Day – 2

**General Interview Preparation**

INTERVIEW PREPARATION

**Key Conversations to Practice Before Your Interview**

Below is a list of sample interview questions that everyone should be prepared to answer (regardless of technology). The content below is a combination of insights from PDP (Professional Development Program), client feedback, and observations we have made regarding success.

1. **Tell me about yourself**

Introduce yourself based on your technical background

* 60-90 Seconds
  + Do not rush.
  + If you have to speak more quickly than usual to keep it to 90 seconds, consider rewording or removing optional/extra information.
  + If you finish in under 60 seconds, consider slowing down and/or adding more relevant information.
* Greeting, name, your title as listed on your portfolio, and your degree and school (graduation year optional)
* Talk about your technical experience (Revature experience; outside experience only if you feel fully confident interviewing on it)
* Explain technologies you have worked with (again, Revature technologies; additional technologies only if you feel prepared to interview on them)
* Explain your reason for going into technology (optional)
* State your overall career goal (optional)

1. **Project Discussion**

"Walk me through your latest project" or a similar version is a very common interview question. You should prepare yourself for it as it can drive the focus of your entire interview. The questions below may be asked outright or only implied. Be sure you think about each of these before entering an interview.

*Project Introduction*

* Purpose of the project: You can play around with the order a bit, but you should include the 5 elements below.
  + Who is the audience for this application?
  + What problem does it solve?
  + Why does the problem need to be solved?
  + How does this application solve it?
  + What is it called?
* Explain any methodologies you followed, Agile or otherwise, and state how that helped your project progression.
* Include information as to whether it was inherited code or a first iteration.

*Project data flow*

"Explain the complete data flow through your last project."

* Different data access layers and any design patterns used for the data flow
* Include the technologies used and how you maintained version control
* Discuss CI/CD strategies if applicable

*Role*

"What was your role in the project?"

* Specific role you took
* How did your role help other teams/team members?
* How did you manage your time constraints during the sprint?

1. **Overcoming challenges**

Many clients will ask soft skills questions relating to your response to challenges. Below are a few good examples/strategies for answering these kinds of questions.

"What was the biggest challenge during your most recent project?" There is a right answer to this question!

* Talk about a challenge that you overcame, how you overcame it, and how you would prevent it from happening or mitigate it earlier in the future. (This is the right answer.)
* Do not be negative about the challenge. Not all challenges are fun, but they all are opportunities to learn.
* Be cautious discussing interpersonal conflict. How you speak about others in the interview indicates how you will speak about your future colleagues and supervisors. Interviewers will take note.

"How did you handle [insert scenario]?"

Utilize STAR stories

Situation - Task - Action - Result

Example: How did you handle a time of great stress?

"[Situation] In my final semester of my graduate degree, I not only had my classes to keep up with, but I worked full time as a Graduate Teaching Assistant, and I also needed to get my Master's thesis finished so it could be approved by the deadline. [Task] I needed to utilize all of my time management skills. [Action] I chose to create a calendar that outlined all of my due dates and goals and to give myself a daily moderate task list of reasonable goals. I also made sure to leave wiggle room and scheduled time for self-care in order to maximize my efficiency. [Result] As a result, I got my thesis submitted and presented early, I got straight As that semester, and I was able to keep on top of all of my responsibilities with my students. As stressful and busy as that semester was, I look back on it with pride."

Practice the above points as they are key conversations that many have during interviews. You’ve got this – you’re amazing.

**What are microservices**

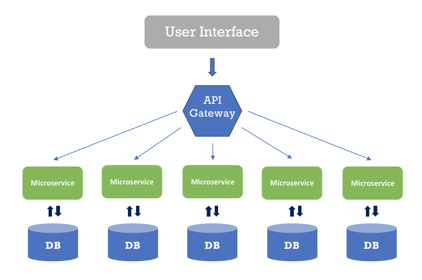
What are microservices

Microservice Architecture describes a way of designing software applications as suites of independently deployable services.

"..the microservice architectural style is an approach to developing a single application as **a suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and independently deployable by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies. " -- **James Lewis and Martin Fowler (2014)**

"Microservices are small, autonomous services that work together." -- **Sam Newman**

In a microservice architecture, we divide an application into **services**. Each service runs a unique process and manages its database. A service can generate alerts, log data, support user interfaces (UIs), handle user identification or authentication, and perform various other tasks.



The microservice paradigm provides development teams with a more decentralized approach for building software. Most companies in the world – *like Amazon, Netflix, Uber, Etsy, etc.,* have adopted the microservices architecture for developing their applications. Over time, these enterprises dismantled their monolithic applications and refactored them into microservice-based architectures. This has given them scaling advantages, greater business agility, and increased profits.

Netflix uses a microservices architecture

Netflix started as a website that allowed us to select DVDs and delivers that to our mailbox. It also began as a monolithic application, built and managed with a traditional development model by a single team of over 100 engineers.

As the company transitioned to a product that delivers streaming content to millions of viewers all over the world, 24 hours a day, **Netflix also had to switch to a microservice architecture** that facilitates receiving content from various sources, getting it into its systems, processing it, and distributing it to users seamlessly.

Each day, Netflix’s API receives hundreds of millions of calls, which travel between microservices to accomplish a task. When you click the play button on a movie, you might trigger a chain of five API calls which keep track of playback, collect content for the user interface, manage streaming, and more.

Netflix also leverages serverless architectures, which fits well with microservices, to encode this content, backup files, secure their assets, and monitor their IT environment.

**Advantages / disadvantages of a microservice architecture (MSA)**

Advantages of Microservices

* Microservices follow the [Single Responsibility Principle](https://deviq.com/single-responsibility-principle/).
* Fault isolation - If one service crashes, it’s quite possible the rest of the application can continue to operate until that service recovers.
* Microservices are language and platform independent.
* Dynamic scaling - we can dynamically scale up/down the service instances whenever required.
* Developers have the freedom to develop and deploy services independently.
* Use of containers allowing for a quick deployment and development of the application.

Disadvantages of Microservices

* Microservice architecture is complex.
* Harder to test and monitor because of the complexity of the architecture.
* Large numbers of microservices are harder to manage and secure.

**Characteristics of a MSA**

Characterics of Microservices Architecture

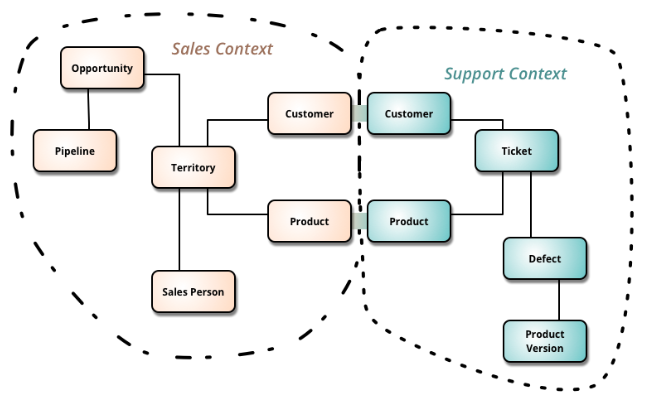
* Microservice architecture encourages us to break our application into **multiple individually deployable components**. Here, a component refers to a unit of software that is independently replaceable and upgradeable. These components expose their functionality as **services** and serve a business purpose. Components are loosely coupled and communicate with each other through pre-defined protocols, such as message queues, HTTP request/response models, and so on.
* Services are organized based on the business capabilities, not by the technology.
* Each service has its own database, either different instances of the same database technology, or entirely independent database systems - an approach called [Polyglot Persistence](https://www.jamesserra.com/archive/2015/07/what-is-polyglot-persistence/#:~:text=Polyglot%20Persistence%20is%20a%20fancy,dealt%20with%20different%20data%20stores).
* MSA provides **infrastructure automation** with [continuous delivery](https://en.wikipedia.org/wiki/Continuous_delivery) and [automated testing](https://smartbear.com/learn/automated-testing/what-is-automated-testing/).
* Any service’s failure should be in **isolation**. Failure of one service should not make the whole application go down. Since services can fail, it’s important to detect the failures and, if possible, restore the service.

Challenges of Microservices Architecture

Microservice architecture is much more complex than legacy systems. This environment becomes more complicated because teams have to manage and support many moving parts. Some of the challenges that an organization faces are:

* **Bounded Context** - The bounded context is a central pattern in [Domain-Driven Design(DDD)](https://en.wikipedia.org/wiki/Domain-driven_design). DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships. Bounded context defines our domain boundaries in the business context.

For example, a product in the sales context refers to an item in the process of being sold. A product in the support context refers to the item that is already sold to the customer which has some defects, so the customer contacts the support team.



* Finding a root cause for problems – Distributed logic with distributed data increases the effort of finding a root cause.
* Version management and cyclic dependencies between services.
* Logging is distributed between services.
* An issue that’s caused by one service can cause trouble elsewhere.

**MSA vs Monolith applications**

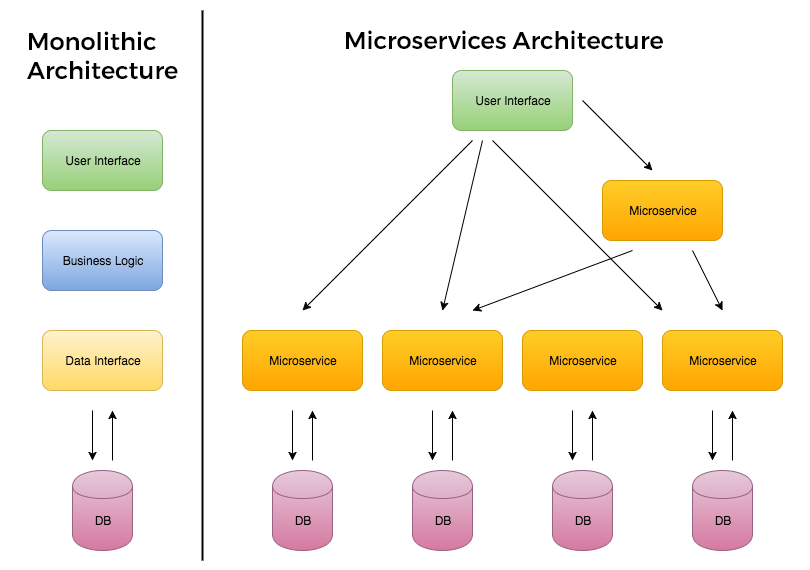
Microservice Architecture vs Monolithic Applications

We build monolithic applications as a single unit. They usually develop these Enterprise Applications in three major parts:

* **User Interface** - might consist of HTML pages and JavaScript running on the user’s browser.
* **Server-Side application** - handles incoming data (like HTTP requests), executes domain logic, retrieves and updates data from the database, and updates the user interface
* **Databases** - stores data on the tables of an RDBMS.

Any minor changes in the system require building and deploying a new version of the server-side application. Over time, it’s often hard to keep a good modular structure, making it harder to manage changes that ought to only affect one module within that application. Scaling requires scaling of the entire application rather than the parts of it that require the most resources.

These frustrations have led to the **microservice architectural style**, building applications as suites of services. When services are independently deployable and scalable, each service provides a firm module boundary. We can develop services in distinct programming languages.



* Services are deployed and updated independently, which gives more flexibility.
* Monolithic uses a shared database. Each microservice uses a single database. Each service can use any database system (MySQL, Oracle, NoSQL) depending on its business logic.
* Monolithic applications are tightly coupled. Here, it is difficult to change technology or language or framework. MSA ensures that the services are loosely coupled, so it's easy to make changes since services are independent to each other.
* Monolithic applications are more compatible with [Waterfall Model](https://en.wikipedia.org/wiki/Waterfall_model). MSA is adaptable with [Agile Methodology](https://en.wikipedia.org/wiki/Agile_software_development) for developing applications.

Summary - Difference between Monolithic and Microservices Architecture

| **Monolithic** | **MSA** |
| --- | --- |
| Simple to develop and deploy. | Complex and hard to develop and deploy. |
| Code changes affect the entire system. | Only the microservice that is changed would be affected. |
| One codebase and one shared database. | A codebase and database for each microservice. |
| Hard to scale or upgrade. | Easy to scale and upgrade. |
| Less expensive and faster to develop. | More expensive and takes more time to develop. |
| Monolithic uses a shared database. | Each microservice uses a single database. Each service can use any database system (MySQL, Oracle, NoSQL) depending on its business logic. |
| Monolithic applications are tightly coupled. Here, it is difficult to change technology or language or framework. | MSA ensures that the services are loosely coupled, so it's easy to make changes since services are independent to each other. |
| The entire system can be affected by a single error or bug. | The entire system is shielded from the error or bug on one microservice. |

**Discovery service and Netflix Eureka Server**

# **Discovery Service**

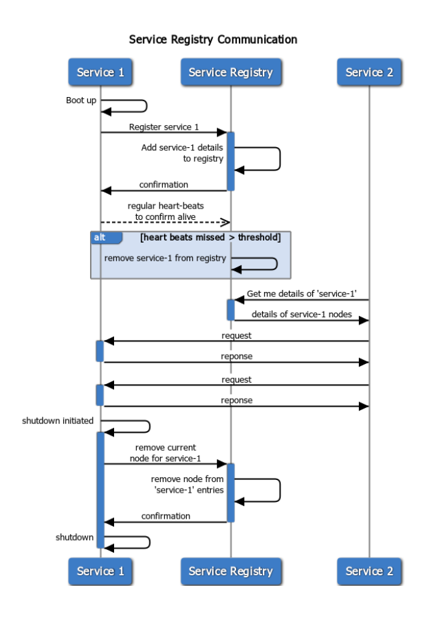
In a microservice architecture, applications are built up of self-sufficient units, which can be deployed and updated independently, and communicate with each other via REST APIs.

