailability
- Fall-back mechanisms

## Server-Side Discovery with Spring Cloud Gateway

- Integrating Eureka with Spring Cloud Gateway
- Dynamic routing based on service registry
- Config example for using Spring Cloud Gateway with Eureka

## Scaling and Performance Considerations

- Horizontal scaling of Eureka servers
- High availability with multiple Eureka instances
- Performance tuning in large-scale applications

## Alternatives to Eureka

- Consul
- Zookeeper
- Kubernetes Service Discovery

## Conclusion

- Recap of Service Discovery benefits
- Overview of Eureka in Spring Boot
- Final thoughts on best practices

# Circuit Breaker Pattern

Resilience4j in Spring Boot

Enhancing Microservice Resilience

V-1.0

## Agenda

- ☐ What is the Circuit Breaker Pattern?
- ☐ Resilience in Microservices Architecture
- ☐ Introduction to Resilience4j
- ☐ Resilience4j Circuit Breaker with Spring Boot
- ☐ Demonstration: Implementing Circuit Breaker
- ☐ Key Takeaways

## What is the Circuit Breaker Pattern?

- **Definition**: A design pattern that helps prevent cascading failures in distributed systems by temporarily stopping the execution of requests to an external service that's failing.
- **Phases**:
  - **Closed**: Requests flow normally.
  - **Open**: Requests are blocked for a time.
  - **Half-Open**: Limited requests are sent to check if the service has recovered.

## Why Use Circuit Breakers?

- Prevent service overloads.
- Improve application resilience.
- Enhance fault tolerance in distributed architectures.
- Handle failures gracefully.

## Resilience in Microservices

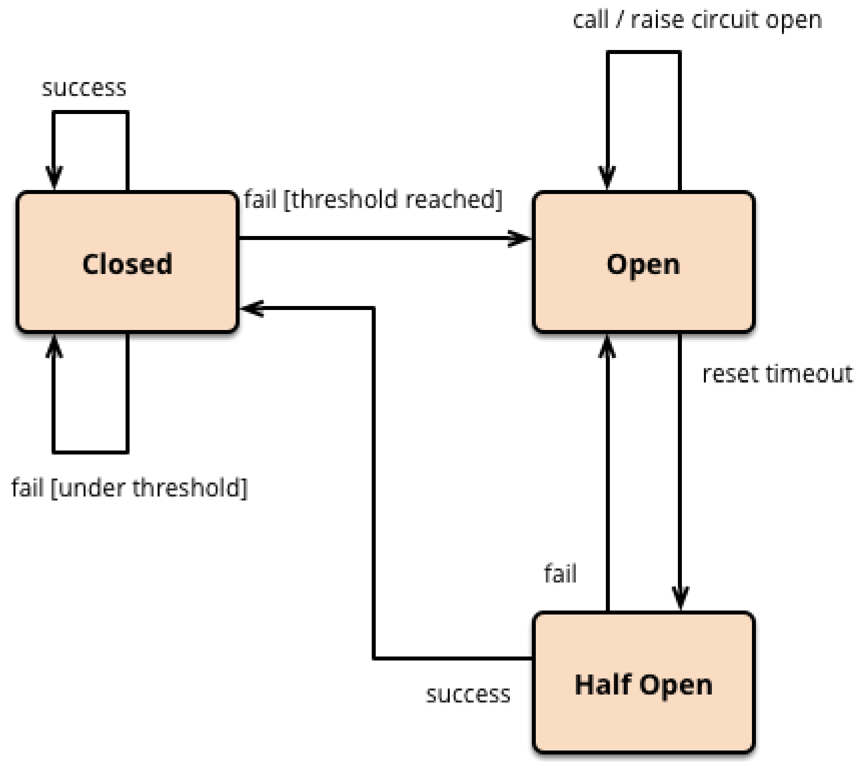
- Challenges in microservice communication:
  - **Latency issues**
  - **Service failures**
  - **Network partitions**
- Solutions: Circuit Breakers, Retries, Bulkheads, Rate Limiters

