Rate Limiter Pattern – Microservice Design Patterns

**Overview:**

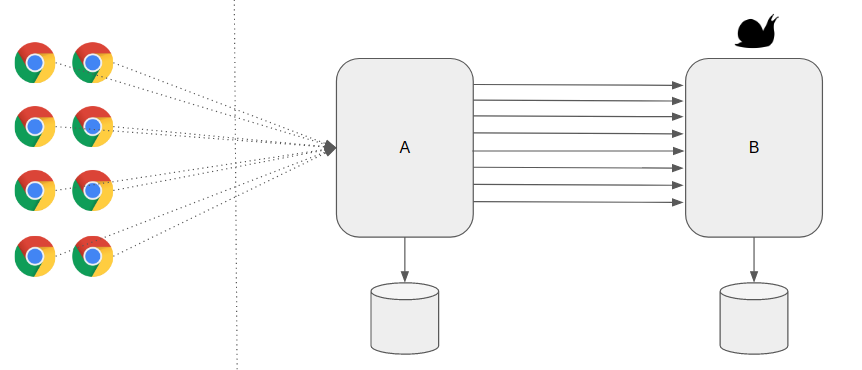
**Need For Resiliency:**

Microservices are distributed in nature. When you work with distributed systems, always remember this number one rule – ***anything could happen***. We might be dealing with network issues, service unavailability, application slowness etc. An issue with one system might affect another system behavior/performance. Dealing with any such unexpected failures/network issues could be difficult to solve.

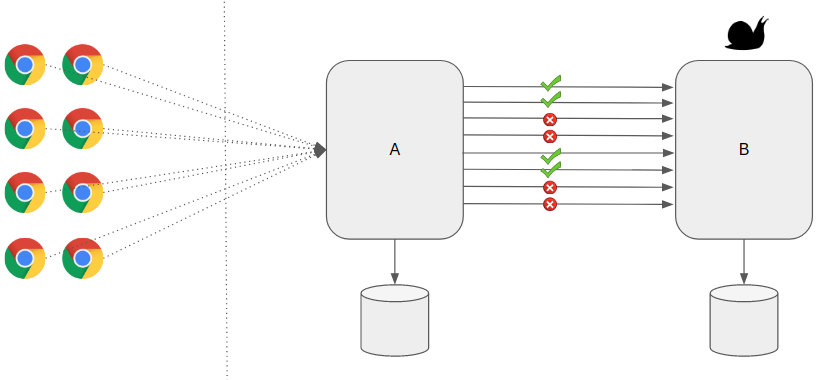
Ability of the system to recover from such failures and remain functional makes the system more ***resilient***. It also avoids any cascading failures to the downstream services.

**Rate Limiter Pattern:**

In Microservice architecture, when there are multiple services (A, B, C & D), one service (A) might depend on the other service (B) which in turn might depend on C and so on. Let’s consider this example. We have 2 services A and B.  Service A depends on Service B. The Service B has to do a lot of ***CPU/IO intensive work*** for the requests it receives. So it usually takes time to respond because of the nature of its work. Service B has a limit on max number of requests it can handle within the given time window.



Now the problem is – Service A receives a lot of requests occasionally and for every request if we depend on Service B, It could add too much load on Service B which could bring the service down. As a defensive measure, Service B wanted to protect itself from receiving too many requests by rejecting calls it can not handle.



***Rate Limiter Pattern*** helps us to make our services highly available just by limiting the number of calls we could make/process in a specific window. In other words, It helps us to control the throughput. When we receive too many requests, the Service might simply reject the call. The client has to retry at a later time or can go with some default/cached values.

**Rate Limiter vs Circuit Breaker:**

Rate Limiter Pattern might sound same as [Circuit Breaker](https://www.vinsguru.com/circuit-breaker-pattern/) in some cases. However there is an important difference!

* Rate Limiter helps to protect the server from over loading by controlling throughput.
* Circuit Breaker helps to keep the client safe and functional when the target server is failing / unresponsive.

