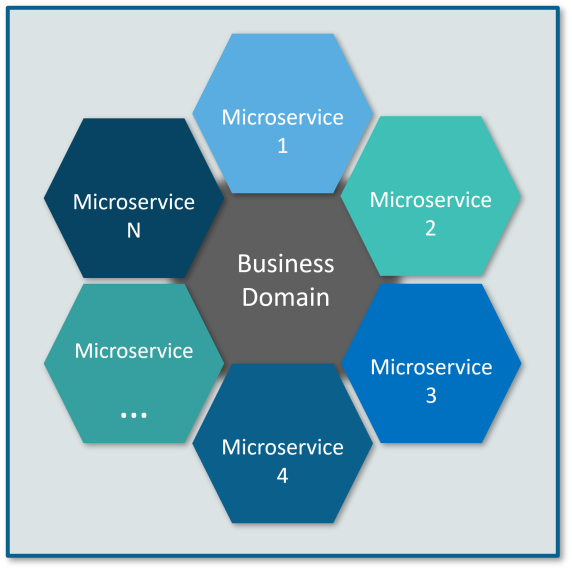
## What Are Microservices?

Microservices is an approach to software architecture that builds a large, complex application from multiple small components that each perform a single function, such as authentication, notification, or payment processing. Each microservice is a distinct unit within the software development project, with its own codebase, infrastructure, and database. The microservices work together, communicating through web APIs or messaging queues to respond to incoming events.

Microservices take a complex monolithic application and turn it into a set of services, which are faster to develop, and much easier to understand and maintain. Each of these services can also be developed independently by a team that is focused on that service. It also allows for continuous deployment, because each microservice can be deployed independently.

**Microservices**, aka ***[Microservice Architecture](https://www.edureka.co/blog/microservice-architecture/" \t "_blank)***, is an architectural style that structures an application as a collection of small autonomous services, modeled around a **business domain.**

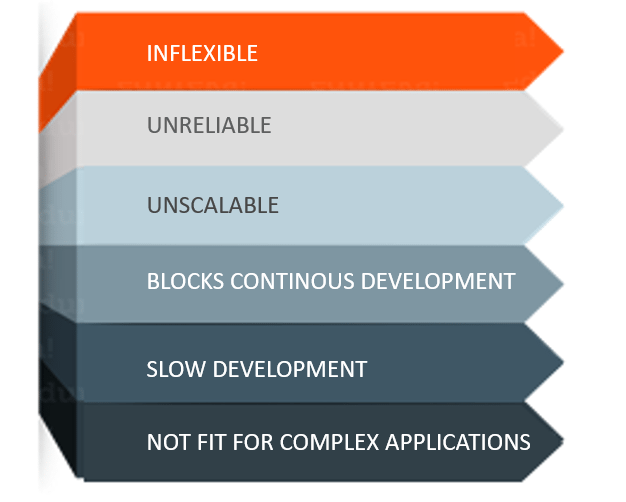


In Microservice Architecture, each service is **self-contained** and implements a **single business capability.**

## ****Why Microservices?****

In layman terms, **Monolithic Architecture is** similar to a big container wherein all the software components of an application are assembled together and tightly packaged.

