
Spring Boot
Microservices

Module-1 – Version -2

MICRO SERVICES – SPRING BOOT

INTRODUCTION TO MICRO SERVICES

Micro Services

- Problems with Monolithic Application
- What is Microservices
- Advantages and disadvantages
- When not to use Microservice
- Use case for MicroServices

Monolith Application

- ***A monolithic application*** *is a single-tiered software application*
  - *Has user interface and data access code combined into a single program from a single platform.*

- Its characteristics
  - *Doesn't have Module but Modularized using* object-based approach to be distributed across multiple computers
  - *Deployed together as a single deployment unit- EAR or WAR.*
  - Long Release Cycles
  - Large Teams

Challenges to Monolith

- Large codebases become mess over the time
- Multiple teams working on single codebase become tedious
- Difficulty in Adapting to Device Explosion
- Scaling up certain parts of the application
- Technology updates/rewrites become complex and expensive tasks

Micro Services

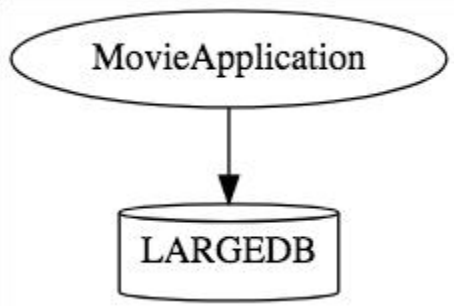
- An engineering approach focused on **decomposing** applications with **well-defined interfaces** - service contracts such as a JSON schema or WSDL
  - **Deployed independent**ly as **services** and operated by **small teams** who own the **entire lifecycle** of the service.
  - Each service handles a specific business domain (logging, auth, orders, customers)
  - Each Service provides the implementation for user interface, business logic, and connection to database.
  - Each Service can be scaled up Individually

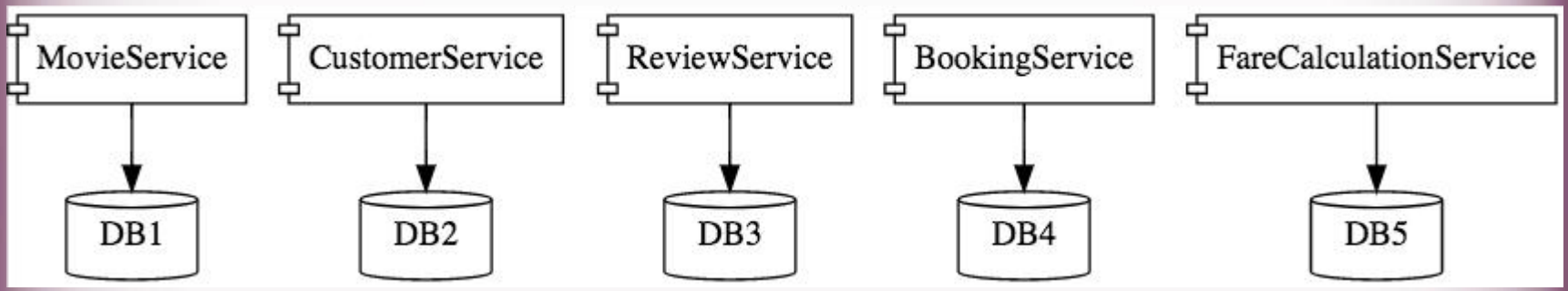
Micro Service

```
class UserApp() {

User getUser() {

// 1. auth user

// 2. get user data

// 3. log user actions
    }

}
```

```
class UserApp() {

void authUser(User user) {
    ... }

User getUserData() { ... }

void logUserActions() { ... }

}
```

Micro Services





Characteristics of Micro Service

- **Single-function**
  - Each and every service has a specific function, or responsibility.
  - A service can do many tasks, but all of them are relevant to a single function.
- **Well-defined interfaces**
  - Services must provide an interface that defines to communicate with it.
  - Defines a list of methods, and their inputs and outputs.
  - JSON Schema or WSDL files

Characteristics of Micro Service

- **Independent**
  - Services doesn’t know about each other implementation.
  - They can get tested, deployed, and maintained independently.
  - Services may be implemented using different language stacks, and communicate with different databases.
    - They can work together to complete a required operation.