One of the challenges in a microservices application is how services discover and connect to each other, because service instances are constantly being created and destroyed according to scaling needs. Fixed IP addresses don't work, so we need a service registry to track the dynamic changes in the network locations of service instances.

Service discovery is a method for application components to locate each other. The service registry is a key part of service discovery.

## Service Registry

* A database containing the network locations of service instances.
* It contains the currently-available instances of each service and their connection details.
* Services query the service registry to retrieve the details for the required microservice and then connect to it.
* The registry maintains a **heartbeat mechanism** to see if services are still up and if not, removes them from the registry.



[Netflix Eureka](https://github.com/Netflix/eureka/wiki) is a good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a POST request. Every 30 seconds it must refresh its registration using a PUT request. A registration is removed by either using an HTTP DELETE request or by the instance registration timing out. A client can retrieve the registered service instances by using an HTTP GET request.

Other examples of service registries - [etcd](https://etcd.io/" \t "_blank), [consul](https://www.consul.io/), [Apache Zookeeper](https://zookeeper.apache.org/)

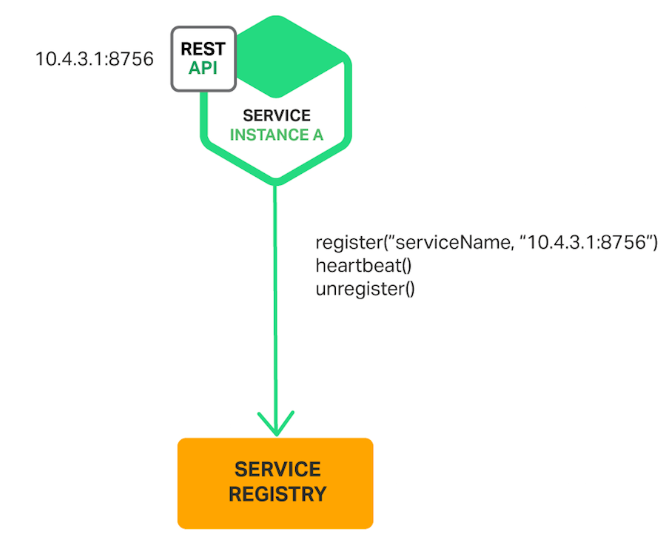
## Microservice Registration Patterns

There are two primary microservice registration patterns to handle the registration and deregistration of service instances in the Service Registry.

1. Self-Registration Pattern
2. Third-party registration Pattern

### Self-Registration Pattern

When using the self-registration pattern, a service instance is responsible for registering and deregistering itself with the service registry. Also, service instances send heartbeat requests to prevent their registration from expiring.



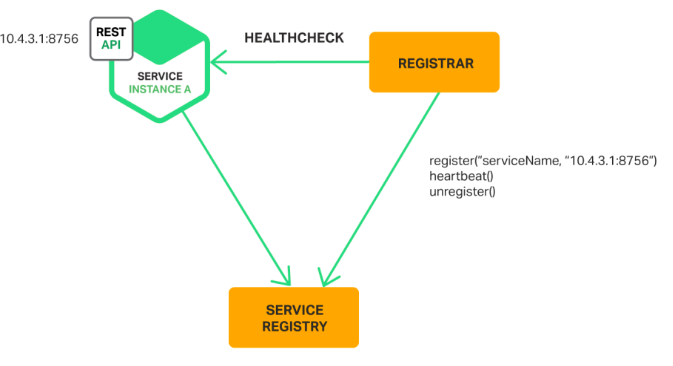
A good example of this approach is the **Netflix OSS Eureka client**. The Eureka client handles all aspects of service instance registration and deregistration. The **Spring Cloud** project, which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. We simply annotate our Java Configuration class with an @EnableEurekaClient annotation.

**Benefit** - This pattern is relatively simple and doesn’t require any other system components.

**Drawbrack** - This pattern couples the service instances to the service registry. We must implement the registration code in each programming language and framework used by our services.

### Third-party registration Pattern

When using the third-party registration pattern, service instances aren’t responsible for registering themselves with the service registry. Instead, another system component known as the **service registrar** handles the registration and deregistration. A service registrar is responsible for identifying that a service has started, registering the service, and unregistering the service when it shuts down or crashes.



Example of a service registrar - [Netflix OSS Prana](https://github.com/Netflix/Prana/wiki),

The service registrar is a built-in component of deployment environments. The [EC2](https://aws.amazon.com/ec2/) instances created by an Autoscaling Group can be automatically registered with an AWS [Elastic Load Balancing](https://aws.amazon.com/elasticloadbalancing/) (ELB). [Kubernetes](https://kubernetes.io/) services are automatically registered and made available for discovery.

**Benefit** - Services are decoupled from the service registry. We don’t need to implement service‑registration logic for each programming language and framework used by our developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

**Drawbrack** - Unless it’s built into the deployment environment, it is yet another highly available system component that we need to set up and manage.

## Service discovery patterns

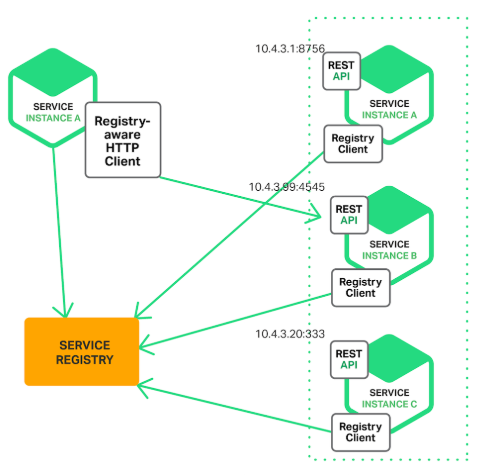
There are two main service discovery patterns:

1. Client-side discovery pattern
2. Server-side discovery pattern

### Client-Side Discovery Pattern

The client contacts a service registry, receives details for available services, and contacts one of them using a load balancing algorithm.

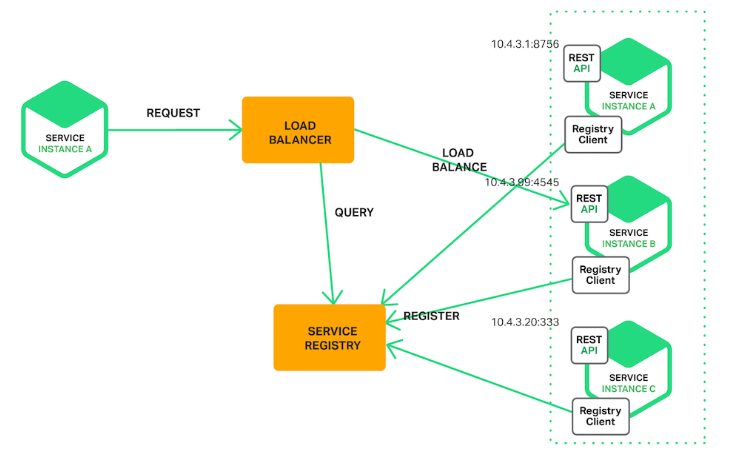
[Netflix OSS](https://netflix.github.io/) provides a great example of the client-side discovery pattern.



### Server-Side Discovery Pattern

The client contacts a load balancer, making a request that indicates which type of service it needs. The load balancer consults the service registry, selects the optimal service (for example, the least loaded one) and routes the request to it.

An example of a server-side discovery mechanism is AWS [Elastic Load Balancing](https://aws.amazon.com/elasticloadbalancing/) (ELB). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn’t a separate service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

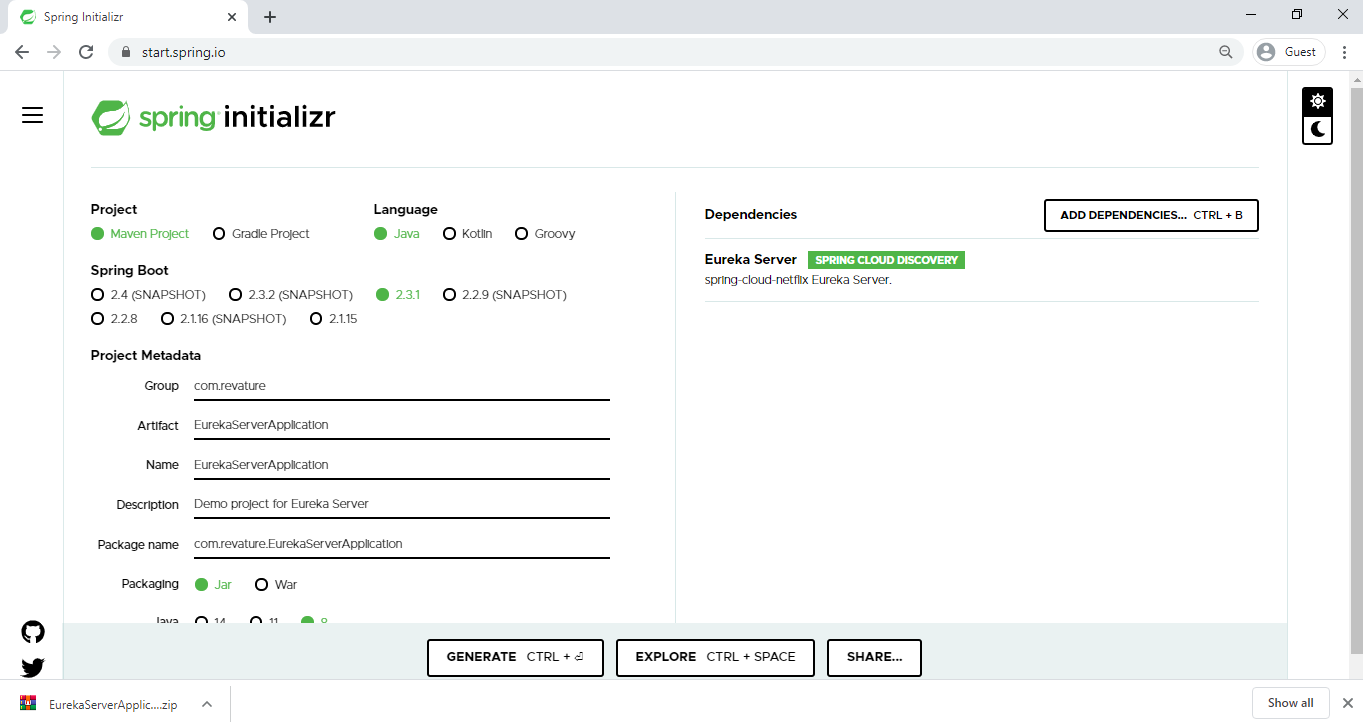


In some deployment environments, we need to set up own service-discovery infrastructure using a service registry such as [Netflix Eureka](https://github.com/Netflix/eureka/wiki), [etcd](https://etcd.io/" \t "_blank), or [Apache Zookeeper](https://zookeeper.apache.org/). In other deployment environments, service discovery is built in. For example, [Kubernetes](https://kubernetes.io/) and [Marathon](https://mesosphere.github.io/marathon/) handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router.

## Example - Netflix Eureka Server

Eureka Server is an application that acts as a service registry. Every Microservice will register itself into the Eureka server and the Eureka server knows all the client applications running on each port and IP address.

Naviagate to the [Spring Initializr](https://start.spring.io/) and create the Spring Boot project with Eureka server dependency. The following image shows the Initializr set up for the Eureka server application:



The preceding image shows the Initializr with Maven chosen as the build tool. It also shows values of com.revature and EurekaServerApplication as the Group and Artifact, respectively.

Verify you have Spring cloud Eureka server dependency is added under the pom.xml file.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

Open the the main application class file and enable the Eureka Server configuration by annotating the class with @EnableEurekaServer.

@SpringBootApplication

@EnableEurekaServer

**public** **class** **EurekaServerApplication** {

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**EurekaServerApplication**.**class**, args);

}

}

When the registry starts, it will complain (with a stacktrace) that there are no replica nodes to which the registry can connect. In a production environment, you will want more than one instance of the registry. For our simple purposes, however, it suffices to disable the relevant logging.

By default, the registry also tries to register itself, so you need to disable that behavior as well.

