**Microservices**

**Smaller services that work together**

Why Microservices?

(Take example of Ecommerce app)

In Monolith applications, presentation, business and data access layer present in same code base and deployed as a single unit. Even though you create multi module project one module is dependent on other.

Benefits:

* Upgrade & Deploying

In Microservices, if any change was made in any service you can change the version of that particular srvc and deploy that srvc itself as code base is different. (only shopping cart)

* Scaling

You can easily scale up and down the particular srvc itself in peak time according to user requirement(like if peak sales in ecommerce)

* Different Tech stack and database this is called polyglot services

i.e, In Microservices as each service is independent each srvc can be developed in any language like java or other and different database.

* Central confirguration management for different environments
* Naming server by Eureka,Fault tolerance by Circuit breaker, Api gateway(find in downside of this doc)
* Less Dependent on other services & easy to test

Drawbacks:

* Automation

Need to build,deploy & monitor all srvcs

* Tracking and debugging

As there are many services if any issue occurs we need to debug many services and track(flow) to find the exact cause.

* Monitoring

Monitoring all the services also becomes difficult even though there are tools(websites) available

* In Monolith app, only single repo is enough and ci/cd pipeline is simple but here there are Mono repo and poly repo.

In Mono Repo, there will be single repo for different services i.e will be different folders for different services. Here we can clone only one repo for all services. For ci/cd pipeline, there are no tools available for multiple ci/cd pipeline for single repo. If any change is occurred in one service only that folder can be deployed by writing scripts so its difficult.

In Poly repo different ci/cd pipeline is available for different services as they are in different repo. But cloning and maintaining multiple repos becomes somewhat difficult

* Configuration management

You need to manage configurations of 100’s of components across different environments.

ServiceOrientedArchitecture Vs Microservices

**Communication b/w Microservices**

By Http i.e rest call Synrconous communication

By Messaging i.e Brokers Asynchronous communication

One will produce messages and other service will consume the messages

**By Spring Boot**

Enable building production ready applications quickly

Provides non-functional features

Embedded servers, metrics & health checks(monitoring), externalized configuration

**By Spring Cloud**

Provides cloud enabled microservices. It is a framework designed to build applications quickly

Important Spring cloud modules

Naming server(Eureka)

Spring cloud load balancer(Client side load balancing)

Feign (Easier Rest Clients)

Visibility & Monotoring with Netflix Api Gateway

Configuration Management with Spring Cloud Config Server

Fault tolerance with Resilience4j

Components of Microservice

**Centralized configuration Management (Spring cloud config server)**

We can maintain config of different services for different environments in central config in git and all microservices will get them. If any change in config happens we can do there.

**Eureka Naming Server (Monitoring & acts as load balancer)**

It is an application that holds information about all client service applications. Each service registers with eureka server known as service registration. When one service needs to call other service calling service asks eureka and eureka server provides instance of other service to the calling service this is known as Service Discovery.

All services will register with Eureka Naming Server. So that we will be knowing which service is up or down. We don’t need port numbers to call one service to other service by feign client by application name we can do as every service is registered with Eureka server.

Dynamically scale up & down.

When one service needs to call other service it will ask Eureka server for the active instances. We will use Spring load balancer for the balancing between different instances. So there is no need of middle tier load balancer seperatley.

So by using Euerka client dependency in your calling function spring cloud load balancer dependency will be there in eureka client. So with both eureka client and feign load balance will happen. In previous spring versions ribbon was used but now its not used.

**API Gateway (Spring Cloud Gateway newer older is Zul)**

It acts as single endpoint to the client side and routes the requests to internal microservices

By using all services are not exposed directly

Authorization & authentication can be done here itself at starting

Service discovery integration

Load balancing

Retries, Circuit Breaker mechanism, Rate limiter

Logging, tracing

**Circuit Breaker(older-Hystrix Circuit Breaker new- Resilience Circuit Breaker)**

In our houses, in abnormal conditions when additional power flows circuit breaker will open and protects internal wiring circuits in house.

In Microservices, a service calls other services to retrieve data and there may be chance that downstream service is down. It may be due to slow network connection, timeouts or temporarily unavailable. So, retrying calls can solve the issue. Sometimes service may be down longer time. So client don’t have knowledge & he will be sending requests continuously. So network resources will be exhausted with low performance & bad user experience. Also, the failure of one service might lead to cascading failures throughout application.

