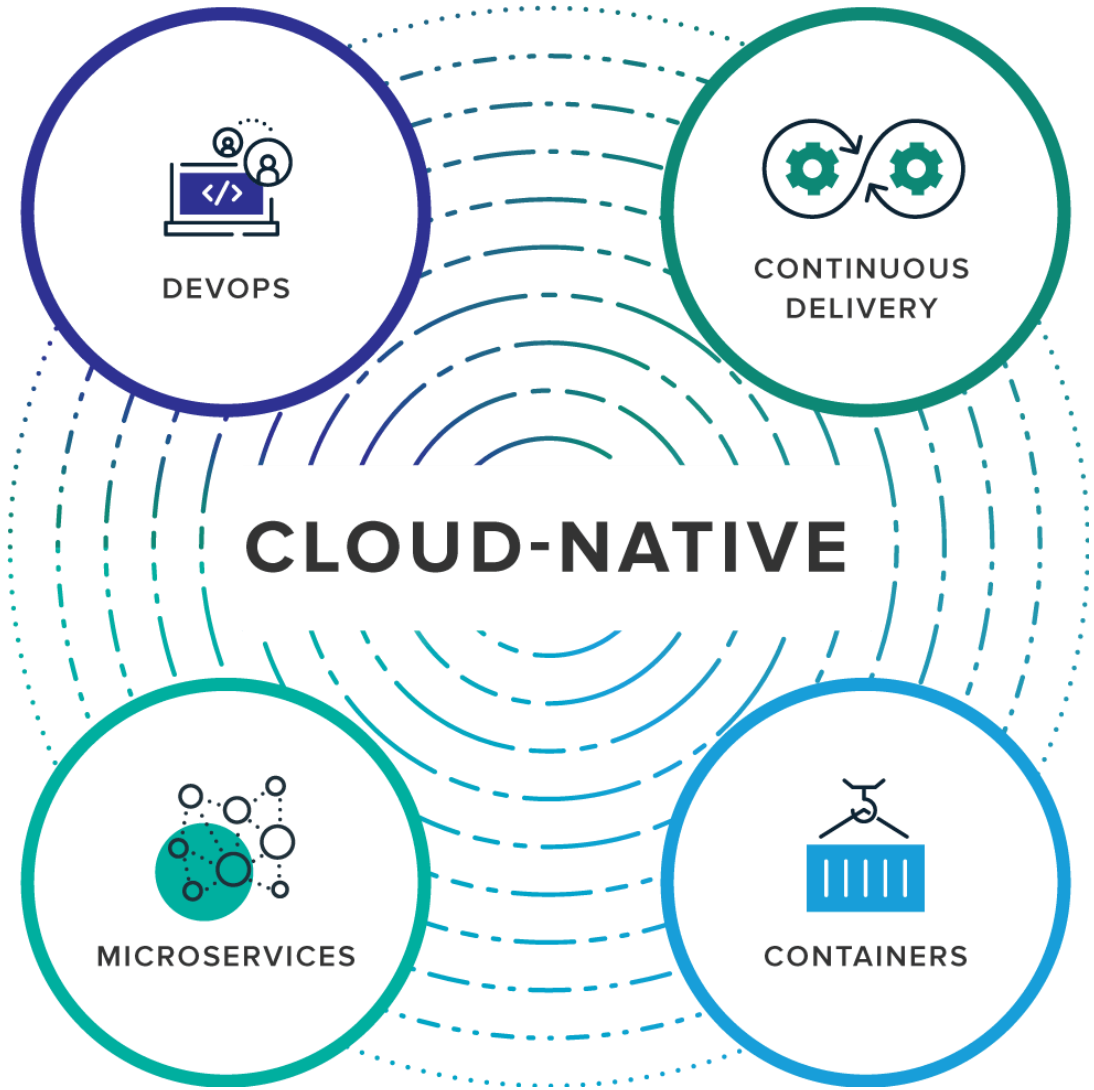
Native Cloud Applications (NCA)

- **NCA** is an approach to building and running applications that fully exploits the advantages of the cloud computing delivery model.
- It is designed specifically for a cloud computing architecture.
- Take advantage of cloud computing frameworks and are composed of loosely-coupled cloud services
- Cloud-native is about how applications are created and deployed, not where.

Transforming organization capability to develop software by leveraging micro-service architecture, agile and Business Driven Development (BDD), DevOps, containerization, and Platform-as-a-Service (PaaS).

- **Micro-services**: Accelerating time-to-market with smaller and logical units of deployment
- **Containerization**: Quickly move, enhance and scale applications with independent containers enabling modular development and deployment
- **Agile Development:** Continuous Delivery, enabled by Agile product development practices, is all about shipping small batches of software to production constantly, through automation
- **DevOps**: Changing organizational culture, skills and approaches in a move toward end-to-end automation and continuous deployment


DEVOPS
CONTINUOUS DELIVERY
CLOUD-NATIVE
MICROSERVICES
CONTAINERS

Native Cloud Applications

NCA is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development.

In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service**. The twelve-factor app is a methodology for building software-as-a-service apps.**

Spring Cloud facilitates these styles of development in a number of specific ways and the starting point is a set of features that all components in a distributed system either need or need easy access to when required.

The twelve-factor app is a methodology (https://12factor.net/)
for building software-as-a-service apps that:

- ❑ Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
- ❑ Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
- ❑ Are suitable for deployment on modern cloud platforms, remove the need for servers and systems administration;
- ❑ Minimize divergence between development and production, enabling continuous deployment for maximum agility;
- ❑ And can scale up without significant changes to tooling, architecture, or development practices.

Note : The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Heroku: Cloud Application Platform

The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and indirectly witnessed the development, operation, and scaling of hundreds of thousands of apps via our work on the Heroku platform.

Heroku is a cloud-based development platform as a service (PaaS) provider. The Heroku platform supports development in Ruby on Rails, Java, Node.js, Python, Scala and Clojure

The Twelve Factors: (for building software-as-a-service)

I. **Codebase**
One codebase tracked in revision control, many deploys

II. **Dependencies**
Explicitly declare and isolate dependencies

III. **Config**
Store config in the environment

IV. **Backing services**
Treat backing services as attached resources

V. **Build, release, run**
Strictly separate build and run stages

VI. **Processes**
Execute the app as one or more stateless processes

VII. **Port binding**
Export services via port binding

VIII. **Concurrency**
Scale out via the process model

IX. **Disposability**
Maximize robustness with fast startup and graceful shutdown

X**. Dev/prod parity**
Keep development, staging, and production as similar as possible

XI. **Logs**
Treat logs as event streams

XII. **Admin processes**
Run admin/management tasks as one-off processes

Characteristics of Microservices

Decentralized

Microservices architectures are distributed systems with decentralized data management.

They don’t rely on a unifying schema in a central database.

