Spring Boot Microservice

Makarand Bhoir

Agenda

- Microservice Introduction
- Monolith vs Microservice
- Benefits of Microservice
- Microservice Architecture
- Spring Cloud Eureka Server
- Spring Cloud Gateway
- Resilience4j

Monolithic Application Architecture

- The term "monolithic" comes from the Greek term's *monos* and *lithos*, together meaning a large stone block. The meaning in the context of IT for monolithic software architecture characterizes the uniformity, rigidity, and massiveness of the software architecture.
- A monolithic code base framework is often written using a single programming language, and all business logic is contained in a single repository.

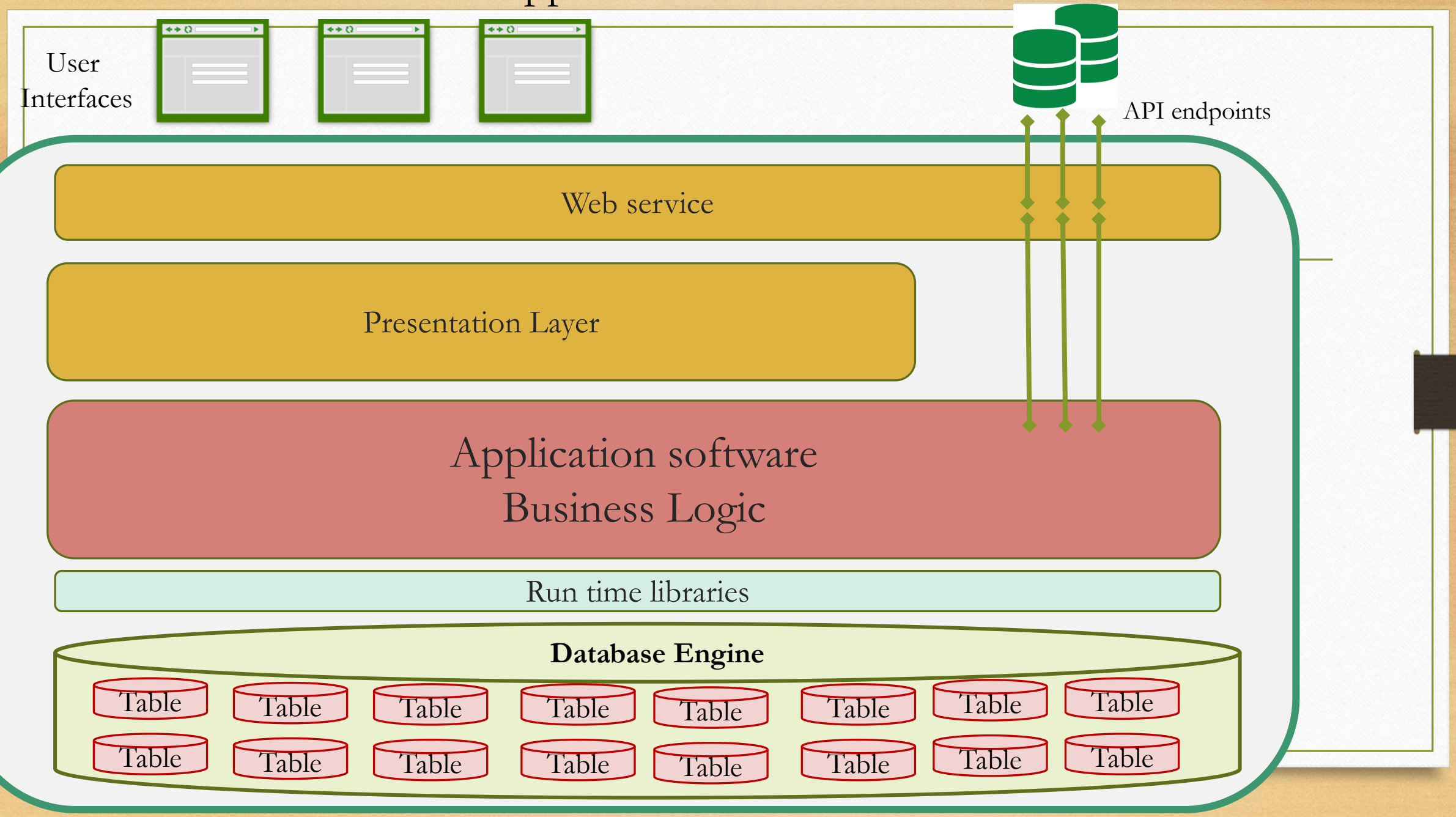
Monolithic Application Architecture

- Typical monolithic applications consist of a single shared database, which can be accessed by different components.

N-tier architecture in Monolithic Apps

- The N-tier architecture allows developers to build applications on several levels. The simplest type of N-tier architecture is the one-tier architecture.
- In this type of architecture, all programming logic, interfaces, and databases reside in a single computer.
- As soon as developers understood the value of decoupling databases from an application, they invented the two-tier architecture, where databases were stored on a separate server.
- This allowed developers to build applications that allow multiple clients to use a single database and provide distributed services over a network.