## Introduction to Resilience4j

- **Resilience4j**: A lightweight, easy-to-use fault tolerance library for Java 8+ that provides implementations for circuit breakers, rate limiters, retries, etc.
- Modular, compared to Netflix Hystrix (retired).
- Key modules:
  - CircuitBreaker
  - Retry
  - RateLimiter
  - Bulkhead
  - TimeLimiter

## How Circuit Breaker Works in Resilience4j

- Circuit Breaker transitions between:
  - **Closed**: All requests pass through.
  - **Open**: Requests are blocked for a time.
  - **Half-Open**: Test requests pass through to see if the service has recovered.
- Configurable parameters:
  - Failure rate threshold
  - Wait duration before opening
  - Sliding window size


call / raise circuit open
success
Closed
fail [threshold reached]
Open
reset timeout
fail [under threshold]
fail
success
Half Open

## Integrating Resilience4j with Spring Boot

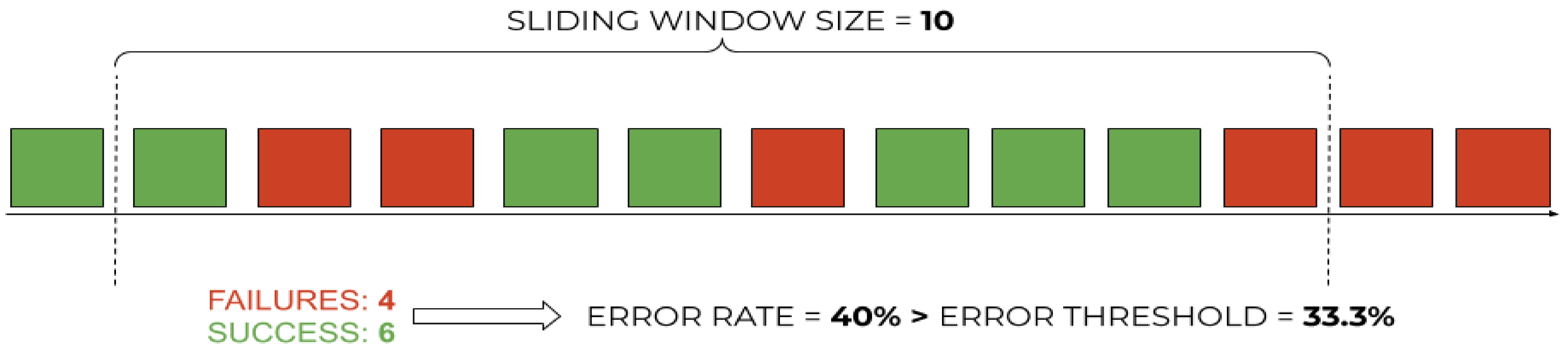
- Resilience4j provides Spring Boot Starter libraries.
- Configuration via application.properties or YAML files.
- Integration steps:
- Add the Resilience4j dependency.
- Configure circuit breaker properties.
- Annotate methods with @CircuitBreaker.

## Dependency Management

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.7.0</version>
</dependency>
```

## Properties

```yaml
resilience4j.circuitbreaker:
    instances:
        message-service:
          registerHealthIndicator: true
          slidingWindowSize: 10
          permittedNumberOfCallsInHalfOpenState: 3
          slidingWindowType: COUNT_BASED
          minimumNumberOfCalls: 5
          waitDurationInOpenState: 5s
          failureRateThreshold: 33.3
          automaticTransitionFromOpenToHalfOpenEnabled: true
```


SLIDING WINDOW SIZE = **10**
FAILURES: **4**
SUCCESS: **6**
ERROR RATE = **40%** **>** ERROR THRESHOLD = **33.3%**

## Code Example

