The Spring Cloud project is an umbrella project from the Spring team that implements a set of common patterns required by distributed systems.**Spring Cloud by itself is not a cloud platform**. Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that follows the Twelve-Factor application principles.

The **cloud-ready solutions** that are developed using Spring Cloud are portable across many cloud providers such as Cloud Foundry, AWS, Heroku, and so on. Built on Spring's "**convention over configuration**" approach, Spring Cloud defaults all configurations by hiding the complexities, and by providing simple declarative configurations to build echo-systems.

Spring Cloud offers many choices of cloud solutions based on their requirements. For example, the service registry can be implemented using popular options such as Eureka, ZooKeeper, or Consul. The components of Spring Cloud are fairly decoupled.

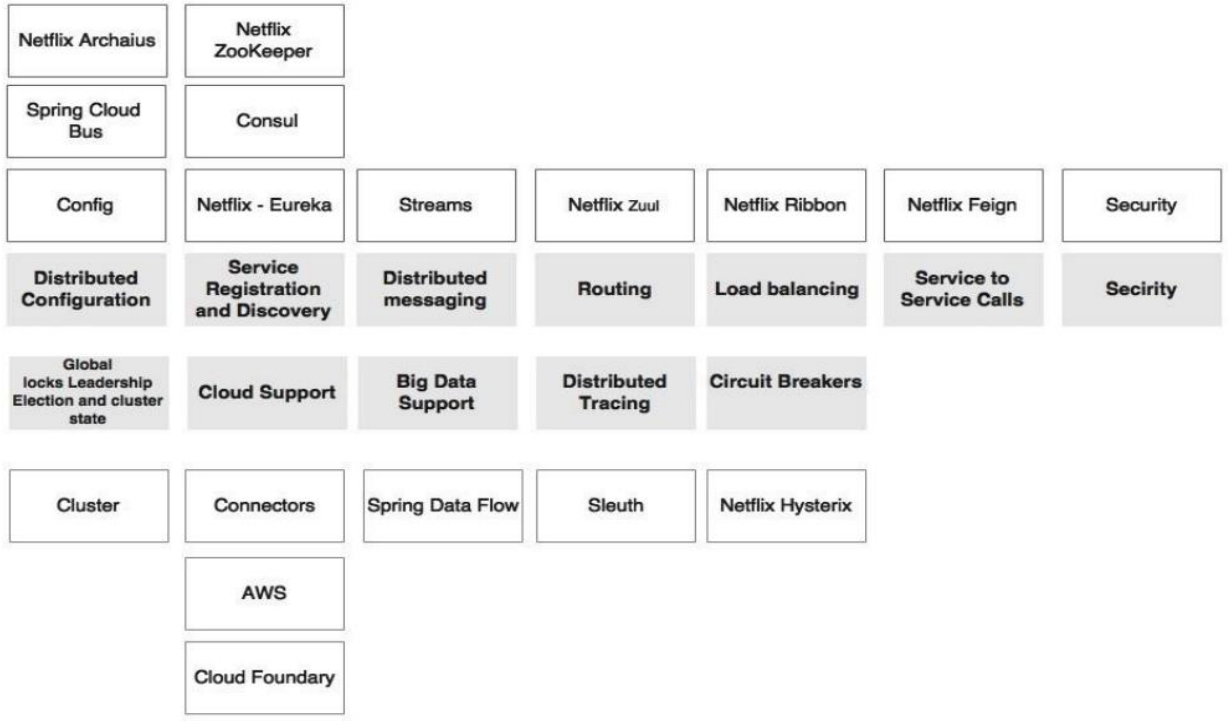
Spring Cloud releases

The Spring Cloud project is an overarching Spring project that includes a combination of different components. The versions of these components are defined in the BOM.

The names of the Spring Cloud releases are in an alphabetic sequence, starting with A, following the names of the London Tube stations. **Angel** was the first release, and **Brixton** is the second release.