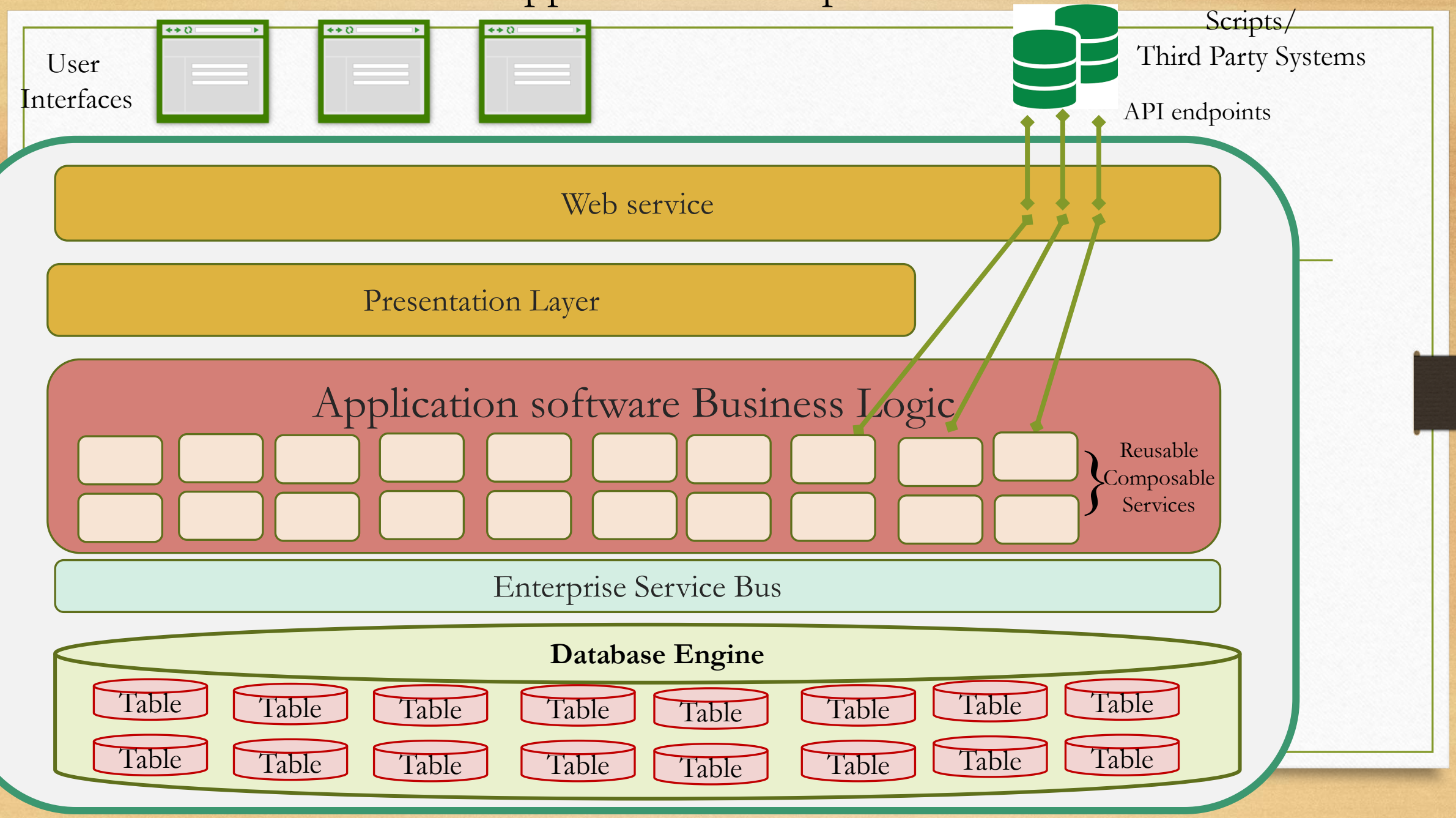
Monolithic Application



Service Oriented Architecture

- Service-oriented architecture was largely created as a response to traditional, monolithic approaches to building applications.
- SOA breaks up the components required for applications into separate service modules that communicate with one another to meet specific business objectives.
- Each module is considerably smaller than a monolithic application, and can be deployed to serve different purposes in an enterprise. Additionally, SOA is delivered via the cloud and can include services for infrastructure, platforms, and applications.