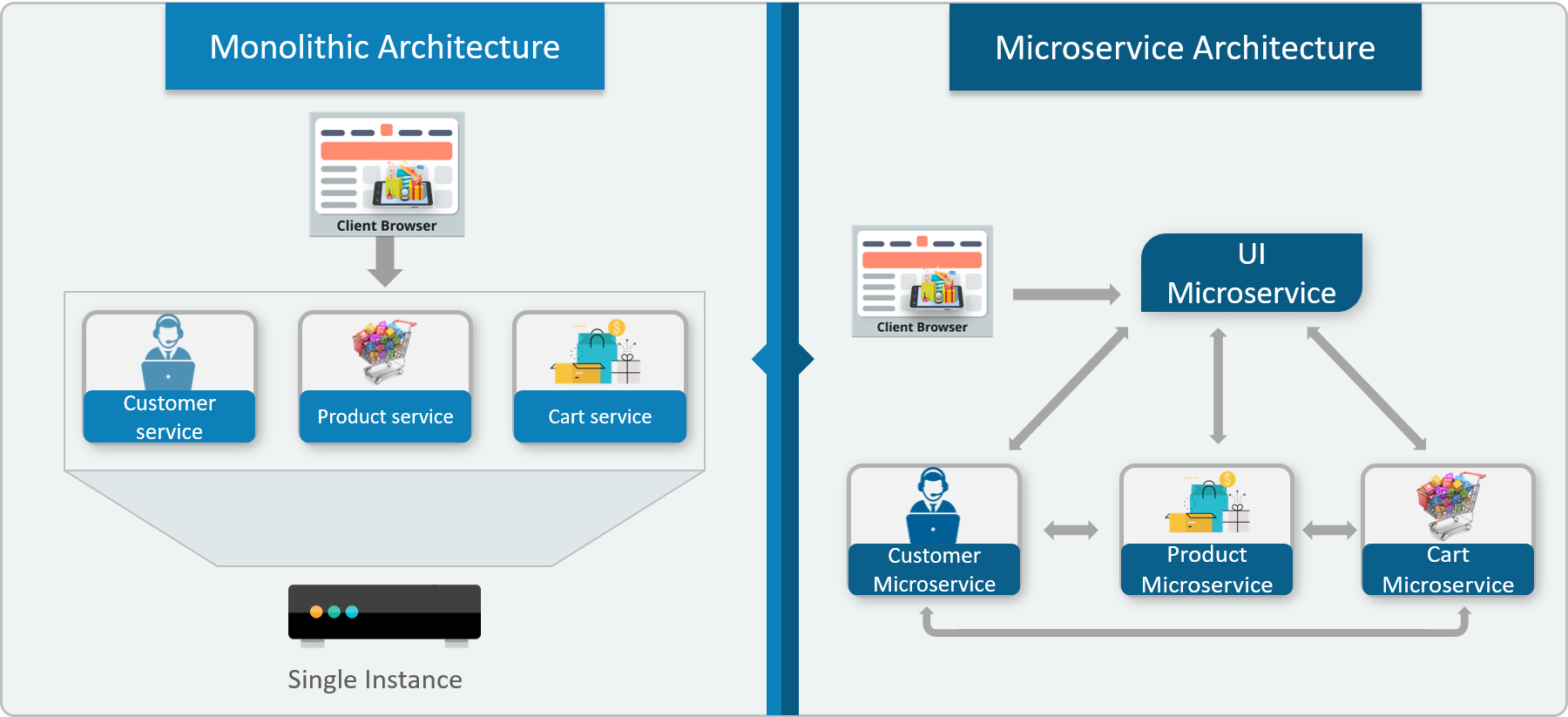
**Listed down are the challenges of Monolithic Architecture:**



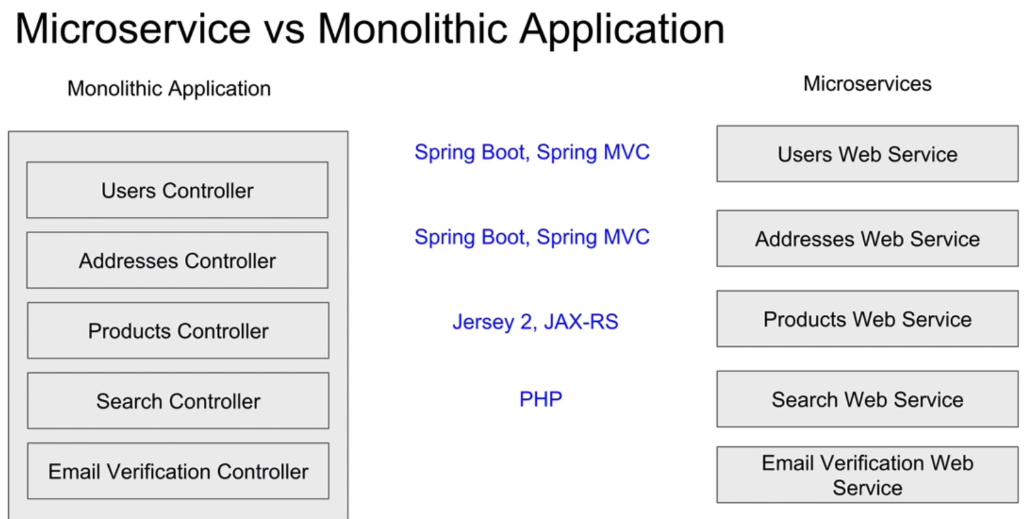
* **Inflexible –** Monolithic applications cannot be built using different technologies
* **Unreliable –** Even if one feature of the system does not work, then the entire system does not work
* **Unscalable –** Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt
* **Blocks Continous Development –** Many features of the applications cannot be built and deployed at the same time
* **Slow Development –** Development in monolithic applications take lot of time to be built since each and every feature has to be built one after the other