Characteristics of Micro Service

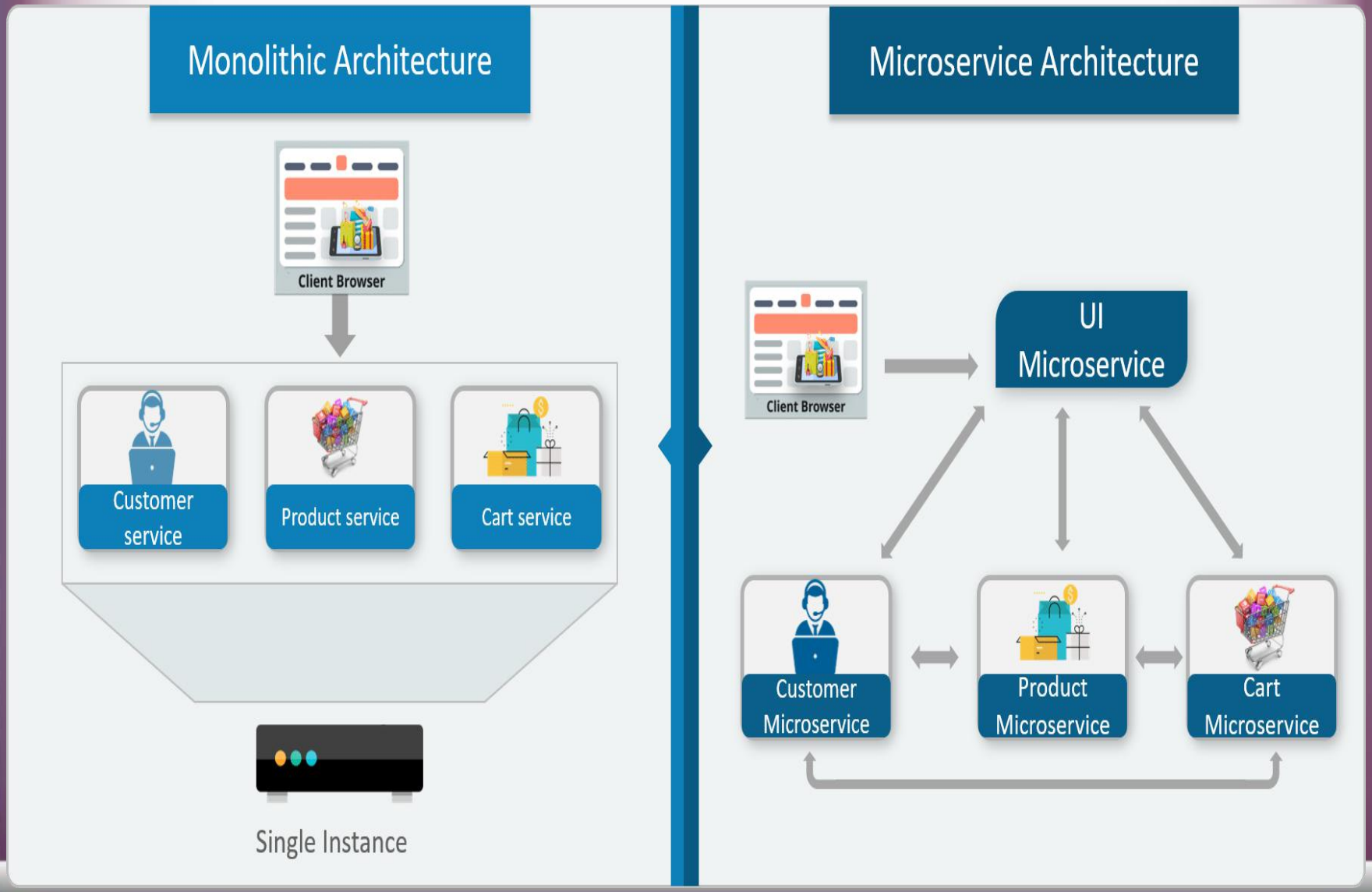
- **Small Teams**
  - Split the work up and team across the services.
  - Each team focuses on a specific service, they don't need to know about internal workings of other teams.
  - Teams can work efficiently, communicate easily,
  - Each service can be deployed rapidly as soon as it's ready.