Monolithic Application: Enterprise SOA Model



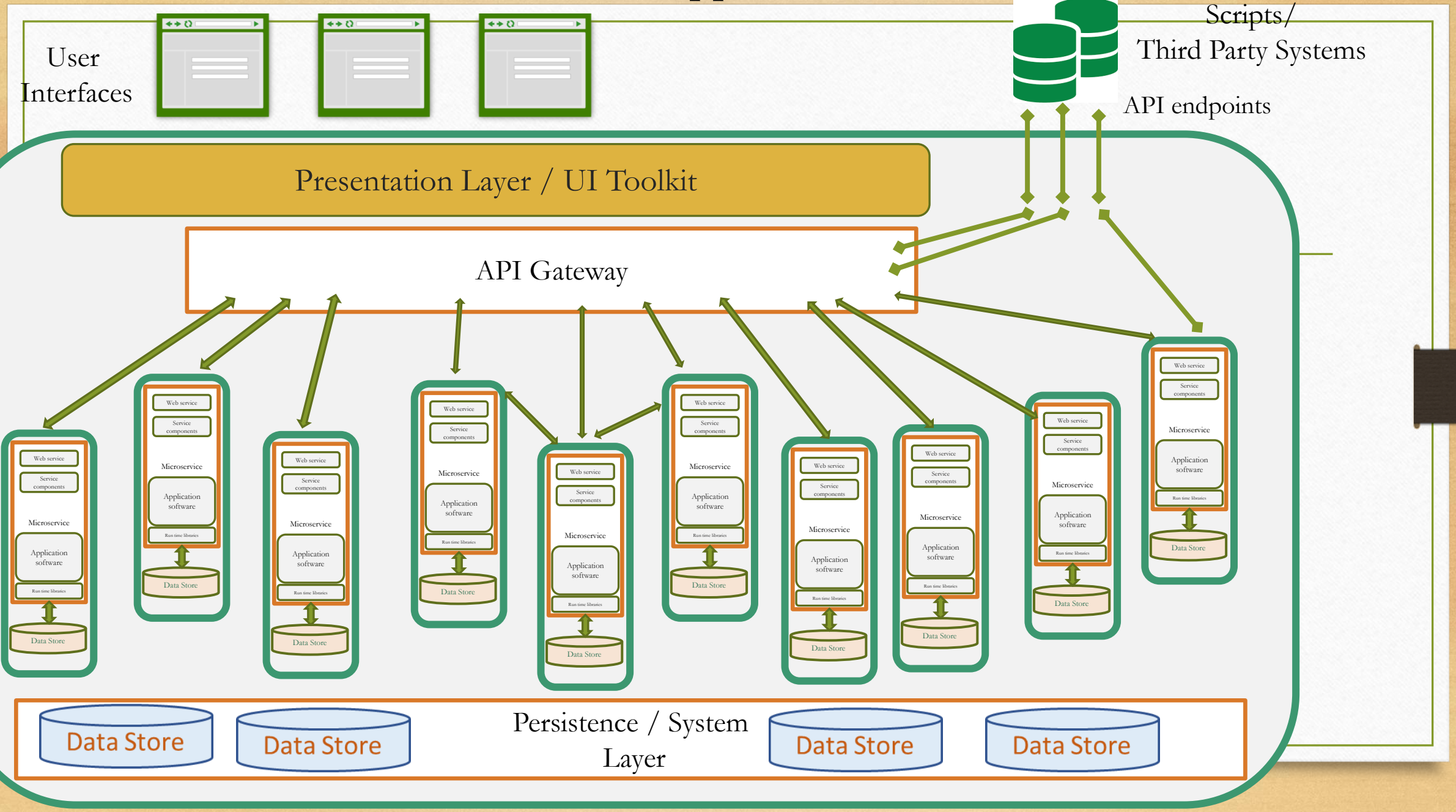
Microservices

- Microservices is an architecture style, in which large complex software applications are composed of one or more services.
- Microservice can be deployed independently of one another and are loosely coupled.
- Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability.

Microservices

- Microservices can be developed in any programming language. They communicate with each other using language-neutral application programming interfaces (APIs) such as Representational State Transfer (REST).
- Microservices also have a bounded context. They don't need to know anything about underlying implementation or architecture of other microservices.

Microservices-based Application

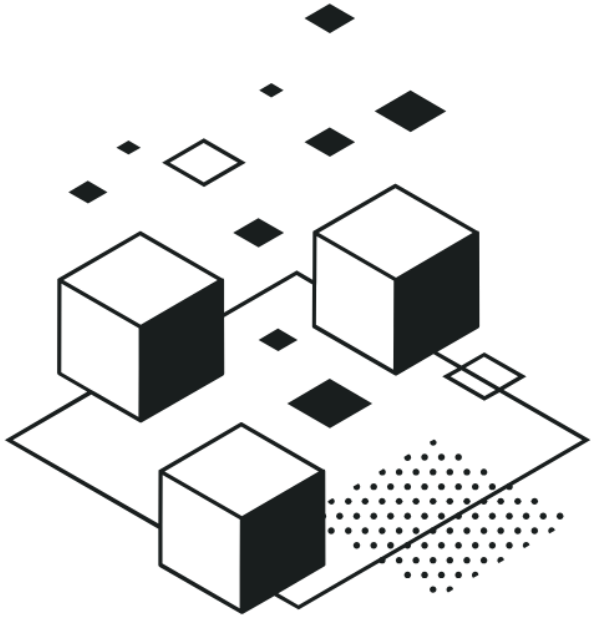


Microservices Architecture

Microservice architectures are the ‘new normal’. Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code.

Spring Boot’s many purpose-built features make it easy to build and run your microservices in production at scale.

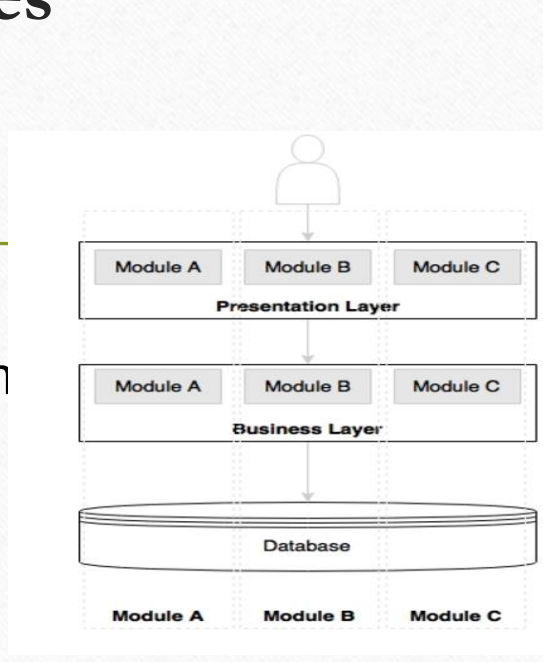
And don’t forget, no microservice architecture is complete without Spring Cloud – easing administration and boosting your fault-tolerance.