Add some properties under application.properties to handle all of these requirements.

server.port=8761

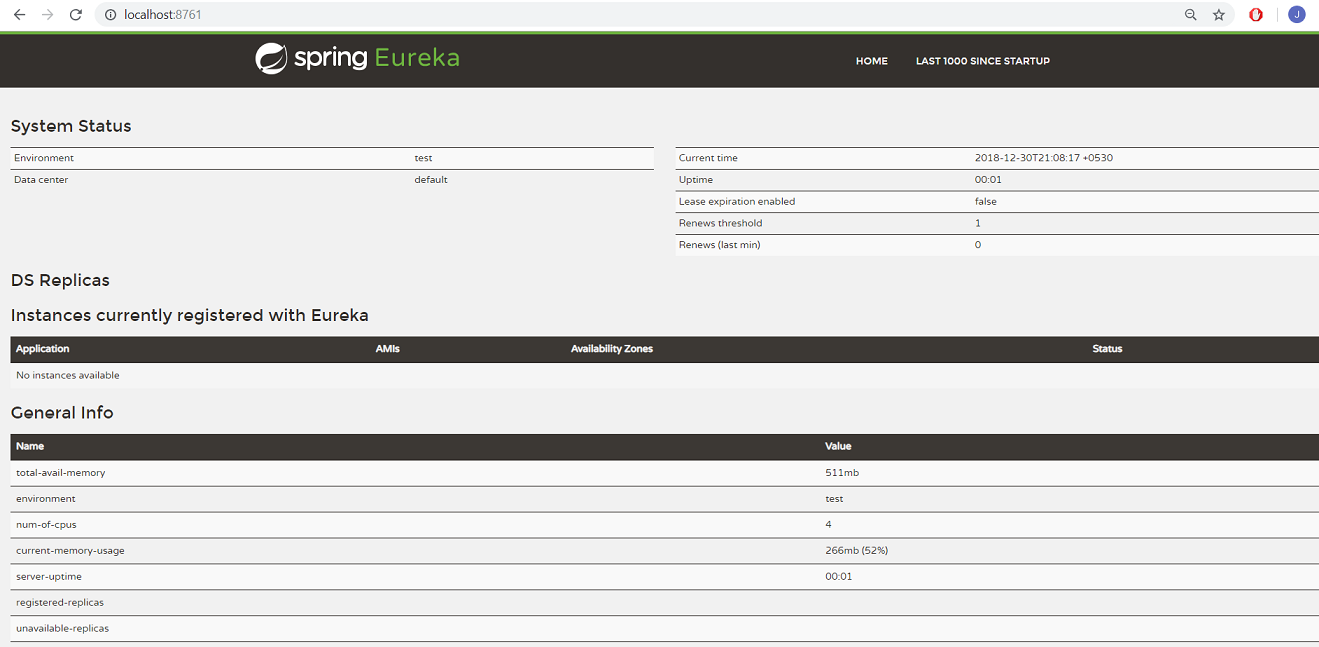
# Port 8761 is the common port for Eureka Servers

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

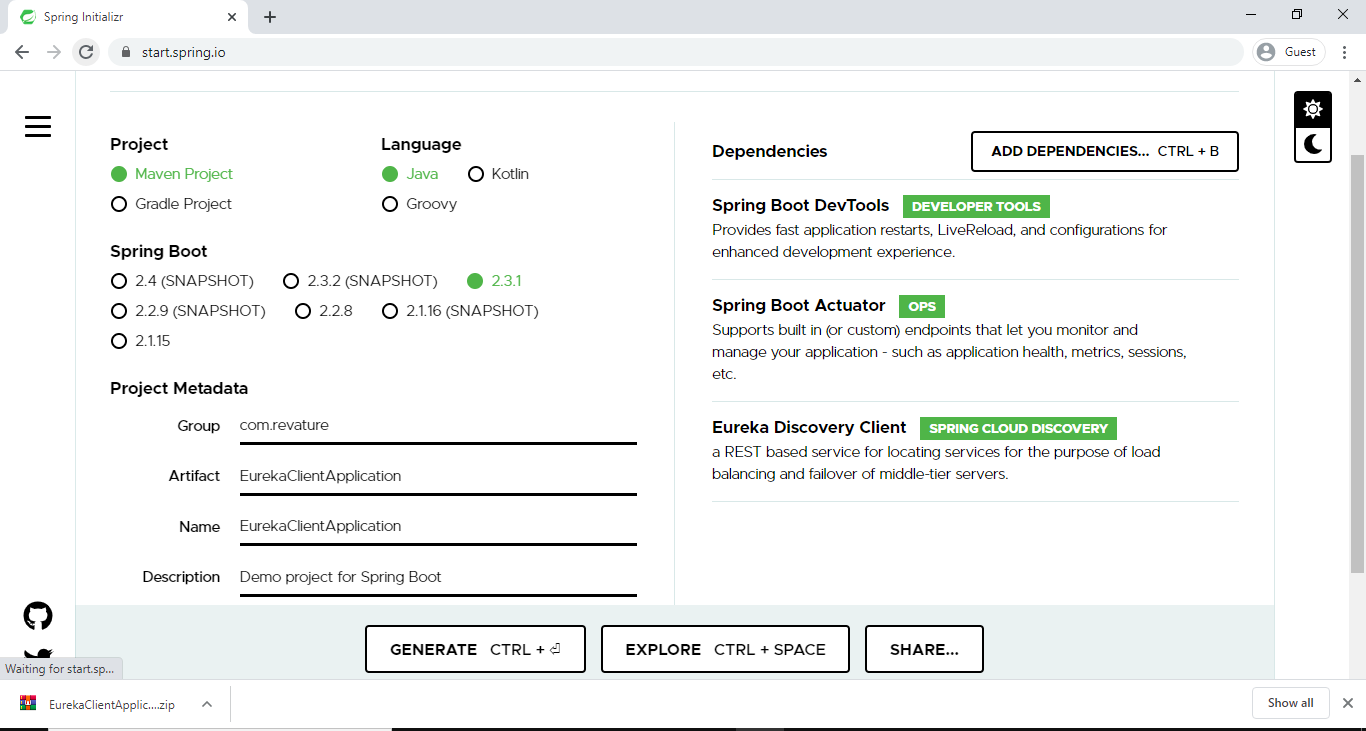
Here we're configuring an application port – 8761 is the default one for Eureka servers.

Run the Eureka server application and visit the URL: <http://localhost:8761/> , you able to see Eureka dashboard



You can see the Eureka server is up and running but no application is registered with it yet. Let's create a Spring Boot Client Application and register that Client Application With Eureka Server.

Navigate to [Spring Initializr](https://start.spring.io/) and create a Spring Boot Application with DevTools, Actuator, and Discovery Client dependencies. The following image shows the Initializr set up for the Eureka Client application:



The preceding image shows the Initializr with Maven chosen as the build tool. It also shows values of com.revature and EurekaClientApplication as the Group and Artifact, respectively.

Open the the main application class file and enable the Eureka client configuration by annotating the class with @EnableDiscoveryClient.

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** **EurekaClientApplication**{

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**EurekaClientApplication**.**class**, args);

}

}

We add a REST Controller class and implement a GET method in the class.

@RestController

**public** **class** **HelloWorldController** {

@GetMapping("/hello-world/{name}")

**public** **String** getHelloWorld (@PathVariable **String** name)

{

**return** "Hello World "+name;

}

}

You need to add the below list of properties in application.properties to register this service in Eureka Server.

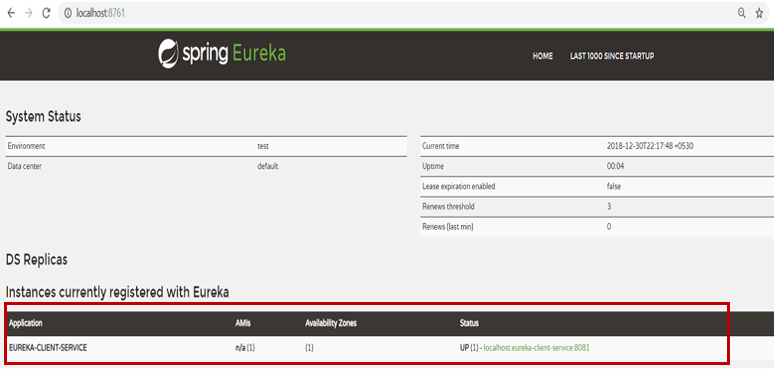
spring.application.name=eureka-client-service

server.port=8081

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

eureka.client.service-url.defaultZone determines the address where the Eureka Server is running so the client application can register itself in Eureka Server.

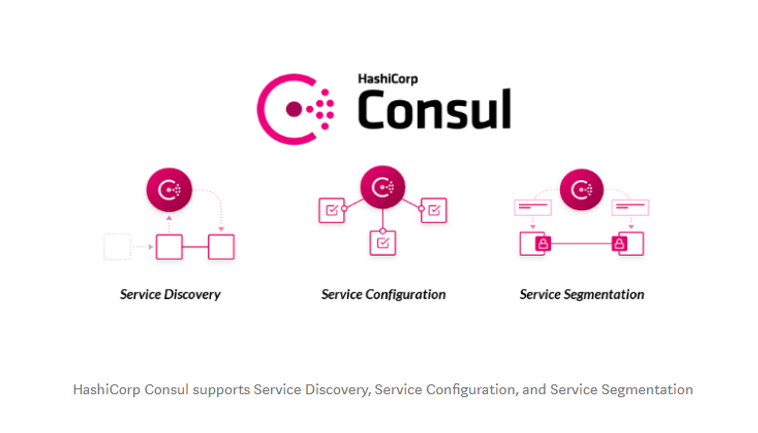
Before running this application, we need to make sure the Eureka Server is up and running. Run the Eureka Client application and navigate to the Eureka Server at <http://localhost:8761/>. This time we see that the client application registered in Eureka Server.



**Consul - Discovery Service**

# **Consul**

Microservices and other distributed systems can enable faster, simpler software development. But there's a trade-off resulting in greater operational complexity around inter-service communication, configuration management, and network segmentation. **[HashiCorp Consul](https://www.hashicorp.com/products/consul/" \t "_blank)** is an open source tool that solves these new complexities by providing service discovery, health checks, load balancing, a service graph, mutual TLS identity enforcement, and a configuration key-value store. These features make Consul an ideal control plane for a **Service Mesh**.



**Consul Agent** - The [Consul agent](https://www.consul.io/docs/agent) is the core process of Consul. The agent maintains membership information, registers services, runs checks, responds to queries, and more. The agent must run on every node that is part of a Consul cluster.

# **Consul Service Mesh**

Service mesh solves the networking and security challenges of operating microservices and cloud infrastructure. Consul is a service mesh solution that offers a software-driven approach to routing and segmentation. It also brings additional benefits such as failure handling, retries, and network observability.

Let's us discuss about the challenges with microservice architecture and how Consul solves these complex challenges and makes easy to work with microservices.

## Service Discovery

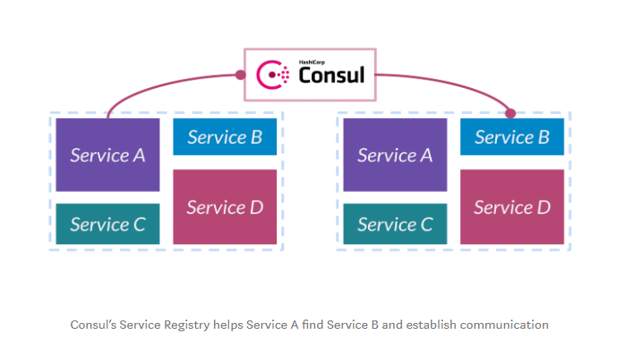
**Service Discovery in a monolithic application** - Let's say service A wants to call service B. So, we would expose a method, mark it as public, in service B and then service A can just call it. They're in the same application. It's just an in-memory function call.

In a monolithic application, the services would expose public functions and it would simply call the function across services.

**Service Discovery in MSA** - Here, we deploy service A and service B as an independent deployable units in a distributed environment. If service A wants to call service B, then service A needs to know the location (IP address) of the service B. So, the challenge here is: "How do these different services discover one another in our distributed environment?"

**Consul’s solution** for this service discovery problem is a central **service registry**.

Consul maintains a central registry that contains the entry for all the upstream services. When a service instance starts, it gets registered on the central registry. The registry is populated with all the upstream instances of the service.



**Example** - When a service A wants to talk to service B, it will discover and communicate with B by querying the registry about the upstream service instances of B.

Consul also provides **health-checks** on these service instances. If one of the service instances or services itself is unhealthy or fails its health-check, the registry would then know about this scenario and would avoid returning the service’s address.

Consul programmatically manages registry, which gets updated when any new service registers itself and becomes available for receiving traffic.

## Service Configuration

**Configuration management in a monolith application** - In monolithic application, we would have all our configuration in a single XML file that configures the whole thing. The advantage of this is that all of our different subsystems, all of our components, had a consistent view of the configuration.

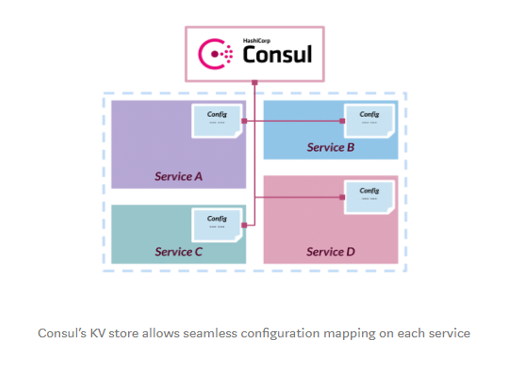
As an example, suppose we wanted to put our application in maintenance mode. We wanted to prevent it from writing to the database so that we could do some upgrades in the background. We would change this configuration file and then all of these subsystems would believe that we're in maintenance mode simultaneously.

**Configuration management in a MSA** - The challenge here is: "How do we think about configuration in our distributed environment?"

The second big challenge we run into is configuration.

**Consul’s solution** for this configuration management is the central **Key-Value store**.

Instead of trying to define the configuration in each service distributed throughout our infrastructure, Consul uses a key-value store to capture it centrally.



**Example** - We can define a key centrally that says whether our system is in maintenance mode or not , "Are we in maintenance mode?" and the values can be "true or false". Also, we have to change some default configuration in each service depending upon the state of our system. When the system is in maintenance mode, we change the "Are we in maintenance mode?" key's value centrally from "false to true". Then, push out the changed state to all our services and configure them. Thus, gives consistent view of configuration across all the services.