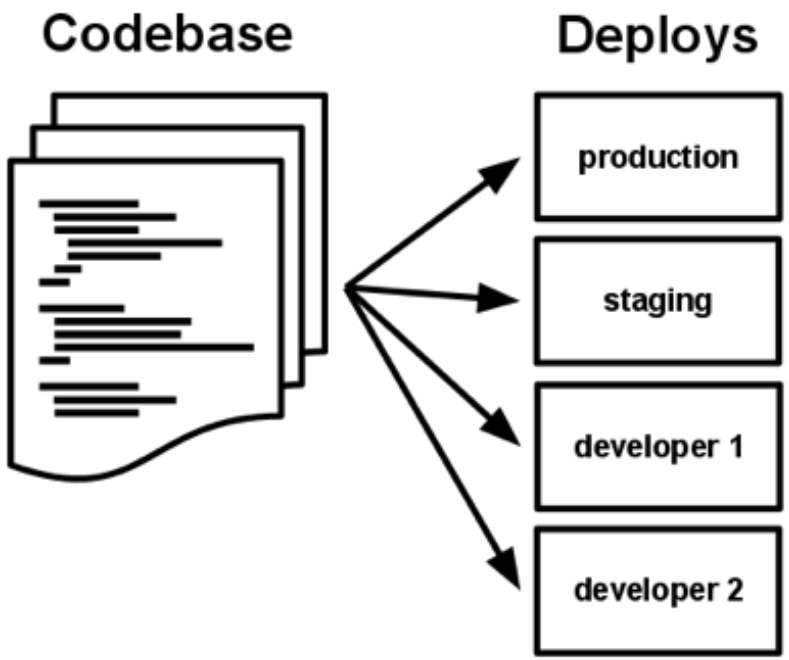
Each microservice has its own view on data models.

Microservices are also decentralized in the way they are developed, deployed, managed, and operated.

I. Codebase
One codebase tracked in revision control, many deploys

A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a code repository, often shortened to code repo or just repo.

A codebase is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).



II. Dependencies
Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as Maven, Gradle etc.,

III. Config
Store config in the environment

An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc).

This includes:

Resource handles to the database, Memcached, and other backing services

Credentials to external services such as Amazon S3 or Twitter

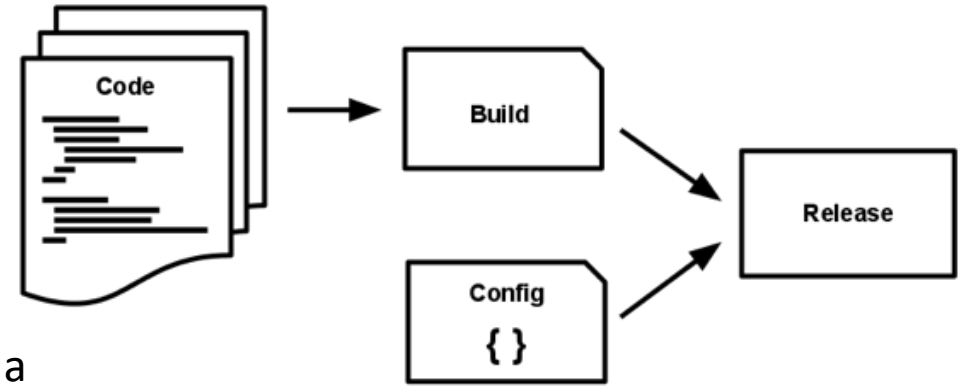
Per-deploy values such as the canonical hostname for the deploy

IV. Backing services
Treat backing services as attached resources

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or MongoDB), messaging/queueing systems (such as RabbitMQ or ActiveMQ) and caching systems (such as Memcached,ehcache).

V. Build, release, run
Strictly separate build and run stages

A codebase is transformed into a deploy through three stages:



The build stage is a transform which converts a code repo into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors dependencies and compiles binaries and assets.

The release stage takes the build produced by the build stage and combines it with the deploy's current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.

The run stage (also known as "runtime") runs the app in the execution environment, by launching some set of the app's processes against a selected release.

VI. Processes
Execute the app as one or more stateless processes

The app is executed in the execution environment as one or more processes.

VII. Port binding
Export services via port binding

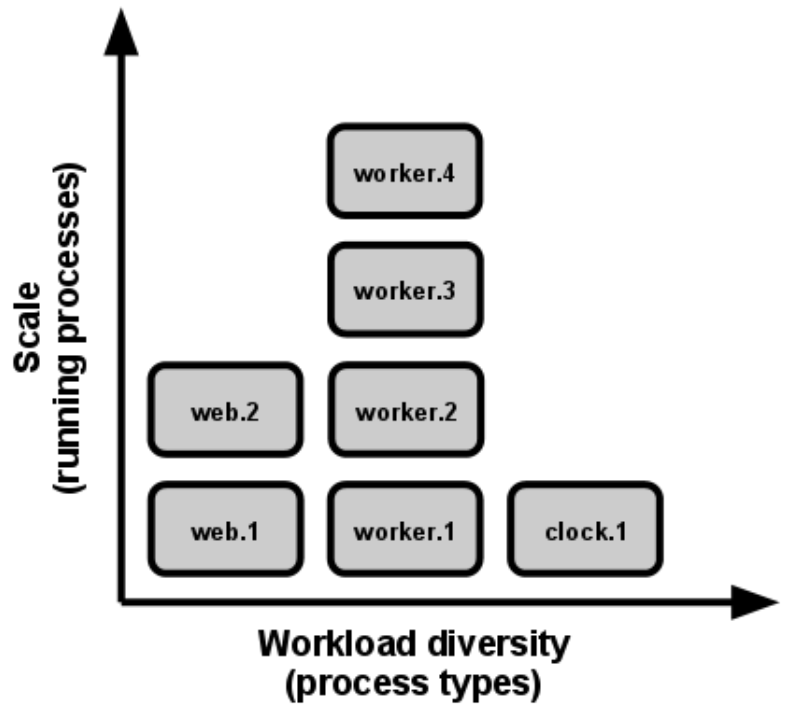
The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service.

The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port

VIII. Concurrency
Scale out via the process model

In the twelve-factor app, processes are a first class citizen.

Processes in the twelve-factor app take strong cues from the unix process model for running service daemons.



IX. Disposability
Maximize robustness with fast startup and graceful shutdown

The twelve-factor app’s processes are disposable, meaning they can be started or stopped at a moment’s notice.

This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys

X. Dev/prod parity
Keep development, staging, and production as similar as possible

The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.

Looking at the three gaps described above:

Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.

Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behaviour in production.

Make the tools gap small: keep development and production as similar as possible.

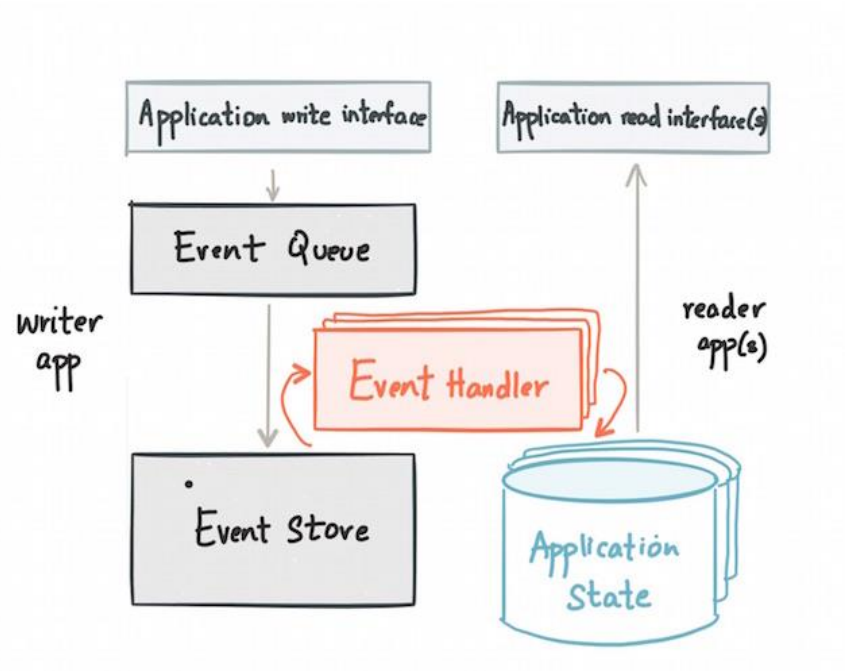
Dev/prod parity

	Traditional app	Twelve-factor app
Time between deploys	Weeks	Hours
Code authors vs code deployers	Different people	Same people
Dev vs production environments	Divergent	As similar as possible

XI. Logs
Treat logs as event streams (ELK stack)

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles.

Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app's behaviour.



XII. Admin processes
Run admin/management tasks as one-off processes

Twelve-factor strongly favors languages which provide a REPL(A *Read–Eval–Print Loop*) shell out of the box, and which make it easy to run one-off scripts.

In a local deploy, developers invoke one-off admin processes by a direct shell command inside the app's checkout directory.

In a production deploy, developers can use ssh or other remote command execution mechanism provided by that deploy's execution environment to run such a process.

Microservice Development

Independent

Different components in a microservices architecture can be changed, upgraded, or replaced independently without affecting the functioning of other components.

Similarly, the teams responsible for different microservices are enabled to act independently from each other.

Do one thing well

Each microservice component is designed for a set of capabilities and focuses on a specific domain.

If developers contribute so much code to a particular component of a service that the component reaches a certain level of complexity, then the service could be split into two or more services.

Polyglot

Microservices architectures don’t follow a “one size fits all” approach.

Teams have the freedom to choose the best tool for their specific problems.

As a consequence, microservices architectures take a heterogeneous approach to operating systems, programming languages, data stores, and tools. This approach is called polyglot persistence and programming.

Black box

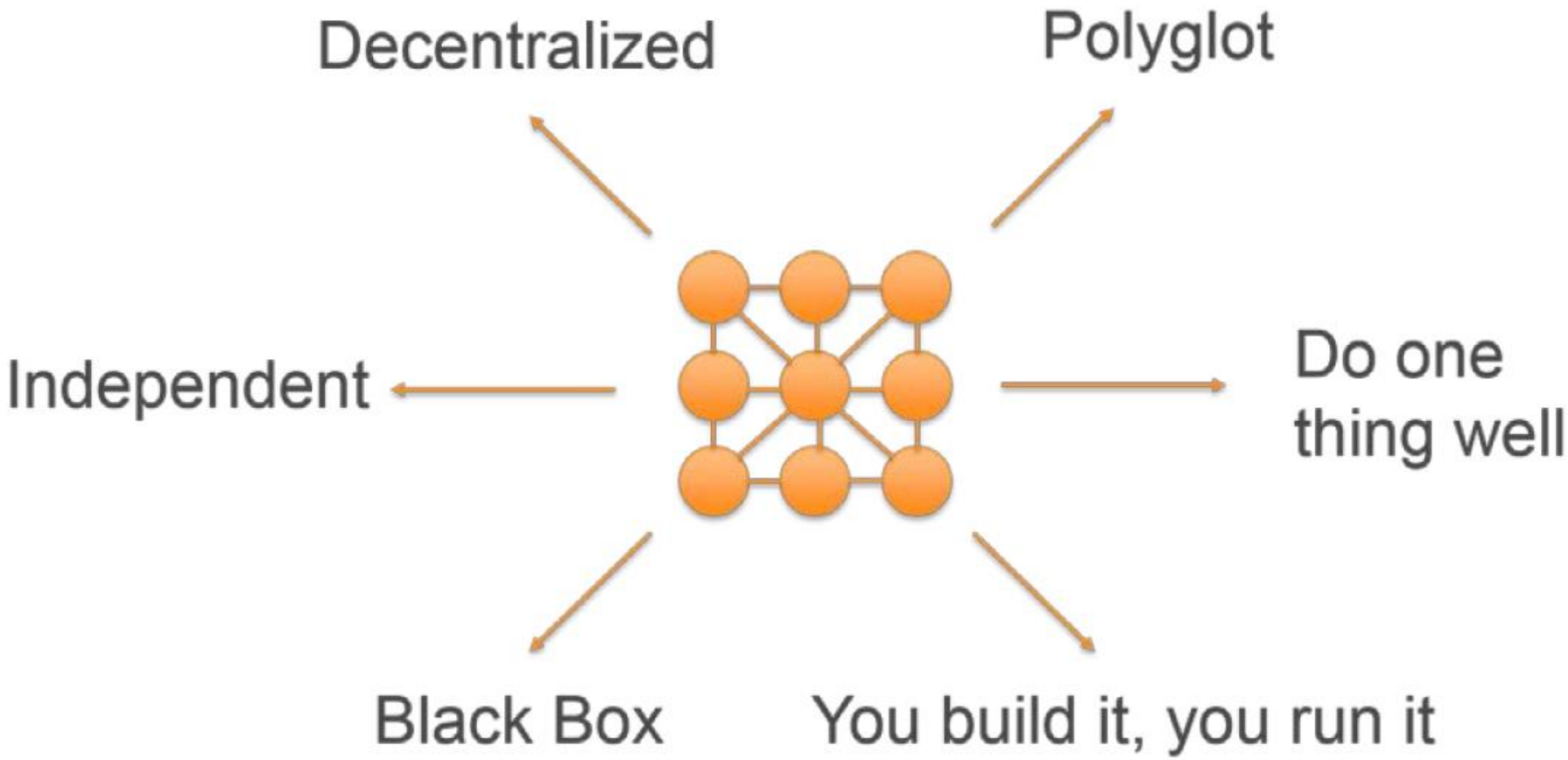
Individual microservice components are designed as black boxes, that is, they hide the details of their complexity from other components.

Any communication between services happens via well-defined APIs to prevent implicit and hidden dependencies.

You build it; you run it

Typically, the team responsible for building a service is also responsible for operating and maintaining it in production.

This principle is also known as DevOps


Decentralized
Polyglot
Independent
Do one thing well
Black Box
You build it, you run it

Benefits of Microservices

Microservices are adopted to address limitations and challenges with agility and scalability that they experience in traditional monolithic deployments.

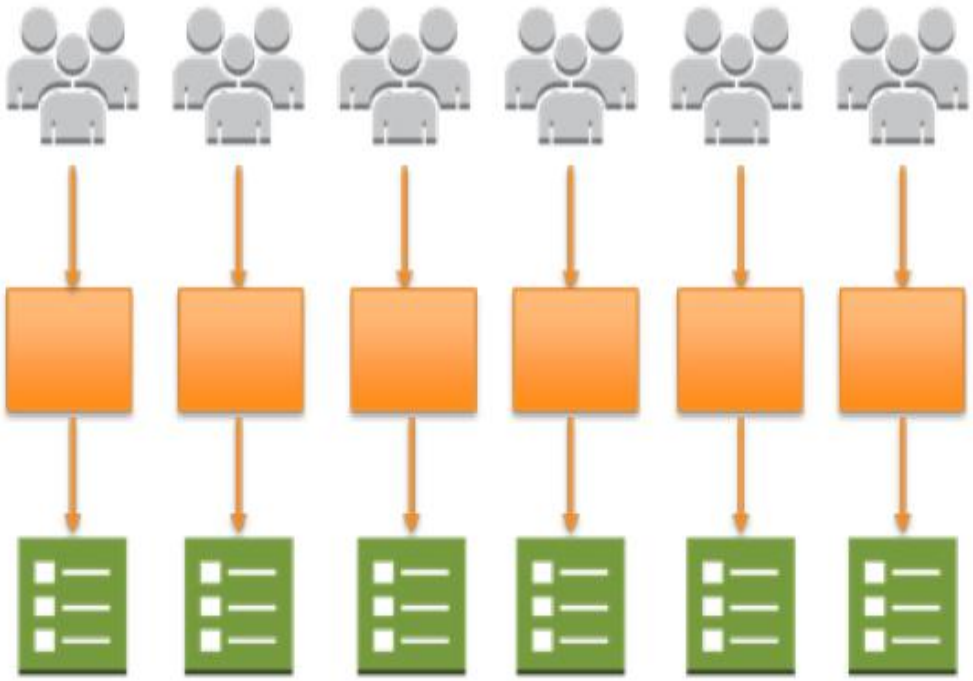
Agility

Microservices foster an organization of small independent teams that take ownership of their services.

