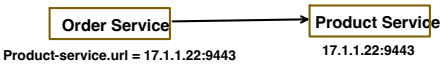
Service Discovery

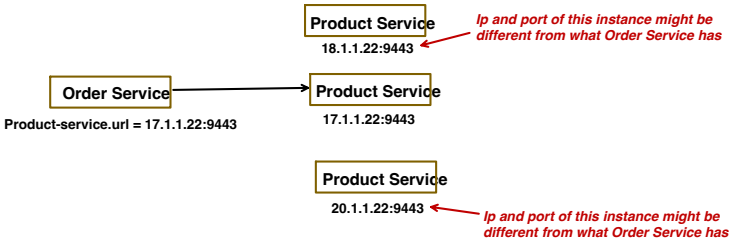
Wednesday, 11 June 2025 12:38 PM

- In Microservices, services or component instances are created and deleted dynamically.
- We can not hardcode the URL of a particular instance, its not scalable and feasible.

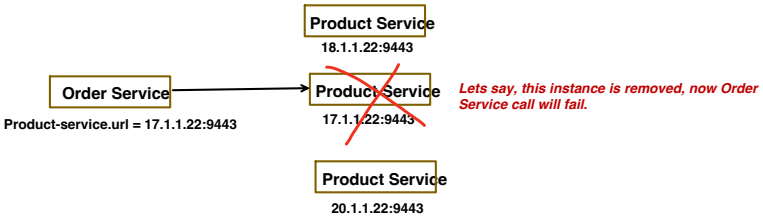


In above example:
- Order service has hardcoded the URL of the Product service and its ok till we have only 1 instance of Product service.

- But in case, when we have multiple Product service instances then….



Also, possible that, particular instance is removed



Now in above example, where multiple instances of Product service is present and Order Service has hardcoded the URL of the Product service, the problem it might face is:

- <u>Single point of failure</u>:
If hardcoded instance of Product Service goes down, Order service will not be able to communicate with any other instance.

- <u>No Load Balancing</u>:
Only one instance of Product Service get overburdened while other instances remain idle.

- <u>Tight Coupling</u>:
Because of hardcoded URL in Order Service, there is tight coupling between Product and Order Service, as without updating the Order Service its not possible to change or move Product Service.

- <u>Difficulty in testing</u>:
Different environment (say production, QA, dev) might uses different URL, which require frequent changes in config and not only cause difficulty in testing but prone to error too.

Solution for above problem is:

SERVICE DISCOVERY (like Eureka)

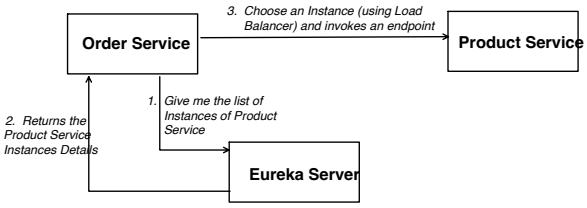
Eureka Server

- Act like a phonebook.
- Has all the instances info of all the registered clients like:
  - Service name
  - Instance id
  - IP
  - Port number
  - Health status

  etc.

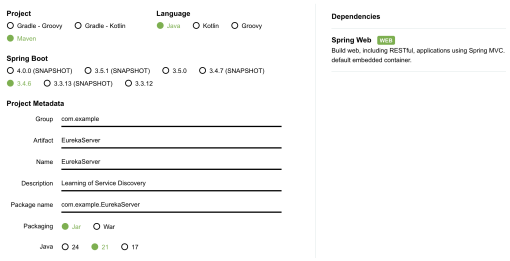
Eureka Client

- Register itself with the Server.
- Discovers an instance of other service via Eureka Server



Lets first set up Eureka Server Application:

Go to Spring Initializer (start.spring.io)



1. pom.xml

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

Version for *"spring-cloud-starter-netflix-eureka-server"* will be automatically resolved by below dependency management.

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>2023.0.1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

2. Enable Eureka Server functionality

Tells Spring boot to create necessary beans, which is required for Eureka Server like:
- *EurekaController*
- *Dashboard etc..*

```java
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

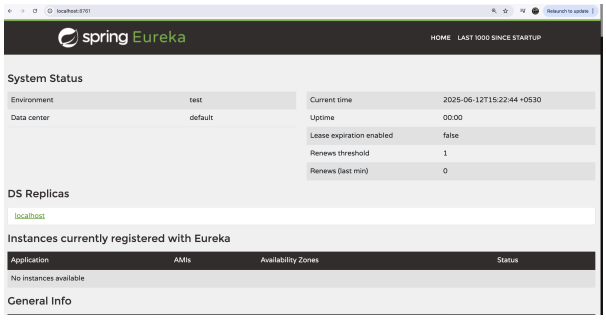
3. application.properties

```
spring.application.name=eureka-server
server.port=8761

# Since it's a server, we don't want it to register and
# also don't want to fetch the instances details
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

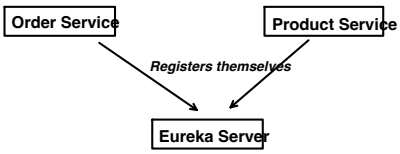
Lets start the Server and see the dashboard:

```
2025-06-12T15:22:06.258+05:30  INFO 43767 --- [eureka-server] [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint(s) beneath base path '/actua
2025-06-12T15:22:06.270+05:30  INFO 43767 --- [eureka-server] [           main] o.s.c.n.e.s.EurekaServiceRegistry        : Registering application EUREKA-SERVER with eurek
2025-06-12T15:22:06.274+05:30  INFO 43767 --- [eureka-server] [      Thread-9] o.s.c.n.e.server.EurekaServerBootstrap  : isAws returned false
2025-06-12T15:22:06.274+05:30  INFO 43767 --- [eureka-server] [      Thread-9] o.s.c.n.e.server.EurekaServerBootstrap  : Initialized server context
2025-06-12T15:22:06.274+05:30  INFO 43767 --- [eureka-server] [      Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl   : Got 1 instances from neighboring DS node
2025-06-12T15:22:06.274+05:30  INFO 43767 --- [eureka-server] [      Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl   : Renew threshold is: 1
2025-06-12T15:22:06.274+05:30  INFO 43767 --- [eureka-server] [      Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl   : Changing status to UP
2025-06-12T15:22:06.276+05:30  INFO 43767 --- [eureka-server] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8761 (http) with context
2025-06-12T15:22:06.276+05:30  INFO 43767 --- [eureka-server] [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
2025-06-12T15:22:06.278+05:30  INFO 43767 --- [eureka-server] [      Thread-9] o.s.EurekaServerInitializerConfiguration : Started Eureka Server
2025-06-12T15:22:06.291+05:30  INFO 43767 --- [eureka-server] [           main] c.e.E.EurekaServerApplication            : Started EurekaServerApplication in 1.513 seconds
```

http://localhost:8761



Now lets set up Eureka Client Application:



Product Service

1. pom.xml

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

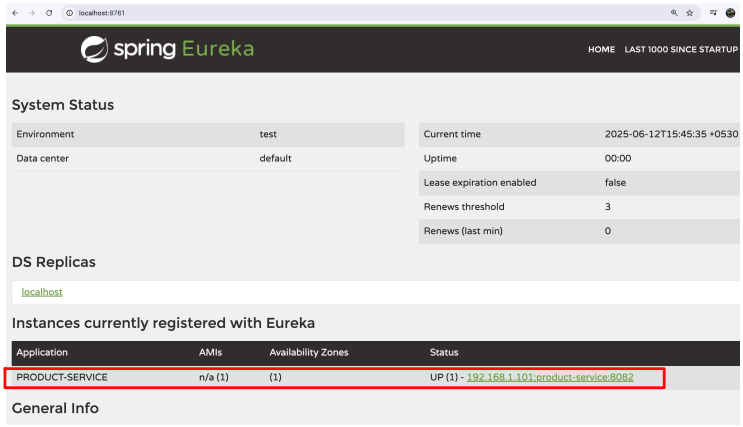
Version for *"spring-cloud-starter-netflix-eureka-client"* will be automatically resolved by below dependency management.