## Service Segmentation

In microservice environment, we have hundreds of unique services across different network zone. Traffic pattern between these services are much more complicated. Many services have complicated [east-west traffic flow](https://en.wikipedia.org/wiki/East-west_traffic). The challenge here is: "How do we think about segmenting this network?" and "How do we partition which services are allowed to talk to which other services?"

The third challenge we run into is segmentation. Consul deals with this is with a feature called **Connect**.

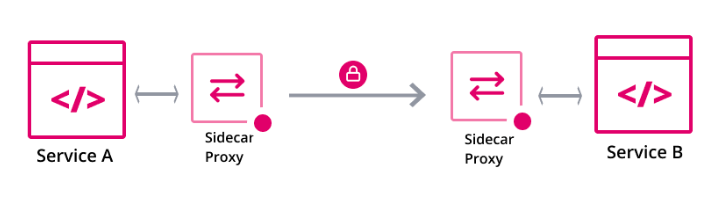
Consul Connect establishes inter-service communication polices by using service graph. With the **Service Graph**, we can define - which service can talk to which other service. So, we might say, "A is able to talk to B" and "C is allowed to talk to D". Here, we're not talking about IP to IP. We're not saying IP 1 can talk to IP 2. We're talking about "service A can talk to service B".

The higher benefit of this is, it is not IP restricted. Rather it’s service level. This makes scalable independent of our distributed network. The policy will be enforced on all instances of service also, there won't be any hardbound firewall rules specific to a service’s IP.

So, when we say "service A can talk to service B", how do we know what is service A and what is service B? and how do we authorize the identity of each service?.

Consul Connect provides service-to-service connection authorization and encryption using **mutual Transport Layer Security (TLS)**. We issue TLS certificates that uniquely identify these services. So we can uniquely say, "This is service A and this is service B. Unlike saying, "There's an IP and we don't actually know what's running at that IP with any strong guarantee."

Consul enforces TLS using an agent-based proxy attached to each service instance. This proxy acts as a sidecar. Applications can use **sidecar proxies** in a service mesh configuration to establish TLS connections for inbound and outbound connections without being aware of Connect at all. Applications may also natively integrate with Connect for optimal performance and security. Connect can help you secure your services and provide data about service-to-service communications.



We probably have many hundreds of services that exist are not TLS aware. So the advantage of imposing it at the proxy layer is that we can get that guarantee of our data being encrypted in transit without needing to re-implement all of these applications. These sidecar proxies prevents us from making any change into the code of original service.

After authorizing the identity of service A and service B, these proxies call the service graph and looks for a policy like this — 'Is there a rule that allows service A to talk to service B?'. If so, then the proxies allow that traffic to take place, A is allowed to talk directly to B through an encrypted channel.

**NOTE:** Consul "connect", HashiCorp's service mesh feature, provides service-to-service networking and security through connection authorization and encryption using mutual Transport Layer Security (mTLS). Applications deployed with the "connect" feature can use sidecar proxies in a service mesh configuration to establish TLS connections for inbound and outbound connections, without being aware of Consul at all.

### Summary - Consul Service Mesh

* Consul addresses the new microservices architecture challenges with **service discovery** and allowing operators to deploy applications into a [**zero-trust network**](https://www.cloudflare.com/learning/security/glossary/what-is-zero-trust/).
* Consul offers us a [service catalog](https://en.wikipedia.org/wiki/Service_catalog), health checks, automatic load balancing, and geo-failover across multiple instances of the same service.
* Consul service mesh uses **mutual TLS** (mTLS) and will automatically generate and distribute the TLS certificates for every service in the mesh. The certificates are used for both:
  + service identity verification
  + service communication encryption
* Consul service mesh deploys **sidecar proxies** locally alongside each service instance, which transparently handles inbound and outbound service connections, automatically verifying and encrptying TLS connections between services.
* Consul also helps us to secure service communication at the network level by enabling us to manage service-to-service communication permissions using [intentions](https://www.consul.io/docs/connect/intentions.html). **Intentions** define service based access control for services in the Consul service mesh and are used to control which services are allowed or not allowed to establish connections.
* In addition to securing our services, Consul service mesh can also intercept data about service-to-service communications and surface it to **monitoring tools**.

# **Spring Cloud Consul**

The Spring Cloud Consul project provides easy integration with **Consul** for Spring Boot applications. With a few simple annotations we can quickly enable and configure the common patterns inside our application and build large distributed systems with [Hashicorp’s Consul](https://www.consul.io/" \t "_blank). The patterns provided include Service Discovery, Control Bus and Configuration, Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

## Example

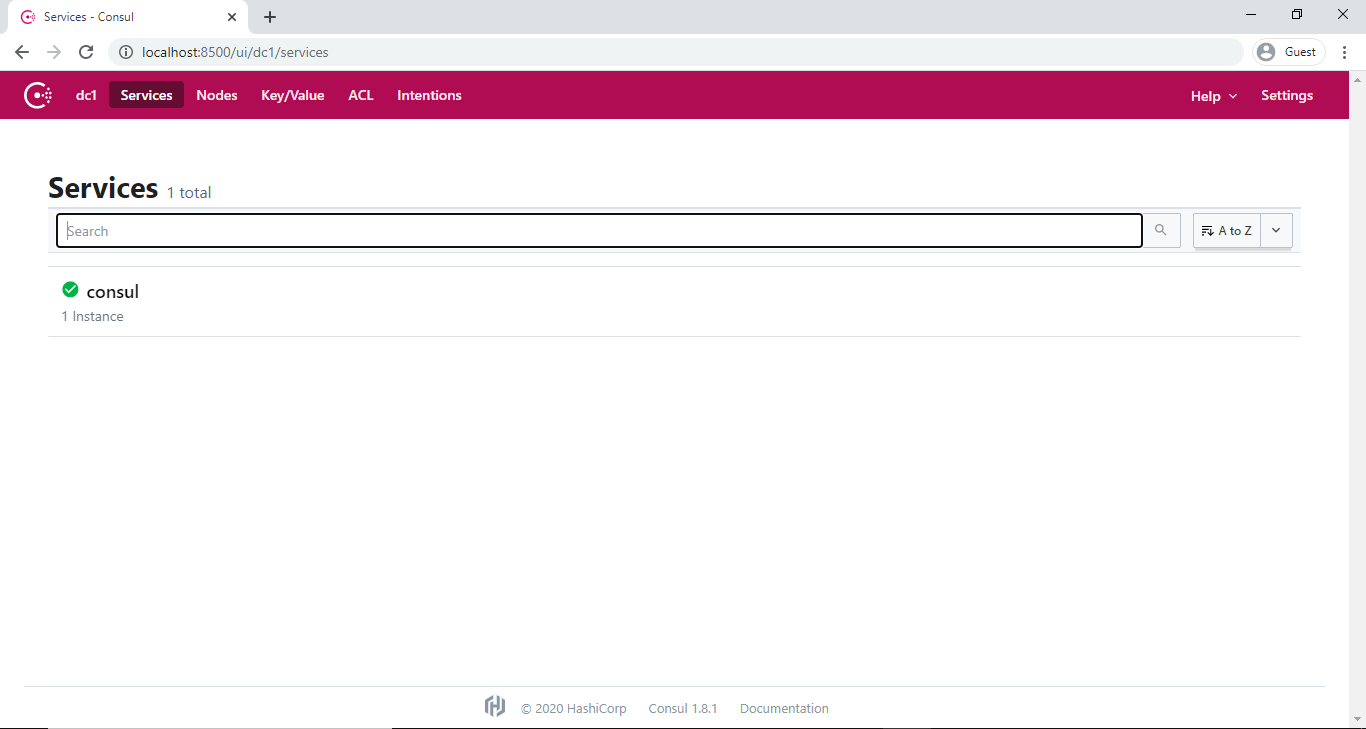
In this example, we will be running consul server using docker. So, we will need to install docker before installing consul.

You can refer this [link](https://gitlab.com/revature_training/docker-team/-/blob/master/modules/docker-concepts/installing-docker.md) to install docker.

After that, follow the below steps to install consul using the docker image:

* First of all, we need to download the docker image of consul from the docker hub. To pull the consul image, run the docker pull consul command on the terminal.
* Next, we need to run the Consul Agent locally on port **8500** (default port for consul) using docker run -d --name consul -p 8500:8500 consul command.
* Then, start the consul server with docker start consul command. Now, the consul container is up and running on the port 8500. You can check if it is available by navigating to <http://localhost:8500/ui> in your browser.

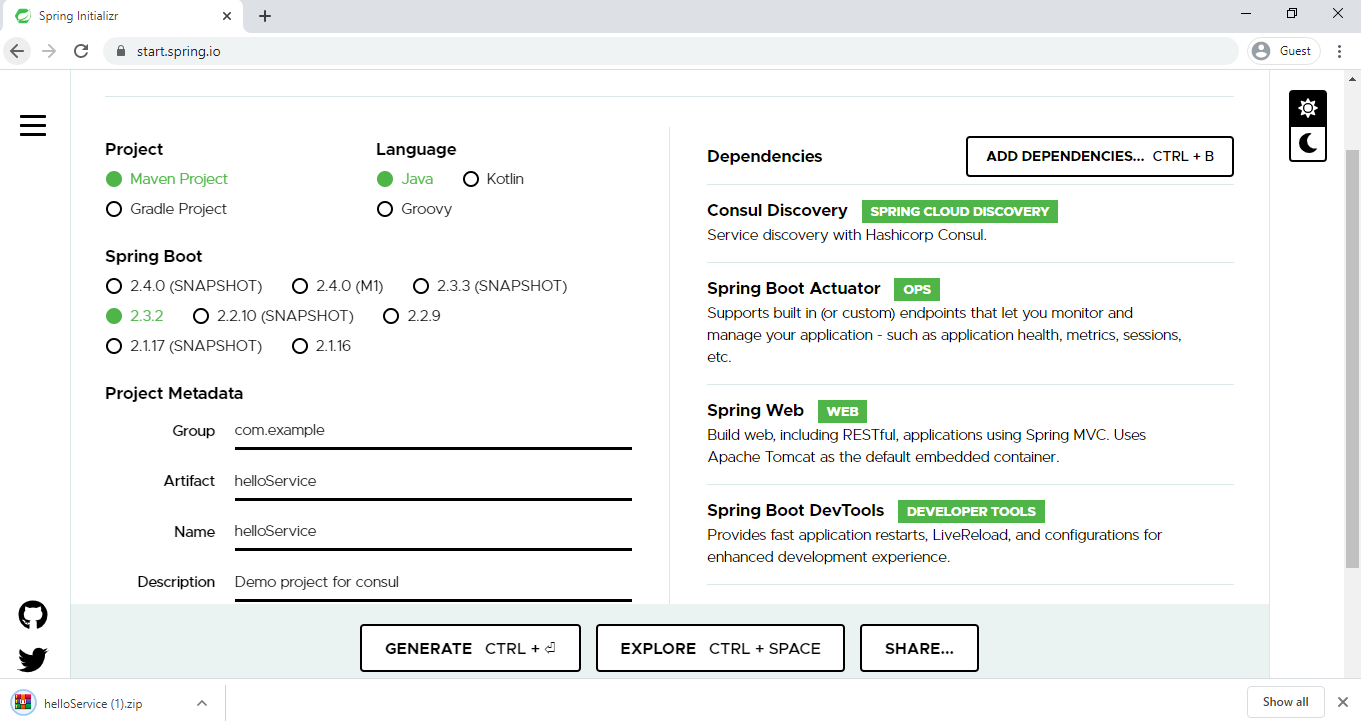
You able to see:



**NOTE:** You can stop the consul server with docker stop consul command.

Now, let us create a simple microservice application and we'll make that service register its instance with the Consul.

Navigate to the [Spring Initializr](https://start.spring.io/) and create the Spring Boot project with Consul Discover, DevTools, Actuator, and Web dependencies. The following image shows the Initializr set up for the Eureka server application:



The preceding image shows the Initializr with Maven chosen as the build tool. It also shows values of com.revature and helloSerive as the Group and Artifact, respectively.

Open the the main application class file and enable the discovery client configuration by annotating the class with @EnableDiscoveryClient.

**package** com.revature.helloService;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient

@SpringBootApplication

**public** **class** **HelloServiceApplication** {

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**HelloServiceApplication**.**class**, args);

}

}

Then, we add a REST Controller class and implement a GET method in that.