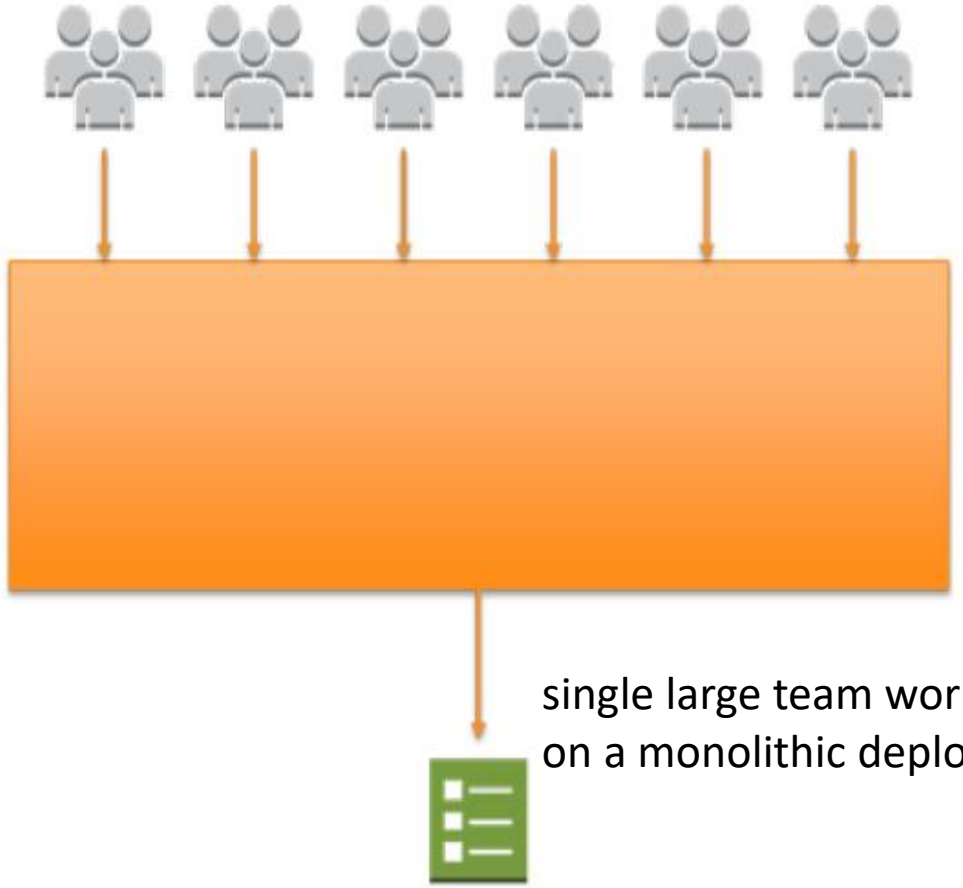
Teams act within a small and well-understood bounded context, and they are empowered to work independently and quickly, thus shortening cycle times.

We benefit significantly from the aggregate throughput of the organization.

Two types of deployment structures:



small independent teams working
on many deployments



single large team working
on a monolithic deployment.

Architectural Complexity

In monolithic architectures, the complexity and the number of dependencies reside inside the code base, while in microservices architectures complexity moves to the interactions of the individual services

Microservice decisions

Microservice decisions

L
SEP

Before implementing a microservices-based application development strategy, we must first evaluate your business readiness and make the necessary business-level decisions to ensure the long-term success of the project.

Compared to the traditional monolithic approach, a microservices strategy involves several different forms of business investment, including financial investments, an investment in the culture of your workplace, and an investment in new development and operations.

Financial investments

Financially, creating a microservices environment will require costs related to the up front architecture and design effort that is required to build many discrete, loosely coupled, asynchronous services that are resilient, scalable, and easy to monitor and manage.

Additionally, we must invest in unit testing, integration testing, and full deployment automation of these microservices.

However, as long-term testing and maintenance costs are reduced.

Monoliths and microservices are both valid application architecture styles, with each style presenting pros and cons.

To proceed with a microservices style, we need to measure these benefits and costs in the context of the specific project or application.

The microservices approach is all about handling a complex system, but in order to do so, the approach introduces its own set of complexity.

When we use microservices we have to work on automated deployment, monitoring, dealing with failure, eventual consistency, and other factors that a distributed system introduces.

There are well-known ways to cope with all this, but it's extra effort.

We have existing monolithic applications that have already proven themselves to be too large and complex to manage.

In this case, we are building a business case to solve a known problem.

However, for new or existing applications that don't yet have this problem but will face significant growth or increased mission criticality, the business case needs to demonstrate the anticipated complexities of:

- ❑ improving resiliency,
- ❑ improving availability,
- ❑ increasing the frequency of deployment & testing
- ❑ managing the complexity of large development teams,
- ❑ and the complexities of achieving deployment automation,
- ❑ increased monitoring with a monolithic design.

The business case, therefore, needs to consider these long-term productivity and agility dividends against the up front costs.

How large and complex is your application?

Large and complex applications tend to be good candidates for microservices development and refactoring.

When identifying and ranking applications by complexity consider the following criteria:

• Look for a gap between how many deployments are performed every year versus how many are desired by the business and application teams. Complex applications typically bundle a large number of changes across a small number of change windows. This decreases agility, generally increases risk, and typically dramatically increases testing costs.

• How large is the testing effort when a change is rolled out? Testing effort is typically directly correlated with complexity.


NETFLIX
OSS

Netflix OSS is a set of frameworks and libraries that Netflix wrote to solve some interesting distributed-systems problems at scale.

Today, for Java developers, it's pretty synonymous with developing microservices in a cloud environment.

Patterns for service discovery, load balancing, fault-tolerance, etc are incredibly important concepts for scalable distributed systems and Netflix brings nice solutions for these.

Spring Cloud Netflix

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus)