**Sample Application:**

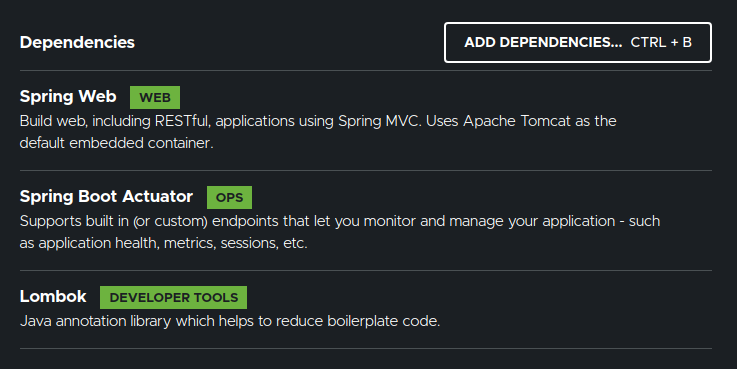
Lets consider a simple ***compute-service*** application which provides below endpoints.

* ***/double/{input}:*** doubles the given input. Unlimited calls allowed.
* ***/square/{input}:*** calculates the square of the given input. Limited calls only. Max 5 calls per minute.

Now Let’s see how to apply Rate Limiter Pattern in this Microservice design.

**Project Set Up:**

Lets first create a Spring Boot project with these dependencies.



We also need this dependency.

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

<version>1.6.1</version>

</dependency>

Copy

If the client wants to find the square of a given number, say 10, it expects the response in the below format from the ***compute-service***. The ***message*** field might contain the error message when the response type is FAILURE.

{

"input": 10,

"output": 100,

"responseType": "SUCCESS",

"message": ""

}

Copy

**Rate Limiter Pattern With Spring Boot:**

* Response Type enum

public enum ResponseType {

SUCCESS,

FAILURE;

}

Copy

* Compute Response DTO

@Data

@AllArgsConstructor(staticName = "of")

public class ComputeResponse {

private int input;

private long output;

private ResponseType responseType;

private String message;

}

Copy

* Rate Limiter with ***resilience4j***

Our compute service might have multiple endpoints. However we would like to limit the max number of calls for certain end points. For example, I would like to limit the calls for calculating squares to 5/minute.

resilience4j.ratelimiter:

instances:

squareLimit:

limitForPeriod: 5

limitRefreshPeriod: 60s

timeoutDuration: 0

Copy

* Controller
  + I apply the ***squareLimit*** configuration only to a specific endpoint.

@RestController

public class ComputeController {

@GetMapping("/double/{input}")

public ComputeResponse doubleValue(@PathVariable int input){

return ComputeResponse.of(input, 2\*input, ResponseType.SUCCESS, Strings.EMPTY);

}

@GetMapping("/square/{input}")

@RateLimiter(name = "squareLimit", fallbackMethod = "squareErrorResponse")

public ComputeResponse getValue(@PathVariable int input){

return ComputeResponse.of(input, input \* input, ResponseType.SUCCESS, Strings.EMPTY);

}

public ComputeResponse squareErrorResponse(int input, Throwable throwable){

return ComputeResponse.of(input, -1, ResponseType.FAILURE, throwable.getMessage());

}

}

Copy

**Demo:**

Our application is ready. Start the compute-service & test.

***Case 1:*** access the endpoint for double multiple times.

* Endpoint

http://localhost:8080/double/10

* Output

{

"input": 10,

"output": 20,

"responseType": "SUCCESS",

"message": ""

}

Copy

***Case 2:***access the endpoint for square multiple times.

* Endpoint

http://localhost:8080/square/10

* First 5 calls every 60 seconds

{

"input": 10,

"output": 100,

"responseType": "SUCCESS",

"message": ""

}

Copy

* Additional calls response

{

"input": 10,

"output": -1,

"responseType": "FAILURE",

"message": "RateLimiter 'squareLimit' does not permit further calls"

}

Copy

**Summary:**

***Rate Limiter Pattern*** is very useful in controlling throughput of a method call. So that server resources can be utilized properly & it is also useful to prevent the server from any malicious attacks.