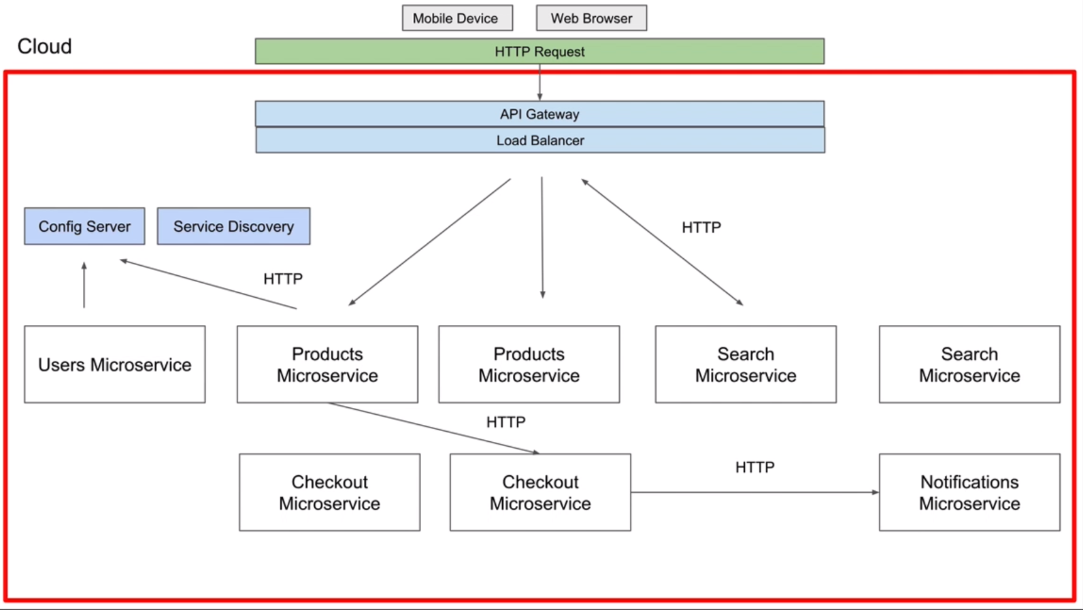
**Not Fit For Complex Applications –**Features of complex applications have tightly coupled dependencies

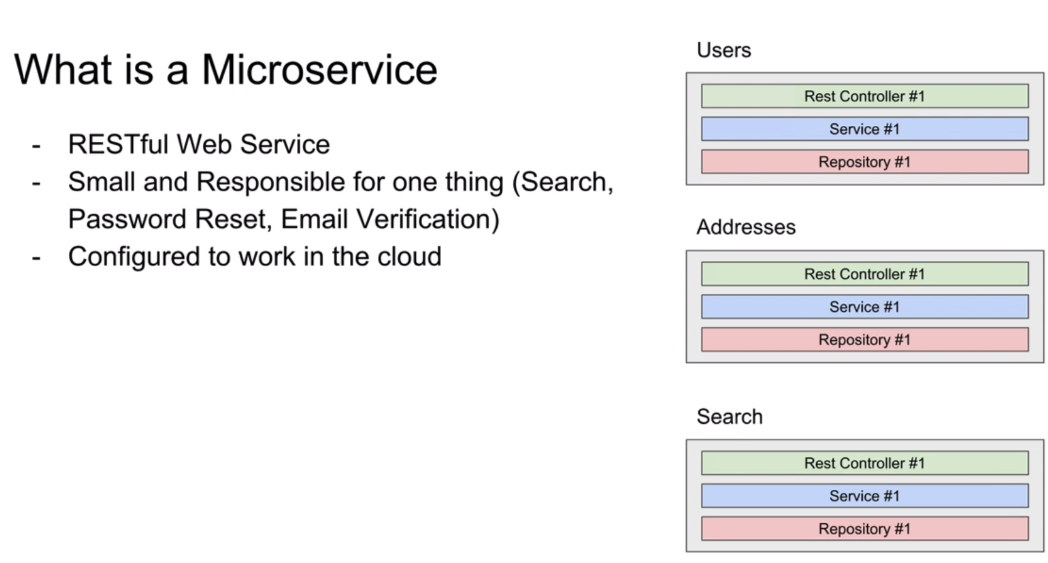
## ****Differences Between Traditional Architecture and Microservices****



The main difference we observe in the above diagram is that all the features initially were under a single instance sharing a single database. But then, with microservices, each feature was allotted a different microservice, handling their own data, and performing different functionalities.