Components of Spring Cloud

Each Spring Cloud component specifically addresses certain distributed system capabilities. The grayed-out boxes at the bottom of the following diagram show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud sub projects addressing these capabilities:



The Spring Cloud capabilities are explained as follows:

Distributed configuration: Configuration properties are hard to manage when there are many microservice instances running under different profiles such as development, test, production, and so on. It is, therefore, important to manage them centrally, in a controlled way. The distributed configuration management module is to externalize and centralize microservice configuration parameters. **Spring Cloud Config** is an externalized configuration server with Git or SVN as the backing repository. Spring Cloud Bus provides support for propagating

configuration changes to multiple subscribers, generally a microservice instance. Alternately, ZooKeeper or HashiCorp's Consul can also be used for distributed configuration management.

Routing: Routing is an API gateway component, primarily used similar to a reverse proxy that forwards requests from consumers to service providers. The gateway component can also perform software-based routing and filtering. **Zuul** is a lightweight API gateway solution that offers fine-grained controls to developers for traffic shaping and request/response transformations

- **Load balancing**: The load balancer capability requires a software-defined load balancer module which can route requests to available servers using a variety of load balancing algorithms. **Ribbon** is a Spring Cloud sub project which supports this capability. Ribbon can work as a standalone component, or integrate and work seamlessly with Zuul for traffic routing.
- **Service registration and discovery**: The service registration and discovery module enables services to programmatically register with a repository when a service is available and ready to accept traffic. The microservices advertise their existence, and make them discoverable. The consumers can then look up the registry to get a view of the service availability and the endpoint locations. The registry, in many cases, is more or less a dump. But the components around the registry make the ecosystem intelligent. There are many subprojects existing under Spring Cloud which support registry and discovery capability. **Eureka, ZooKeeper, and Consul** are three sub projects implementing the registry capability.
- **Service-to-service calls**: The **Spring Cloud Feign** sub project under Spring Cloud offers a declarative approach for making RESTful service-to-service calls in a **synchronous** way. The declarative approach allows applications to work with **POJO** (**Plain Old Java Object**) interfaces instead of low-level HTTP client APIs. Feign internally uses reactive libraries for communication.
- **Circuit breaker**: The circuit breaker sub project implements the circuit breaker pattern. The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service. It also automatically reconnects back to the primary service when the service is back to normal. It finally provides a monitoring dashboard for monitoring the service state changes. The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard respectively.
- **Global locks, leadership election and cluster state:** This capability is required for cluster management and coordination when dealing with large deployments. It also offers global locks for various purposes such as sequence generation. The Spring Cloud Cluster project implements these capabilities using **Redis, ZooKeeper, and Consul**.
- **Security:** Security capability is required for building security for cloud-native distributed systems using externalized authorization providers such as OAuth2. The Spring Cloud Security project implements this capability using customizable authorization and resource

servers. It also offers SSO capabilities, which are essential when dealing with many microservices.

Big data support: The big data support capability is a capability that is required for data services and data flows in connection with big data solutions. The Spring Cloud Streams and the Spring Cloud Data Flow projects implement these capabilities. The Spring Cloud Data Flow is the re-engineered version of Spring XD.

Distributed tracing: The distributed tracing capability helps to thread and correlate transitions that are spanned across multiple microservice instances. **Spring Cloud Sleuth** implements this by providing an abstraction on top of various distributed tracing mechanisms such as **Zipkin and HTrace** with the support of a 64-bit ID.

Distributed messaging: Spring Cloud Stream provides declarative messaging integration on top of reliable messaging solutions such as **Kafka, Redis**, and **RabbitMQ**.

Cloud support: Spring Cloud also provides a set of capabilities that offers various connectors, integration mechanisms, and abstraction on top of different cloud providers such as the Cloud Foundry and AWS.