Features

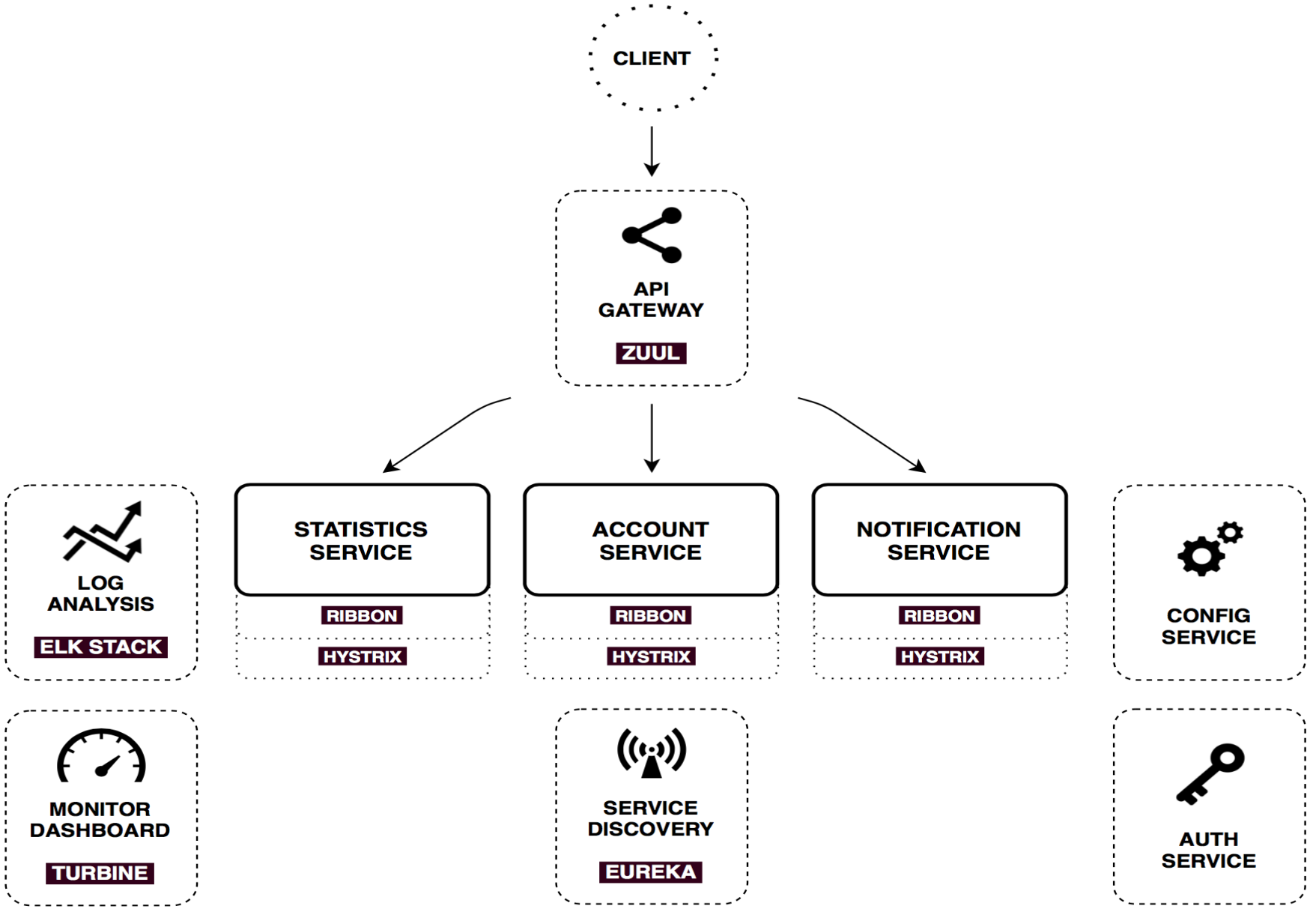
Spring Cloud main Features:

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Distributed messaging

Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.

The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon)..



Tracing System

(Zipkin)

Service Discovery:

Eureka instances can be registered and clients can discover the instances using Spring-managed beans

Circuit Breaker:

Hystrix clients can be built with a simple annotation-driven method decorator
Embedded Hystrix dashboard with declarative Java configuration

Declarative REST Client:

Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations

Client Side Load Balancer:

Ribbon

External Configuration:

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.

Router and Filter:

Automatic registration of **Zuul** filters, and a simple convention over configuration approach to reverse proxy creation

Eureka Service Registry

@EnableEurekaServer

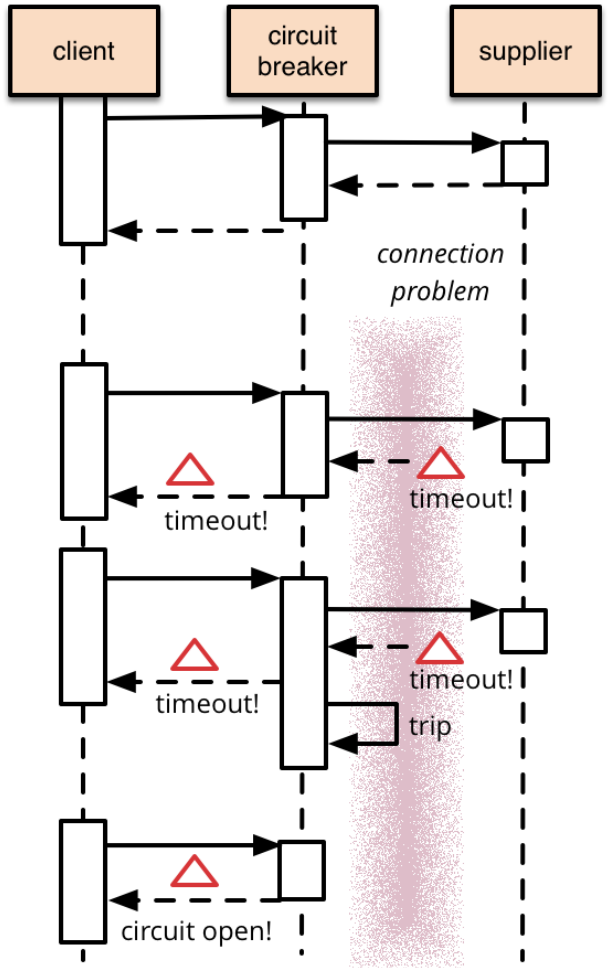
Spring Cloud's @EnableEurekaServer to standup a registry that other applications can talk to. This is a regular Spring Boot application with one annotation added to enable the service registry.

@EnableDiscoveryClient activates the Netflix Eureka DiscoveryClient implementation. Discovery client that both registers itself with the registry and uses the Spring Cloud DiscoveryClient abstraction to interrogate the registry for it's own host and port.

There are other implementations for other service registries like Hashicorp's Consul or Apache Zookeeper

Netflix’s Hystrix

Circuit breaker is a design pattern used in modern software development. It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.



The basic idea behind the circuit breaker is very simple. We wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.

Usually we will also want some kind of monitor alert if the circuit breaker trips.

Netflix’s Hystrix library provides an implementation of the Circuit Breaker pattern: when we apply a circuit breaker to a method, Hystrix watches for failing calls to that method, and if failures build up to a threshold, Hystrix opens the circuit so that subsequent calls automatically fail.

While the circuit is open, Hystrix redirects calls to the method, and they’re passed on to our specified fallback method.

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 currently only works in a class marked with @Component or @Service.

Spring Boot Actuator includes a number of additional features to help you monitor and manage your application when it's pushed to production. We can choose to manage and monitor our application using HTTP endpoints, with JMX or even by remote shell

Spring Actuator - Hystrix Health Endpoint

If we enabled Hystrix in your microservice, Spring Actuator will automatically add the Hystrix Health to your application's health endpoint