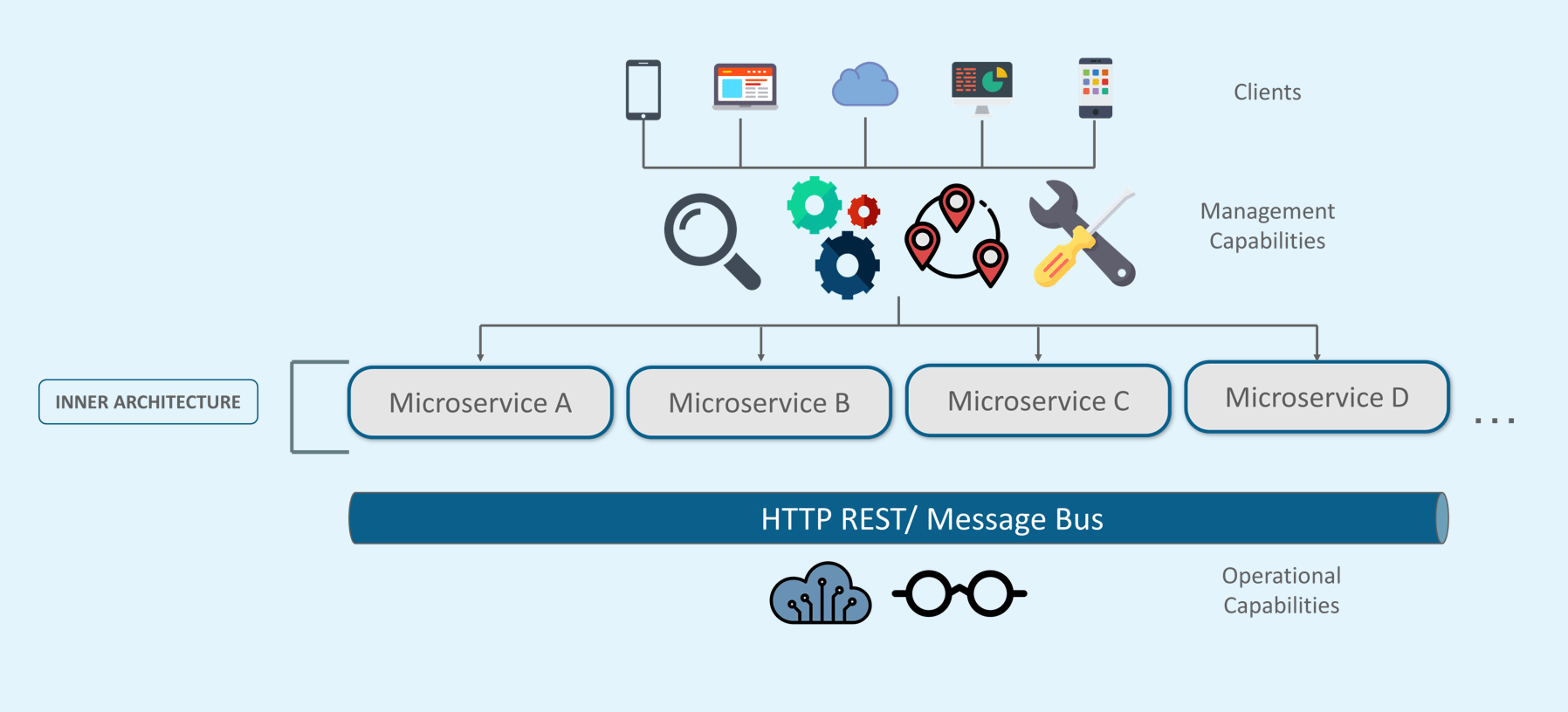






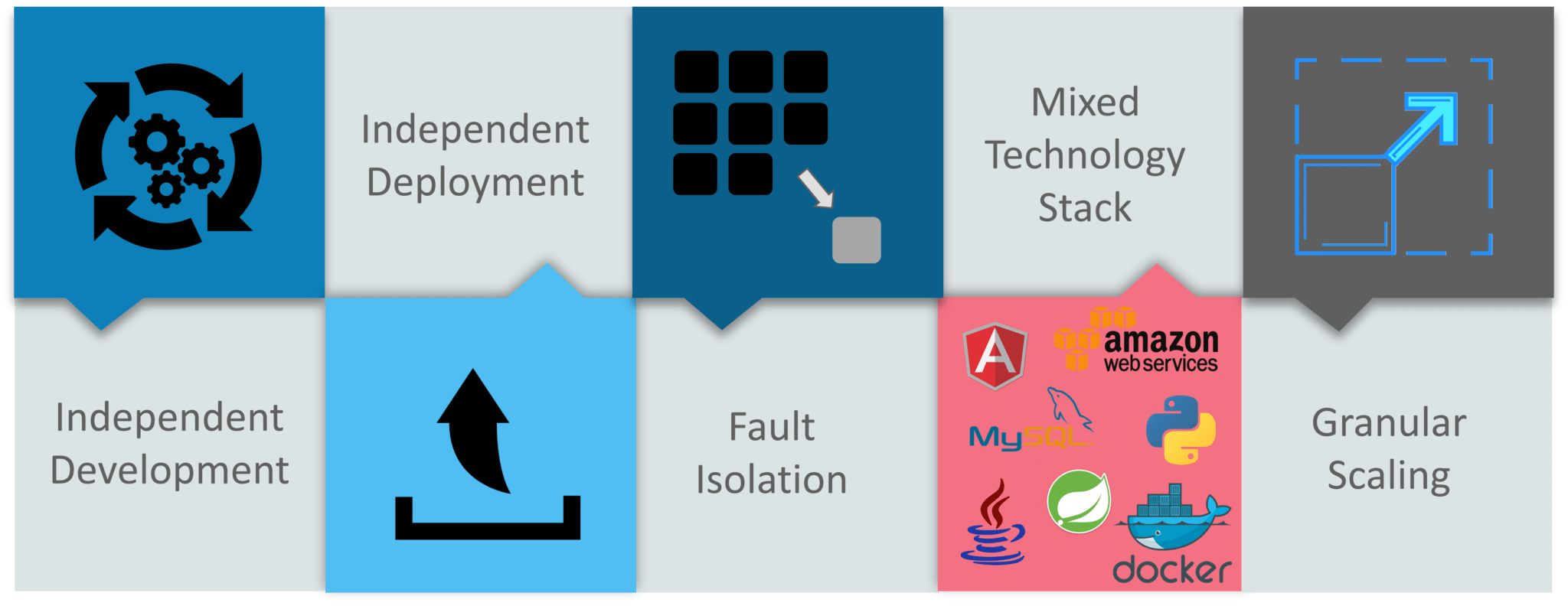
## 

## ****Microservice Architecture****



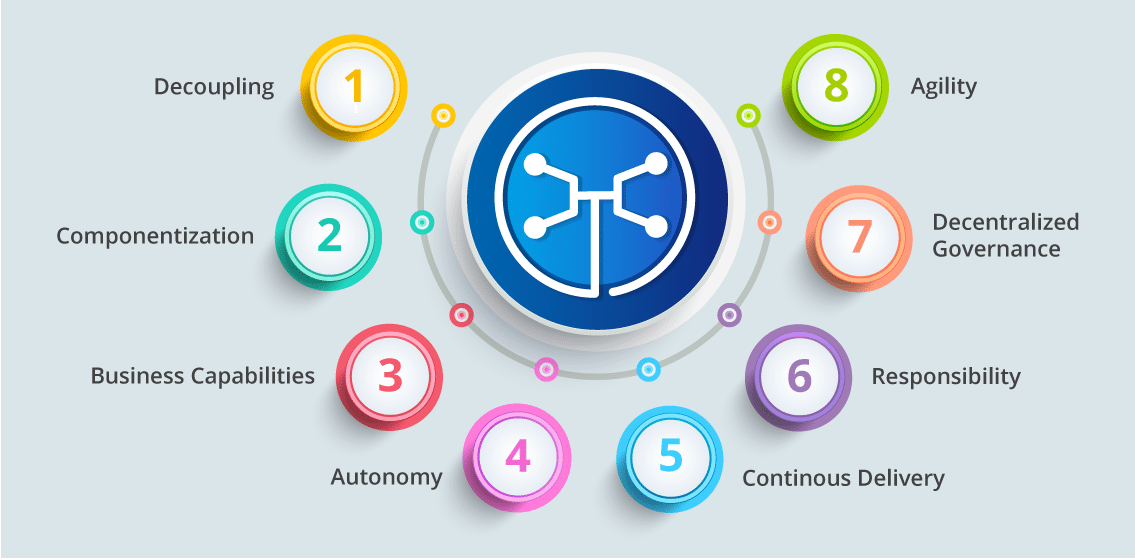
* Different clients from different devices try to use different services like search, build, configure and other management capabilities
* All the services are separated based on their domains and functionalities and  are further allotted to individual microservices
* These microservices have their own **load balancer** and **execution environment** to execute their functionalities & at the same time captures data in their own databases
* All the microservices communicate with each other through a stateless server which is either **REST** or **Message Bus**
* Microservices know their path of communication with the help of **Service Discovery**and perform operational capabilities such as automation, monitoring
* Then all the functionalities performed by microservices are communicated to clients via **API Gateway**
* All the internal points are connected from the API Gateway. So, anybody who connects to the API Gateway automatically gets connected to the complete system