Circuit Breaker pattern works based on 3 states

Closed, Open, Half-open

Closed : downstream is working fine

Open : Suppose downstream service is down, so all requests will be failed. So we can configure if 10 requests comes change closed to open state. So at this time no requests will be going to downstream and user will be getting response as downstream service is down. How much time it will be in open state? We can configure upto 10sec remain in open state. After 10 sec if some request comes the state will be from open state to Half – open state

Half-open: In this state, if suppose 10 requests comes 5 requests will be blocked and 5 requests will be passed. In those 5 requests if more than 60%(threshold value) will be processed successfully then it will move to closed state(that is order service is up) or else it will move to open state. In open state again it waits for 10 secs and it will come to Half-open & process continuous.

Ex: In app.properties

resilience4j.ratelimiter.instances.sample-api.maxRetryAttempts=5

resilience4j.ratelimiter.instances.sample-api.waitDuration=1s

@GetMapping(“/sample-api”)

@Ratelimiter(name=”default”)

@Bulkhead(name=”default”)

Public String sampleApi(){

return “rate limiting response”;

}

Ratelimiter means in specific time I need to call only specific no.of requests. Ex: In above ex mention below config In 10 sec 2 calls

In application.properties

//2 requests in 10secs

resilience4j.ratelimiter.instances.default.limitForPeriod=2

resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s

Bulkhead means to limit the number of concurrent calls to a particular service.

In above example in app.properties

resilience4j.bulkhead.instances.default.maxConcurrentCalls =10

**The Need For The Cloud**

Loads on applications is not consistent throughout the year:

* Consider a startup company, which could have a potential to grow very fast.
* Imagine an online shopping portal, whose customer load varies throughout the year.
* Other online business portals, such the insurance sector, which have peak load at a particular time of the year, and very little load during other times.

If you provision your infrastructure to be able to handle the peak system lead, what would that infrastructure be doing the rest of the time?

Most of it would just be sitting idle, watching the fun.

That is exactly the problem cloud wants to solve.

Cloud intends to make it easy to provision infrastructure when you need, and just as easy to release it when you don’t. This feature of the cloud has created a lot of buzz around it, over the last decade or so.

The thing is, you cannot take any application and just put it on the cloud.

A lot of things need to be right, in order to make an application cloud-enabled. They are known as **cloud-native applications**.

**Advantages of Cloud**

##### Cost Reduction

**Pay for what you use.** There are significant cost reductions with using infrastructure from the cloud, because you only use what you need for the duration you need them.

##### Scalability And Elasticity

When applications are built well, they can auto scale based on the load. Elasticity means increase or decrease based on work load. Scalability is always used to address the increase in workload in an organization.

##### Reliability

Since your applications are cloud-enabled, they are designed to withstand certain kinds of outages. For example, applications with microservices architecture have fault-tolerance built into them.

### Improved collaboration &  Unlimited storage capacity

Cloud applications improve collaboration by allowing groups of people to quickly and easily share information in the cloud via shared storage.

### Challenges With Developing Cloud Applications

#### Security

Security is a major concern, as the infrastructure is not what you own, and it can be dynamically acquired and released.  Your data gets stored and processed by a third-party vendor and you cannot see it.

## Internet Connectivity

Cloud services are dependent on a high-speed internet connection. So businesses that are relatively small and face connectivity issues should ideally first invest in a good internet connection so that no downtime happens. It is because internet downtime might incur vast business losses.

## Performance and Investment

When your business applications move to a cloud or a third-party vendor, so your business performance starts to depend on your provider as well. Another major problem in cloud computing is investing in the right cloud service provider.

Before investing, you should look for providers with innovative technologies. The performance of the BI’s and other cloud-based systems are linked to the provider’s systems as well

# **Cloud Service Models**

Infrastructure as a Service (IaaS), Platform as a Service (PaaS), [Software as a Service (SaaS)](https://www.javatpoint.com/cloud-service-models#SaaS)

