**Spring Cloud Components Vs Openshift**

**Service Registry**

**Eureka:-**Eureka is a REST (REpresentational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

Tight integration between Ribbon and Eureka allows declarative use of Eureka when the caller is using the Ribbon library.

**OpenShift**:-In OpenShift, a Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Additional backing pods can be added to, or removed from a service, while the service itself remains consistently available, enabling anything that depends on the service to refer to it through a consistent address.

Contrary to a third-party service registry, the platform in charge of service replication can provide a current and accurate report of service replicas at any moment. The service abstraction is also a critical platform component that is as reliable as the underlying platform itself. This means that the client does not need to keep a cache and account for the failure of the service registry itself. Ribbon can be declaratively configured to use OpenShift instead of a service registry, without any code changes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Load Balancer**

**Ribbon:-**Ribbon allows load balancing among a static list of instances that are declared, or however many instances of the service that are discovered from a registry lookup.

**gRPC:-**The more modern gRPC is a replacement for Ribbon that’s been developed by Google and adopted by a large number of projects.

While Ribbon uses simple text-based JSON or XML payloads over HTTP, gRPC relies on Protocol Buffers for faster and more compact serialization. The payload is sent over HTTP/2 in binary form. The result is better performance and security, at the expense of compatibility and tooling support in the existing market

**OpenShift Service:-**OpenShift provides load balancing through its concept of service abstraction. The cluster IP address exposed by a service is an internal load balancer between any running replica pods that provide the service. Within the OpenShift cluster, the service name resolves to this cluster IP address and can be used to reach the load balancer. For calls from outside and when going through the router is not desirable, an external IP address can be configured for the service.

A service is a fixed cluster IP address. This IP address is a virtual IP address that can be used to discover/call the actual endpoints in your Pods/containers. How does this IP address know which pods/containers are eligible to be discovered? It uses a \_Label Selector\_\_ that picks pods that have labels that you define. For example, let’s say we want a Kubernetes Service with a selector of “app=cassandra AND tier=backend”. This would give me a service with a virtual IP that discovers any pods that match that label (have both app=cassandra AND tier=backend). This selector is actively evaluated so that any pods that leave the cluster or any pods that join the cluster (based on what labels they have) will automatically start to participate in the service discovery

Note an instance of a Kubernetes Service is not a “thing” or an appliance or a docker container or anything..it’s a virtual “thing”… so there are no single points of failure. It’s an IP address that gets routed by Kubernetes.

Since the IP is fixed for a given environment (Dev, QA, etc) we don’t care about caching it: it’ll never change. Now if we use DNS, our app can be configured to talk to services at <http://awesomefooservice/> and even when we move from Dev to QA to Prod, insofar we have those Kubernetes Services in each environment, our app doesn’t need to change.

Using Kubernetes Services as we did in the Service Discovery section above, we accomplish proper load balancing (again, without all of the overhead of the service registries, custom clients, DNS drawbacks, etc). When we interact with a Kubernetes Service via its DNS (or IP), Kubernetes will by default load balance across the pods in the cluster (remember, the cluster is defined by the labels and label selectors). If you don’t want the extra hops in the load balancing, no worries; this virtual IP is routed directly to the Pods, it does not hit a physical network

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Circuit Breaker**

**Hystrix:-**Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

Hystrix implements both the circuit breaker and bulkhead patterns.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Externalized Configuration**

**Spring Cloud Config:-**Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

**OpenShift ConfigMaps:-**ConfigMaps can be used to store fine-grained information like individual properties, or coarse-grained information like entire configuration files or JSON blobs. They provide mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Distributed Tracing**

**Sleuth/Zipkin:-**Spring Cloud Sleuth generates trace IDs for every call and span IDs at the requested points in an application. This information can be integrated with a logging framework to help troubleshoot the application by following the log files, or broadcast to a Zipkin server and stored for analytics and reports.

**Jaeger:-**Jaeger, inspired by Dapper and OpenZipkin, is an open source distributed tracing system that fully conforms to the Cloud Native Computing Foundation (CNCF) OpenTracing standard. It can be used for monitoring microservice-based architectures and provides distributed context propagation and transaction monitoring, as well as service dependency analysis and performance latency optimization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Proxy/Routing:-**

**Zuul:-**Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security, and more. Zuul supports multiple routing models, ranging from declarative URL patterns mapped to a destination, to groovy scripts that can reside outside the application archive and dynamically determine the route.

**Istio/ingress:-**Istio is an open platform-independent service mesh that provides traffic management, policy enforcement, and telemetry collection. Istio is designed to manage communications between microservices and applications. Istio is still in pre-release stages.