- **Entire Lifecycle**
  - The team is responsible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.
  - No separate team for coding, deployment. Etc

Characteristics of Micro Service

- **Minimizing Communication**
  - Only essential cross-team communication should be through the interface that each service provides.
  - They all need to agree on the external interface, so that communication between services is clearly defined.
- **Cloud Enabled**
  - Dynamic Scaling

Monolithic vs Micro Service



When Not to use Micro Services

- Complexity is a key factor
  - *"…don't even consider microservices unless you have a system that's too complex to manage as a monolith…"* –***Martin Fowler***
- **If complexity isn't problem, microservices aren't the solution.**
- Micro service architecture also brings with it significant overhead
  - Design
  - Interoperability of services
  - Management
  - Use of system resources.

Challenges with Micro service

- Development has some inherent complexities

- **Quick Setup needed**
  - Should be able to create micro services quickly.
- **Automation**
  - Smaller component build ,deployment, monitoring etc should automated
- **Visibility :**
  - Should be able to monitor and identify problems automatically.
  - Need great visibility around all the components.
- **Debugging**
  - Centralized Logging and Dashboards are essential to debug problems.

Challenges with Micro service

- **Configuration Management**
  - Need to maintain configurations for hundreds of components across environments.
  - Should Have Configuration Management solution
- **Dynamic Scale Up and Scale Down**
  - Applications should be easily scaled up and down
- **Pack of Cards :**
  - If a Service at the bottom of the call chain fails, it can have knock on effects on all other services.
  - Services should be fault tolerant by Design.
- **Consistency**
  - Should have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring

Solutions to Challenges

- **Spring Boot**
  - *Enable building production ready applications quickly*
  - Provide non-functional features
  - embedded servers (easy deployment with containers)
  - metrics (monitoring)
  - health checks (monitoring)
  - externalized configuration

Solutions to Challenges

- **Spring Cloud**
  - *Cloud solutions – From Spring and Open Sourced*
- **Dynamic Scale Up and Down.**
  - Using a combination of Naming Server (Eureka)
  - Client Side Load Balancing
  - Feign (Easier REST Clients)
- **Visibility and Monitoring**
  - Zipkin Distributed Tracing
  - API Gateway
- **Configuration Management**
  - Spring Cloud Config Server
- **Fault Tolerance**
  - Cloud Circuit Breaker

SPRING CLOUD EUREKA

Service Register and Discovery

- Application is decomposed into Many Microservices.
  - Each deployed in different servers and different ports.

- Services may need to communicate with each other to execute some tasks
  - ***Using ip address and port number has many limitations on invoking the services***

Service Register and Discovery

- "**Service Registration**".
  - Services are registered in a centralized location with an identifier and server details.