|  |  |  |
| --- | --- | --- |
| IAAS | PAAS | SAAS |
| Advantages:  It’s a pay-as-you-use service where a third party provides you with infrastructure(hardware) services, like storage and virtualization, as you need them, via a cloud, through the internet.  You don’t have to maintain or update your own on-site datacenter because the provider does it for you.  **Focus on the core business, On-demand scalability** | Advantages:  It is where a provider hosts the hardware and software on its own infrastructure and delivers this platform to the user as an integrated solution, solution stack, or service through an internet connection.  You write the code, build, and manage your apps, but you do it without the headaches of software updates or hardware maintenance. The environment to build and deploy is provided for you.  Scalability  Applications deployed can scale from one to thousands of users without any changes to the applications. | Advantages:  Delivers an entire application that is managed by a provider, via a web browser.  One to Many  SaaS services are offered as a one-to-many model means a single instance of the application is shared by multiple users.  No special software or hardware versions required  All users will have the same version of the software and typically access it through the web browser. SaaS reduces IT support costs by outsourcing hardware and software maintenance and support to the IaaS provider.  Multidevice support  SaaS services can be accessed from any device such as desktops, laptops, tablets, phones, and thin clients.  easy to buy & low maintanance  SaaS pricing is based on a monthly fee or annual fee subscription |
| Disadvantages:  **Security (need to choose trustworthy provider)**  Most of the IaaS providers are not able to provide 100% security.  Migration issues  It is difficult to migrate VM from one IaaS provider to the other, so the customers might face problem related to vendor lock-in. | Disadvantages:  It may happen that some applications are local, and some are in the cloud. So there will be chances of increased complexity when we want to use data which in the cloud with the local data.  Data Privacy  Corporate data, whether it can be critical or not, will be private, so if it is not located within the walls of the company, there can be a risk in terms of privacy of data | Disadvantages:  Security  Actually, data is stored in the cloud, so security may be an issue for some users.  **Total Dependency on Internet**  Latency issue  Since data and applications are stored in the cloud at a variable distance from the end-user, there is a possibility that there may be greater latency when interacting with the application compared to local deployment |
| Examples:  AWS, Microsoft Azure & Google Cloud | Examples:  Redhat Openshift, Salesforce & AWS Elastic Beanstalk | Examples:  Google Apps like gmail & etc, Salesforce |
| It provides a virtual data center to store information and create platforms for app development, testing, and deployment.  Used by network architects. | It provides virtual platforms and tools to create, test, and deploy apps.  Used by developers | It provides web software and apps to complete business tasks.  Used by end users. |

**Apache Camel**

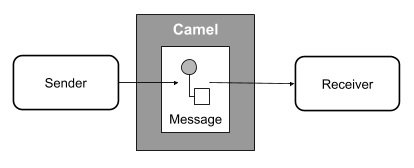
Why Name Camel ?

* it stmands for Concise Application Message Exchange Language (i.e. the Java [DSL](https://camel.apache.org/manual/dsl.html) for routing)
* a Camel can carry 4 times the load of other beasts of burden
* You can go a long way with Camel!

What is Apache Camel ?

Apache Camel is an open source framework that provides rule-based routing and mediation engine. Apache Camel essentially provides an implementation of various EIPs. It makes integration easier by providing connectivity to a very large variety of transports and APIs. For example, you can easily route JMS to JSON, JSON to JMS, HTTP to JMS, FTP to JMS, even HTTP to HTTP, and connectivity to Microservices. You simply need to provide appropriate endpoints at both ends. Camel is extensible and thus in future more endpoints can be added easily to the framework.

Camel is a black box that receives messages from some endpoint and sends it to another one. Within the black box, the messages may be processed or simply redirected.



So why have a framework for this? In practical situations as seen in the introduction case study, there may be many senders and many receivers each following its own protocol such as ftp, http and jms. The system may require many complex rules such as message from sender A should be delivered only to B & C. In situations, you may have to translate the message to another format that the receiver expects. This translation may be subject to certain conditions based on the message contents. So essentially you may need to translate between protocols, glue components together, define routing rules, and provide filtering based on message contents.

**Enterprise Integration Patterns (EIP)**

EIP are design patterns that aim to provide solutions for integration challenges. There are many patterns

Pipeline pattern (opposite of multicast pattern)

It is default pattern when you specify multiple outputs. Here its not the **same** message sent to multiple destinations, but a sent through a chain (the pipes and the filters). Output from transformOrder will be input to validateOrder

Ex:

from("jms:queue:order:in").pipeline("direct:transformOrder", "direct:validateOrder", "jms:queue:order:process");

Or

from("jms:queue:order:in")

.to("direct:transformOrder")

.to("direct:validateOrder")

.to("jms:queue:order:process");

Multicast (opposite to pipeline pattern)

Multicast allows us to route the **same message** to a set of different endpoints and process them in a different way. i.e it multicasts the message to different destinations & all destinations gets same copdy of the original message.

By default, processing on different endpoints is not done in parallel, but this can be changed by using *parallelProcessing()* DSL statement.