```
@CircuitBreaker(name = "myService", fallbackMethod = "fallbackMethod")
public String myService() {
    // Call to an external service
}


public String fallbackMethod(Throwable t) {
    return "Service is currently unavailable.";
}
```

## Demo: Circuit Breaker Implementation in Spring Boot

- Short demo of how to create a circuit breaker-enabled service in Spring Boot.
- **Steps**:
  - Setup dependencies.
  - Create a service class.
  - Configure circuit breaker.
  - Test with an external service failure.

## Circuit Breaker Metrics and Monitoring

- Micrometer integration: Monitor circuit breaker states.
- Actuator: Use Spring Boot Actuator for health checks and exposing circuit breaker metrics.
- Example:

/actuator/metrics/resilience4j.circuitbreaker.state

## Fallback Strategies

- Fallback methods ensure the application doesn’t crash.
- Fallbacks can return default responses or cached data.
- Example:
  - Static response
  - Another service call

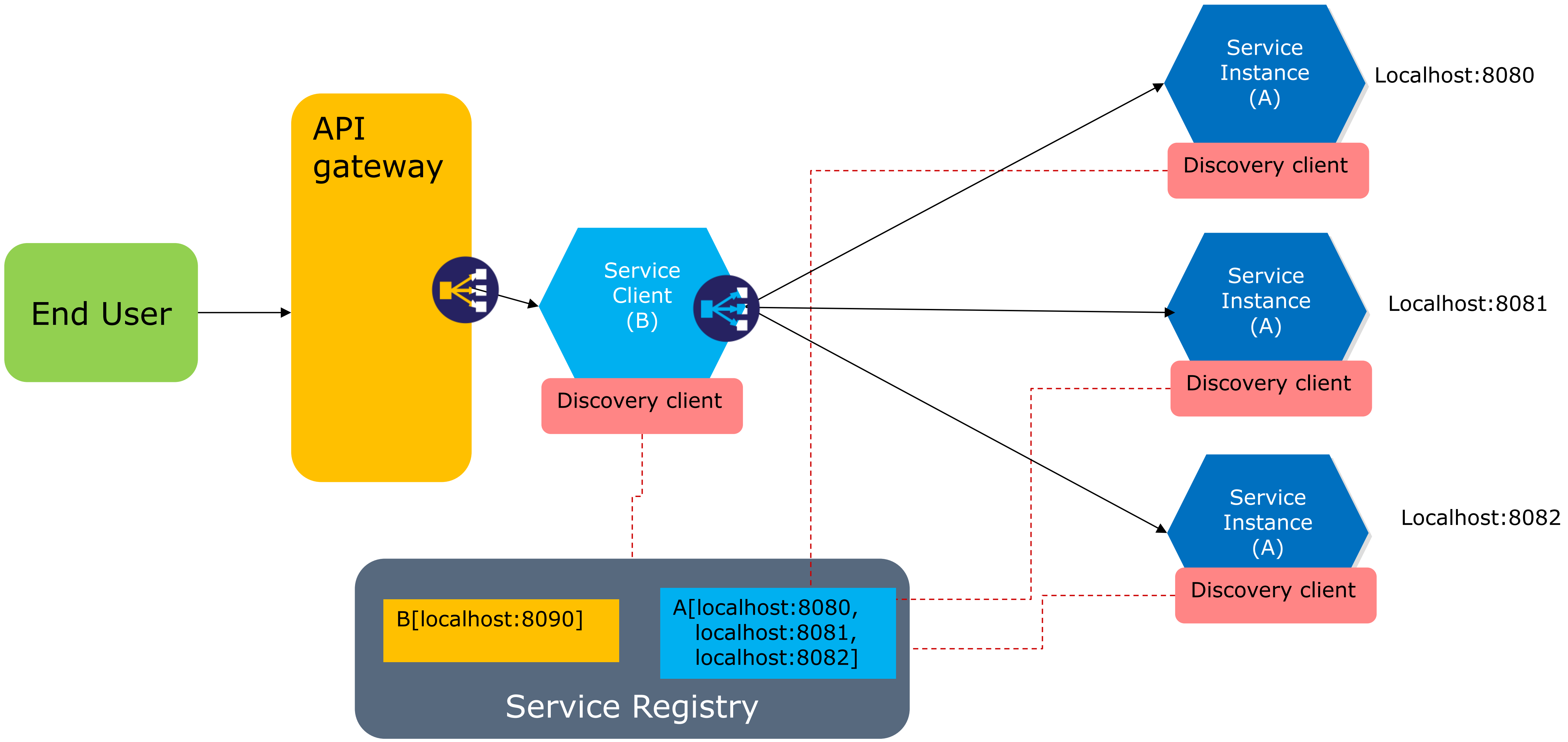
## Key Takeaways

- Circuit breaker pattern improves the resilience of microservices.
- Resilience4j offers a modular and efficient way to implement circuit breakers.
- Easy integration with Spring Boot via annotations and configuration.

- Questions?