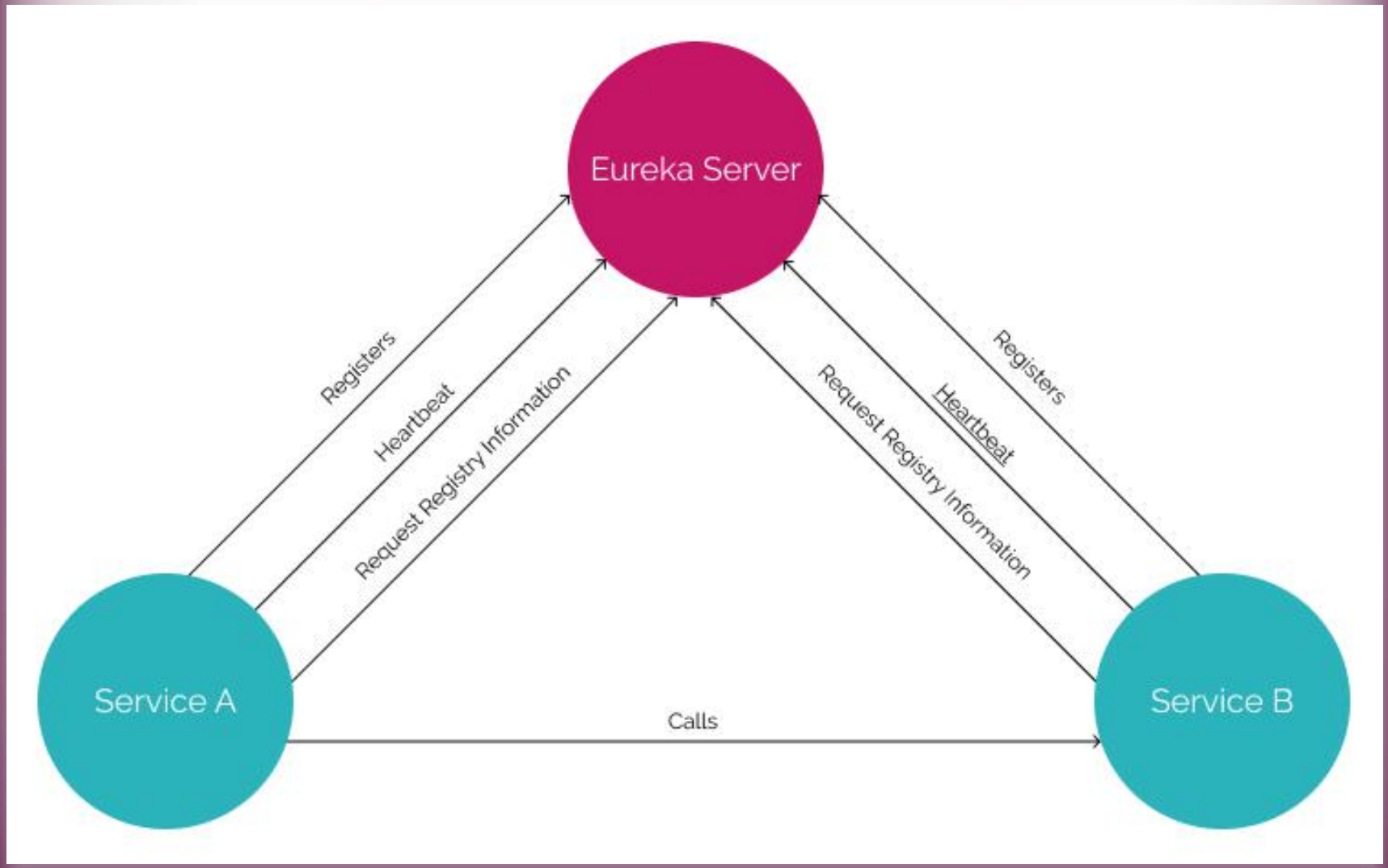
- "**Service Discovery**"
  - Microservice should be able to look up the list of registered services from the centralized location

- Implementations of **Service Registration and Discovery** Server
  - **Netflix Eureka**
  - **Consul**
  - **Zookeeper**

Netflix Eureka Server

- Micro services will register with **Eureka** server
- Services provide the following details
  - host name, ip address, port, and health indicators etc…
- **serviceId**.
  - The Unique Id given to the registered Services
    - Also Used to by other services for access

Service Registry



Netflix Eureka Server

- **@EnableEurekaServer**
  - Added to the main Spring Boot Application configuration class
  - Used to Notify the **Spring** Container that this application is a registration service by **Netflix** technology
  - It also activates Eureka Server related configurations

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

Server Config Properties

```
server:
  port: 8761

spring:
  application:
    name: eureka-server-registry
eureka:
  client:
    fetch-registry: false
    register-with-eureka: false
    service-url:
      defaultZone: http://localhost:8761/eureka
```

Configuration Properties

- **fetch-registry**=false **(default: true)**
  - A Eureka instance is also a Eureka client
  - It can fetch the registry containing the details of other instances.
  - **"false"** means it will not be fetching the registry information

- **register-with-eureka**=false **(default: true)**
  - Should this server register itself as a client;
  - **"false"** means it prevents itself from acting as a client.