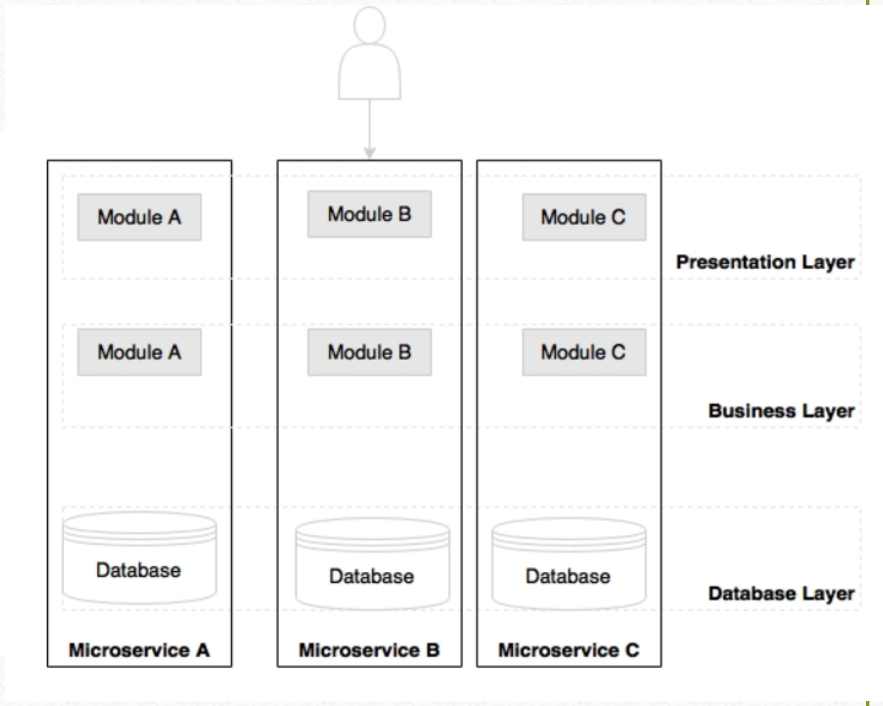


Benefits of Microservices

The transition from monolithic applications to microservices can many benefits, including:

1. Improved scalability
2. Increased flexibility
3. Faster delivery time
4. Better resource utilization
5. Technology diversity





Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

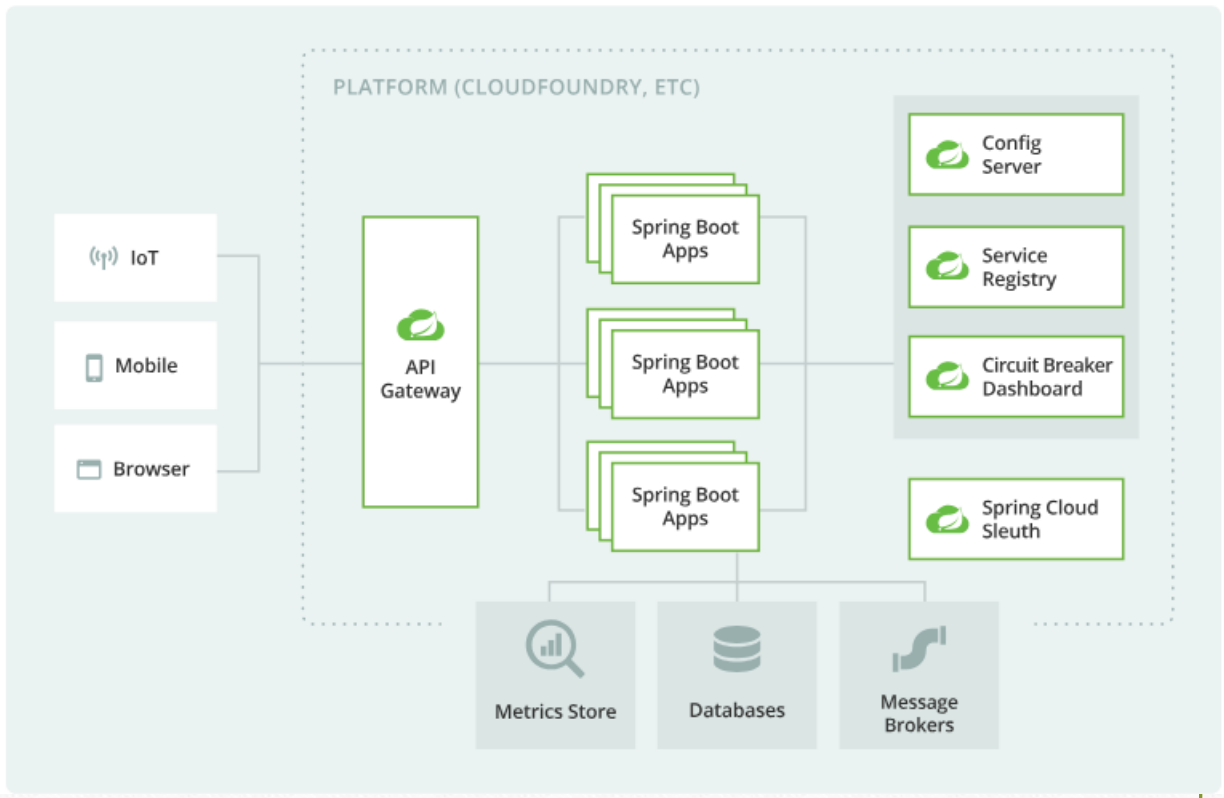
Microservices resilience with Spring Cloud

The distributed nature of microservices brings challenges. Spring helps you mitigate these.

With several ready-to-run cloud patterns, Spring Cloud can help with

service discovery, load-balancing, circuit-breaking, distributed tracing, Monitoring and

API gateway.