## API Gateway

## Service Discovery and Load Balancing



## Introduction to API Gateway Pattern

- **What is an API Gateway?**
  - A server that acts as an entry point for all client requests.
  - It routes requests to the appropriate backend microservices.
- **Why use an API Gateway?**
  - Centralized access control
  - Cross-cutting concerns (logging, security, caching)
  - Reduces complexity for clients
  - Helps decouple front-end clients from multiple microservices

## Key Features of an API Gateway

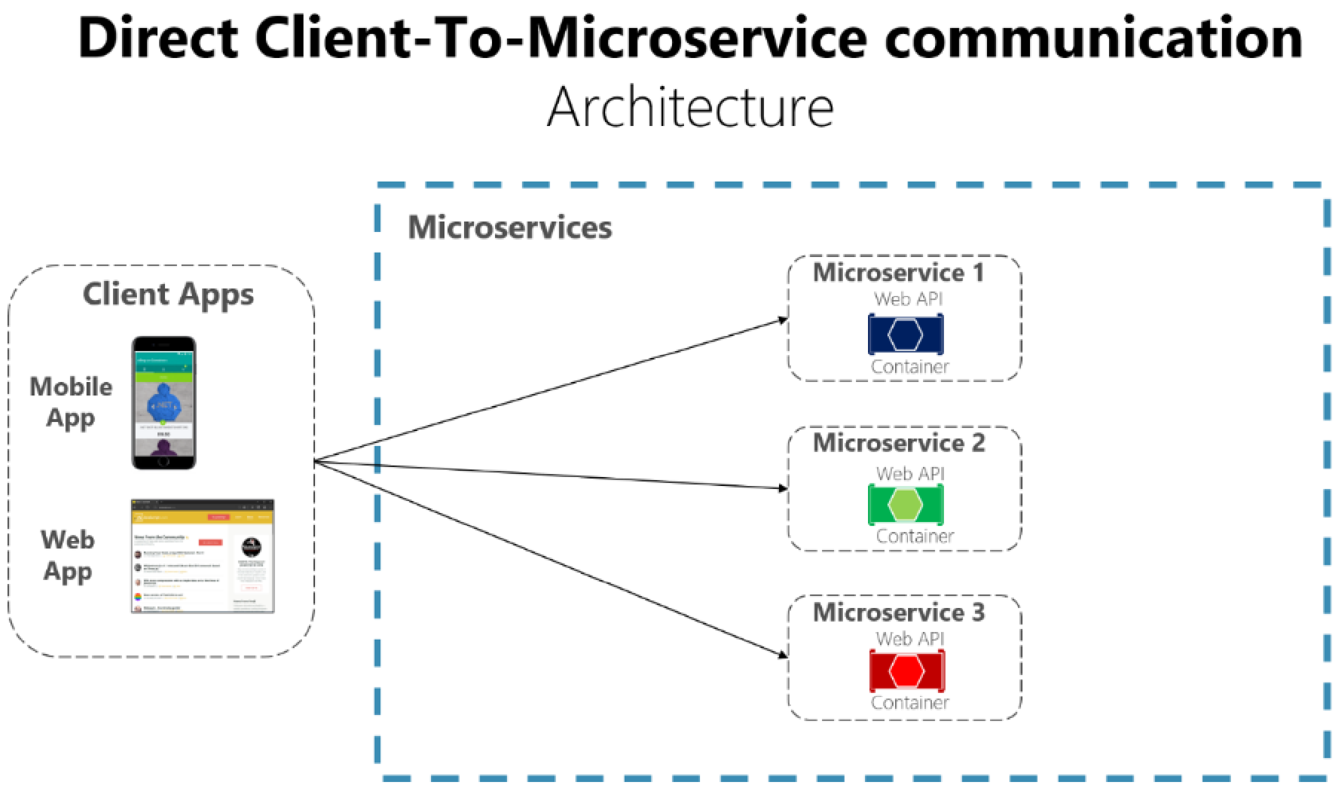
- **Request Routing**
  - Directs client requests to the right microservice
- **Load Balancing**
  - Distributes incoming traffic across microservices
- **Security**
  - Authentication, authorization, and SSL termination
- **Rate Limiting**
  - Controls the number of requests per client
- **Logging & Monitoring**
  - Tracks request flows, error logs, and metrics

## Spring Cloud Gateway Overview

- **What is Spring Cloud Gateway?**
  - A project built on top of Spring Boot for building API gateways.
  - Provides features like routing, filters, security, and more.

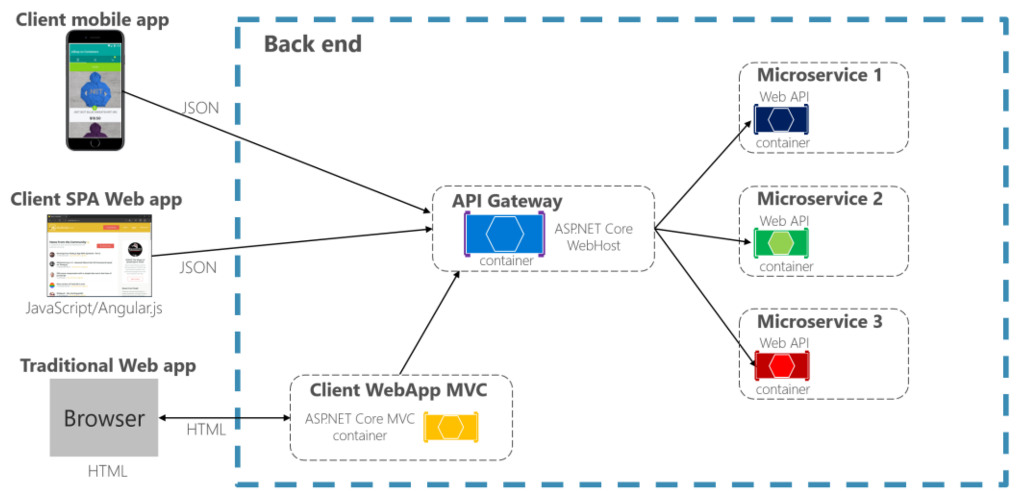
- **Why use Spring Cloud Gateway?**
  - Built on top of Project Reactor for reactive programming