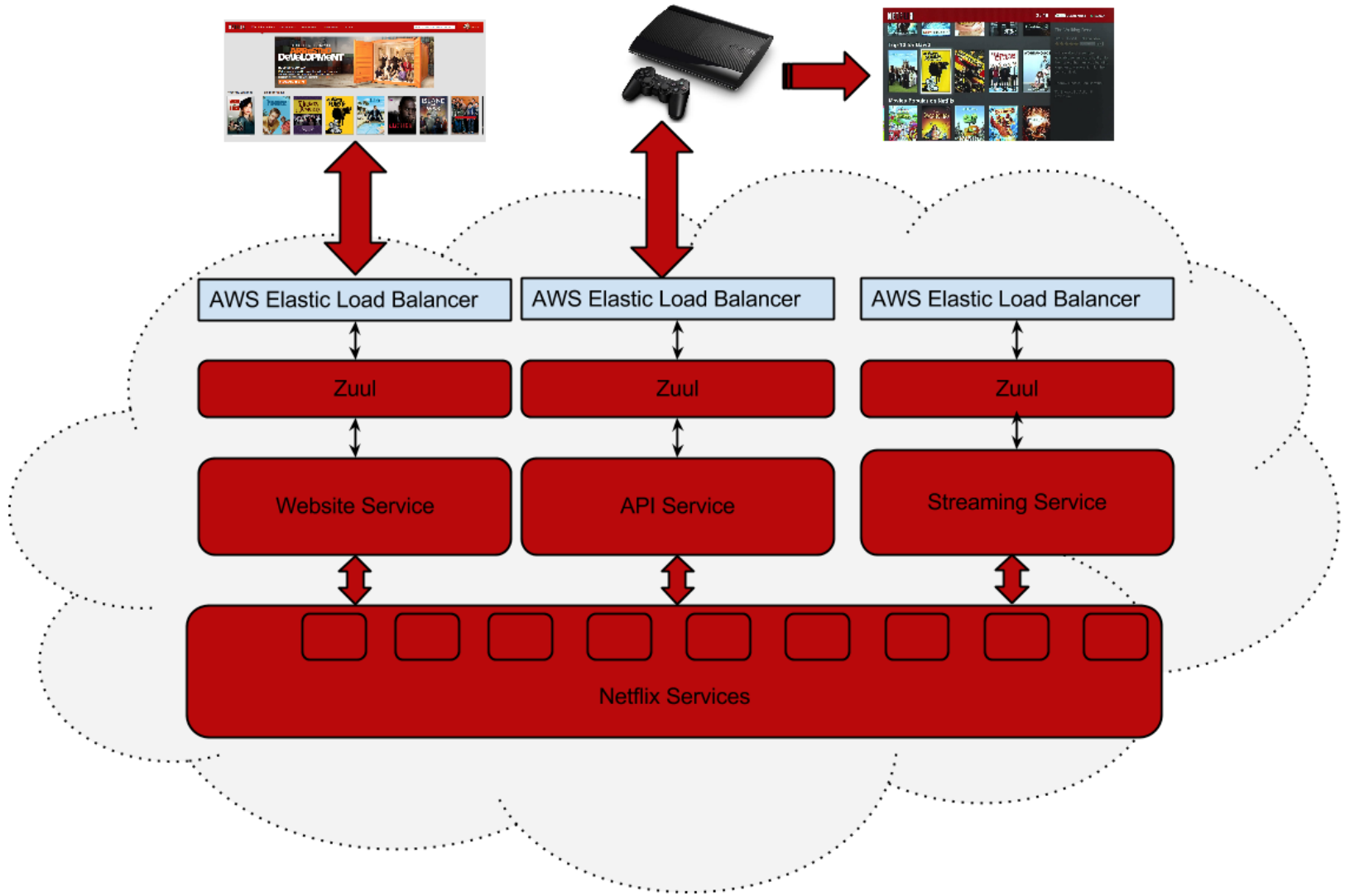
Netflix Zuul

What is Zuul?

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.

The volume and diversity of Netflix API traffic sometimes results in production issues arising quickly and without warning. We need a system that allows us to rapidly change behavior in order to react to these situations.

Zuul uses a range of different types of filters that enables us to quickly and nimbly apply functionality to our edge service. These filters help us perform the following functions:


AWS Elastic Load Balancer
AWS Elastic Load Balancer
AWS Elastic Load Balancer
Zuul
Zuul
Zuul
Website Service
API Service
Streaming Service
Netflix Services

Authentication and Security - identifying authentication requirements for each resource and rejecting requests that do not satisfy them.

Insights and Monitoring - tracking meaningful data and statistics at the edge in order to give us an accurate view of production.

Dynamic Routing - dynamically routing requests to different backend clusters as needed.

Stress Testing - gradually increasing the traffic to a cluster in order to gauge performance.

Load Shedding - allocating capacity for each type of request and dropping requests that go over the limit.

Static Response handling - building some responses directly at the edge instead of forwarding them to an internal cluster

Multiregion Resiliency - routing requests across AWS regions in order to diversify our ELB usage and move our edge closer to our members

Spring Cloud Config Server

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.

Spring Boot Actuator and Spring Config Client are on the classpath any Spring Boot application will try to contact a config server on http://localhost:8888 (the default value of spring.cloud.config.uri):

http url : http://localhost:8888/env

Spring Cloud Config Server features:

HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)

Encrypt and decrypt property values (symmetric or asymmetric)

Embeddable easily in a Spring Boot application using @EnableConfigServer

Config Client features (for Spring applications):

Bind to the Config Server and initialize Spring Environment with remote property sources

Encrypt and decrypt property values (symmetric or asymmetric)

Difference between @RibbonClient and @LoadBalanced

@LoadBalanced

Used as a marker annotation indicating that the annotated RestTemplate should use a RibbonLoadBalancerClient for interacting with your service(s).

In turn, this allows you to use "logical identifiers" for the URLs you pass to the RestTemplate. These logical identifiers are typically the name of a service. For example:

restTemplate.getForObject("http://some-service-name/user/{id}", String.class, 1);

where some-service-name is the logical identifier.

Is @RibbonClient required?

No! If you're using Service Discovery and you're ok with all of the default Ribbon settings, you don't even need to use the @RibbonClient annotation.

When should I use @RibbonClient?

There are at least two cases where you need to use @RibbonClient

- You need to customize your Ribbon settings for a particular Ribbon client
- You're not using any service discovery

Customizing your Ribbon settings:

Define a @RibbonClient

@RibbonClient(name = "some-service", configuration = SomeServiceConfig.class)

name - set it to the same name of the service you're calling with Ribbon but need additional customizations for how Ribbon interacts with that service.

configuration - set it to an @Configuration class with all of your customizations defined as @Beans. Make sure this class is not picked up by @ComponentScan otherwise it will override the defaults for ALL Ribbon clients.

Using Ribbon without Service Discovery

If you're not using Service Discovery, the name field of the @RibbonClient annotation will be used to prefix your configuration in the application.properties as well as "logical identifier" in the URL you pass to RestTemplate.

Define a @RibbonClient

@RibbonClient(name = "myservice")

then in your application.properties

myservice.ribbon.eureka.enabled=false
myservice.ribbon.listOfServers=http://localhost:5000,
http://localhost:5001

What is a Feign Client?

Netflix provides Feign as an abstraction over Rest-based calls, by which Microservice can communicate with each other, But developers don’t have to bother about Rest internal details.

Declarative REST Client: Feign

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

Spring Cloud Consul

Spring Cloud Consul provides Consul integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

With a few simple annotations we can quickly enable and configure the common patterns inside your application and build large distributed systems with Hashicorp's Consul. The patterns provided include Service Discovery, Distributed Configuration and Control Bus.