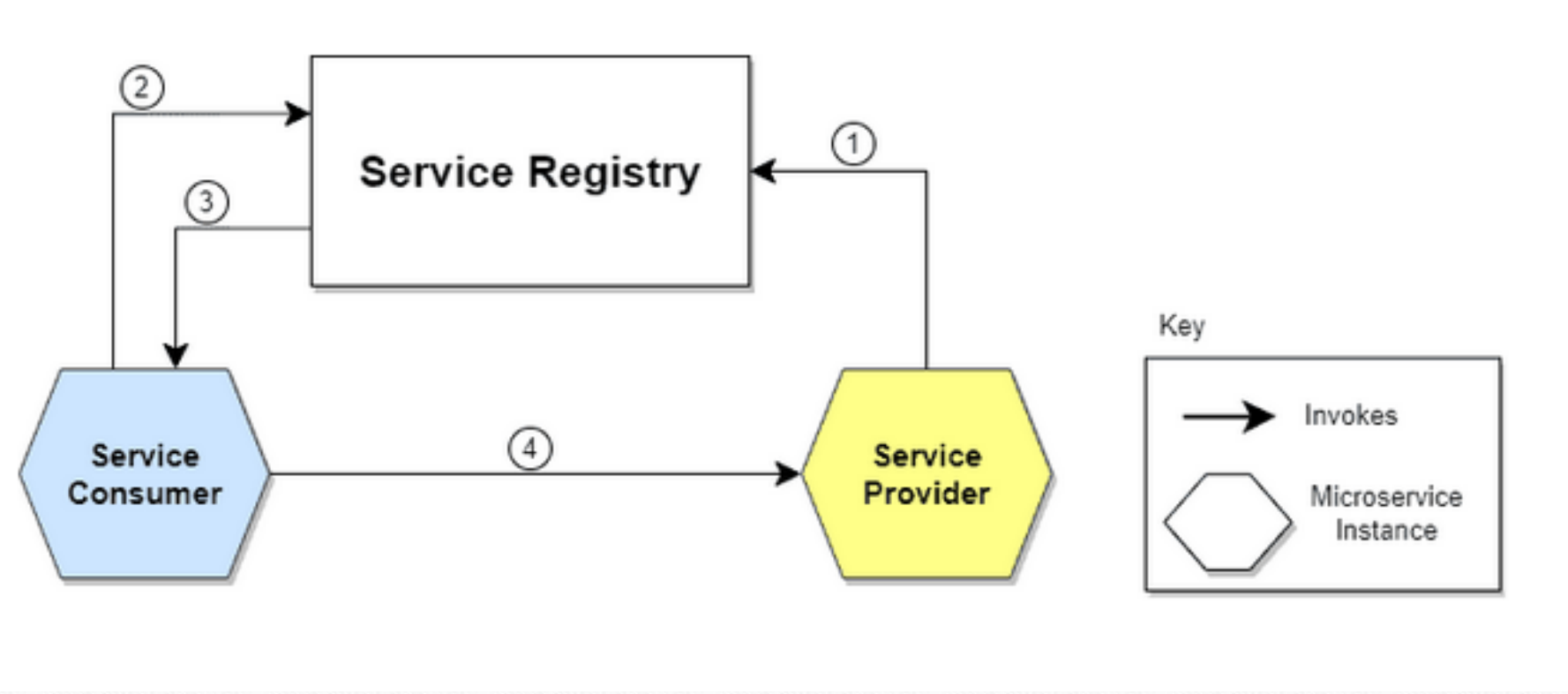
Spring Cloud Eureka

Service discovery in Microservices

- A microservices-based application typically runs in virtualized or containerized environments. The number of instances of a service and its locations changes dynamically.
- We need to know where these instances are and their names to allow requests to arrive at the target microservice. This is where tactics such as Service Discovery come into play.
- The Service Discovery mechanism helps us know where each instance is located. In this way, a Service Discovery component acts as a registry in which the addresses of all instances are tracked.
- The instances have dynamically assigned network paths. Consequently, if a client wants to make a request to a service, it must use a Service Discovery mechanism.

How Does Service Discovery Works?

- Service Discovery handles things in two parts. First, it provides a mechanism for an instance to register. Second, it provides a way to find the service once it has registered.



What is Microservices Load Balancing?

- At its core, load balancing for microservices aims to distribute incoming network traffic across multiple instances of a service to ensure no single instance is overwhelmed with too much traffic. This results in:
  - Optimal Resource Utilization: Traffic distribution ensures that all service instances are used effectively.
  - Enhanced Application Availability: In the event that a service instance fails, traffic is rerouted to healthy instances.
  - Reduced Latency: Requests are directed to the nearest or quickest service instance, minimizing response times.

Spring Netflix Eureka

- Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.
- With a few simple annotations, you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.
- The patterns provided include Service Discovery (Eureka).