**package** com.revature.helloService.controllers;

**import** java.util.Date;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** **HelloServiceController** {

@GetMapping("/hi")

**public** **String** **message**() {

**return** "Hello User !! " + **new** **Date**();

}

}

We configure the below properties in application.properties file to register this service in Consul Server.

spring.application.name= hello-service

server.port=9999

#Setting desired port and name for this service

spring.cloud.consul.discovery.metadata.map = user=admin

#consul metadata

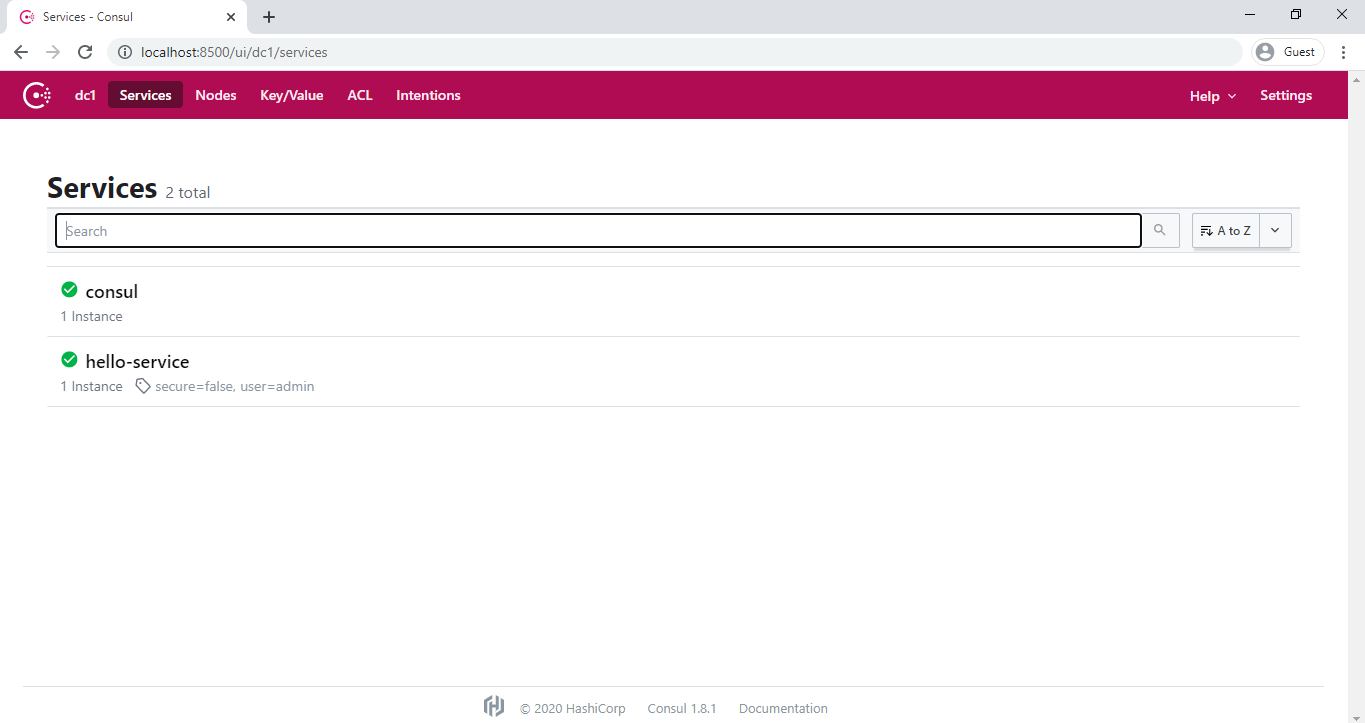
When we run this helloService application, it registers itself in Consul server automatically.

In the console, you able to see the logs of hello-service registering with consul.

2020-08-03 09:38:52.052 INFO 17104 --- [ restartedMain] o.s.c.c.s.ConsulServiceRegistry : Registering service with consul: NewService{id='hello-service-9999', name='hello-service', tags=[secure=false], address='host.docker.internal', meta={map=user=admin}, port=9999, enableTagOverride=null, check=Check{script='null', dockerContainerID='null', shell='null', interval='10s', ttl='null', http='http://host.docker.internal:9999/actuator/health', method='null', header={}, tcp='null', timeout='null', deregisterCriticalServiceAfter='null', tlsSkipVerify=null, status='null', grpc='null', grpcUseTLS=null}, checks=null}

2020-08-03 09:38:52.417 INFO 17104 --- [ restartedMain] c.r.h.HelloServiceApplication : Started HelloServiceApplication in 5.844 seconds (JVM running for 6.993)

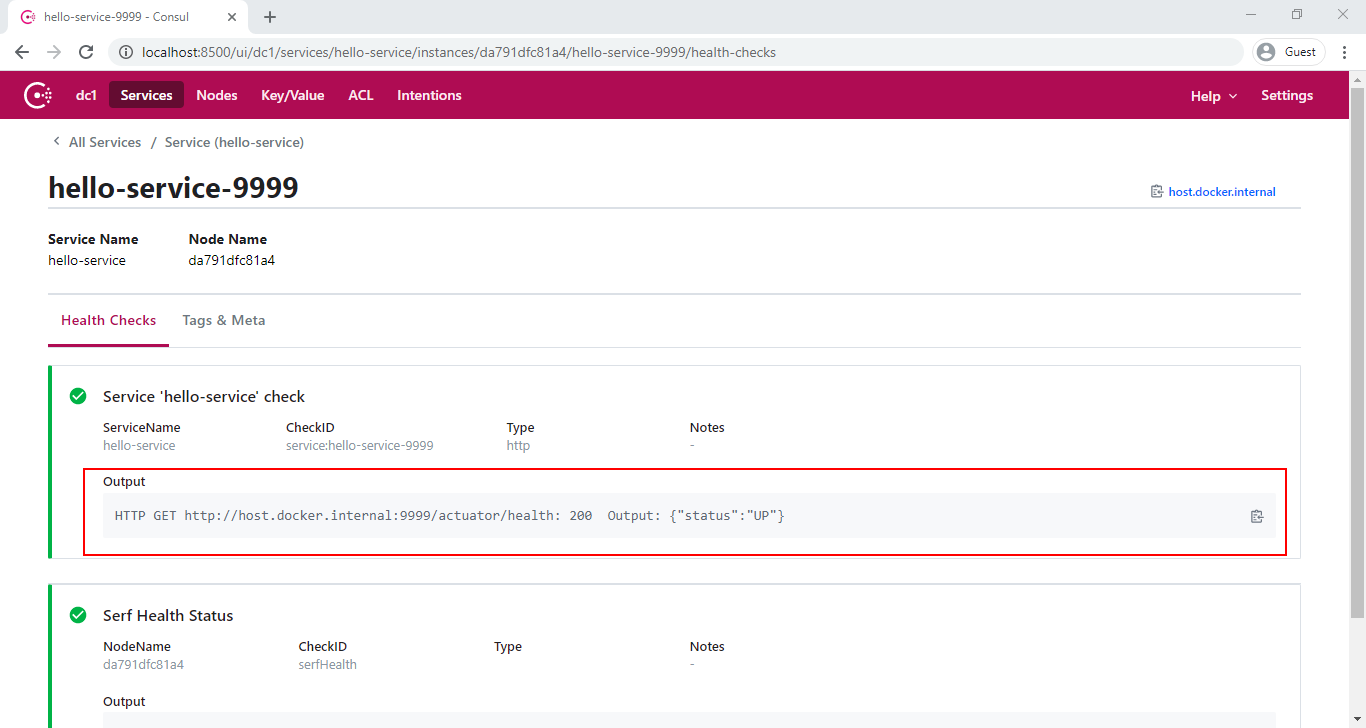
To verify that, visit <http://localhost:8500/ui/> on our browser. You able to see the hello-service registered in the Consul server:



**HTTP Health Checks** - The health check for a Consul instance defaults to "/health", which is the default locations of a useful endpoint in a Spring Boot Actuator application. When you visit <http://localhost:9999/actuator/health>, you able to see:

{"status":"UP"}

This implies that our hello-service is healthy. Also, you can able to see the health status of hello-service on the consul dashboard:



We can change default health check path using spring.cloud.consul.discovery.healthCheckUrl property. Also, we can set the health check intervals using spring.cloud.consul.discovery.healthCheckInterval property.

In application.properties file,

spring.cloud.consul.discovery.healthCheckUrl = http://myserver.com:${server.port}/status

spring.cloud.consul.discovery.healthCheckInterval = 15s

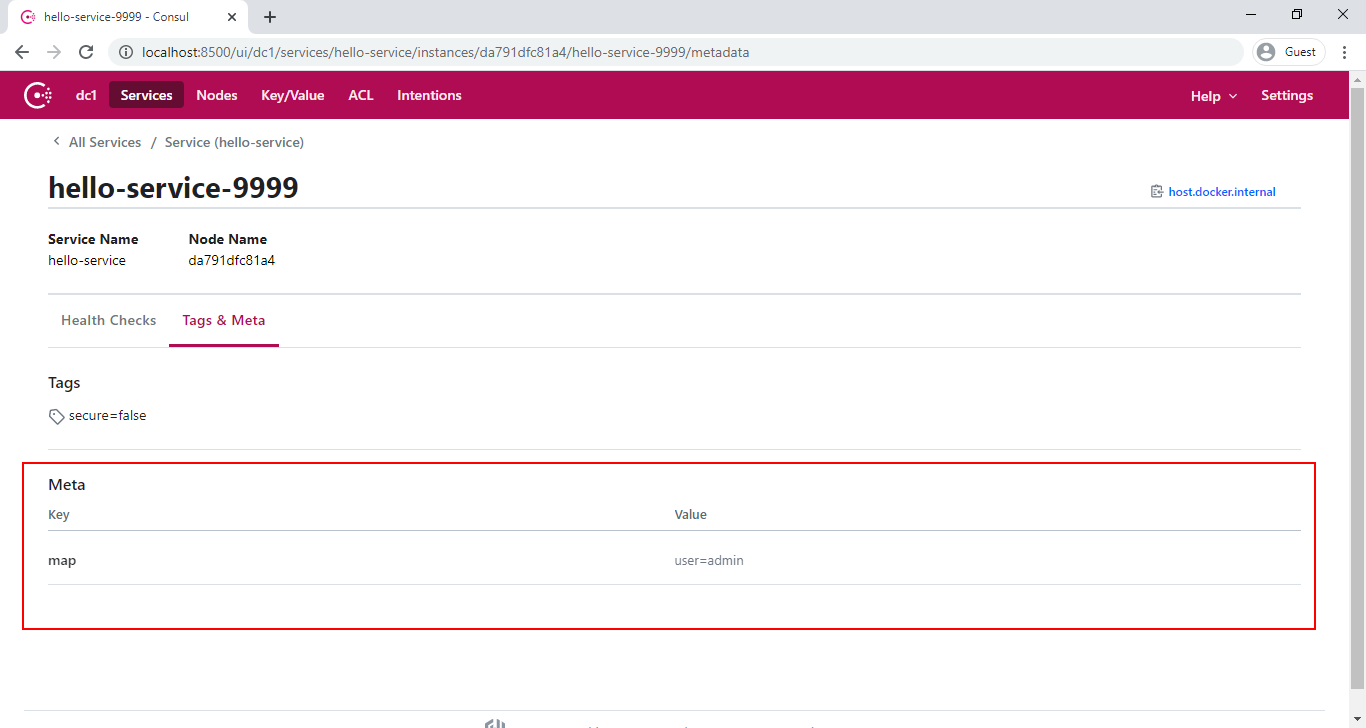
You can disable the health check by setting management.health.consul.enabled=false.

**Metadata** - Consul supports metadata on services. Spring Cloud’s ServiceInstance has a Map<String, String> metadata field which is populated from a services meta field. To populate the meta field set values on spring.cloud.consul.discovery.metadata.map property.

In application.properties file,

spring.cloud.consul.discovery.metadata.map = user=admin

The above configuration will result in a service who’s meta field contains user → admin. Also, you able to see this meta field populated from hello-service on the consul dashboard:

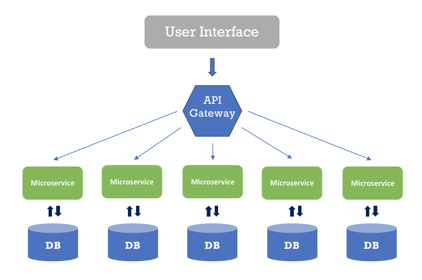


**API Gateway**

# **API Gateway**

An API gateway sits between a client and a collection of backend services. The API gateway takes all API requests from a client, determines which services are needed, and combines them into a synchronous experience for the user.

For most microservices-based applications, an API gateway acts as a single entry point into a system. The API gateway is responsible for request routing, composition, and protocol translation, and can streamline the system. It handles some requests by simply routing them to the appropriate backend service, and handles others by invoking multiple backend services and aggregating the results. If there are failures in the backend services, the API gateway can mask them by returning cached or default data.



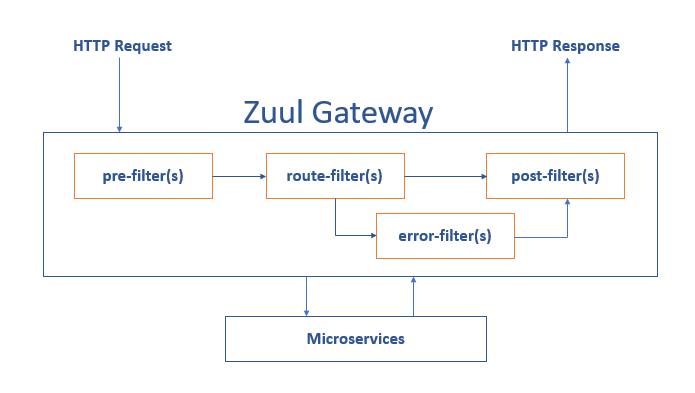
[Netflix Zuul](https://github.com/Netflix/zuul/wiki) is a good example of an API Gateway.

## Zuul Implementation

"Zuul is the front door for all requests from devices and web sites to the backend of the Netflix streaming application. As an edge service application, Zuul is built to enable dynamic routing, monitoring, resiliency and security. It also has the ability to route requests to multiple Amazon Auto Scaling Groups as appropriate." -- Zuul Wiki Page.