## ****Advantages Of Microservices****



* **Independent Development** – All microservices can be easily developed based on their individual functionality
* **Independent Deployment** – Based on their services, they can be individually deployed in any application
* **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
* **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
* **Granular Scaling** –  Individual components can scale as per need, there is no need to scale all components together

## ****Microservices Features****



* **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
* **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
* **Business Capabilities** – Microservices are very simple and focus on a single capability
* **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
* **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
* **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
* **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
* **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

## ****Companies using Microservices****

There is a long list of companies using Microservices to build applications, these are just to name a few:

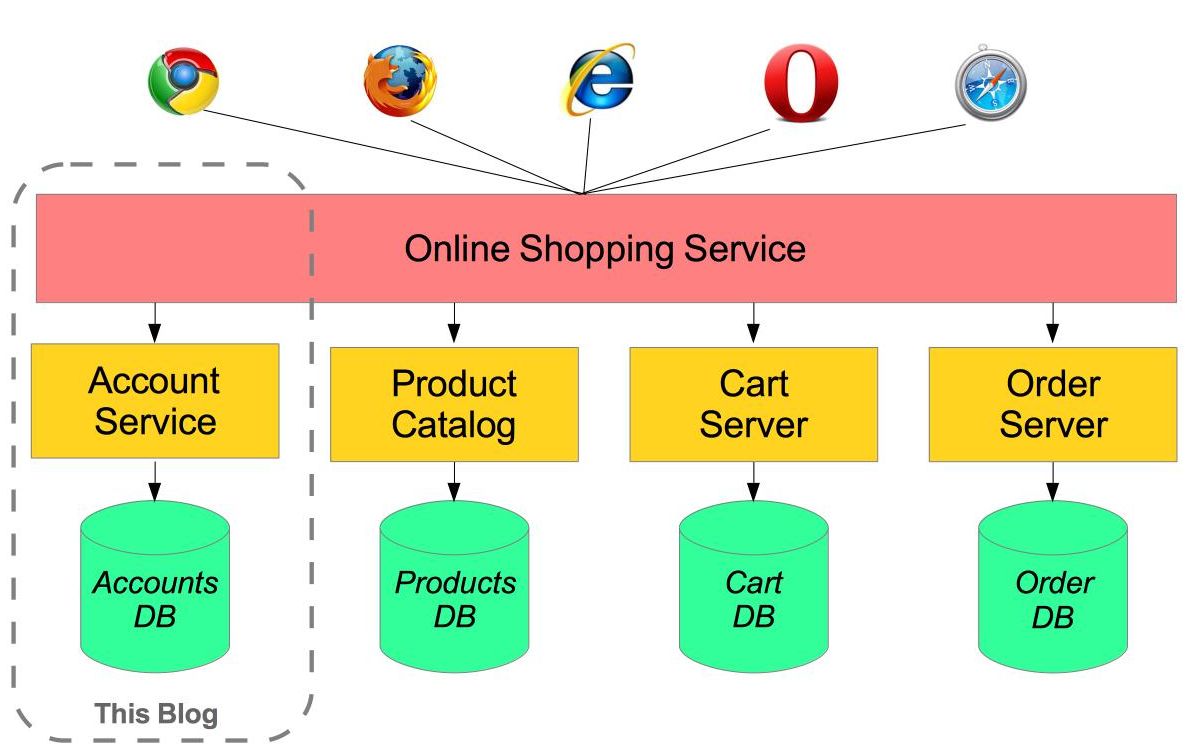


### Spring Cloud

*Spring Cloud provides solutions to cloud enable your microservices. It leverages and builds on top of some of the Cloud solutions opensourced by Netflix (Netflix OSS).*