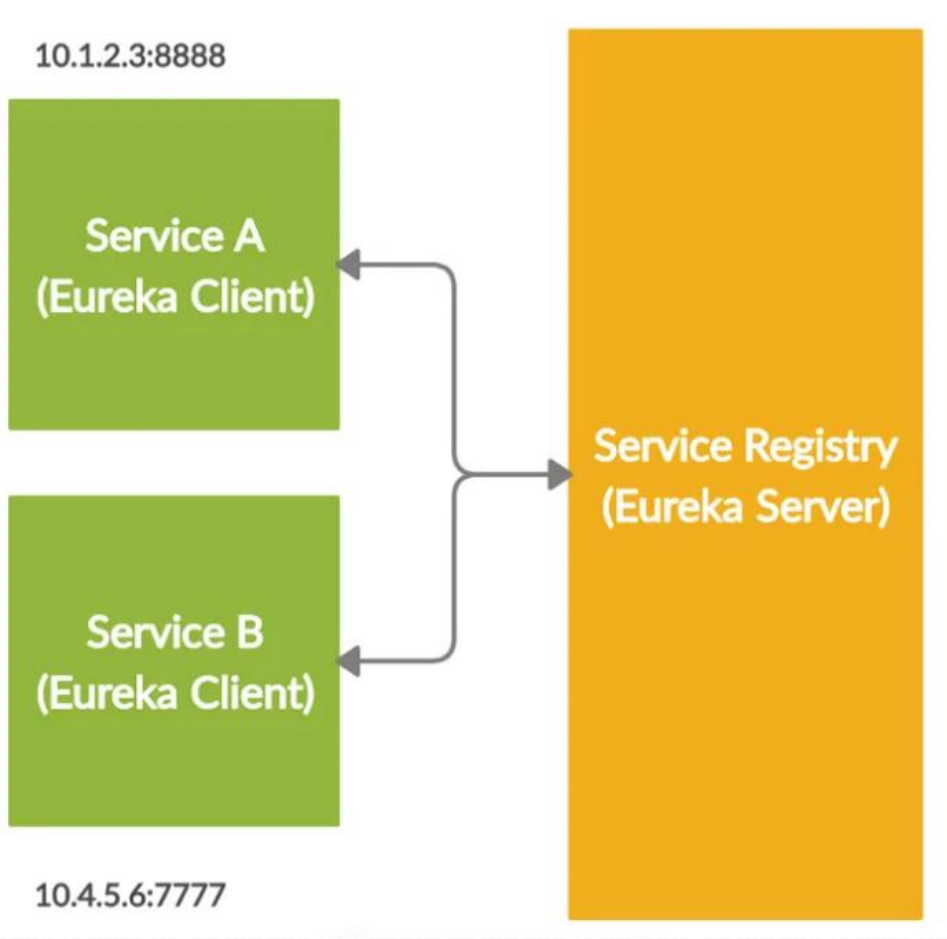
Spring Netflix Eureka - Features

Spring Cloud Netflix features:

- Service Discovery: Eureka instances can be registered and clients can discover the instances using Spring-managed beans
- Service Discovery: an embedded Eureka server can be created with declarative Java configuration

Spring Netflix Eureka

- Eureka is a REST based service which is primarily used for acquiring information about services that you would want to communicate with. This REST service is also known as Eureka Server.
- The Services that register in Eureka Server to obtain information about each other are called Eureka Clients.



Spring Netflix Eureka

Eureka mainly consists of main components, let's see what they are:

1. Eureka Server: It is an application that contains information about all client service applications. Each microservice is registered with the Eureka server and Eureka knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

2. Eureka Client: It's the actual microservice and it registers with the Eureka Server, so if any other microservice wants the Eureka Client's address then they'll contact the Eureka Server. All operations with Eureka Clients may take some time to be reflected on Eureka Server, and then on other Eureka Clients

Spring Netflix Eureka Server

Implementing a Eureka Server for service registry is as below:

- adding spring-cloud-starter-netflix-eureka-server to the dependencies
- enabling the Eureka Server in a @SpringBootApplication by annotating it with @EnableEurekaServer
- configuring some properties

Spring Netflix Eureka Server

Dependencies need to add in pom.xml

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

<dependencies>
<dependencyManagement>
 <dependencies>
<dependency>              <groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-parent</artifactId>
<version>Greenwich.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Spring Netflix Eureka Server

Then we'll create the main application class:

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

Spring Netflix Eureka Server

Finally, we'll configure the properties in YAML format, so an application.yml will be our configuration file:

```
server:
 port: 8761
eureka:
 client:
  registerWithEureka: false
  fetchRegistry: false
```

Spring Netflix Eureka Client

- For a @SpringBootApplication to be discovery-aware, we have to include a Spring Discovery Client (for example, spring-cloud-starter-netflix-eureka-client) into our classpath.

- Then we need to annotate a @Configuration with either @EnableDiscoveryClient or @EnableEurekaClient. Note that this annotation is optional if we have the spring-cloud-starter-netflix-eureka-client dependency on the classpath.

- The latter tells Spring Boot to use Spring Netflix Eureka for service discovery explicitly. To fill our client application with some sample-life, we'll also include the spring-boot-starter-web package in the pom.xml and implement a REST controller.

Spring Netflix Eureka Client

Dependencies need to add in pom.xml

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

Spring Netflix Eureka Client

Then we'll create the main application class:

```
@SpringBootApplication
@RestController
public class EurekaClientApplication{
@Autowired
@Lazy
private EurekaClient eurekaClient;      @Value("$
{spring.application.name}")
private String appName;
public static void main(String[] args) {
SpringApplication.run(EurekaClientApplication.class, args);     }
@Override
public String greeting() {
return String.format("Hello from '%s'!",
eurekaClient.getApplication(appName).getName());
} }
```

Working with Eureka Server

- *Lab:* Working with Spring Service Registration and Discovery using Netflix Eureka

Lab Guide:

https://spring.io/guides/gs/service-registration-and-discovery

Spring Cloud Gateway

Spring Cloud Gateway

- Spring Cloud Gateway is a powerful API gateway built on Spring Framework 5.0, designed to provide a unified entry point for microservices architectures.
- It acts as a reverse proxy and acts as a single point of entry for all requests to your microservices.