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>2023.0.1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

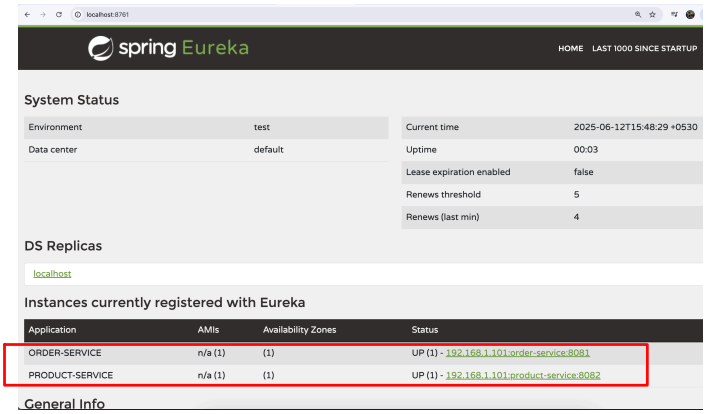
2. application.properties

```
server.port=8082
spring.application.name=product-service

#path of the Eureka Server
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#by-default values are true only, so we can even skip below configs
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
```

Lets start the Product Service Server and see the dashboard again:



Similarly for Order Service, we can register it with Eureka Server



Now lets see, how Order Service can invoke Product Service:

Using RestTemplate

Without Service Discovery

```
public void callProductAPI(String id) {
    RestTemplate restTemplate = new RestTemplate();
    String response = restTemplate.getForObject( url: "http://localhost:8082/products/"+id, String.class);
    System.out.println("Response from Product api call is: " + response);
}
```

Specifically mentioning the URL

With Service Discovery

```
Import org.springframework.cloud.client.ServiceInstance;
Import org.springframework.cloud.client.discovery.DiscoveryClient;
```

```
@Autowired
DiscoveryClient discoveryClient;

public void callProductAPI(String id) {

    RestTemplate restTemplate = new RestTemplate();
    List<ServiceInstance> instances = discoveryClient.getInstances( serviceId: "product-service");
    URI uri = instances.get(0).getUri();

    String response = restTemplate.getForObject( url: uri + "/products/" + id, String.class);

    System.out.println("Response from Product api call is: " + response);
}
```

1. Fetching the instances of "product-service"
2. load balancer logic to choose a particular instance

So with RestTemplate, load balancing (choosing an instance for the product service) logic need to be handled.

Using FeignClient

So with FeignClient, load balancing is handled automatically and by the framework.

So we need to provide the Load Balancer dependency too, apart from *"spring-cloud-starter-netflix-eureka-client"*

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

Earlier without Service Discovery:

```
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

We need to provide the URL

With Service Discovery:

```
@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

No need of the URL, only the registered name of the product service is required.
and
Load Balancing will also be handled internally by the framework.

As a curious engineer, few questions comes to our mind:

Service Registration doubts:
---
1. **How does Eureka Server know whether a client is UP or DOWN?**
2. **Where and how the data is stored?**
3. **What if Eureka Server itself goes down? Is it a single point of failure?**

Discovery Doubts:
---
1. **It can cause latency issue, as each call now required 2 hops, first it has to invoke Eureka Server and then the actual call.**
2. **What if the local cache is stale? Can this lead to calling a dead instance?**

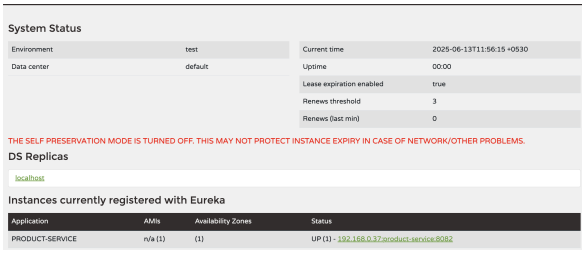
Lets try to find an answer one by one:

1. **How does Eureka Server know whether a client is UP or DOWN?**

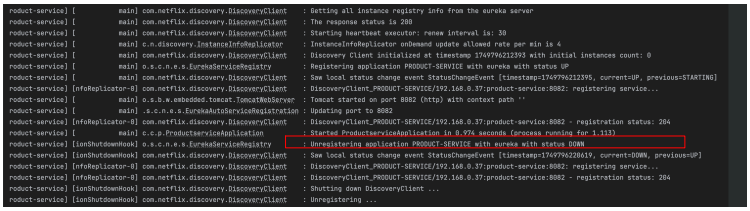
Through Client de-registration request

Through Client Heart Beat

When client application is gracefully shut down, then eureka client sends the de-registration request to Eureka Server, it mark client status as DOWN.



Client(product-service) application logs :



Eureka Client periodically sends Heart Beat to the Eureka Server.

Lets say, if client shut down without sending any de-registration request (bcoz of Network issue).

then Eureka server wait for the heart beat from Client for a particular interval (decided at the time of registration) and if no Heart Beat received within that time interval, it remove the client instance itself.

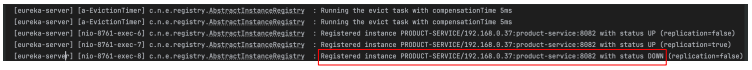
Client(product-service) ***application.properties:***

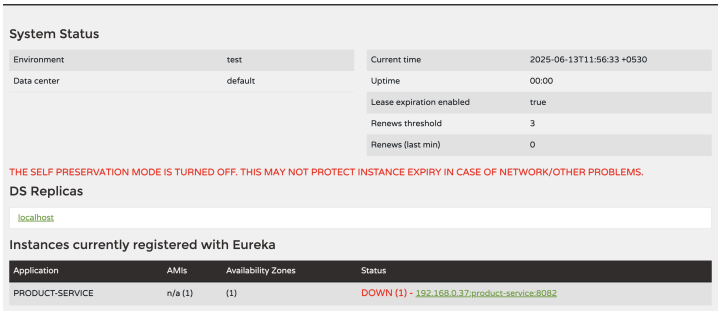
```
server.port=8082
spring.application.name=product-service

#path of the Eureka Server
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#by-default values are true only, so we can even skip below configs
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true

#every 60seconds client will send the heart beat to server
eureka.instance.lease-renewal-interval-in-seconds=60

#telling server to wait this much time, if no heart beat received in this time, then remove me.
#for testing, kept it less than lease-renewal-interval
eureka.instance.lease-expiration-duration-in-seconds=5
```

After this much seconds, client will send the new heart beat to server. Default time is 30sec

Telling server to wait only this much time for the heart beat. After that, you can remove it from the list. Default time is 90sec

Server (Eureka-server) application logs :



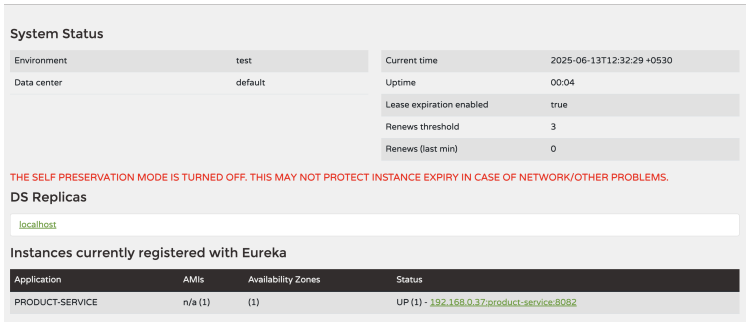


Eureka Server ***application.properties:***

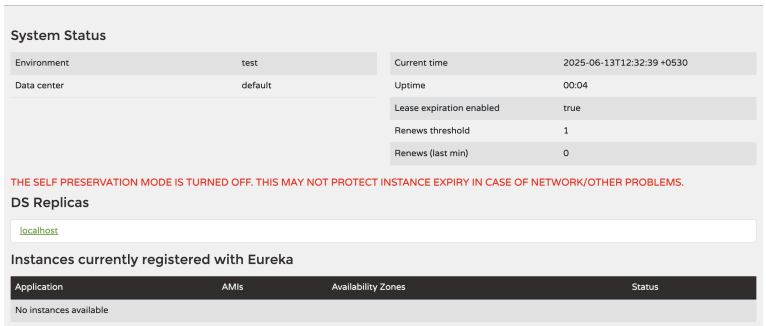
```
spring.application.name=eureka-server
server.port=8761

# Since it's a server, we don't want it to register and
# also don't want to fetch the instances details
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

#by default Server do not remove the client, even heart beat is missed.
#so we need to turn this OFF. And allow the server to remove if client heart beat missed.
eureka.server.enable-self-preservation=false

#how frequently eviction task runs
eureka.server.eviction-interval-timer-in-ms=6000
```

Allowing server to remove the instance, if heart beat is not received.