Spring Cloud Consul features:

Service Discovery: instances can be registered with the Consul agent and clients can discover the instances using Spring-managed beans

Supports Ribbon, the client side load-balancer via Spring Cloud Netflix

Supports Zuul, a dynamic router and filter via Spring Cloud Netflix

Distributed Configuration: using the Consul Key/Value store

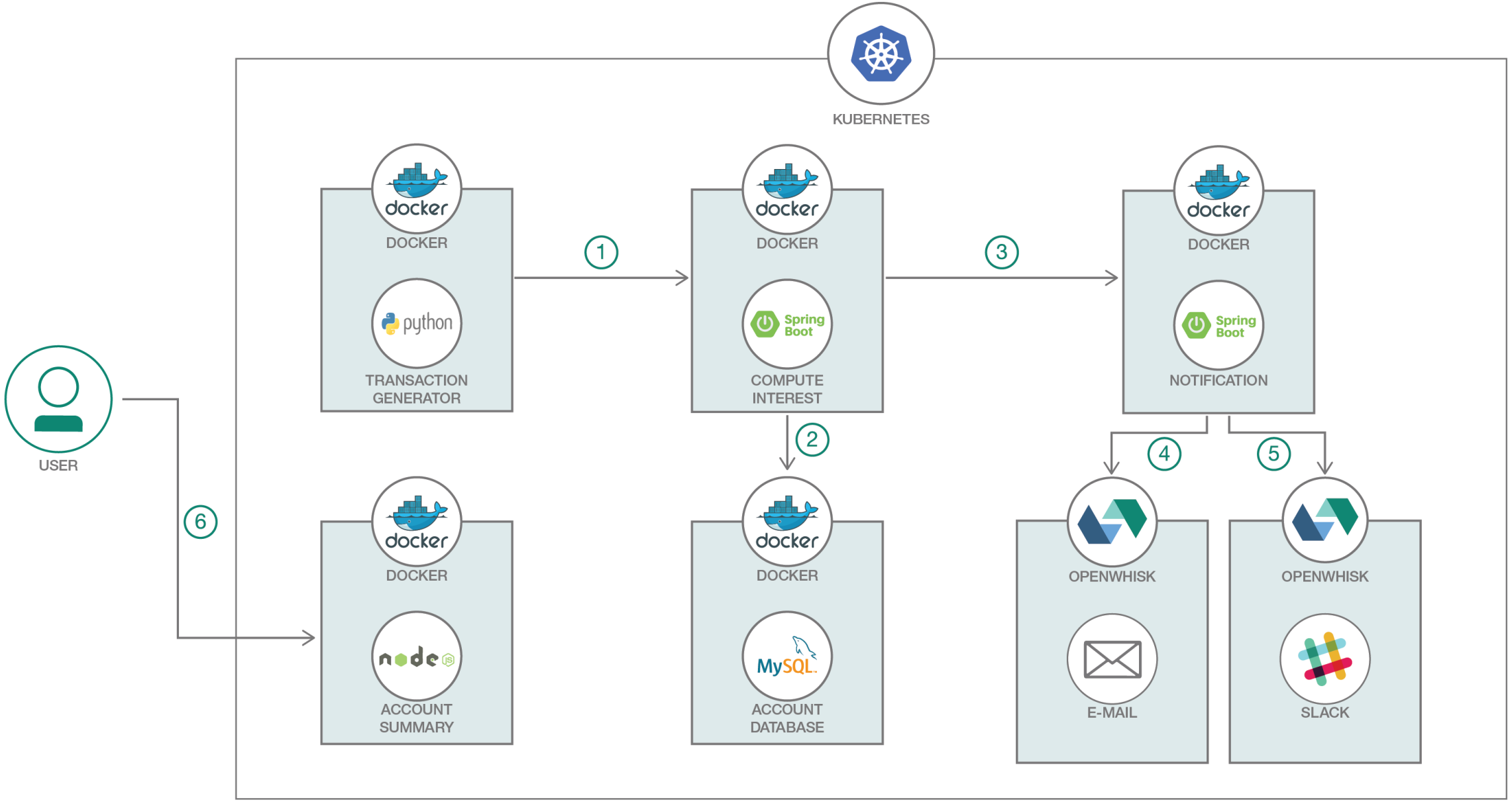
Control Bus: Distributed control events using Consul Events

Build and deploy Java Spring Boot microservices on a Kubernetes cluster

- Spring Boot manages the build of the microservices (jar/war) and the dependency injection (DI) context at runtime.

- Docker manages the packaging of microservice in a container and initializing the application's phase-specific variables through OS/cluster-level environment variables.

- Kubernetes manages the actual network layout and provides the service discovery, load-balancing, and scaling.


KUBERNETES
USER
DOCKER
TRANSACTION GENERATOR
DOCKER
COMPUTE INTEREST
DOCKER
NOTIFICATION
DOCKER
ACCOUNT SUMMARY
DOCKER
ACCOUNT DATABASE
OPENWHISK
E-MAIL
OPENWHISK
SLACK
1
2
3
4
5
6

- The Transaction Generator service written in Python simulates transactions and pushes them to the Compute Interest microservice.

- The Compute Interest microservice computes the interest and then moves the fraction of pennies to the MySQL database to be stored. The database can be running within a container in the same deployment or on a public cloud such as Bluemix.

- The Compute Interest microservice then calls the notification service to notify the user if an amount has been deposited in the user's account.

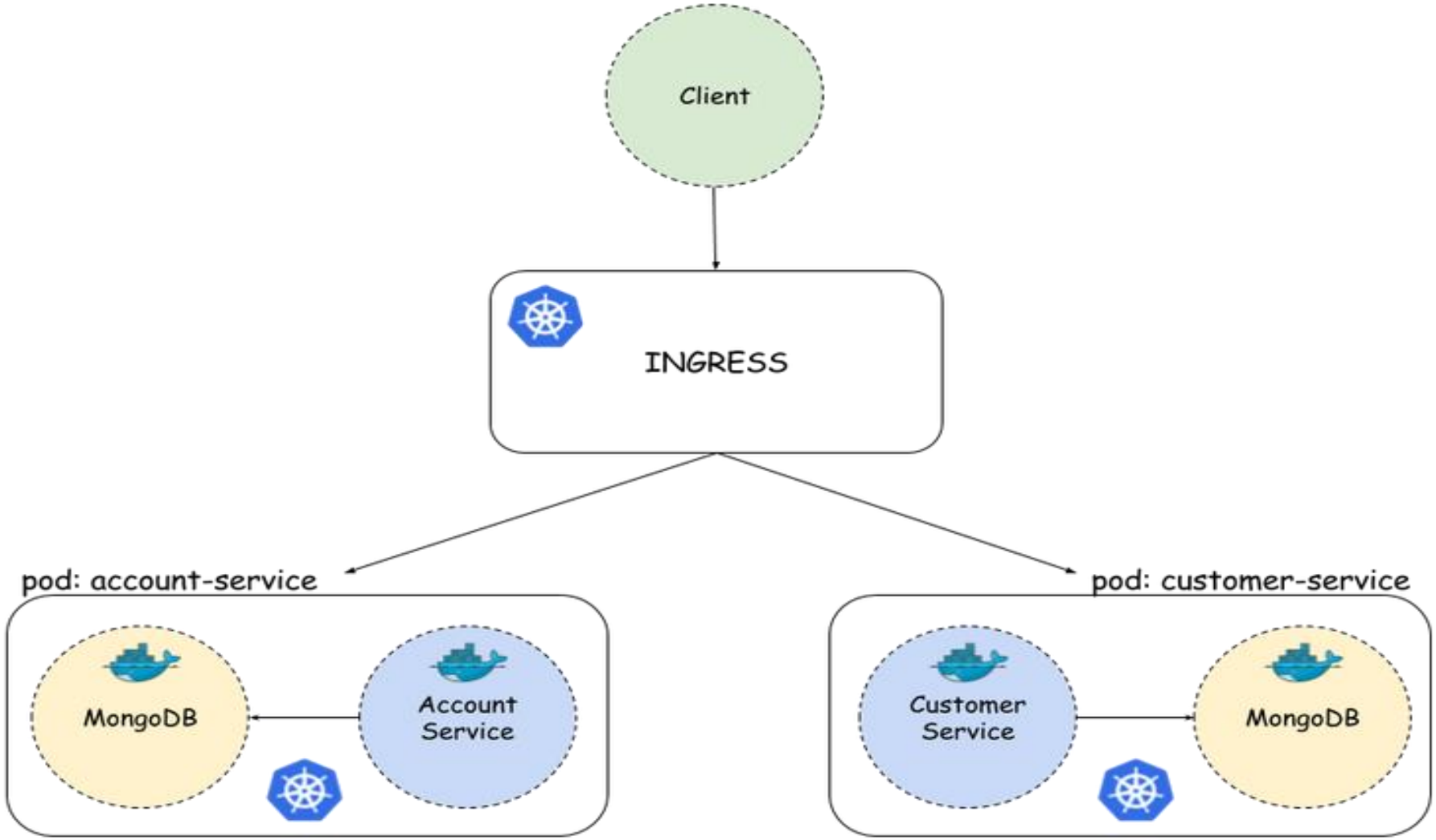
- The Notification service uses OpenWhisk actions to send an email message to the user. You can also invoke an OpenWhisk action to send messages to Slack.