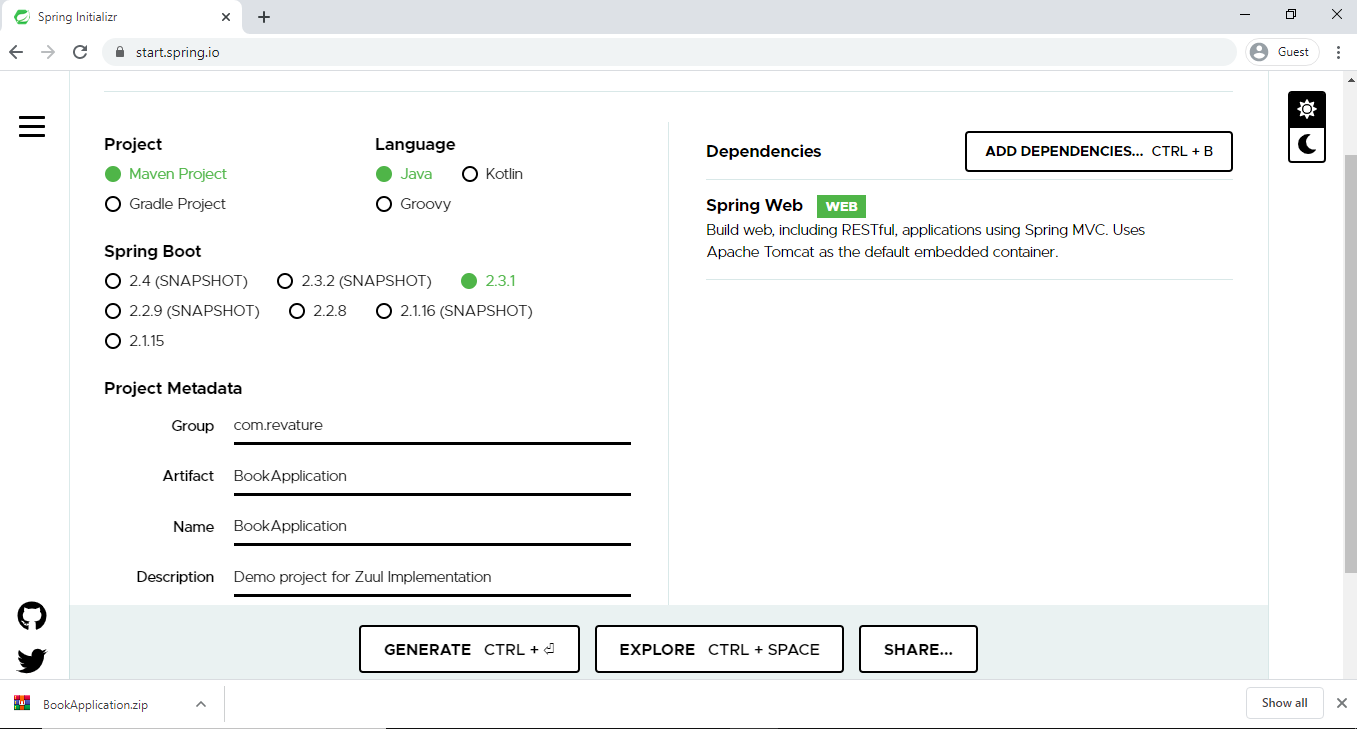
Zuul has mainly four types of filters that enable us to intercept the traffic in different timelines of the request processing for any particular transaction. We can add any number of filters for a particular url pattern.

* pre filters – executed before the request is routed.
* post filters – executed after the request has been routed.
* route filters – used to route the request.
* error filters – executed when an error occurs while handling the request.



Let's implement our own API Gateway with Spring Cloud and Zuul.

Let's create a service with a few REST endpoints for testing the proxy later. Navigate to [Spring Initializr](https://start.spring.io/) and create a Spring Boot Application with Spring Web dependency. The following image shows the Initializr set up for the Book application:



The preceding image shows the Initializr with Maven chosen as the build tool. It also shows values of com.revature and BookApplication as the Group and Artifact, respectively.

The BookApplication class marked with the @RestController annotation that return values from @RequestMapping methods - available() and checkedOut(). They handle requests to the /available and /checked-out paths, each of which returns the String name of a book.

@RestController

@SpringBootApplication

**public** **class** **RoutingAndFilteringBookApplication** {

@RequestMapping(value = "/available")

**public** **String** **available**() {

**return** "Spring in Action";

}

@RequestMapping(value = "/checked-out")

**public** **String** **checkedOut**() {

**return** "Spring Boot in Action";

}

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**RoutingAndFilteringBookApplication**.**class**, args);

}

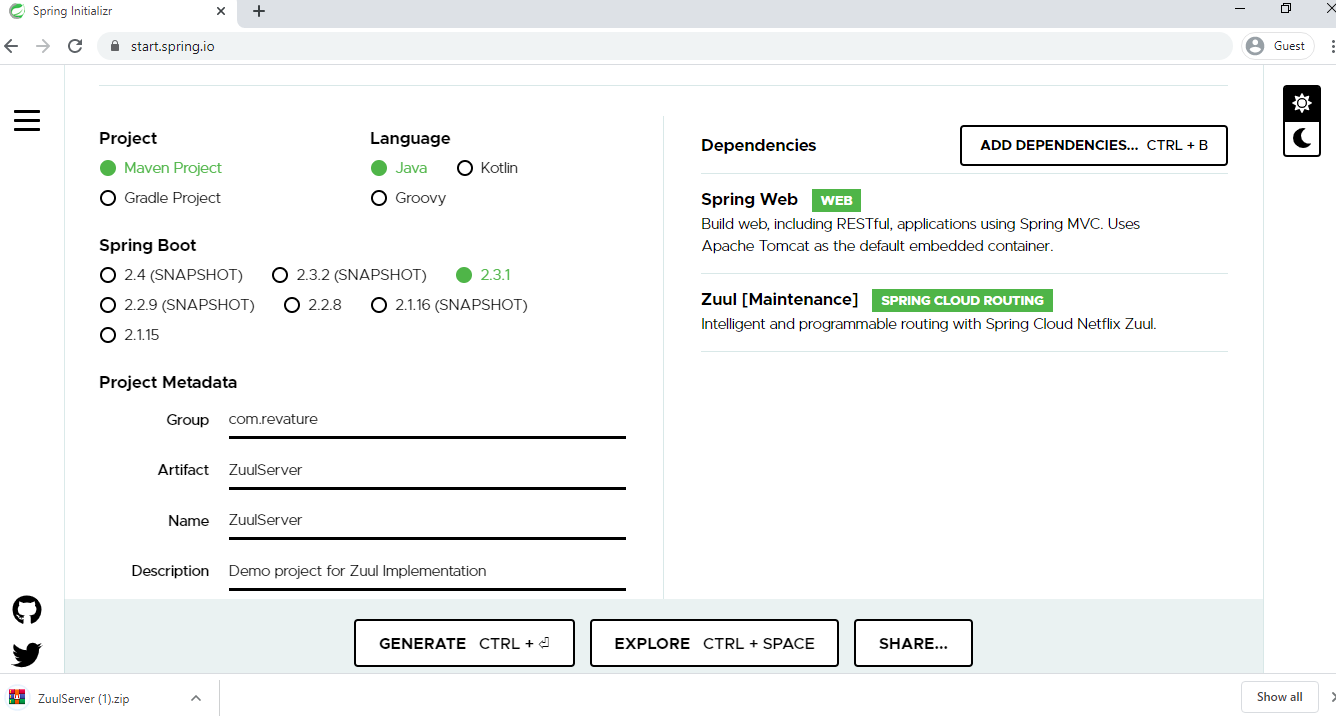
}

In an application.properties file, set the server port and the application name.

spring.application.name=book

server.port=8081

Let's create a Zuul gateway which will proxy the book service. Navigate to [Spring Initializr](https://start.spring.io/) and create a Spring Boot Application with Spring Web and Zuul dependencies.



The preceding image shows the Initializr with Maven chosen as the build tool. It also shows values of com.revature and ZuulServer as the Group and Artifact, respectively.

Enable the embedded Zuul proxy by annotating the class with @EnableZuulProxy which turns the Gateway application into a reverse proxy that forwards relevant calls to other services - such as our book application.

@EnableZuulProxy

@SpringBootApplication

**public** **class** **RoutingAndFilteringGatewayApplication** {

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**RoutingAndFilteringGatewayApplication**.**class**, args);

}

//Adding Zuul Filter class

@Bean

**public** **SimpleFilter** **simpleFilter**() {

**return** **new** **SimpleFilter**();

}

}

To forward requests from the Gateway application, we need to tell Zuul the routes that it should watch and the services to which to forward requests that are made to those routes. We specify routes by setting properties under zuul.routes. Each of our microservices can have an entry under zuul.routes.NAME, where NAME is the application name (as stored in the spring.application.name property).

In the application.properties file, set the zuul routes and server port.

zuul.routes.books.url=http://localhost:8081

ribbon.eureka.enabled=false

# explicitly disables the use of Eureka in Ribbon

server.port=8080

Spring Cloud [Netflix Zuul](https://github.com/Netflix/zuul/wiki) uses [Netflix’s Ribbon](https://github.com/Netflix/ribbon/wiki) to perform client-side load balancing. By default, Ribbon would use [Netflix Eureka](https://github.com/Netflix/eureka/wiki) for service discovery. For this simple example, we skip service discovery by setting ribbon.eureka.enabled to false.

Spring Cloud Netflix picks up, as a filter, any @Bean that extends com.netflix.zuul.ZuulFilter and is available in the application context. Here, we enable the pre filter.

**public** **class** **SimpleFilter** **extends** **ZuulFilter** {

**private** **static** **Logger** log = **LoggerFactory**.**getLogger**(**SimpleFilter**.**class**);

@Override

**public** **String** **filterType**() {

**return** "pre";

}

@Override

**public** **int** **filterOrder**() {

**return** 1;

}

@Override

**public** **boolean** **shouldFilter**() {

**return** true;

}

@Override

**public** **Object** **run**() {

**RequestContext** ctx = **RequestContext**.**getCurrentContext**();

**HttpServletRequest** request = ctx.**getRequest**();

log.**info**(**String**.**format**("%s request to %s", request.**getMethod**(), request.**getRequestURL**().**toString**()));

**return** null;

}

}

Filter classes implement four methods:

* filterType() - returns a string that stands for the type of the filter (pre,post, route, error)
* filterOrder() - gives the order in which this filter is to be run, relative to other filters.
* shouldFilter() - contains the logic that determines when to run this filter
* run()- contains the functionality of the filter.

Zuul filters store request and state information in the RequestContext. You can use that to get at the HttpServletRequest and then log the HTTP method and URL of the request before it is sent on its way.

Make sure that both applications are running. In a browser, visit one of the book application’s endpoints through the Gateway application (<https://localhost:8080/books/available> or <https://localhost:8080/books/checked-out>). The request’s method gets logged by the Gateway application before it is handed on to the Book application, as the following sample logging output shows:

2019-10-02 10:58:34.694 INFO 11608 --- [nio-8080-exec-4] c.e.r.filters.pre.SimpleFilter : GET request to http://localhost:8080/books/available

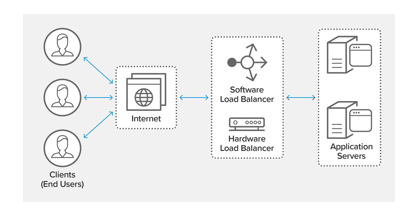
We access the book service endpoints through the Gateway service.

**Load balancing**

# **Load Balancing**

In a microservice architecture, multiple instances of each microservice are deployed for availability and scalability. This limits the impact of the failure of a single instance and maintains overall system reliability. Successfully adopting this architecture requires a **load balancing mechanism** to ensure that incoming requests are spread across all of the available instances, rather than overloading some instances at the same time, while other instances are under-utilized.

A **load balancer** acts as the traffic cop sitting in front of our servers, and routs client requests across all servers capable of fulfilling those requests. It does so in a manner that maximizes speed and capacity utilization, and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts sending requests to it.



[Netflix Ribbon](https://github.com/Netflix/ribbon/wiki) is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients.

### Ribbon — Load Balancing

When Zuul receives a request, it picks up one of the physical locations available and forwards requests to the actual service instance. Internally, Zuul uses Netflix Ribbon to lookup for all instances of the service from the service discovery (Eureka Server) and routes the external request to an appropriate service instance.

ribbon:

MaxAutoRetries: 1

MaxAutoRetriesNextServer: 2

OkToRetryOnAllOperations: true

By default Ribbon uses [round robin](https://en.wikipedia.org/wiki/Round-robin_scheduling) to lookup services available. If we want to handle load balancing ourselves, we can add the below properities in the Zuul Server

ribbon:

eureka:

enabled: false

listOfServers: localhost:8000,localhost:9092,localhost:9999

**Circuit breaker design pattern**

# **Circuit Breaker**

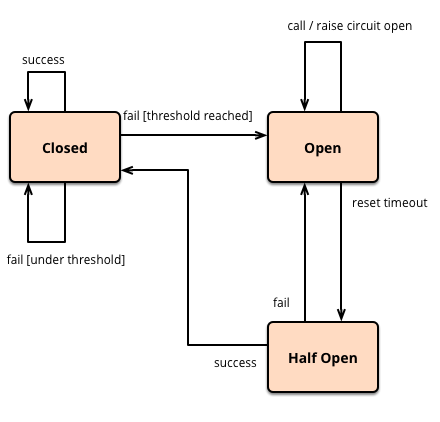
Services sometimes collaborate when handling requests. When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency that it's essentially unusable. Precious resources such as threads might be consumed in the caller 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.