How often, server check for dead instances



Because in above config:
- Client will send the heart beat after - 60seconds
- Server will wait for the heart beat only for - 5 seconds
So server, removed the client instance from the list, even Client application is running and up.



2. How does Eureka Server stores the data

Eureka Server only stores the data in-memory : Map<String, Lease<InstanceInfo>>

- **Key : *appName/instanceId***
  ex:
  PRODUCT-SERVICE/192.157.2.27:product-service:8082
- value: ***InstanceInfo object***
  - Instance ID
  - App name
  - IP address
  - Host name
  - Port
  - Status (UP, DOWN)
  - Last renewed timestamp
  - Lease duration
    Etc.

3. What if Eureka Server itself goes down? Is it a single point of failure?

- **Yes, a Single Eureka Server is a Single point of failure, if it goes down.**
- **Usually, 3 nodes cluster is used.**

Say, we have 3 Eureka Server on different machine or container:

- **eureka-1 at port 8761**
- **eureka-2 at port 8762**
- **eureka-3 at port 8763**

Now, each Server is a client too. As they have to register themselves, fetch the registry, replicate changes.

application.properties for eureka server 1:

```
spring.application.name=eureka-server
server.port=8761
eureka.instance.hostname=localhost

eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone= http://localhost:8762/eureka/, http://localhost:8763/eureka/
```

application.properties for eureka server 2:

```
spring.application.name=eureka-server
server.port=8762
eureka.instance.hostname=localhost

eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone= http://localhost:8761/eureka/, http://localhost:8763/eureka/
```

application.properties for eureka server 3:

```
spring.application.name=eureka-server
server.port=8763
eureka.instance.hostname=localhost

eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone= http://localhost:8761/eureka/, http://localhost:8762/eureka/
```

And in Client application.properties:

eureka.client.service-url.defaultZone= http://eureka-1:8761/eureka, http://eureka-2:8762/eureka, http://eureka-3:8763/eureka

Now even if 1 server is down, it will not impact the client availability.

4. It can cause latency issue, as each call now required 2 hops, first it has to invoke Eureka Server and then the actual call.

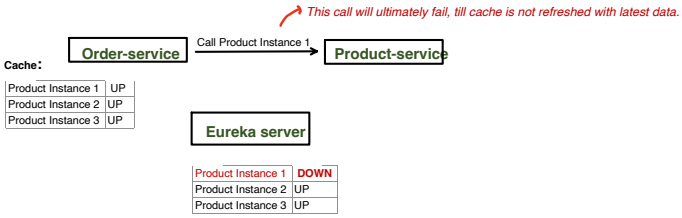
- Eureka Server does not get called for every request.
- At startup, Client say (order-service) fetch the registry
  eureka.client.fetch-registry=true
- And Cache it locally, all future request, uses this local copy to find the instance.

we can control this with config like below, after every 30 seconds client will refresh its cache copy:

eureka.client.registry-fetch-interval-seconds=30

5. What if the local cache is stale? Can this lead to calling a dead instance?

It's a valid scenario and you can say it's a trade off.



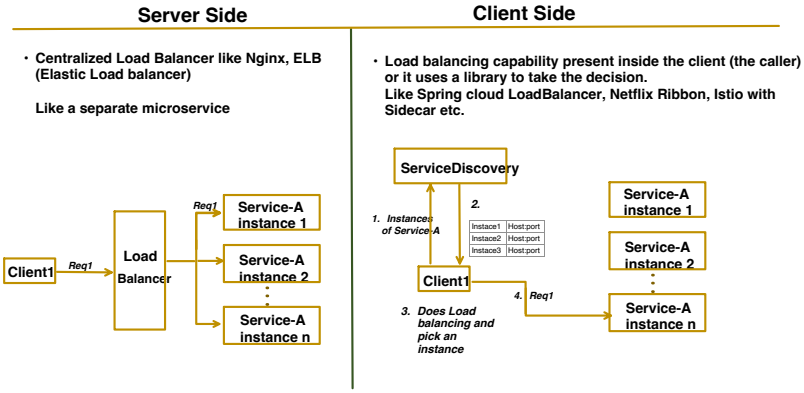
Client Side Load Balancer

Wednesday, 18 June 2025 3:37PM

Load Balancer:

- Helps in distributing traffic to multiple instances of a servers.
- Helps in preventing single server from being overloaded with huge traffic.

Load Balancer Types:



Client Side Load Balancer:

Pre-requisite :

Let's see Client Side Load Balancing with RestTemplate:

In Service Discovery topic, we have seen that:

- After setting up the Service Discovery.
- In RestTemplate, we fetch the List of Instances using DiscoveryClient object.
- Then client is manually choosing the instances from the list and invoking the endpoint.

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    DiscoveryClient discoveryClient;

    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {

        List<ServiceInstance> instances = discoveryClient.getInstances( serviceId: "product-service");
        URI uri = instances.get(0).getUri();

        String response = restTemplate.getForObject( url: uri + "/products/" + id, String.class);

        System.out.println("Response from Product api call is: " + response);
    }
}
```

```java
@Configuration
public class Config {

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

Now, lets use "Spring Cloud Load Balancer" and see the above example again:

1. Add dependency

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

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    RestTemplate restTemplate;

    @Value("${product.service.baseurl}")
    String productBaseURL;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {

        String response = restTemplate.getForObject( url: productBaseURL + "/products/" + id, String.class);
        System.out.println("Response from Product api call is: " + response);
    }
}
```

Now, we don't need DiscoveryClient here.

```java
@Configuration
public class Config {

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

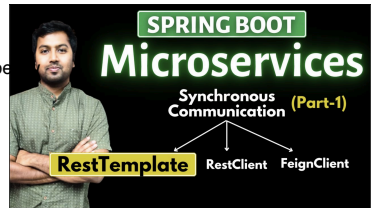
With this annotation framework start intercepting the request, call Service Discovery, get instances & apply load balancing algorithm.

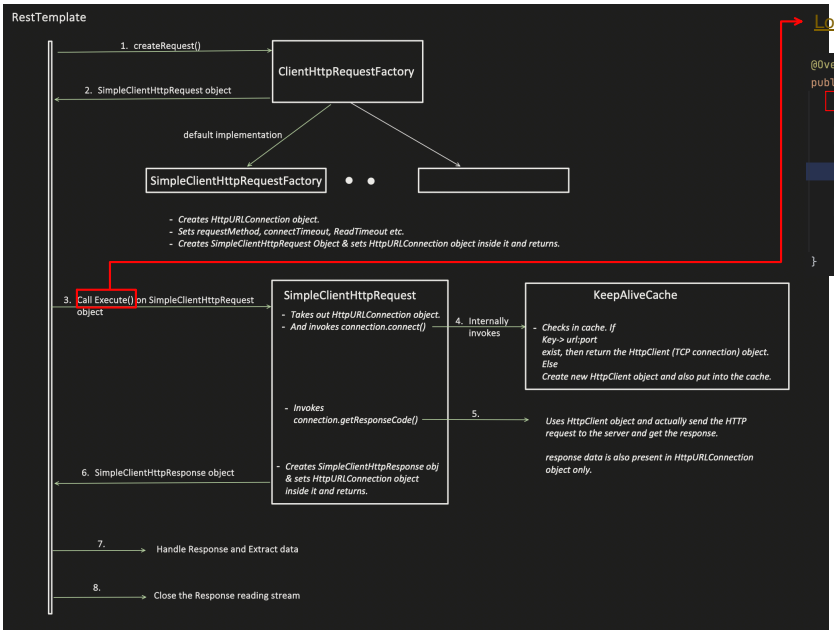
application.properties

```
server.port=8081

spring.application.name=order-service

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

product.service.baseurl = http://product-service
```

Now we are simply using the product service name.

