# Health Check

## Context

You have applied the Microservice architecture pattern. Sometimes a service instance can be incapable of handling requests yet still be running. For example, it might have ran out of database connections. When this occurs, the monitoring system should generate a alert. Also, the load balancer or service registry should not route requests to the failed service instance.

## Problem

How to detect that a running service instance is unable to handle requests?

## Forces

- An alert should be generated when a service instance fails
- Requests should be routed to working service instances

## Solution

A service has an health check API endpoint (e.g. HTTP `/health`) that returns the health of the service. The API endpoint handler performs various checks, such as

- the status of the connections to the infrastructure services used by the service instance
- the status of the host, e.g. disk space
- application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

## Examples

In our application all the microservice are configured with **Actuator** for health check you can run the application locally or on server and access the health of the application by endpoint **/actuator/health.**

Add maven dependency of Actuator.

# Scaling Microservice

The book, The Art of Scalability, describes a really useful, three dimension scalability model: the scale cube.

Scaling an application by running clones behind a load balancer is known as X-axis scaling. The other two kinds of scaling are Y-axis scaling and Z-axis scaling. The microservice architecture is an application of Y-axis scaling but let's also look at X-axis and Z-axis scaling.

## X-axis scaling

X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.

One drawback of this approach is that because each copy potentially accesses all of the data, caches require more memory to be effective. Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.

## Y-axis scaling

Unlike X-axis and Z-axis, which consist of running multiple, identical copies of the application, Y-axis axis scaling splits the application into multiple, different services. Each service is responsible for one or more closely related functions. There are a couple of different ways of decomposing the application into services. One approach is to use verb-based decomposition and define services that implement a single use case such as checkout. The other option is to decompose the application by noun and create services responsible for all operations related to a particular entity such as customer management. An application might use a combination of verb-based and noun-based decomposition.

## Z-axis scaling

When using Z-axis scaling each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed. Another common routing criteria is the customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.

Z-axis splits are commonly used to scale databases. Data is partitioned (a.k.a. sharded) across a set of servers based on an attribute of each record. In this example, the primary key of the RESTAURANT table is used to partition the rows between two different database servers. Note that X-axis cloning might be applied to each partition by deploying one or more servers as replicas/slaves. Z-axis scaling can also be applied to applications. In this example, the search service consists of a number of partitions. A router sends each content item to the appropriate partition, where it is indexed and stored. A query aggregator sends each query to all of the partitions, and combines the results from each of them.

Z-axis scaling has a number of benefits.

- Each server only deals with a subset of the data.
- This improves cache utilization and reduces memory usage and I/O traffic.
- It also improves transaction scalability since requests are typically distributed across multiple servers.
- Also, Z-axis scaling improves fault isolation since a failure only makes part of the data in accessible.

Z-axis scaling has some drawbacks.

- One drawback is increased application complexity.
- We need to implement a partitioning scheme, which can be tricky especially if we ever need to repartition the data.
- Another drawback of Z-axis scaling is that doesn’t solve the problems of increasing development and application complexity. To solve those problems we need to apply Y-axis scaling.

# Pattern: API Composition

## Context

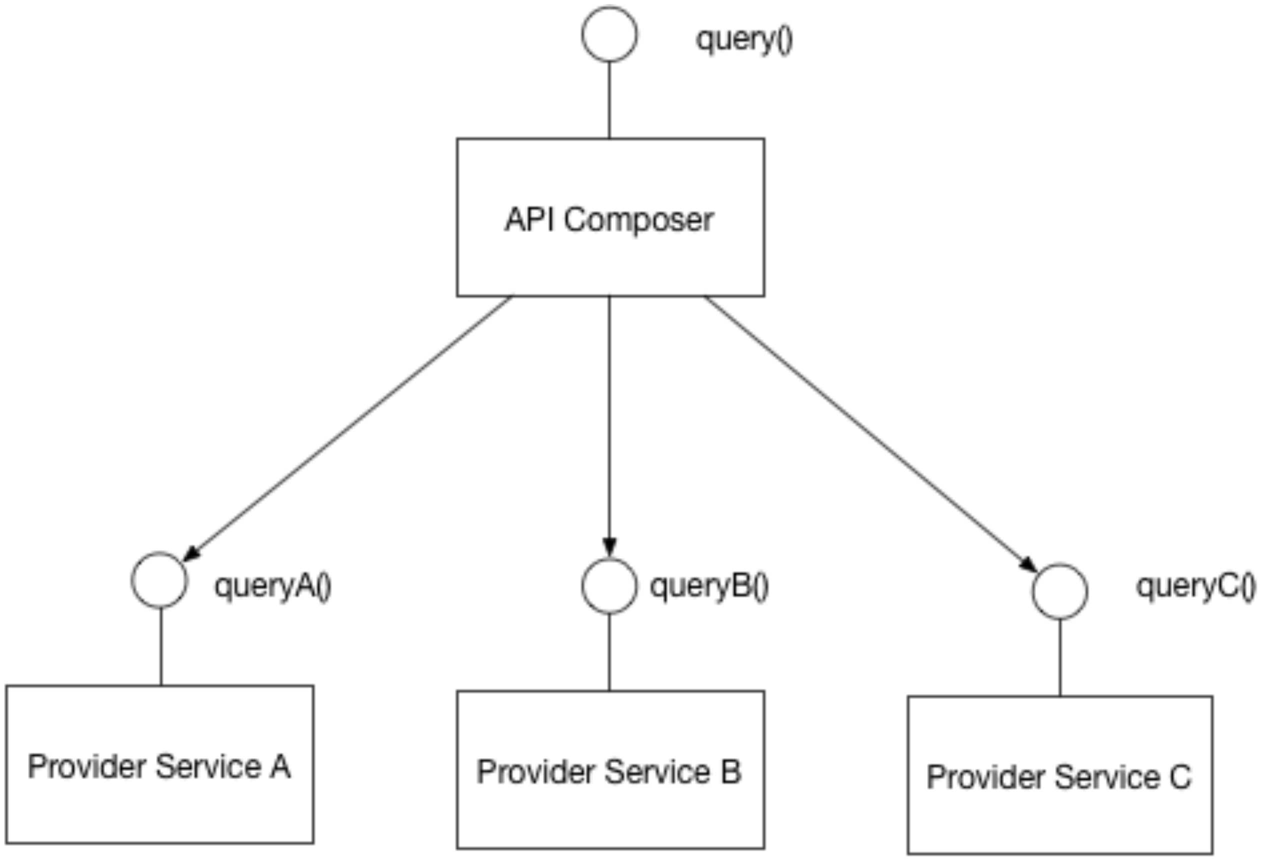
You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services.