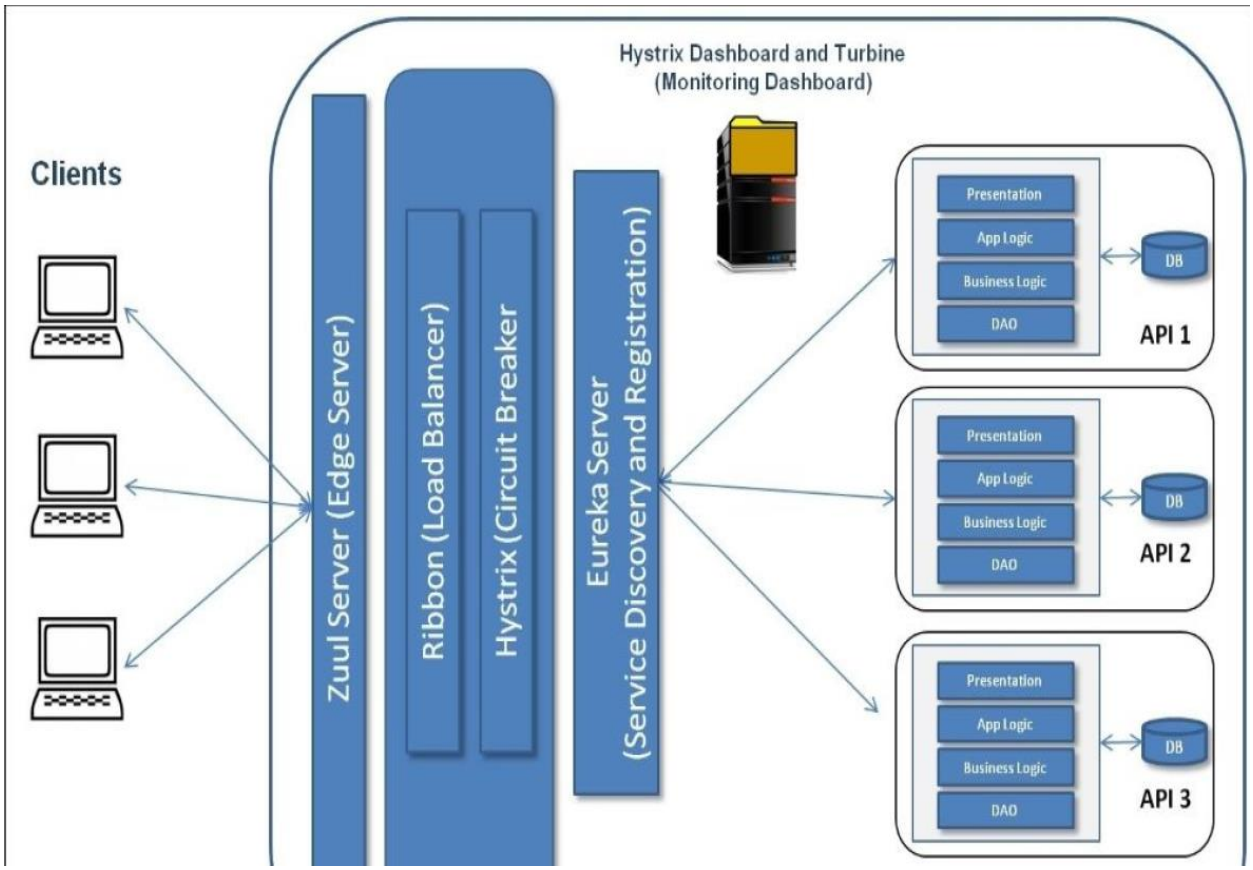
Spring Cloud and Netflix OSS

Many of the Spring Cloud components which are critical for microservices' deployment came from the Netflix Open Source Software (Netflix OSS) center. Netflix is one of the pioneers and early adaptors in the microservices space.

In order to manage large scale microservices, engineers at Netflix came up with a number of homegrown tools and techniques for managing their microservices. These are fundamentally crafted to fill some of the software gaps recognized in the AWS platform for managing Netflix services.

Later, Netflix open-sourced these components, and made them available under the Netflix OSS platform for public use. These components are extensively used in production systems, and are battle-tested with large scale microservice deployments at Netflix.

Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism, well-integrated and aligned with Spring Boot and the Spring framework.



Fleet-management-echosystem