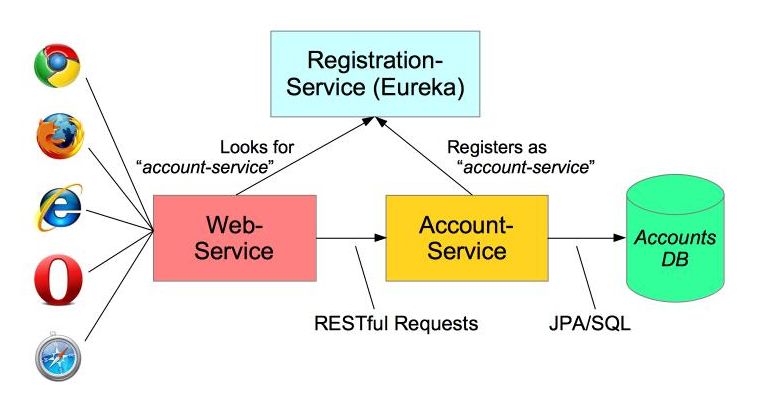
Microservices with Spring

[Microservices](http://martinfowler.com/articles/microservices.html) allow large systems to be built up from a number of collaborating components. It does at the process level what Spring has always done at the component level: loosely coupled processes instead of loosely coupled components.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/shopping-system.jpg)

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:

Inevitably there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/mini-system.jpg)

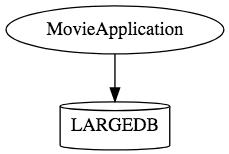
In this article I aim to clarify how things work by building the simplest possible system step-by-step. Therefore, I will only implement a small part of the big system - the user account service.

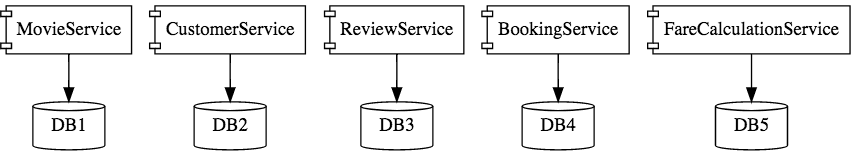
The *Web-Application* will make requests to the *Account-Service* microservice using a RESTful API. We will also need to add a *discovery* service – so the other processes can find each other.

# Service Registration

When you have multiple processes working together they need to find each other. The developers at Netflix had this problem when building their systems and created a registration server called Eureka (“I have found it” in Greek). Fortunately for us, they made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server.

## How does Microservice Architecture look like?

This is how a monolith would look like. One application for everything. 

This is how the same application would look like when developed using Microservices Architecture. 

Microservice Architectures involve a number of small, well designed, components interacting with messages. https://www.springboottutorial.com/images/Microservices-Chain-Example.png

## Challenges with Microservice Architectures

While developing a number of smaller components might look easy, there are a number of inherent complexities that are associated with microservices architectures.

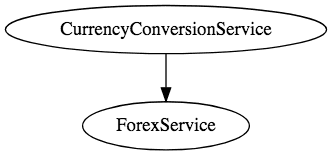
Lets look at some of the challenges:

* Quick Setup needed : You cannot spend a month setting up each microservice. You should be able to create microservices quickly.
* Automation : Because there are a number of smaller components instead of a monolith, you need to automate everything - Builds, Deployment, Monitoring etc.
* Visibility : You now have a number of smaller components to deploy and maintain. Maybe 100 or maybe 1000 components. You should be able to monitor and identify problems automatically. You need great visibility around all the components.
* Bounded Context : Deciding the boundaries of a microservice is not an easy task. Bounded Contexts from Domain Driven Design is a good starting point. Your understanding of the domain evolves over a period of time. You need to ensure that the microservice boundaries evolve.
* Configuration Management : You need to maintain configurations for hundreds of components across environments. You would need a Configuration Management solution
* Dynamic Scale Up and Scale Down : The advantages of microservices will only be realized if your applications can scaled up and down easily in the cloud.
* Pack of Cards : If a microservice at the bottom of the call chain fails, it can have knock on effects on all other microservices. Microservices should be fault tolerant by Design.
* Debugging : When there is a problem that needs investigation, you might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.

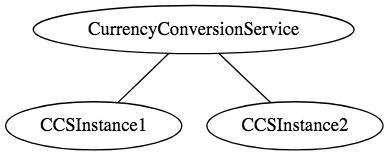
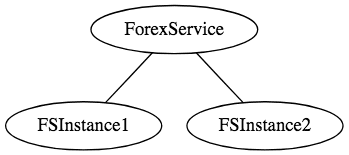
n this series of articles, we would create two microservices:

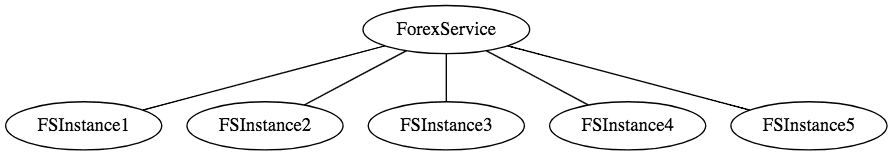
* Forex Service - Abbreviated as FS
* Currency Conversion Service - Abbreviated as CCS

The diagram below shows the communication between CCS and FS. We would establish communication between these two components.