Red Hat is a participant in the Istio project.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* Spring Cloud and Openshift both claim to be the best environment for developing and running microservices, but they are both very different in nature and address different concerns.
* **How do I handle application service registry and discovery?**

To benefit from the many Microservices you create in a distributed, elastic cloud environment, a proper service registration and discovery strategy is critical to allow them to communicate with each other or to be consumed. As a pioneer in cloud-native Microservices implementation, Netflix has built and released a set of opinionated libraries known as Netflix OSS, such as Eureka, Zuul, Ribbon, etc. And they are well integrated with the Spring framework. Typically, services are registered to Eureka, and Ribbon/Zuul handle the service discovery and basic load-balancing. .

Today, Kubernetes natively supports service registry, discovery and load balancing. You no longer need Netflix OSS for these tasks. Instead, you design the system the Kubernetes way. You can leverage Kubernetes’ Service type to expose your Microservices, then it automatically registered into Kubernetes system via virtual IP or DNS . The discovery will be through the familiar DNS lookup style. For Service-based clusters, Kubernetes provides a built-in load balancer to distribute the workload among Pods (containers).

* **Should I use Kubernetes virtual IP or DNS based service lookup?**

There are two options for you: Environment variable or DNS. Environment variable is enabled out-of-box by Kubernetes primer. Under the covers, Kubernetes exposes the service virtual IP within the cluster so that other Pods/Services can invoke the target service via this environment variable entry. The downside of this approach is the sequence of the services’ creation among depending Kubernetes services matters. And in general, relying on IP addresses is not a good idea.

A better solution is the DNS-based approach. Let’s say I have an “Order” microservice and I name it order-service as part of the name in the service’s yaml definition. The front end web service can simply invoke the Order service via REST endpoint <http://order-service/id>. This abstracts the application from the underlying infrastructure. To use the DNS approach, you need to install the Kubernetes DNS add-on if you are building your own Kubernetes infrastructure. But if you are using Kubernetes as a Service,, the DNS service is automatically enabled in your cluster.

* **Kubernetes handles application resiliency, but what about gracefully handling failures of dependent services?**

Kubernetes ensures application resiliency or fault-tolerant through its built-in self-recovery mechanism. Implemented as ReplicaSet, Kubernetes ensures your application Pod (containers) always trying to meet the desired state. For example, at any given time, there are always 3 order service Pods running in my cluster.

This is cool and taken care by the Kubernetes for a service. But you still need to design your Microservices to react to dependency failures. For example, if a downstream service is down, or if there’s a database or storage outage. If this happens, you don’t want this service issue to impact the entire application by blocking resources or bringing down the entire user experience. You want to gracefully degrade or fail-safe this service. For a Java-based implementation, It recommend the Netflix Hystix library that provides commands to implement failure-handling patterns like Circuit Breaker or Bulkhead, as well as a dashboard to view the system’s overall health. It integrates well with Spring Boot based container applications; the Hystrix dashboard itself can be easily packaged and deployed as containers and managed by Kubernetes.

* **How do I expose the services for external consumption? LoadBalancer or Ingress or NodePort?**

You need to determine how your client-side application, either Web 2.0 or Mobile apps, can access your Microservices hosted on Kubernetes cluster. More than likely, you do not want to expose your core business logic or data Microservices directly to Internet—instead, you should build a BFF (Back-end for Front-end) or an API gateway that in turns determines which back end service to consume. Under this design, access to the back end data microservices are typically only through the Kubernetes internal network. The front end tier (BFF or API gateway) becomes the Internet-facing component.

Then the question becomes how to expose this front end service tier for external consumption. You have two choices in Kubernetes: LoadBalancer or Ingress (there is a third option—NodePort—but I don’t recommend it for production use).

Services with LoadBalancer provide an externally-accessible IP address that sends traffic to the correct port on your cluster nodes. LoadBalancer is easy to enable and allows your client app to access REST API via a service-defined HTTP path. The drawbacks of LoadBalancer are consuming more IPs (billable resource) for your application, relying on the Cloud vendor implementation, and lack of central entry point to handle common tasks such as TLS termination.

The other approach is Ingress. Rather than creating a LoadBalancer service for each service that you want to expose to the public, Ingress provides a unique public route that lets you forward public requests to services inside your cluster based on their individual paths. This approach allows a central entry point to you application. So you can easily add DNS route and global load balancing across multiple Kubernetes regions. It is a promising solution.