  - Easy integration with Spring ecosystem

# Direct Access: No API Gateway



# Architecture Diagram

Using a single custom **API Gateway service**



Using multiple **API Gateways / BFF**

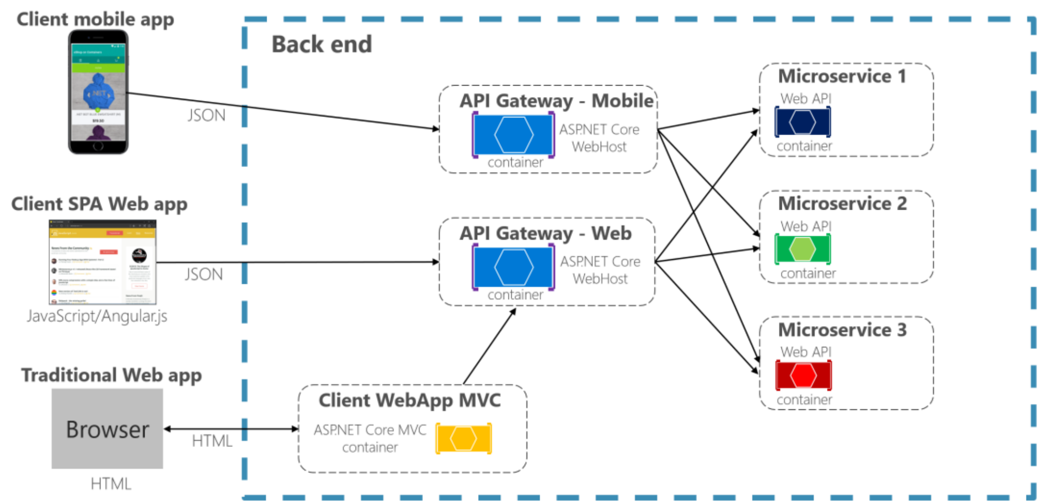


Image Courtesy: Microsoft

## API Gateway Flow in Spring Boot

1. **Client Request**
2. **API Gateway (Spring Cloud Gateway)**
   1. Request routed to appropriate microservice
3. **Microservice Communication**
   1. Response sent back through the gateway to the client

- Setting up Spring Cloud Gateway
- Dependencies in pom.xml:
- xml
- Copy code
- <dependency>
-     <groupId>org.springframework.cloud</groupId>
-     <artifactId>spring-cloud-starter-gateway</artifactId>
- </dependency>
- Main Class Setup:
- java
- Copy code
- @SpringBootApplication
- public class ApiGatewayApplication {
-     public static void main(String[] args) {
-         SpringApplication.run(ApiGatewayApplication.class, args);
-     }
- }

- Configuring Routes in API Gateway
- Example of Route Configuration in application.yml:
- yaml
- Copy code

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081
          predicates:
            - Path=/users/**
        - id: order-service
          uri: http://localhost:8082
          predicates:
            - Path=/orders/**
```