In RestTemplate topic, we have already seen be
diagram and understood it in-depth:





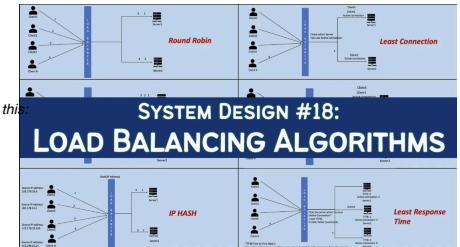
LoadBalancerInterceptor.java class get invokes first

```java
@Override
public <T> ServiceInstance choose(String serviceId, Request<T> request) {
    ReactiveLoadBalancer<ServiceInstance> loadBalancer = loadBalancerClientFactory.getInstance(serviceId);
    if (loadBalancer == null) {
        return null;
    }
    Response<ServiceInstance> loadBalancerResponse = Mono.from(loadBalancer.choose(request)).block();
    if (loadBalancerResponse == null) {
        return null;
    }
    return loadBalancerResponse.getServer();
}
```

1. Picks the load balancing algorithm

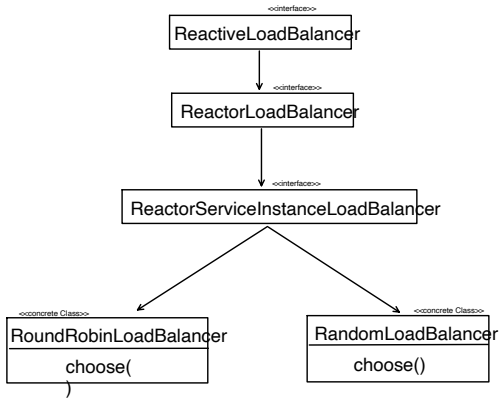
2. Internally calls Service Discovery and picks one instance based on above load balancing algorithm.

In HLD playlist, I have already discussed different types of Load balancer algorithm in depth, have a look, if there is any doubt with this.



`ReactiveLoadBalancer<ServiceInstance> loadBalancer = loadBalancerClientFactory.getInstance(serviceId);` selects load balancer algorithm for specific SERVICE ID

- Each Load Balancing Algorithm is attached to a ServiceId. ServiceId is nothing but a client name like : "*product-service*".
- By default Spring Cloud Load Balancer picks RoundRobinLoadBalancer algorithm.
- Means if we are not going with Default Algorithm, then we need to define for which client (or serviceId) which algorithm we need.
- Spring Cloud Load Balancer supports only 2 algorithms i.e. RoundRobin and Random.
- If we want different algorithm like Weighted, Least Connection etc. then either we need to write custom implementation or switch to service mesh like Istio with Side car.



If we want to use say RandomLoadBalancer instead of Default RoundRobinLoadBalancer, we can override it in configuration

(Same as previous)

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    RestTemplate restTemplate;

    @Value("${product.service.baseurl}")
    String productBaseURL;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {

        String response = restTemplate.getForObject( url: productBaseURL + "/products/" + id, String.class);
        System.out.println("Response from Product api call is: " + response);
    }
}
```

(Same as previous)

```java
@Configuration
public class Config {

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

application.properties (Same as previous)

```
server.port=8081

spring.application.name=order-service

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

product.service.baseurl = http://product-service
```

Now Changing the Load Balancing algorithm for Product Service from default RoundRobinLoadBalancer to RandomLoadBalancer.

```java
@SpringBootApplication
[redacted]
@LoadBalancerClient(name = "product-service", configuration = LoadBalancerProductClientConfig.class)
public class OrderserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderserviceApplication.class, args);
    }
}
```

Normal Config class(like above) generally invokes only at application startup.
But this LoadBalancerClient config invokes at runtime too, when "product-service" is used: "http://product-service/xyz"

```java
@Configuration
public class LoadBalancerProductClientConfig {

    @Bean
    public ReactorLoadBalancer<ServiceInstance> productClientLoadBalancer(
            LoadBalancerClientFactory factory) {

        return new RandomLoadBalancer(
                factory.getLazyProvider( name: "product-service", ServiceInstanceListSupplier.class),
                serviceId: "product-service");
    }
}
```

ServiceInstanceListSupplier, is actually the one, which helps to invoke Service Discovery and ask for instances of "product-service"

New RandomLoadBalancer(ServiceInstanceListSupplier, serviceId)
we pass serviceId because it mapped this algorithm to this service ID only.
So for "product-service" only RandomLoadBalancer is used.

What if, for "product-service" I want Algorithm A and for all other clients (or ServiceId), I want Algorithm B

```java
@SpringBootApplication
[redacted]
@LoadBalancerClients(defaultConfiguration= LoadBalancerGlobalConfig.class,
        value = {
        @LoadBalancerClient(name = "product-service", configuration = LoadBalancerProductClientConfig.class)
        })
public class OrderserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderserviceApplication.class, args);
    }
}
```

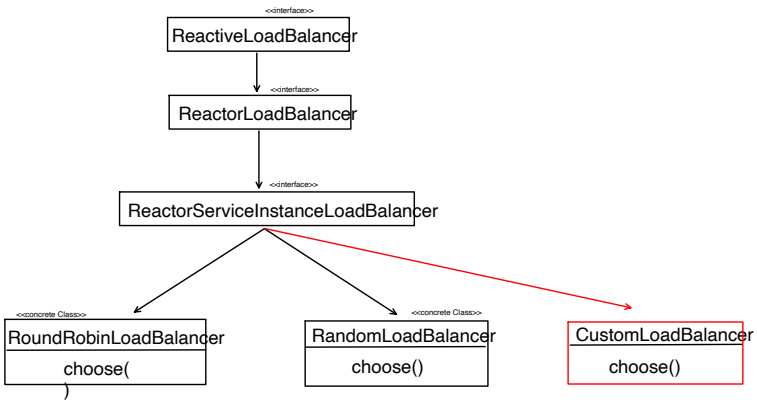
First child clients config will be loaded, then default one.

- As we already know, this *LoadBalancerClients* config is loaded at runtime for each service ID, So this *environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME)* will resolved at runtime.
- *@ConditionalOnMissingBean* is required at Global config class, because say for *"product-service"* two ReactorLoadBalancer bean should not get created one from *LoadBalancerProductClientConfig* and another from *LoadBalancerGlobalConfig*.
- So below, for "product-service" Random Load balancer algorithm will be used and for all other services "RoundRobin Algorithm will be used.

```java
@Configuration
@ConditionalOnMissingBean(ReactorLoadBalancer.class)
public class LoadBalancerGlobalConfig {

    @Bean
    public ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(
            LoadBalancerClientFactory factory, Environment environment) {

        String serviceId = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new RoundRobinLoadBalancer(
                factory.getLazyProvider(serviceId, ServiceInstanceListSupplier.class),
                serviceId
        );
    }
}
```

```java
@Configuration
public class LoadBalancerProductClientConfig {

    @Bean
    public ReactorLoadBalancer<ServiceInstance> productClientLoadBalancer(
            LoadBalancerClientFactory factory) {

        return new RandomLoadBalancer(
                factory.getLazyProvider( name: "product-service", ServiceInstanceListSupplier.class),
                serviceId: "product-service");
    }
}
```

If we want to use say our CustomLoadBalancer logic:




```java
public class MyCustomLoadBalancer implements ReactorServiceInstanceLoadBalancer {

    private final ObjectProvider<ServiceInstanceListSupplier> serviceInstanceSuppliers;
    private final String serviceId;

    public MyCustomLoadBalancer(ObjectProvider<ServiceInstanceListSupplier> serviceInstanceSuppliers,
                                String serviceId) {
        this.serviceInstanceSuppliers = serviceInstanceSuppliers;
        this.serviceId = serviceId;
    }

    @Override
    public Mono<Response<ServiceInstance>> choose(Request request) {
        return serviceInstanceSuppliers.getIfAvailable().get().next().map(instances -> {
            if (instances == null || instances.isEmpty()) {
                return new EmptyResponse();
            }
            //Custom load balancing algorithm, i am just picking first instance for demo
            return new DefaultResponse(instances.get(0));
        });
    }
}
```

```java
@Configuration
public class LoadBalancerProductClientConfig {

    @Bean
    public ReactorLoadBalancer<ServiceInstance> productClientLoadBalancer(
            LoadBalancerClientFactory factory) {

        return new MyCustomLoadBalancer(
                factory.getLazyProvider( name: "product-service", ServiceInstanceListSupplier.class),
                serviceId: "product-service");
    }
}
```

During Service Discovery, we have already seen how FeignClient handles load balancer automatically.