Configuration Properties

- **eureka.client.serviceUrl.defaultZone**
  - In a standalone mode
    - points to the local server address.
    - Switch off the client-side behavior
    - Does not keep trying and failing to reach its peers.

EUREKA DISCOVERY CLIENT

Discovery From Eureka Server

- **EurekaClientConfigBean**
  - Implementation of EurekaClientConfig.
  - Properties are prefixed by eureka.client.
- **Discovery clients**
  - Clients can look up and fetch information of other services
  - Can communicate without IP addresses and port numbers

Discovery Clients

- **@EnableDiscoveryClient**
  - Added in the main spring boot configuration class
  - ***Works with any Discovery Client implementations***
  - ***Example : Eureka, Consul, Zookeeper***
- **@EnableEurekaClient**
  - Works only with Eureka

Discovery Client Service

```
@SpringBootApplication
@EnableEurekaClient
public class BloodDonarServiceApplication {

public static void main(String[] args) {

SpringApplication.run(BloodDonarServiceApplication.class, args);

}

}
```

application.yml

```yaml
spring:
  application:
    name: product-service
server:
  port: 5454

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

SUMMARY

SPRING WEBCLIENT

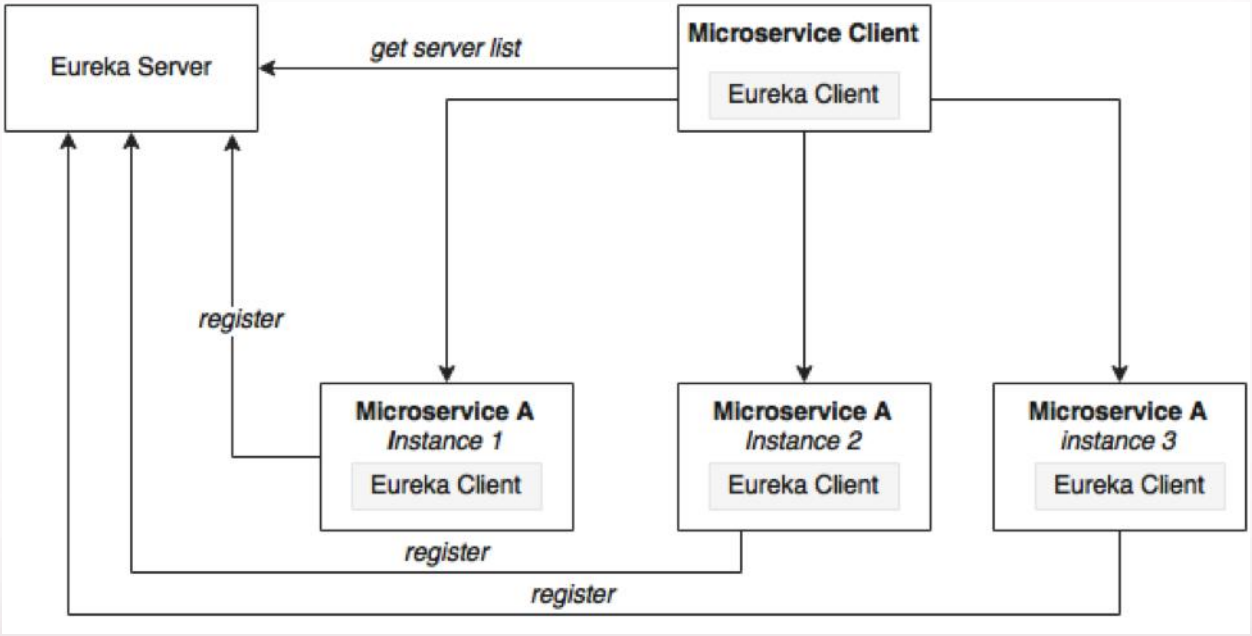
WebClient

- Introduced from Spring 5
  - Its a non-blocking client with support for Reactive Streams.
  - It’s a replacement for *RestTemplate* .
  - Its more functional and is fully reactive.
  - It's included in the spring-boot-starter-weblux dependency

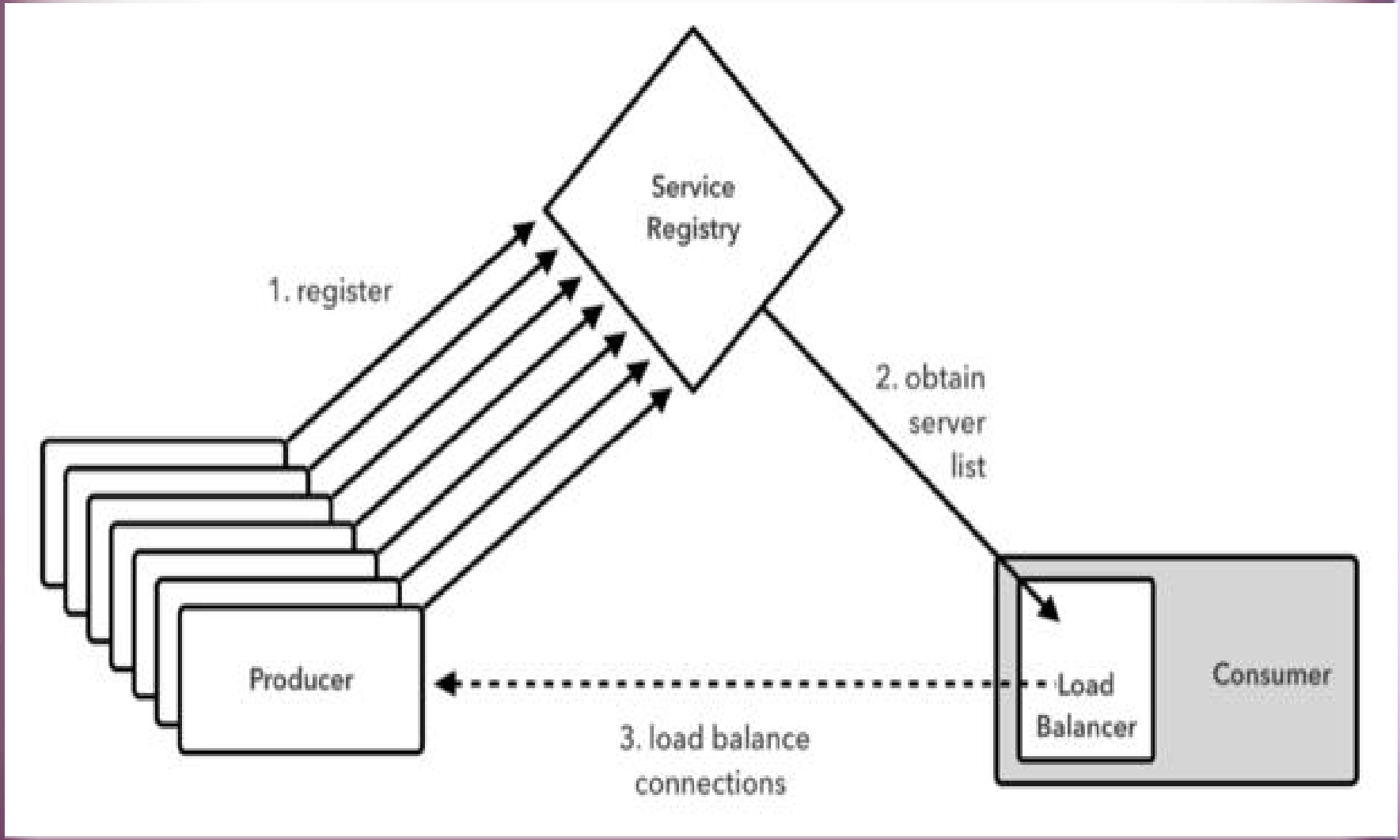
Load Balancing

- A distributed system may consist of many services running on different computers.
- When the number of users is large, a service is created multiple replicas.
- Each replica runs on a different computer.
- **A Load Balancer**" helps to distribute incoming traffic equally among servers.

Load Balancing



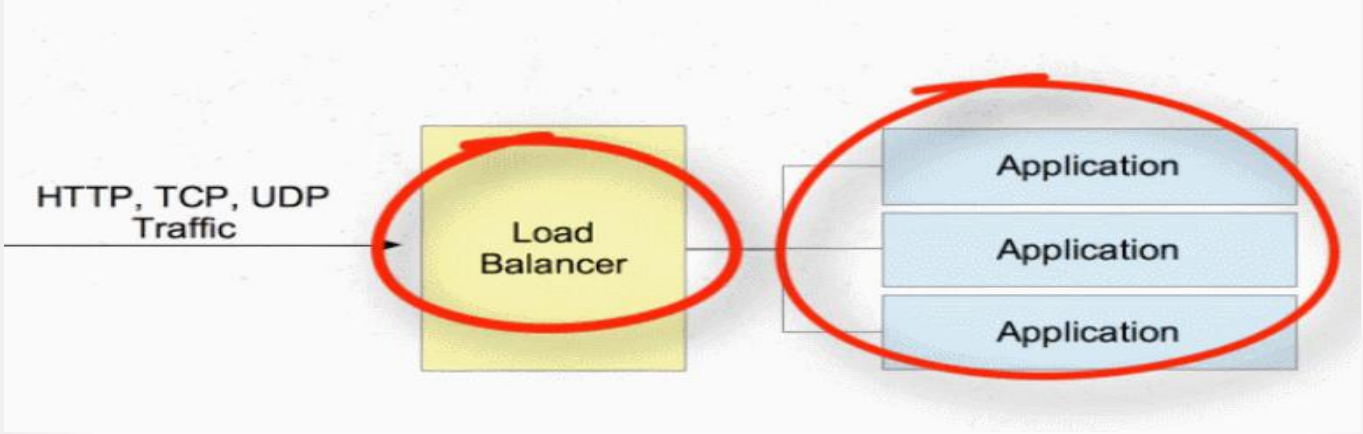
Working of Load Balancer

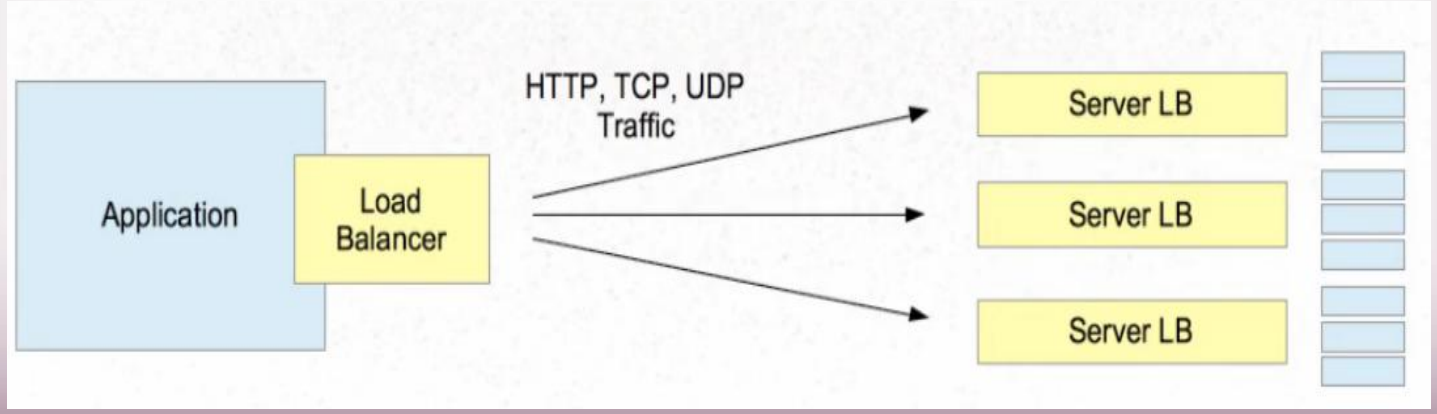


Load Balancing on Server Side

- Load Balancers are components placed at the Server Side.
- When the requests come from the **Client**, they will go to the load balancer, and the load balancer will designate a **Server** for the request.
- It may use algorithm like random designation.
- Most load balancers are hardware, integrated with software to control load balancing.
- Need a separate server to host the load balancer instance which has the impact on cost and maintenance.

Server Side and Client Side





Scalability

- As the traffic to a Service increases it can hinder its performance
  - Should be managed properly.
- Load balancing, gives ability to add a physical or virtual server to accommodate demand without causing a service disruption.
- As new servers come online, the load balancer recognizes them and seamlessly includes them in the process.

Load Balancing with Web Client

- Add spring-cloud-starter-load balancer dependencies
- Web Client can be configured to work with Load Balancer client

- **WebClient.Builder**
  - Used to create a load-balanced WebClient
  - Add the @Bean annotation

Load Balancing with Web Client

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

    @Bean
    WebClient client(WebClient.Builder builderRef) {
        return builderRef.build();
    }
```