- Explanation:
- Define route IDs, URIs, and paths for microservices.

- Filters in API Gateway
- Global Filters:
- Applied to all routes.
- Examples: Security, logging.
- Route-Specific Filters:
- Applied to specific routes.
- Example of a Filter:
- yaml
- Copy code

```
- id: order-service
  uri: http://localhost:8082
  filters:
    - AddRequestHeader=Order,Service
```

- Load Balancing with API Gateway
- Enable Load Balancing:
- Use Spring Cloud's spring-cloud-starter-loadbalancer.
- Example Configuration:
- yaml
- Copy code
- spring:
-   cloud:
-     gateway:
-       discovery:
-         locator:
-           enabled: true

## Security in API Gateway

- **Authentication & Authorization:**
  - Use Spring Security with OAuth2 or JWT.
- **Example Configuration:**
  - Add JWT validation to requests before forwarding them to microservices.

- Rate Limiting in API Gateway
- Example Configuration for Rate Limiting:
- yaml
- Copy code

```
- id: rate_limiter
  uri: http://localhost:8080
  filters:
    - name: RequestRateLimiter
      args:
        redis-rate-limiter:
          replenishRate: 10
          burstCapacity: 20
```

## API Gateway Best Practices

- Handle timeouts and retries effectively.
- Ensure security policies are applied centrally.
- Monitor the gateway for performance bottlenecks.
- Implement caching for faster responses.

## ☐ Conclusion

- **Recap:**
  - API Gateway pattern simplifies microservice management.
  - Spring Cloud Gateway provides a powerful tool for implementing it in Spring Boot.