So for load balancer everything is same like RestTemplate, no change at all.

Using FeignClient

So with FeignClient, load balancing is handled automatically and by the framework.

So we need to provide the Load Balancer dependency too, apart from *"spring-cloud-starter-netflix-eureka-client"*

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

Earlier without Service Discovery:

We need to provide the URL

```
@FeignClient(name = "product-service",
        url = "${feign.client.product-service.url}")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

With Service Discovery:

No need of the URL, only the registered name of the product service is required.
and
Load Balancing will also be handled internally by the framework.

```
@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

```
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    ProductClient productClient;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {

        String response = productClient.getProductById(id);
        System.out.println("Response from Product api call is: " + response);
    }
}
```

```
@SpringBootApplication
@EnableFeignClients
@LoadBalancerClient(name = "product-service", configuration = LoadBalancerProductClientConfig.class)
public class OrderserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderserviceApplication.class, args);
    }
}
```

```
@Configuration
public class LoadBalancerProductClientConfig {

    @Bean
    public ReactorLoadBalancer<ServiceInstance> productClientLoadBalancer(
            LoadBalancerClientFactory factory) {

        return new RandomLoadBalancer(
                factory.getLazyProvider( name: "product-service", ServiceInstanceListSupplier.class),
                serviceId: "product-service");
    }
}
```

Resilience4j: Building Fault-Tolerant Microservice

Saturday, 12 July 2025 3:05 PM

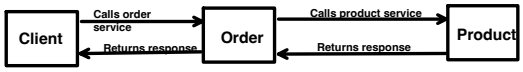
What is Fault-Tolerant Microservice?

- Fault Tolerant Microservice is a service, which continues to work even when downstream system fails.
- Instead of crashing or Cascading failures, it handles the failure gracefully.

What its required?

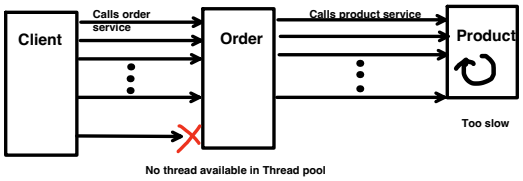
- As in microservices, different services communicate over the network.
- And its possible that 1 service unavailability could bring down the whole system.

Normal Scenario:



Lets say because of either buggy code or Infrastructure issue "Product" service becomes too slow:

- Say, product service become slow and taking 60 seconds to respond.
- If proper fault tolerant is not applied, then a call from Order service will hang for 60sec.
- Thus, a thread is blocked for that period.



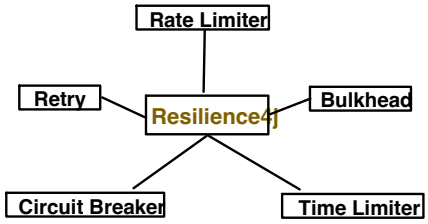
- if sudden burst of request comes from Order to Product service, all threads will now get blocked (waiting for response).
- And because of this, clients of Order service also get blocked.
- Eventually, services will run out of threads and start rejecting the request.

This is know as CASCADING FAILURE.

means, issue is at Product service, but now propagated to other services too.

To avoid this, we need to build Fault tolerant microservice.

And here comes:



Lets go in depth one by one in recommended logical ordering:

RateLimiter → Bulkhead → TimeLimiter → CircuitBreaker → Retry

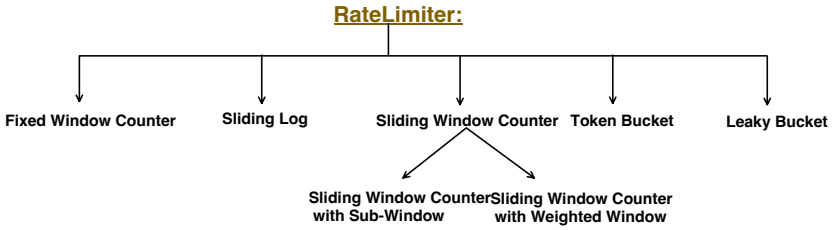
Why Ordering required?

For example: Always apply RateLimiter before Retry, so that retry logic doesn't overwhelm the system with already limited traffic.
Else we end up in retry the traffic, which will get blocked by RateLimiter later.

RateLimiter:

It controls the number of requests allowed to a microservice in a given time window.

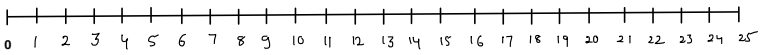
So in other words, it protect our system from sudden traffic spike (like in DDOS attack).



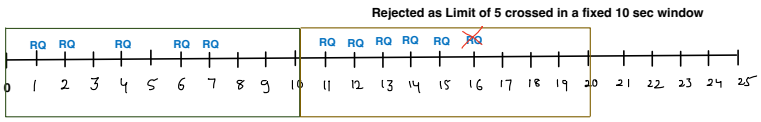
Fixed Window Counter:

- **Counts how many request happens in a fixed window.**
- **If the count crossed the Limit, then reject the request.**

For example:
Limit = 5
Fixed Window Size = 10 second



Time period in seconds



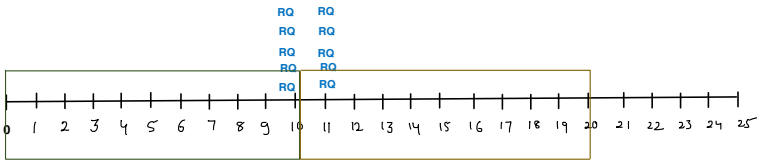
Time period in seconds

Window1 (0-10 sec)
Limit = 5

Window2 (10-20sec)
Limit = 5

Disadvantage:

- **If traffic comes at the edge (end edge of 1st window and starting edge of 2nd window), then there is a chance of sudden spike.**
  like below even though Limit = 5 in 10second, but here 10 requests comes in just 1 second.



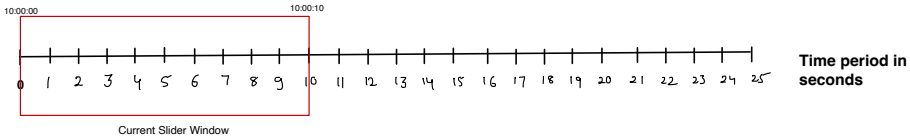
Time period in seconds

Window1 (0-10 sec)
Limit = 5

Window2 (10-20sec)
Limit = 5

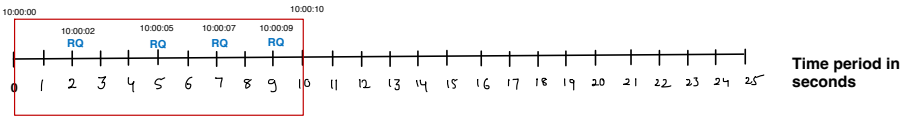
Sliding Log:

- For each accepted request, it stores the exact timestamp.
- For each request comes in, it first check, how many request happened in last X seconds (Window size).
- If no of requests reach the LIMIT, it will reject the request.



Time period in seconds

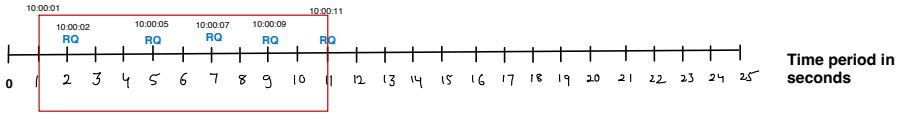
Window Size = 10 seconds
Limit = 5



Time period in seconds

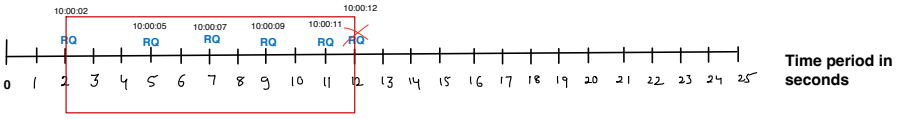
Window Size = 10 seconds
Limit = 5

Now, when window size crossed, slider moves on.
So it consider only those request in count, which are in slider window only.