## Problem

How to implement queries in a microservice architecture?

## Solution

Implement a query by defining an *API Composer*, which invoking the services that own the data and performs an in-memory join of the results.



## Example

An API Gateway often does API composition. In our application Netflix Zuul API getaway is configure to achieve composition pattern.

## Resulting context

This pattern has the following benefits:

- It a simple way to query data in a microservice architecture

This pattern has the following drawbacks:

- Some queries would result in inefficient, in-memory joins of large datasets.

# Resilience (Circuit Breaker)

## Context

You have applied the Microservice architecture. Services sometimes collaborate when handling requests. When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable. Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

## Problem

How to prevent a network or service failure from cascading to other services?

## Forces

## Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

## Example

`RegistrationServiceProxy` from the Microservices Example application is an example of a component, which is written in Scala, that uses a circuit breaker to handle failures when invoking a remote service.

```
@Component
```

```
class RegistrationServiceProxy @Autowired()(restTemplate:
RestTemplate) extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _

  @HystrixCommand(commandProperties=Array(new
HystrixProperty(name="execution.isolation.thread.timeoutInMil
liseconds", value="800")))
  override def registerUser(emailAddress: String, password:
String): Either[RegistrationError, String] = {
    try {
      val response =
restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(emailAddress, password),
        classOf[RegistrationBackendResponse])
      response.getStatusCode match {
        case HttpStatus.OK =>
          Right(response.getBody.id)
      }
    } catch {
      case e: HttpClientErrorException if e.getStatusCode ==
HttpStatus.CONFLICT =>
        Left(DuplicateRegistrationError)
    }
  }
}
```

The `@HystrixCommand` arranges for calls to `registerUser()` to be executed using a circuit breaker.

The circuit breaker functionality is enabled using the `@EnableCircuitBreaker` annotation on the `UserRegistrationConfiguration` class.

```
@EnableCircuitBreaker
class UserRegistrationConfiguration {
```

# Pattern: Distributed tracing

## Context

You have applied the Microservice architecture pattern. Requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc.

## Problem

How to understand the behaviour of an application and troubleshoot problems?

## Forces

- External monitoring only tells you the overall response time and number of invocations - no insight into the individual operations
- Any solution should have minimal runtime overhead
- Log entries for a request are scattered across numerous logs

## Solution

Instrument services with code that

- Assigns each external request a unique external request id
- Passes the external request id to all services that are involved in handling the request
- Includes the external request id in all log messages
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

This instrumentation might be part of the functionality provided by a Microservice Chassis framework.

## Examples

The Microservices Example application is an example of an application that uses client-side service discovery. It is written in Scala and uses Spring Boot and Spring Cloud as the Microservice chassis. They provide various capabilities including Spring Cloud Sleuth, which provides support for distributed tracing. It instruments Spring components to gather trace information and can delivers it to a Zipkin Server, which gathers and displays traces.

The following Spring Cloud Sleuth dependencies are configured in `build.gradle`:

```
dependencies {
  compile "org.springframework.cloud:spring-cloud-sleuth-stream"
  compile "org.springframework.cloud:spring-cloud-starter-sleuth"
  compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit"
```

RabbitMQ is used to deliver traces to Zipkin.

The services are deployed with various Spring Cloud Sleuth-related environment variables set in the `docker-compose.yml`:

```
environment:
  SPRING_RABBITMQ_HOST: rabbitmq
  SPRING_SLEUTH_ENABLED: "true"
  SPRING_SLEUTH_SAMPLER_PERCENTAGE: 1
  SPRING_SLEUTH_WEB_SKIPPATTERN: "/api-docs.*|/autoconfig|/configprops|/dump|/health|/info|/metrics.*|/mappings|/trace|/swagger.*|.*\\.png|.*\\.css|.*\\.js|/favicon.ico|/hystrix.stream"
```

This properties enable Spring Cloud Sleuth and configure it to sample all requests. It also tells Spring Cloud Sleuth to deliver traces to Zipkin via RabbitMQ running on the host called `rabbitmq`.

The Zipkin server is a simple, Spring Boot application:

```
@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinServer {

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

}
```

It is deployed using Docker:

```
zipkin:
  image: java:openjdk-8u91-jdk
  working_dir: /app
  volumes:
    - ./zipkin-server/build/libs:/app
  command: java -jar /app/zipkin-server.jar --
server.port=9411
  links:
    - rabbitmq
  ports:
    - "9411:9411"
  environment:
    RABBIT_HOST: rabbitmq
```