To prevent a network or service failure from cascading to other services, we use the [circuit breaker pattern](https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern#:~:text=Circuit%20breaker%20is%20a%20design,failure%20or%20unexpected%20system%20difficulties.) for building a microservice application.

A circuit breaker wraps a function call with a monitor where the monitor will be tracking potential failures. If the service is in the failed stage, the circuit will send the error message without making a call. If the service is up, then breaker forwards the call to the needed service.

The circuit breaker has 3 distinct states –

1. **Closed State** - When service is up and running, the circuit breaker remains in the closed state and all calls pass through to the services.
2. **Open State** - When the number of failures exceeds a predetermined threshold the breaker trips, and it goes into the Open state. In the OPEN state the circuit breaker returns an error for all calls to the service without making the calls to the Supplier Microservice.
3. **Half-Open State** - The circuit breaker makes a trial call to the failed service periodically to check if it has recovered. If the call to the service times out, the circuit breaker remains in the Open state. If the call returns successfully, then the circuit switches to the closed state.



[Netflix's Hystrix](https://github.com/Netflix/Hystrix/wiki) library provides an implementation of the circuit breaker pattern. It helps to control the interaction between services by providing fault and latency tolerance. It improves overall resilience of the system by isolating failing services and halting the cascading effect of failures.

When you apply a circuit breaker to a method, Hystrix watches for failing calls to that method, and if failures build up to a threshold then Hystrix opens the circuit so that subsequent calls automatically fail. While the circuit is open, Hystrix redirects calls to the method, and they are passed to your specified fallback method.

### Example

Let's create a few microservice instances and test how the Hystrix works.

Create a client application (Bookstore) with only the Web dependency.

@RestController

@SpringBootApplication

**public** **class** **BookstoreApplication** {

@RequestMapping(value = "/recommended")

**public** **String** **readingList**() {

**return** "Spring in Action (Manning), Cloud Native Java (O'Reilly), Learning Spring Boot (Packt)";

}

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**BookstoreApplication**.**class**, args);

}

}

In application.properties file, set server.portto consume the Bookstore service.

server.port=8090

Create another application (Reading) with the Web and Hystrix dependencies.

@EnableHystrix

@RestController

@SpringBootApplication

**public** **class** **ReadingApplication** {

@Autowired

**private** **BookService** bookService;

@Bean

**public** **RestTemplate** **rest**(**RestTemplateBuilder** builder) {

**return** builder.**build**();

}

//To retrieve the list from the Bookstore service.

@RequestMapping("/to-read")

**public** **String** **toRead**() {

**return** bookService.**readingList**();

}

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**ReadingApplication**.**class**, args);

}

}

@EnableHystrix annotation tells Spring Cloud that the reading application uses circuit breaker (Hystrix) and to enable their monitoring, opening, and closing.

Spring Cloud Netflix Hystrix looks for any method annotated with the @HystrixCommand annotation and wraps that method in a proxy connected to a circuit breaker so that Hystrix can monitor it. This works only in a class marked with @Component or @Service.

Let's create a BookService class and mark with @Service which contains a method annotated with the @HystrixCommand annotation.

@Service

**public** **class** **BookService** {

**private** **final** **RestTemplate** restTemplate;

**public** **BookService**(**RestTemplate** rest) {

**this**.restTemplate = rest;

}

@HystrixCommand(fallbackMethod = "reliable")

**public** **String** **readingList**() {

URI uri = URI.**create**("http://localhost:8090/recommended");

**return** **this**.restTemplate.**getForObject**(uri, **String**.**class**);

}

**public** **String** **reliable**() {

**return** "Cloud Native Java (O'Reilly)";

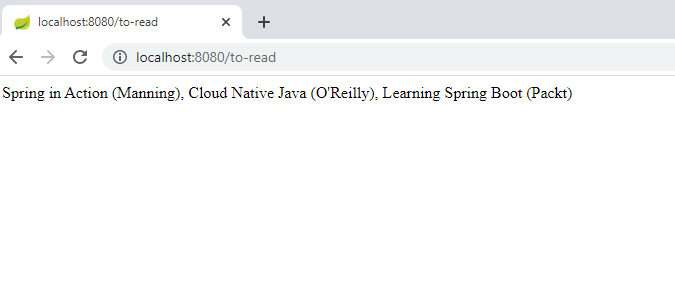
}

}

The RestTemplate is injected into the constructor of the BookService when it is created. Here, we have applied @HystrixCommand to our original readingList() method. Also, there is a new method here: reliable(). The @HystrixCommand annotation has reliable as its fallbackMethod. If, for some reason, Hystrix opens the circuit on readingList(), we have an excellent placeholder reading list ready for our users.

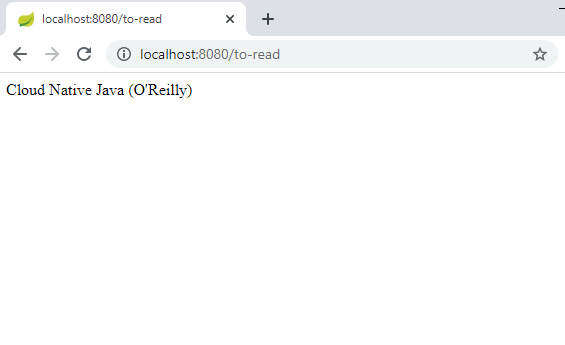
To test our circuit breaker, first let's run both the Bookstore service and the Reading service.

Then, visit localhost:8080/to-read:



Now, stop the bookstore application. Our list source is gone, but Hystrix loads the reliable abbreviated list to stand in the gap.

If we visit localhost:8080/to-read, we able to see the reliable abbreviated list:



**Messaging Queues**

# **Message Queues**

A message queue is a software component used for passing data (messages) between services. It exposes an interface for adding, reading, and removing messages. Generally speaking, messages are persisted.

Examples of Message Queues - [RabbitMQ](https://www.rabbitmq.com/), [Amazon SQS](https://aws.amazon.com/sqs/), and [Apache Kafka](https://kafka.apache.org/).

Message queues facilitate asynchronous communication between discrete services. There can be multiple **producers** (services adding messages to the queue) and **consumers** (services taking messages from the queue). The producers and consumers are decoupled in time and space; a producer need not know when or by whom a message will be consumed. This feature has the following benefits:

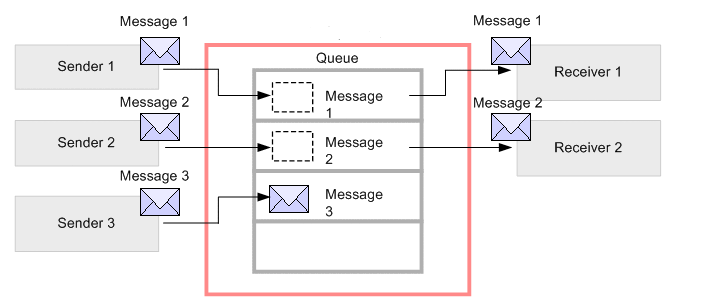
* **Resiliency** - Isolated failures stay isolated. A consumer can go down temporarily and begin processing messages when it comes back online, without affecting other services.
* **Scalability** - To increase the rate at which messages are added or processed, simply add more producers or consumers.
* **Visibility** - Examining the queue itself can provide valuable insight into the health of an application.

## Messaging Models

### Point-to-Point Messaging Model

In the Point-to-Point messaging model, the producer is called as a sender and the consumer is called as a receiver. The most important characteristics of the point-to-point model are as follows:

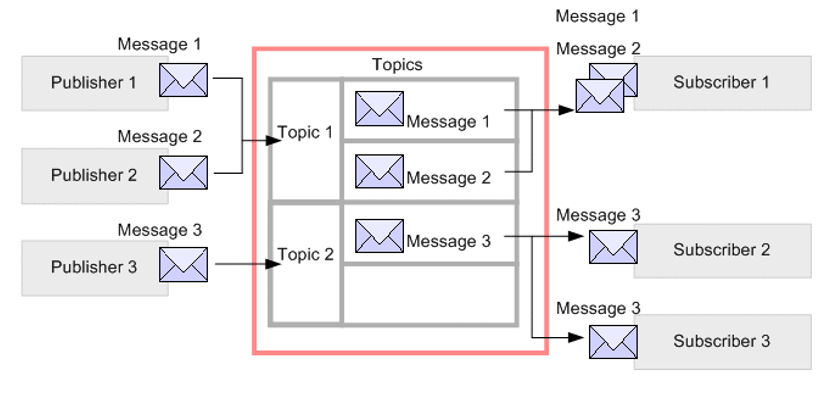
* Senders and Receivers are exchange messages through a virtual channel called a queue. A queue is a destination to which producers send messages and a source from which receivers consume messages.
* Each message is delivered to only one receiver. Multiple receivers may listen on a queue, but each message in the queue may only be consumed by one of the queue’s receivers.
* Messages are ordered. A queue delivers messages to destined receiver in the order they were placed in the queue by the message server. As messages are consumed, they are removed from the head of the queue (unless message priority is used).



The sender sends the messages and those messages are registered in the queue. The messages registered in the queue are delivered to destined receivers running at that time (message1 and message2). Then, the delivered messages are deleted/removed from the queue. If the destined receiver doesn't exist, the messages are accumulated in the queue (message3).

### Pub/Sub Messaging Model

The pub/sub messaging model allows a message producer (also called a publisher) to broadcast a message to one or more consumers (called subscribers). Publisher and Subscribers exchange means through an intermediary channel called as **topic**. **Publishers** produce messages to a topic and **Subscribers** subscribe to a topic and consume messages from a topic. Publishers doesn't have the knowledge about the subscribers.



A publisher creates and sends messages to a topic. Subscribers subscribe to a topic, to receive messages from it. Communication can be one-to-many, many-to-one, and many-to-many.

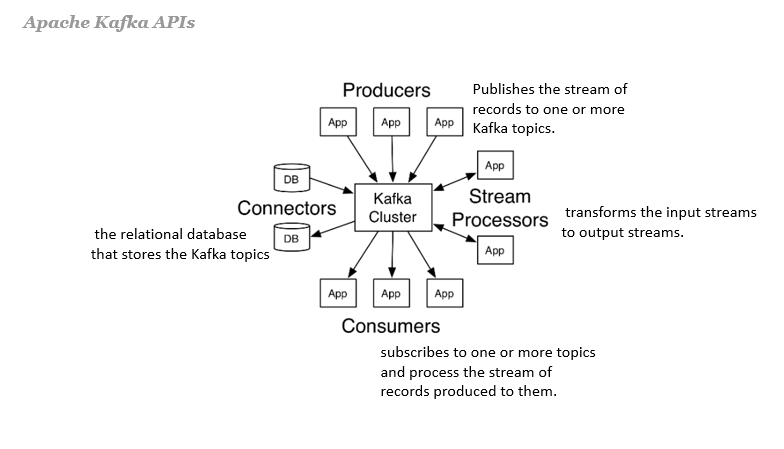
## Apache Kafka

We use Apache Kafka to enable communication between producers and consumers using message-based topics.

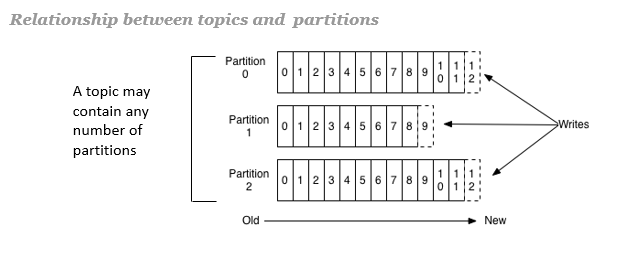
Apache Kafka is a distributed streaming platform, used to

* Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
* Store streams of records in a fault-tolerant durable way.
* Process streams of records as they occur.

Kafka run as a cluster on one or more servers that can span multiple datacenters. The **Kafka cluster** stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp.



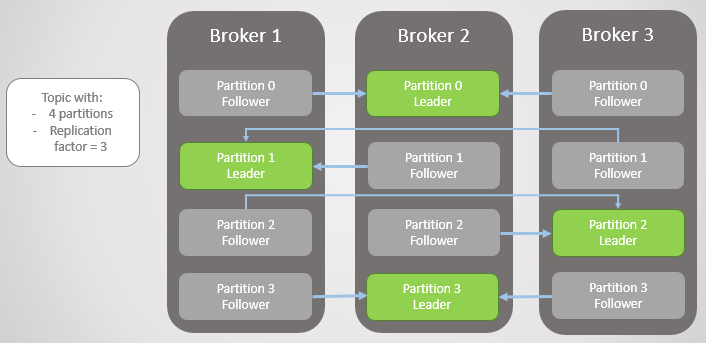
* **Kafka Broker** - A Kafka broker (server) handles all requests from clients (produce, consume, and metadata) and persists and replicates the data within the cluster. There can be one or more brokers in a cluster. For the purpose of managing and coordinating, Kafka broker uses **ZooKeeper**. Also, uses it to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system.
* **Kafka Topic** - A Topic is a category/feed name to which records are stored and published.
* A **Kafka Producer** pushes the message into the Kafka Topic. A **Kafka Consumer** pulls the message from the Kafka Topic.
* **Partitions in Kafka** - Kafka topics are divided into a number of partitions, which contain immutable sequence of records records. Each record in a partition is assigned and identified by its unique offset.