Time period in seconds

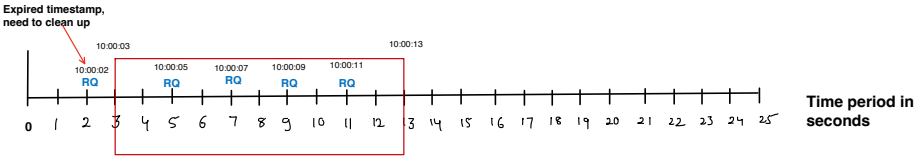
Window Size = 10 seconds
Limit = 5



Time period in seconds

Window Size = 10 seconds
Limit = 5

Now, new request coming, but Total no of request in window is already 5. Limit is reached so new RQ will be rejected.



Time period in seconds

Window Size = 10 seconds
Limit = 5

Now, if new request comes and in the current window only 4 requests are there, so new request will be accepted.

Disadvantage:

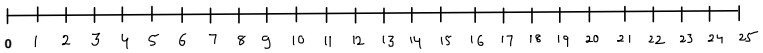
- Need to maintain timeframe info for each accepted request. Which needs more memory.
- Extra overhead of clean up of expired timestamps.

Sliding Window Counter:

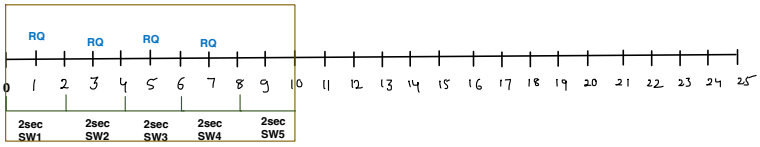
Sliding Window Counter with Sub-Window:

- **Main window is divided into equal sized sub-windows.**
- **Keep Tracks of the number of request in each sub-window.**
- **To decide if request need to be rejected or not, it consider only active sub-windows.**
- **Main window slides(by sub-window size) over time.**

Window Size = 10 seconds
Sub-Window = 2 seconds
Limit = 5



Time period in seconds



Time period in seconds

Main Window (10 sec)
Sub window=2sec
Limit = 5

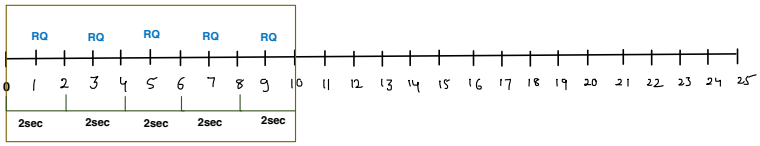
Say for new Request RQ coming at 9th sec time period:

It will sum up the requests in all active Sub-windows:

SW1(0sec-2sec) + SW2(2sec-4sec) + SW3(4sec-6sec) + SW4(6sec-8sec) + SW5 (8sec-10sec)

1 + 1 + 1 + 1 + 0 = 4

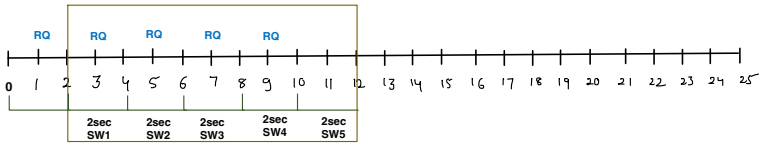
So its less than Limit 5, so RQ coming at 9th sec time period will be allowed



Time period in seconds

Main Window (10 sec)
Sub window=2sec
Limit = 5

Now when time crosses 10th second, main window has to slide. So it Slide(jump) according to sub-window size.



Time period in seconds

Main Window (10 sec)
Sub window=2sec
Limit = 5

Say for new Request RQ coming at 11th sec time period:

It will sum up the requests in all active Sub-windows only:

SW1(2sec-4sec) + SW2(4sec-6sec) + SW3(6sec-8sec) + SW4(8sec-10sec) + SW5 (10sec-12sec)

1 + 1 + 1 + 1 + 0 = 4

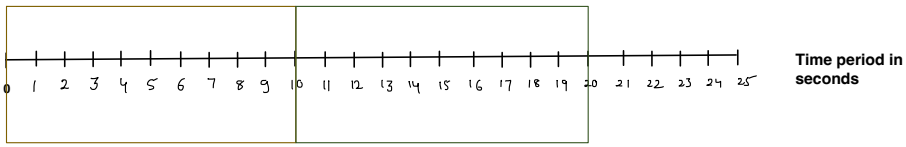
So its less than Limit 5, so RQ coming at 11th sec time period will be allowed

Disadvantage:

- More complex than Fixed window logic.
- Need to maintain Sub window info and request count per sub window. Which needs more memory.

Sliding Window Counter with Weighted Window:

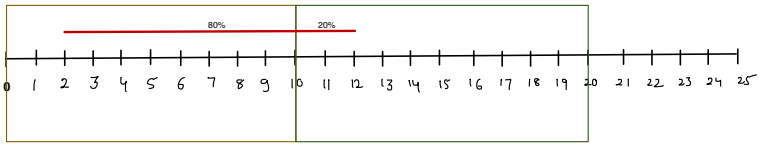
- It tries to simplify the logic by removing Sub-window granularity and bringing some approximation.
- It combined Fixed Window + Sliding Log Concept.



Time period in seconds

Fixed Window (0-10 sec) Fixed Window (10-20 sec)

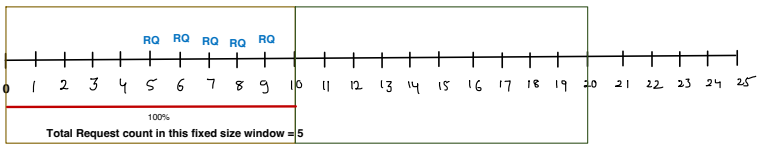
Slider window (equal to size of Fixed Window, say 10 sec) can be between two Fixed Size Window and uses % to determine, how much its in previous window and how much in current window and determine the traffic based on that.



Time period in seconds

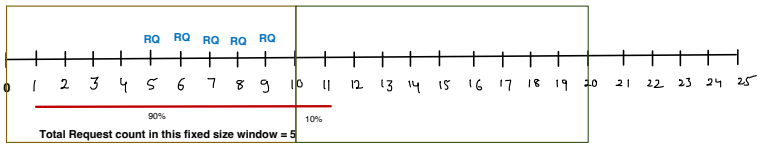
Fixed Window (0-10 sec) Fixed Window (10-20 sec)

Say, Limit = 5 in a slide window.



Time period in seconds

Fixed Window (0-10 sec) Fixed Window (10-20 sec)



Time period in seconds

Fixed Window (0-10 sec) Fixed Window (10-20 sec)

Now lets say RQ is coming at 11th second of time period.

Now, it uses a formula to compute, how much request from each window it should consider.

Total Requests in Current Fixed Window + 90% of Total Request in previous window (i.e. 5)
(using 90% based on above example shown, it can be different)

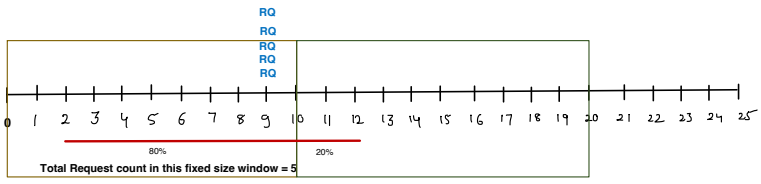
Total Requests in Current Fixed Window (10-20sec) = 0
90% of Total Request in previous window (i.e. 5) = 4.5 say we Ceil/round it to ~5

0 + ~5 = 5

Limit is reached of the slider window so Request is Rejected.

Disadvantage:

- It assume uniform distribution of traffic, but that might not be true sometimes.