- Additionally, an OpenWhisk action to send messages to Slack can also be invoked.

- The user retrieves the account balance by visiting the Node.js web interface.

With Kubernetes, gateway (Zuul) and discovery (Eureka) Spring Boot services are not required, because similar features are available on Kubernetes out of the box.

Kubernetes Ingress acts as a gateway for our microservices.


Client
INGRESS
pod: account-service
MongoDB
Account Service
pod: customer-service
Customer Service
MongoDB

Application Context Hierarchies

If we build an application context from SpringApplication or SpringApplicationBuilder, then the Bootstrap context is added as a parent to that context.

It is a feature of Spring that child contexts inherit property sources and profiles from their parent, so the "main" application context will contain additional property sources, compared to building the same context without Spring Cloud Config.

bootstrap.yml or bootstrap.properties

It's only used/needed if you're using Spring Cloud and your application's configuration is stored on a remote configuration server (e.g. Spring Cloud Config Server).

A Spring Cloud application operates by creating a "bootstrap" context, which is a parent context for the main application. Out of the box it is responsible for loading configuration properties from the external sources, and also decrypting properties in the local external configuration files.

bootstrap.yml or bootstrap.properties can contain additional configuration (e.g. defaults) but generally we need to put bootstrap config here.

Typically it contains:

location of the configuration server (spring.cloud.config.uri)
name of the application (spring.application.name)
some encryption/decryption information

Upon startup, Spring Cloud makes an HTTP call to the config server with the name of the application and retrieves back that application's configuration.

application.yml or application.properties

Contains standard application configuration - typically default configuration since any configuration retrieved during the bootstrap process will override configuration defined here.

Spring Cloud Commons

Spring Cloud Commons: Common Abstractions

Patterns such as service discovery, load balancing and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (e.g. discovery via Eureka or Consul).

@EnableDiscoveryClient

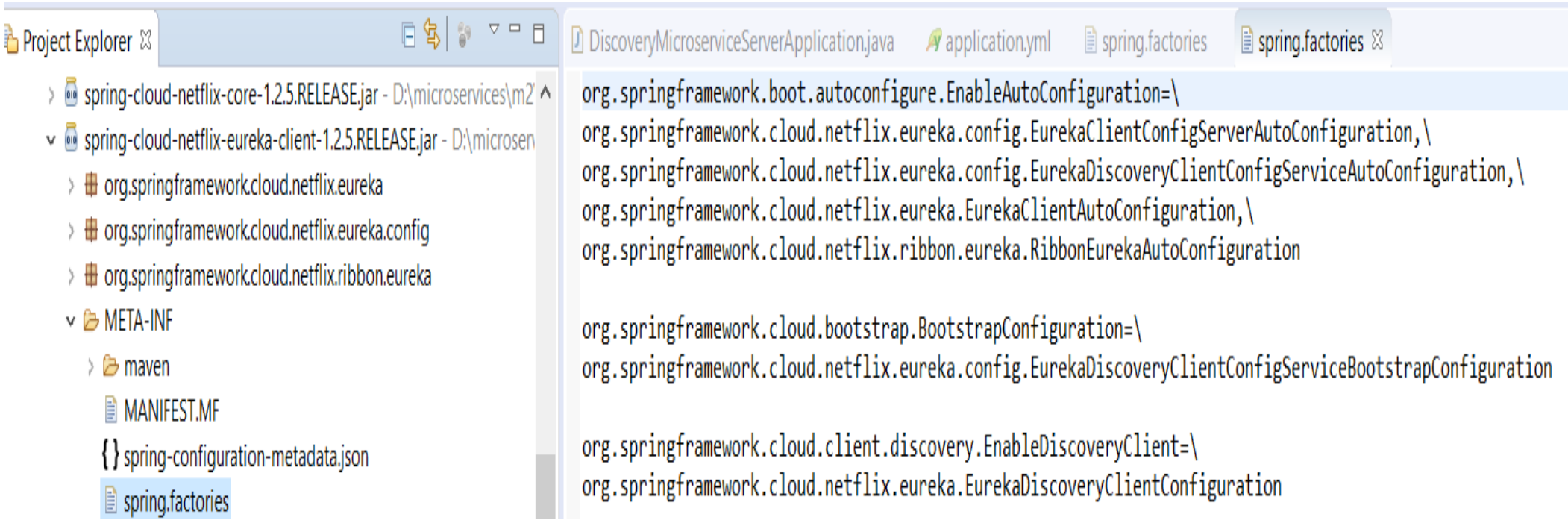
Commons provides the @EnableDiscoveryClient annotation.

This looks for implementations of the DiscoveryClient interface via META-INF/spring.factories.

Implementations of Discovery Client will add a configuration class to spring.factories under the org.springframework.cloud.client.discovery.EnableDiscoveryClient key.

Examples of DiscoveryClient implementations: are Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery and Spring Cloud Zookeeper Discovery.

By default, implementations of DiscoveryClient will auto-register the local Spring Boot server with the remote discovery server. This can be disabled by setting autoRegister=false in @EnableDiscoveryClient.


Project Explorer
spring-cloud-netflix-core-1.2.5.RELEASE.jar - D:\microservices\m2
spring-cloud-netflix-eureka-client-1.2.5.RELEASE.jar - D:\microser
org.springframework.cloud.netflix.eureka
org.springframework.cloud.netflix.eureka.config
org.springframework.cloud.netflix.ribbon.eureka
META-INF
maven
MANIFEST.MF
spring-configuration-metadata.json
spring.factories
DiscoveryMicroserviceServerApplication.java
application.yml
spring.factories
spring.factories
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.cloud.netflix.eureka.config.EurekaClientConfigServerAutoConfiguration,\
org.springframework.cloud.netflix.eureka.config.EurekaDiscoveryClientConfigServiceAutoConfiguration,\
org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration,\
org.springframework.cloud.netflix.ribbon.eureka.RibbonEurekaAutoConfiguration
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
org.springframework.cloud.netflix.eureka.config.EurekaDiscoveryClientConfigServiceBootstrapConfiguration
org.springframework.cloud.client.discovery.EnableDiscoveryClient=\
org.springframework.cloud.netflix.eureka.EurekaDiscoveryClientConfiguration

Spring RestTemplate as a Load Balancer Client

RestTemplate can be automatically configured to use ribbon.

To create a load balanced RestTemplate create a RestTemplate @Bean and use the @LoadBalanced qualifier.

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
```

Ignore Network Interfaces

Sometimes it is useful to ignore certain named network interfaces so they can be excluded from Service Discovery registration

application.yml

```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

We can also force to use only specified network addresses using list of regular expressions:

application.yml

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```