We would want to be able to dynamically scale up and scale down the number of instances of each of these services.

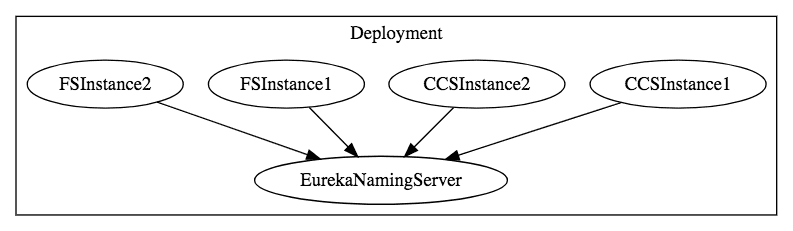
 

And the number of instances for each service might vary with time. Below picture shows a specific instance where there are 5 instances of the Forex Service. 

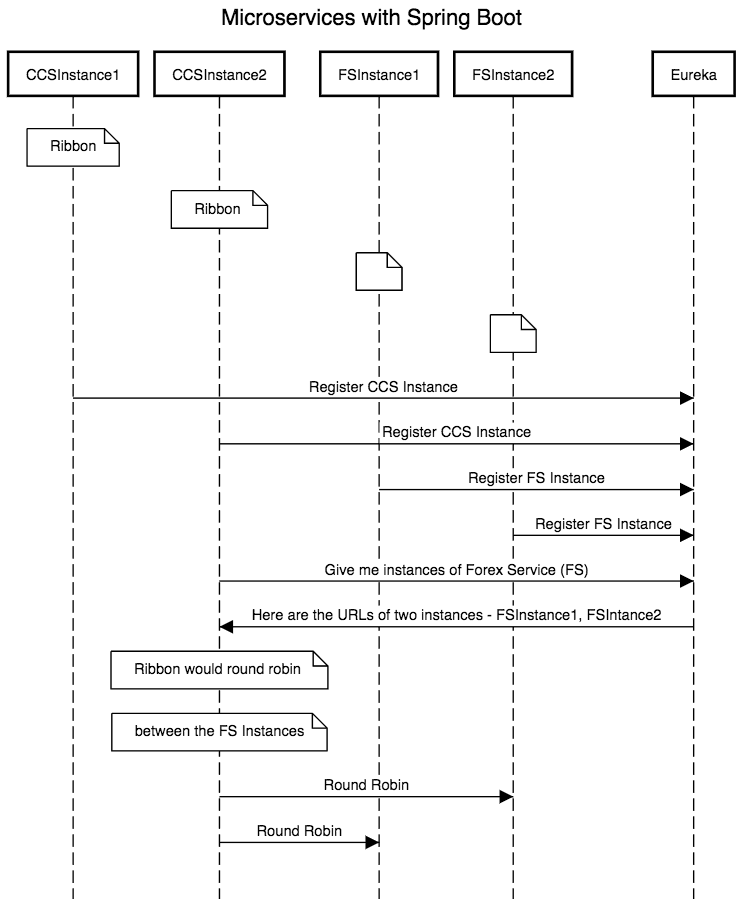
Implementing a solution for dynamic scale up and down needs to answer two questions

* How does the Currency Conversion Service (CCS) know how many instances of Forex Service (FS) are active?
* How does the Currency Conversion Service (CCS) distribute the load between the active instances.

Because we want this to be dynamic, we cannot hardcode the urls of FS in CCS. Thats why we bring in a Naming Server.



All instances of the components (CCS and FS) register with the Eureka Naming Server. When FS needs to call the CCS, it will ask Eureka Naming Server for the active instances. We will use Ribbon to do Client Side Load Balancing between the different instances of FS.

A high level sequence diagram of what would happen when there is a request from CCS to FS is shown below: 

Here are the next series of Articles

* Creating a Forex Microservice - We will create a simple rest service based on Spring Boot Starter Web and Spring Boot Started JPA. We will use Hibernate as JPA implmentation and connect to H2 database.
* Create the CCS - Currency Conversion Service - We will create a simple rest service using feign to invoke the Forex Microservice
* Use Ribbon for Load Balancing
* Implement Eureka Naming Service and connect FS and CCS through Eureka.