Time period in seconds

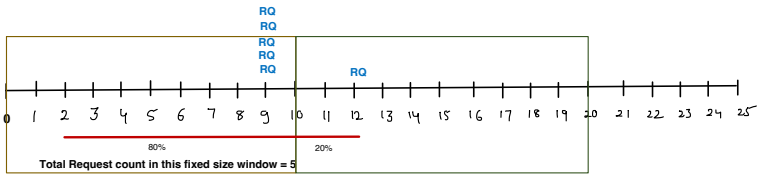
Fixed Window (0-10 sec) Fixed Window (10-20 sec)

Now, new RQ coming at 12th second

Total Requests in Current Fixed Window + 80% of Total Request in previous window (i.e. 5)
(using 80% based on above example shown, it can be different)

0 + 4 = 4

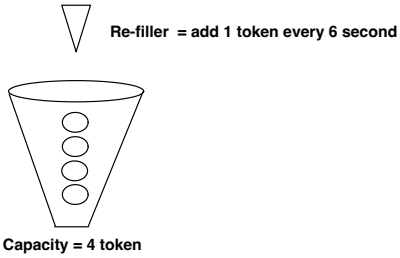
4 is less than Limit 5, so request is permitted

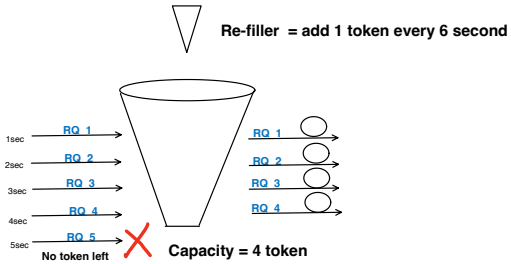


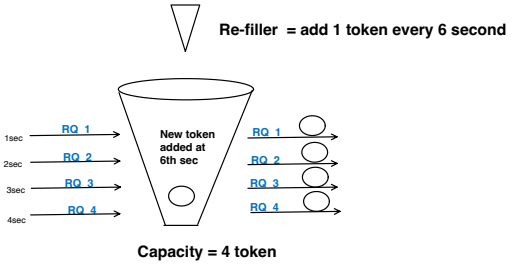
Fixed Window (0-10 sec) **Fixed Window (10-20 sec)**

Token Bucket:

- **Limited Tokens are placed in a bucket.**
- **Each request consume one token.**
- **If there is no token left, request is denied.**
- **There is a re-filler, which adds the token in bucket at regular time interval.**
- **If Bucket capacity is full and can not hold more token, token overflow will happen and token will be rejected.**







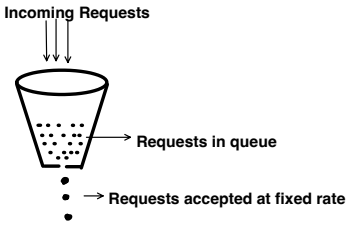
Disadvantage:

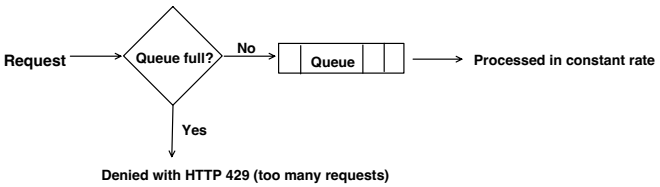
- **A quick burst is possible if token capacity is not set properly.**
  Say system remains idle for some time and token keep on accumulating, and when sudden burst comes, it will be allowed as token is present.

Leaky Bucket:

- **Requests are Queued and accepted at fixed rate.**

- **If queue is full, incoming request will overflow.**





Disadvantage:

- **Latency increases, as request might have to wait in queue for its turn.**
- **Queue size is a bottle neck. So size should be properly set.**

Implementation in Spring boot:

Resilience4j by-default applies **token bucket algorithm**

Pom.xml dependency

```xml
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
    <version>2.1.0</version>
</dependency>
```

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    OrderService orderService;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {
        orderService.invokeProductAPI(id);

    }
}
```

```java
@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

Internally it uses AOP functionality, that's why the method on which this @RateLimiter annotation applies, need to be public and Spring managed bean.

```java
@Component
public class OrderService {

    @Autowired
    ProductClient productClient;

    @RateLimiter(name = "productRateLimiter", fallbackMethod = "rateLimitedFallback")
    public void invokeProductAPI(String id) {
        String response = productClient.getProductById(id);
        System.out.println("Response from Product api call is: " + response);
    }

    public void rateLimitedFallback( String id, Throwable t) {
        System.out.println("Rate limit exceeded. Try later");
        //throw exception here and handle it gracefully
    }
}
```

application.properties

```properties
server.port=8081
spring.application.name=order-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#bucket filled with 2 tokens every 10second,
#if there is no token wait for 1sec before rejecting the request
resilience4j.ratelimiter.instances.productRateLimiter.limitForPeriod=2
resilience4j.ratelimiter.instances.productRateLimiter.limitRefreshPeriod=10s
resilience4j.ratelimiter.instances.productRateLimiter.timeoutDuration=1s
```

Fallback method, return type and parameter(additionally only Throwable need to add more) should match with the original method, else default fallback method provided by RateLimiter framework will get invoked.

Output:

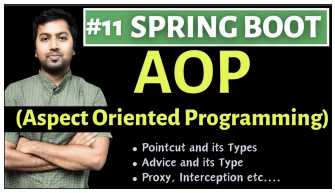
when frequently tried to hit the API, after frequent 2 calls, tokens finished and for next call, got the Rate limiting message. And after waiting for 10seconds, again token is available and able to hit the api again.

```
Response from Product api call is: fetch the product details with id:1
Response from Product api call is: fetch the product details with id:1
Rate limit exceeded. Try later
Response from Product api call is: fetch the product details with id:1
Response from Product api call is: fetch the product details with id:1
Rate limit exceeded. Try later
Response from Product api call is: fetch the product details with id:1
Response from Product api call is: fetch the product details with id:1
Rate limit exceeded. Try later
```

In similar way, @RateLimiter annotation can be used, if we are using RestTemplate or RestClient. No change in that.

If want to write custom rate limiting logic, we can easily do it, using AOP functionality

Explained AOP in depth in this:




```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface CustomRateLimiter {
    int limit();              // max requests
    int windowInSeconds();    // sliding window duration
}
```

```java
@Aspect
@Component
public class RateLimiterAspect {


    @Around("@annotation(customRateLimiter)")
    public Object rateLimit(ProceedingJoinPoint pjp,
                            CustomRateLimiter customRateLimit) throws Throwable {
        //custom rate limiting logic, if request accepted, invoke the method
        return pjp.proceed();
    }

}
```

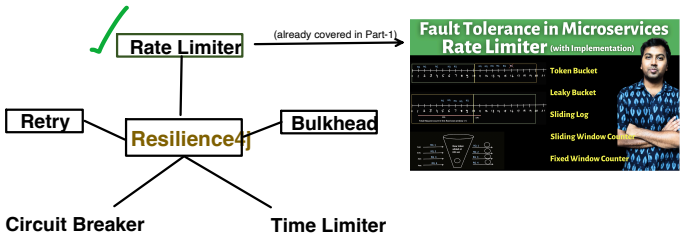
```java
@CustomRateLimiter(limit = 5, windowInSeconds = 60)
public String getProducts() {
    //service call
    return "List of products";
}
```

Next will cover: Bulkhead, TimeLimiter, Circuit breaker and Retry

Bulkhead: Fault-Tolerant Microservice (Part-2)

Wednesday, 16 July 2025 5:50 AM

To build Fault tolerant microservices: Resilience4j provides below mechanisms



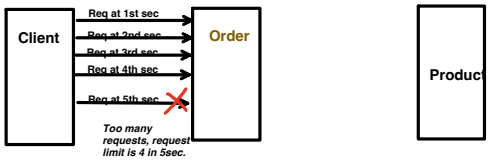
Lets see in depth of:

Bulkhead:

Rate Limiter:

- Protects **our application from our clients** by limiting how many requests we accept within a specific time window from them.
- also Rate limiter never talks about the concurrency.

Ex: let say our application is "Order".

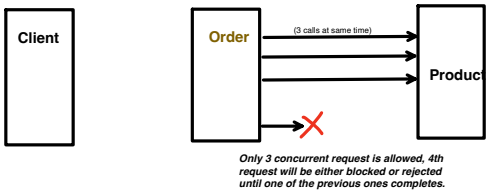


Bulkhead:

- It helps to control **how many concurrent requests can go to downstream service**.

Ex: let say our application is "Order"

Use-Case: Let say, "Product" is small and light weight service and can only handle 3 concurrent request.



Above use case is resolved through "Semaphore Bulkhead"

- It also protects **our application from our downstream services** by limiting how many threads we allocate to them.