Ex: both route1 & route2 gets same copy sent from input file

from(“file:files/input”)

.multicast()

.to("direct:route1","direct:route2").end();

from("direct:route1")

.transform(body().append("World"))

.to("file://" + DESTINATION\_FOLDER\_WORLD);

from("direct:route2")

.transform(body().prepend("Hello"))

.to("file://" + DESTINATION\_FOLDER\_HELLO);

## **Content Based Router**

Content Based Router is a message router which routes a message to its destination based on a message header, part of payload or basically anything from message exchange which we consider as content.

Ex:

from(“files:file/input”)

.choice()

.when(simple(“${file:ext}==xml”))

.log(“xml file”)

.otherwise()

.log(“not a xml file”)

.to()

Splitter

The splitter allows us to **split the incoming message into a number of pieces and processing each of them individually. It breaks** the composite message into a series of individual messages, each containing data related to one item.

Ex:

from("file://"SOURCE\_FOLDER”)

.convertBodyTo(String.class)

.split(body(),”/n”)

.to(“activemq:myqueue”);

In above ex: It takes input and splits into multiple lines into multiple messages and sends to the queue. We can also use delimiter , or something else.

Aggregator

The [Aggregator](http://www.enterpriseintegrationpatterns.com/Aggregator.html) from the [EIP patterns](https://camel.apache.org/components/3.20.x/eips/enterprise-integration-patterns.html) allows you to combine a number of messages together into a single message.

Ex: Here if multiple files are sent the ‘to’ element in body contains same the combine all messages.

from("file:files/aggregate-json")

.unmarshal().json(JsonLibrary.Jackson, CurrencyExchange.class)

.aggregate(simple("${body.to}"),newArrayListAggregationStrategy())

.completionSize(3)

.to("log:aggregate-json");

Note: Here newArrayListAggregationStrategy actual logic will be present

Routing Slip

Routes message dynamically according to the routingslip endpoints

Ex:

String routingSlip = "direct:endpoint1,direct:endpoint3";

from("timer:dynamicRouting?period={{timePeriod}}")

.transform().constant("My Message is Hardcoded")

.dynamicRouter(method(dynamicRouterBean));

from(“from:endpoint1”) .to()

from(“from:endpoint3”) .to()

Deadletter channel:

It is common and it should be expected that sometimes problems can happen, for example, database deadlocks, which can cause a message not to be delivered as expected. However, in certain cases, trying again with a certain delay will help and a message will get processed.

**Dead Letter Channel allows us to control what happens with a message once it fails to be delivered.** Using Dead Letter Channel we can specify whether to propagate the thrown Exception to the caller and where to route the failed Exchange.

When a message fails to be delivered, Dead Letter Channel (if used) will move the message to the dead letter endpoint.

DLC allows to catch and handle exceptions in your routes. Without it, any error or exceptions will be silently ignored, which can lead to data loss or other unexpected behaviour. By this, you can ensure all things are logged properly and allows to debug or troubleshoot and fix problems.

Ex:

@Override

public void configure() throws Exception {

errorHandler(deadLetterChannel("log:dead?level=ERROR") //sending to this endpoint when error occurs we can also mention activemq

.maximumRedeliveries(3).redeliveryDelay(1000)

.retryAttemptedLogLevel(LoggingLevel.ERROR));

from("file://" + SOURCE\_FOLDER + "?delete=true")

.process(exchange -> {

throw new IllegalArgumentException("Exception thrown!");

});

Best Practices for Apache Camel

1 You can define endpoint in property file and take from them

Ex: logging-endpoint=log:logendpoint

from(“file:files/input)

to(“{{logging-endpoint }}”)

2 Make sure camel context is running in standalone spring boot

To ensure the spring boot application keeps running until being stopped or the JVM terminated, typically only need when running spring boot standalone, i.e not with spring-boot-stater-web . Normally we use this dependency spring-boot-stater-web. This will launch web application due to this context will be running continuously, if not configure below property.

In app.properties mention this

camel.springboot.main-run-controller=true

3 logging.level.org.apache.camel.impl=DEBUG

logging.level.org.apache.camel.spring.boot = INFO// when program works fine

//can be used in app.properties if you need more information

4 wiretap it is additional endpoint to which all previous information will be sent here also

From()

.wiretap()

.to

5 Using Dead letter channel as mentioned in eip patterns above

DLC allows to catch and handle exceptions in your routes. Without it, any error or exceptions will be silently ignored, which can lead to data loss or other unexpected behaviour. By this, you can ensure all things are logged properly and allows to debug or troubleshoot and fix problems.