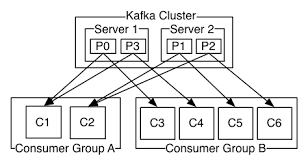
A topic can also have **multiple partition logs** distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. This allows multiple consumers to read from a topic in parallel.

In Kafka, **replication** is implemented at the partition level. Each partition is replicated across a configurable number of servers for fault tolerance.

Each partition has one server which acts as the **leader** and zero or more servers which act as **followers**. The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.



* **Consumer group** - Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.



### Apache Kafka as a Messaging System

Messaging traditionally has two models: queuing and publish-subscribe

* **Queuing** or **Point to Point**- In a queue, a pool of Kafka consumers may read from a server. Also, each record goes to one of them here.
  + Strength - It permits us to divide up the processing of data over multiple consumer instances, that help us scale our processing.
  + Weakness - It is not multi-subscriber, as soon as one consumer reads the data it’s gone.
* **Publish-Subscribe** - Publish-Subscribe allows us broadcast data to multiple consumer/subscriber, but has no way of scaling processing since every message goes to every subscriber.

In Kafka, these two concepts are generalized by the Kafka consumer group. As with a queue the consumer group allows us to divide up processing over a collection of processes (the members of the consumer group). As with publish-subscribe, Kafka allows us to broadcast messages to multiple consumer groups.

The advantage of Kafka's model is that every topic has both these properties: — it can scale processing and is also multi-subscriber — there is no need to choose one or the other.

Kafka has stronger ordering guarantees than a traditional messaging system, too. There is no parallelism in the processing in the traditional system, Kafka performs it well with the notion of parallelism. Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes.

### Spring for Apache Kafka

The Spring for Apache Kafka project applies core Spring concepts to the development of Kafka-based messaging solutions. It provides a "template" as a high-level abstraction for sending messages. It also provides support for Message-driven POJOs with @KafkaListener annotations and a "listener container".

We add the below maven dependency to get started with Spring Boot and Kafka

<!-- https://mvnrepository.com/artifact/org.springframework.kafka/spring-kafka -->

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

<version>2.5.2.RELEASE</version>

</dependency>

**Features:**

* [KafkaTemplate](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaTemplate.html) - A template for executing high-level operations.
* [KafkaMessageListenerContainer](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/KafkaMessageListenerContainer.html) - used to construct an instance with the supplied configuration properties.
* [@KafkaListener](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/annotation/KafkaListener.html) - Annotation that marks a method to be the target of a Kafka message listener on the specified topics.

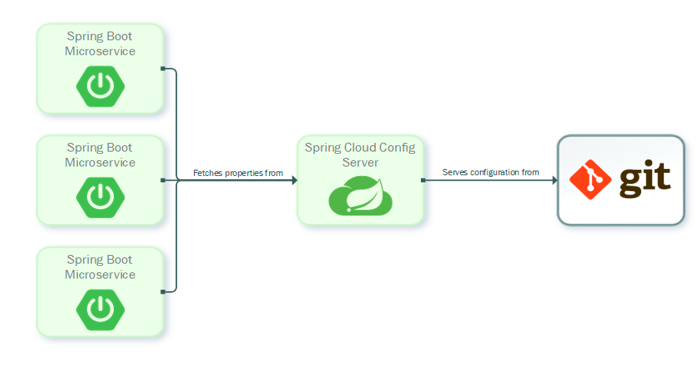
**Config Server**

# **Configuration Servers**

In the cloud, configuration can’t simply be embedded inside the application. The configuration has to be flexible enough to cope with multiple applications, environments, and service instances, as well as deal with dynamic changes without downtime.

[Spring Cloud Config](https://spring.io/projects/spring-cloud-config) is designed to ease these burdens and offers integration with version control systems like Git to help you keep our configuration safe.

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With Spring Cloud Config Server, we have a central place to manage external properties for applications across all environments. Services can consume their application properties from the Config Server rather than loading them locally from the file system or classpath. Configuration is not stored in the Config Server itself but pulled from a Git repository. This allows us to manage our application configuration with all the benefits of version control.



***Spring Cloud Config Server*** - Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the @EnableConfigServer annotation.

@SpringBootApplication

@EnableConfigServer

**public** **class** **ConfigServer** {

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**ConfigServer**.**class**, args);

}

}

We can also set a default configuration git repository in application.properties file:

server.port: 8888

spring.cloud.config.server.git.uri: file://${user.home}/config-repo

The Config Server stores the configuration data with the help of EnvironmentRepository and serves a Environment object to the client. The Environment resources are parametrized by three variables:

* {application} - maps to spring.application.name on the client side.
* {profile} - maps to spring.profiles.active on the client.
* {label} - it is a server side feature labelling a "versioned" set of config files.

***Spring Cloud Config Client*** - Spring Cloud Config Client binds to the Config Server and initialize Spring Environment with remote property sources. All client applications that want to consume the Config Server need a bootstrap.yml (or an environment variable) with the server address set in spring.cloud.config.uri (it defaults to "[http://localhost:8888](http://localhost:8888/)"). The client application has the following bootstrap configuration in the bootstrap.yml file:

spring:

application:

name: foo

profiles:

active: dev,mysql

Day -3

**Containerization & orchestration**

Containerization

When we move software from a developer's laptop to a test environment, from a staging environment into production, and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud, problems may arise when the supporting software environment is not identical.

For example, You're going to test software using Python 2.7, and then it's going to run on Python 3 in production and something weird will happen. Or you'll rely on the behavior of a certain version of an SSL library and a future update changes that behavior. You'll run your tests on Debian and production is on Red Hat, and all sorts of weird things happen.

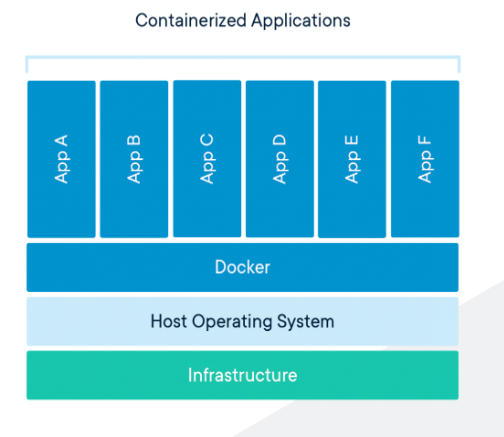
It's not just different software that can cause problems. If there is any difference in the network topology or the security policies and storage, these may also cause problems.

**Containers** are a solution to this problem. A container is a standard unit of software that packages up an application and all its dependencies so the application runs quickly and reliably from one computing environment to another.

**Containerization** is the process of packaging an application along with its required libraries, frameworks, and configuration files together so that it can be run in various computing environments efficiently. In simpler terms, containerization is the encapsulation of an application and its required environment.

[Docker](https://www.docker.com/) is one of the world's leading software containerization platforms. It encapsulates our microservice into a **Docker container** which can then be independently maintained and deployed. Each of these containers will be responsible for one specific business functionality.

With Docker, we can make our application independent of the host environment. Since we use a microservice architecture, we can now encapsulate each microservice in separate Docker containers. Docker containers are lightweight, resource-isolated environments through which we can build, maintain, ship, and deploy our application.



You can find a detailed explanation of Docker [here](https://gitlab.com/revature_training/docker-team).

References

* [Containerization - Video Tutorial](https://www.youtube.com/watch?v=0qotVMX-J5s)
* [Docker - Content Library](https://www.docker.com/resources)
* [Containerization explained: what it is, benefits and applications](https://www.masterdc.com/blog/what-is-containerization-benefits-explained/)
* [Docker - Video Tutorial for Beginners](https://www.youtube.com/watch?v=fqMOX6JJhGo)

Container Orchestration

Container orchestration automates the deployment, management, scaling, and networking of containers.

Container orchestration tools provide a framework for managing containers and microservice architectures at scale. There are many container orchestration tools that can be used for container lifecycle management. Some popular options are [Kubernetes](https://kubernetes.io/), [Docker Swarm](https://docs.docker.com/engine/swarm/swarm-tutorial/), and [Apache Mesos](http://mesos.apache.org/).

**Kubernetes** orchestration allows us to build application services that span multiple containers, schedule containers across a cluster, scale those containers, and manage their health over time.

Kubernetes eliminates many of the manual processes involved in deploying and scaling containerized applications. We can cluster together groups of hosts, either physical or virtual machines, running Linux containers, and Kubernetes gives you the platform to easily and efficiently manage those clusters. A **Kubernetes cluster** is a set of node machines for running containerized applications.

**Distributed tracing**

# **Distributed Tracing**

One of the challenges in a microservice architecture is the ability to debug issues. A user action might trigger a chain of downstream microservice calls. IT and DevOps teams have to use distributed tracing to monitor applications and trace the logs related to a particular user action across microservices, which can be useful for troubleshooting any issue.

Distributed Tracing tells the story of an end-to-end request, including everything from mobile performance to database health.

Distributed tracing provides a solution for describing and analyzing cross-process transactions. Some of the use cases of distributed tracing as described in [Google’s Dapper paper](https://research.google/pubs/pub36356/) include anomaly detection, diagnosing steady state problems, distributed profiling, resource attribution, and workload modeling of microservices.

A **trace** begins when a user sends an initial request to an entry point of our application. As requests travel between services, each segment is recorded as a **span**, which represents time spent in services or resources of those services. Every span is tagged with a unique ID, a distributed tracing engine can identify and analyze all of the data correlated to the original request. All the spans of a request are combined into a single distributed trace to give us a picture of an entire request.

Some distributed tracing tools are:

* [Zipkin](https://zipkin.io/) - Zipkin is a distributed tracing system. It is an open source project that provides mechanisms for sending, receiving, storing, and visualizing traces. Also allows us to correlate activity between servers and get a much clearer picture of exactly what is happening in our services.
* [Jaeger](https://www.jaegertracing.io/) - Jaeger uses distributed tracing to follow the path of a request through different microservices. Jaeger includes tools to monitor distributed transactions, optimize performance and latency, and perform [root cause analysis](https://en.wikipedia.org/wiki/Root_cause_analysis) (RCA), a method of problem solving.
* [Prometheus](https://prometheus.io/) - Prometheus is a open-source software system used for event monitoring and alerting.
* [OpenTelemetry](https://opentelemetry.io/) - [OpenTracing](https://opentracing.io/" \t "_blank)and [OpenCensus](https://opencensus.io/" \t "_blank) have merged to form [OpenTelemetry](https://opentelemetry.io/" \t "_blank). OpenTracing offers a vendor-neutral API for adding tracing to applications and delivering that data into distributed tracing systems. OpenCensus allow us to collect application metrics and distributed traces. OpenTelemetry provides a single set of APIs, libraries, agents, and collector services to capture distributed traces and metrics from your application. You can analyze them using [Prometheus](https://prometheus.io/), [Jaeger](https://www.jaegertracing.io/), and other [observability](https://lightstep.com/observability/) tools.

[Spring Cloud Sleuth](https://cloud.spring.io/spring-cloud-sleuth/reference/html/) provides Spring Boot auto-configuration for distributed tracing. It is used to generate and attach the trace ID, span ID to the logs so that these can then be used by tools like [Zipkin](https://app.revature.com/(https:/zipkin.io/)" \t "_blank) and [ELK stack](https://aws.amazon.com/elasticsearch-service/the-elk-stack/) for storage and analysis.

**Monitoring tools**

# **Monitoring Tools**

Monitoring is a process of reporting, gathering and storing data. Monitoring Microservices helps the organizations to -

* Understand the overall health of the application.
* Glean insight into the performance of each individual service that makes up an application.
* Ensure the API transactions are available and performing well.
* Isolate problematic transactions and endpoints.
* Optimize the end-user experience.

Some of the monitoring tools are:

* **Zipkin** - [Zipkin](https://zipkin.io/?utm_source=thenewstack&utm_medium=website" \t "_blank) is an open-source tracing system designed specifically to trace calls between microservices. It is especially useful for analyzing latency problems.
* **Grafana** - [Grafana](https://grafana.com/) is a multi-platform open source analytics and interactive visualization web application.
* **Prometheus** – [Prometheus](https://prometheus.io/) is an open-source software used for event monitoring and alerting. It is widely used to store and query “time-series data,” which is data that describes actions over time. Prometheus is often combined with other tools, especially [Grafana](https://grafana.com/), to visualize the time series data and to provide dashboards.
* **CAdvisor** – [CAdvisor](https://github.com/google/cadvisor" \t "_blank) is an open source tool to monitor Kubernetes resource usage and performance.
* **Sensu** - [Sensu](https://sensu.io/" \t "_blank) is an open source infrastructure and application monitoring solution that monitors servers, services, and application health, and sends alerts and notifications with third-party integration.
* **Nagios** - [Nagios](https://www.nagios.org/) is an open source software tool for [continuous monitoring](https://en.wikipedia.org/wiki/Continuous_monitoring#:~:text=Continuous%20monitoring%20is%20the%20process,support%20efficient%20and%20effective%20operations.). It helps you to monitor a system, network, and infrastructure. It is used for continuous monitoring of systems, applications, service and business process in a DevOps culture.