Use-Case: Let say our Order Service has 2 APIs (endpoints)

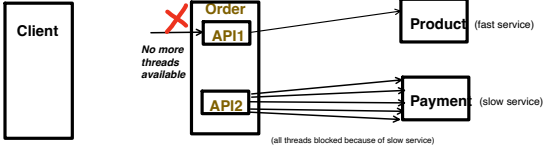
***API 1** : make a downstream call to Product service.*
***API 2** : make a downstream call to Payment service.*

Product service takes ~100 milliseconds to process a request (Fast Service)
and
Payment service takes ~5second to process a request (Slow service)

*What if there is a sudden spike in **API 2** traffic and more and more calls are made to Payment service (say calls are not concurrent).*

Since Payment service is slow, it holds or blocks the thread longer(for ~5seconds) and its possible that all the threads of Order service now blocked (by Payment service only).

*Now even the Order service fast endpoint **(API 1)** becomes unavailable as there is no threads available.*



Above use case is resolved through "Thread Pool Bulkhead"

Bulkhead

- **Semaphore Bulkhead**
- **Thread Pool Bulkhead**

Semaphore Bulkhead:

- **Limits the number of concurrent calls using counter.**
- **If the limit is reached, further calls are rejected immediately or blocked for specific wait time.**

it limits the number of concurrency call, via Semaphore Lock. I have already explained in depth about Semaphores in Multithreading topic of Java playlist. Have a look if there is any doubt with Semaphore locks.

Pom.xml dependency

```xml
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
    <version>2.1.0</version>
</dependency>
```

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    OrderService orderService;

    @GetMapping("/{id}")
    public void callProductAPI(@PathVariable String id) {
        orderService.invokeProductAPI(id);

    }
}
```

```java
@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping(value = "/products/{id}")
    String getProductById(@PathVariable("id") String id);
}
```

```java
@Component
public class OrderService {

    @Autowired
    ProductClient productClient;

    @Bulkhead(name = "productService", type = Bulkhead.Type.SEMAPHORE, fallbackMethod = "productFallback")
    public void invokeProductAPI(String id) {
        String response = productClient.getProductById(id);
        System.out.println("Response from Product api call is: " + response);
    }

    public void productFallback( String id, Throwable t) {
        System.out.println("too many concurrent request, pls try again later");
        //throw exception here and handle it gracefully
    }
}
```

application.properties

```properties
server.port=8081
spring.application.name=order-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#only 2 concurrent calls allowed and
#for further req, if there is no thread available, request will fail immediately
resilience4j.bulkhead.instances.productService.maxConcurrentCalls=2
resilience4j.bulkhead.instances.productService.maxWaitDuration=0
```

Accepted values for maxWaitDuration: It accepts values in the following format:

<number><time-unit>

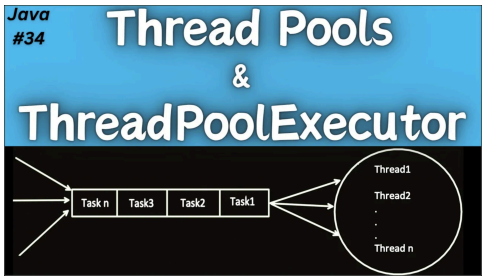
Supported time units:
- **ms** — milliseconds
- **s** — seconds
- **m** — minutes
- **h** — hours
- **d** — days (not typical, but valid)

Example:
resilience4j.bulkhead.instances.productService.maxWaitDuration=0 # Reject immediately
resilience4j.bulkhead.instances.productService.maxWaitDuration=300ms # Wait 300 milliseconds
resilience4j.bulkhead.instances.productService.maxWaitDuration=2s # Wait 2 seconds
resilience4j.bulkhead.instances.productService.maxWaitDuration=1m # Wait 1 minute

Thread Pool Bulkhead:

- **Assign a dedicated thread pool for each service.**
- **Only that pool is used to call the downstream.**

If you have any doubts, how Thread Pools and Thread Pool Executor works, kindly have a look at this video in Java playlist, I have already discussed in depth.



- *The AOP proxy intercepts your method call.*
- *It **submits your method to the Bulkhead's thread pool** (configured via application.properties).*

CompletableFuture.supplyAsync(() -> {
return ourMethodLogic(); // The whole method body runs here
}, bulkheadThreadPoolExecutor);

- *It returns a **CompletableFuture** to the caller for async response.*
- *Inside your method, CompletableFuture.completedFuture(...) just **wraps the result** in a completed future — it **does NOT run anything in a thread pool**.*

```java
@Component
public class OrderService {

    @Autowired
    ProductClient productClient;

    @Bulkhead(name = "productService", type = Bulkhead.Type.THREADPOOL, fallbackMethod = "productFallback")
    public CompletableFuture<String> invokeProductAPI(String id) {
        return CompletableFuture.completedFuture(productClient.getProductById(id));
    }

    public CompletableFuture<String> productFallback(String id, Throwable t) {
        //even though its async, we need to properly handle the exception
        //else our client will get 500 response with BulkheadFullException
        return CompletableFuture.completedFuture( value: "Product Service is busy");
    }
}
```

application.properties

```
server.port=8081
spring.application.name=order-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#max 3 threads in the pool and that's the max pool size too.
#if all 3 threads are busy, max 2 more requests will be put into queue
#from 6th request onwards, it will start rejecting the requests.
resilience4j.thread-pool-bulkhead.instances.productService.coreThreadPoolSize=3
resilience4j.thread-pool-bulkhead.instances.productService.maxThreadPoolSize=3
resilience4j.thread-pool-bulkhead.instances.productService.queueCapacity=2
```

ThreadPool for productService will be created with these configurations.

Testing the Output:

Just added System.out.println for testing purpose.

```java
@Component
public class OrderService {

    @Autowired
    ProductClient productClient;

    @Bulkhead(name = "productService", type = Bulkhead.Type.THREADPOOL, fallbackMethod = "productFallback")
    public CompletableFuture<String> invokeProductAPI(String id) {
        System.out.println("Thread name is:" + Thread.currentThread().getName());
        return CompletableFuture.completedFuture(productClient.getProductById(id));
    }

    public CompletableFuture<String> productFallback(String id, Throwable t) {
        //even though its async, we need to properly handle the exception
        //else our client will get 500 response with BulkheadFullException
        System.out.println("Product Service is busy");
        return CompletableFuture.completedFuture( value: "Product Service is busy");
    }
}
```

Output:

Max thread pool size is = 3 and Queue size is = 2

Invoked the API 6 times, lets see the output:

```
2025-07-17T10:15:08.998+05:30  INFO 82435 --- [order-service] [           main] c.c.o.OrderserviceApplication            : Started OrderserviceApplication in 1.485
2025-07-17T10:20:08.830+05:30  INFO 82435 --- [order-service] [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints via configura
2025-07-17T10:20:09.593+05:30  INFO 82435 --- [order-service] [nio-8081-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'd
2025-07-17T10:20:09.593+05:30  INFO 82435 --- [order-service] [nio-8081-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2025-07-17T10:20:09.594+05:30  INFO 82435 --- [order-service] [nio-8081-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Thread name is:bulkhead-productService-1  ——  Thread1, RQ1
Thread name is:bulkhead-productService-2  ——  Thread2, RQ2
Thread name is:bulkhead-productService-3  ——  Thread3, RQ3 and RQ4 and RQ5 is inserted into the queue
Product Service is busy   RQ6 is rejected, as there is no thread available and no space in queue
Thread name is:bulkhead-productService-1  ——  Thread1 becomes free and picked RQ4 from queue
Thread name is:bulkhead-productService-2  ——  Thread2 becomes free and picked RQ5 from queue
```

Time Limiter:

- **Time Limiter is used to prevent async call from hanging indefinitely.**
- **Time Limiter is non blocking in Resilience4j.**
- **Means, it is mainly designed for asynchronous operations, that returns a reactive type like Mono, Flux etc.**

But currently we have covered Blocking calls like RestTemplate, RestClient and FeignClient.
And we have already covered how to control the time out of the blocking calls.

Feign Client example:

application.properties

```
#request and connection timeout applicable to only Product-service FeignClient
feign.client.config.product-service.connectTimeout=3000
feign.client.config.product-service.readTimeout=5000
```

Time Limiter context is same but for reactive calls. Will cover TimeLimiter, when will start Spring Web flux (reactive programming)