Reactive Load Balancer

- **ReactiveLoadBalancer**
  - The Implementation of the interface uses a Round-Robin
  - It gets instance from reactive ServiceInstanceListSupplier
  - It retrieves available instances from Service Discovery using a Discovery Client available in the classpath.
- **@LoadBalanced**
  - Need to have a load balancer implementation in the classpath.
  - If the Spring Cloud LoadBalancer is in the class path Then, ReactiveLoadBalancer is used
    - Helps to **use "logical identifiers"** for the URLs passed

Http Methods

- **get()**
  - indicates that we are making a *GET* request.
  - We know that the response will be a single object, so we're using a Mono as explained before.

  - `client.get()`
  - `Client.post()`

WebClient

- `client.get().uri("lb://PRODUCT-SERVICE/api/v1/items")`
- **URI**
  - Takes a service name => **lb://**`PRODUCT-SERVICE`
- The “lb“ in the “uri” will use the LoadBalancerClient
  - Load Balancer Client will resolve to an actual host and port
  - Replaces the URI

Rest Client - Controller

```
@RestController
public class ClientController {
@Autowired
private WebClient client;

@GetMapping(path = "/hotels")
public Flux<String> getAllHotels(){
return client.get()
    .uri("lb://HOTEL-SERVICE/api/v1/hotels")
     .retrieve()
      .bodyToFlux(String.class);
      }
 }
```

Rest Client - Controller

```
@PostMapping(path = "/hotels")
public Mono<HotelDto> create(@RequestBody HotelDto
dto)
{
    return client.post()
        .uri("lb://HOTEL-SERVICE/api/v1/hotels/{id}")
        .body(Mono.just(dto), HotelDto.class)
        .retrieve()
        .bodyToMono(HotelDto.class);
}
```

Rest Client - Controller

```
@DeleteMapping(path = "/hotels/{id}")
public Mono<Void> removeHotelById(@PathVariable("id")
int id){


  return client.delete()
     .uri("lb://HOTEL-SERVICE/api/v1/hotels/{id}" ,id)
        .retrieve()
        .bodyToMono(Void.class);
}
}
```

Rest Client - Controller

```
@GetMapping(path = "/hotels/{id}")
public Mono<String> getHotelById(@PathVariable("id")
int id){

return client.get()
    .uri("lb://HOTEL-SERVICE/api/v1/hotels/{id}",id)
     .retrieve()
      .bodyToMono(String.class);

}
```

To Register Multiple Instances

```
server:
  port: ${port:0}
eureka:
  instance:
    instance-id:
${spring.application.name}:${spring.application.instance_id:${random.value}}
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:8761/eureka
```

Get the Instance's Port Number

```
@Autowired
private ServletWebServerApplicationContext
webServerAppCtxt;


System.out.println(webServerAppCtxt.getWebServer().get
Port());
```