Spring Cloud Gateway

Spring Cloud Gateway features:

- Built on Spring Framework and Spring Boot
- Able to match routes on any request attribute.
- Predicates and filters are specific to routes.
- Circuit Breaker integration.
- Spring Cloud DiscoveryClient integration
- Easy to write Predicates and Filters
- Request Rate Limiting
- Path Rewriting

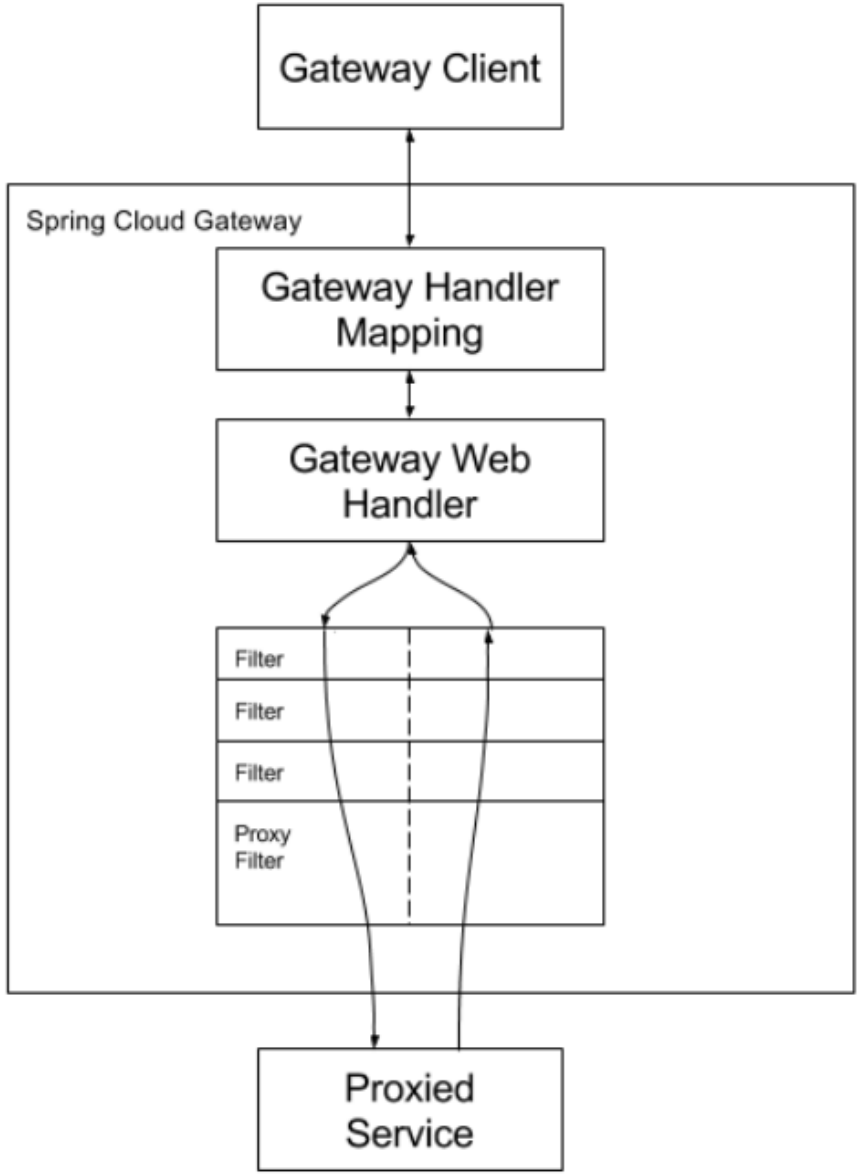
Spring Cloud Gateway

- Dependency needed for Cloud Gateway in Spring Boot:

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

Spring Cloud Gateway

- Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler.
- This handler runs the request through a filter chain that is specific to the request.
- The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent.
- All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.



Spring Cloud Gateway

Most important things to consider:

- **Route:** The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.

- **Predicate:** This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.

- **Filter:** These are instances of Spring Framework GatewayFilter that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

Spring Cloud Gateway

- Gateway Configuration in application.yml:

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: serviceA
          uri: http://localhost:8081
          predicates:
            - Path=/service/**
        - id: serviceB
          uri: http://localhost:8082
          predicates:
            - Path=/service/**
```

Spring Cloud Gateway

- Gateway Configuration in Java file:

```java
@Configuration
public class GatewayConfiguration {
 @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
     return builder.routes()
             .route(r -> r.path("/servicea/**")
                     .uri("http://localhost:8081/"))
             .route(r -> r.path("/serviceb/**")
                     .uri("http://localhost:8082/"))
             .build();
 }
}
```

Working with API Gateway

- *Lab:* Working with Spring Cloud Loadbalancer

Lab Guide:

- https://spring.io/guides/gs/spring-cloud-loadbalancer

Spring Cloud Circuit Breaker – Resilience4J

Resilience4J

- Resilience4j is a lightweight fault tolerance library designed for functional programming.

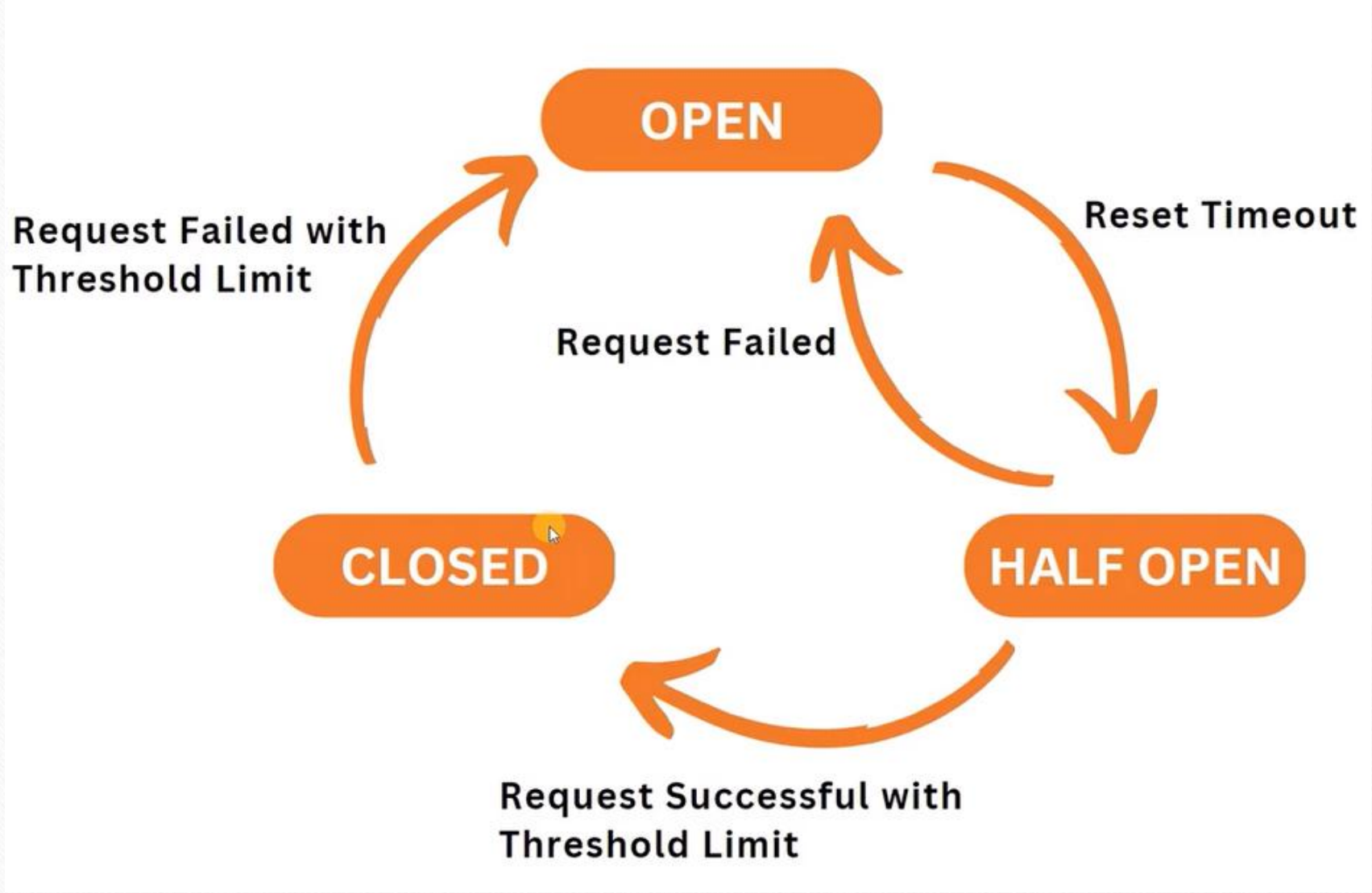
Resilience4J – Circuit Breaker

- Circuit Breaker is a resilience pattern that helps to prevent cascading failures in distributed systems. It acts as a safety mechanism that isolates failing components to prevent further errors from propagating.

- The CircuitBreaker uses a sliding window to store and aggregate the outcome of calls. It can choose between a count-based sliding window and a time-based sliding window.

- The count-based sliding window aggregrates the outcome of the last N calls. The time-based sliding window aggregrates the outcome of the calls of the last N seconds.

Spring Cloud Circuit Breaker



Resilience4J – Circuit Breaker

- Open State:
  - When a component fails a certain number of times within a specified window, the circuit breaker transitions to the open state.
  - In this state, all subsequent requests are immediately rejected without attempting to call the component.

- Closed State:
  - If the component successfully recovers from the failure, the circuit breaker transitions to the closed state.
  - In this state, requests are allowed to proceed normally.

Resilience4J – Circuit Breaker

- Half-Open State:
  - After a specified wait duration, the circuit breaker enters the half-open state. In this state, a limited number of requests are allowed to pass through.
  - If these requests are successful, the circuit breaker transitions to the closed state. Otherwise, it remains in the open state.

- Fallback Mechanism:
  - It allows to configure a fallback mechanism to be executed when the circuit breaker is open.
  - This allows you to provide a graceful degradation of service or return a default response.

Resilience4J – Circuit Breaker

- Properties and Configuration:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-resilience4j</artifactId>
</dependency>
```

- Configuration

  In Controller :

```
@CircuitBreaker(name = "orderMS", fallbackMethod =
"fallbackMethod")
```

  In Properties:

```
13    resilience4j:
14      circuitbreaker:
15        instances:
16          orderMS:
```

Resilience4J – Circuit Breaker

Parameter	Description	Default Value
name	Unique identifier for the circuit breaker.	default
failureRateThreshold	Percentage of failed calls within a sliding window that triggers the open state.	50%
waitDuration	Duration the circuit breaker remains open after it trips.	60 seconds
slidingWindowSize	Size of the sliding window used to calculate the failure rate.	10
recordFailurePredicate	Predicate used to determine if a call should be considered a failure.	Default predicate based on exceptions

Working with Spring Cloud Circuit Breaker

- *Lab:* Working with Circuit Breaker

Lab Guide:

- https://spring.io/guides/gs/cloud-circuit-breaker

Thank You

Makarand Bhoir