# Microservices com Spring Cloud: Para que servem?

Uma Arquitetura de Microsserviços é uma abordagem arquitetural que visa componentizar o código em serviços independentes e pequenos.

Spring Cloud é um conjunto de ferramentas desenvolvidas pela equipe do framework Spring que dá suporte à implementação de uma Arquitetura de Microsserviços na linguagem Java.

## Para que serve uma Arquitetura de Microsserviços?

Para entender a motivação dos Microsserviços, precisamos entender o seu antagonista: o Monólito.

### O que há de errado com o meu Monólito?

> "Quando você tem uma base de código só, você é tão estável quanto a sua parte menos estável."
> Phil Calçado na palestra da QCon SP 2013 Evoluindo uma Arquitetura inteiramente sobre APIs: o caso da SoundCloud

A palavra Monólito vem do grego e significa "pedra única" ("mono" é único e "lithos" é pedra).

Um sistema monolítico é um sistema em que há um **único** "entregável" , o artefato que é implantado (feito o deploy) nos ambientes de homologação e produção. Na plataforma Java, o artefato mais comum é um WAR.

Para uma aplicação pequena, o monólito não é um problema:

- Os desenvolvedores estão acostumados com monólitos.
- As IDEs, frameworks e demais ferramentas são otimizadas para o desenvolvimento de monólitos.
- A refatoração do monólito é simples e, muitas vezes, auxiliada por IDEs.
- Implantar o monólito nos diferentes ambientes é uma tarefa muito simples: basta copiar o entregável para o(s) servidor(es) e pronto!
- O monitoramento é muito simples: ou está ou não está no ar.

Porém, quando o **tamanho da aplicação começa a crescer**, começam a surgir várias complicações.

Algumas são **complicações no desenvolvimento**:

- muitas funcionalidades requerem um **time grande**. E a comunicação entre as pessoas em um time torna-se mais complexa quanto maior for o time.
- quanto maior a **base de código**, mais **complexa** e difícil de entender. A tentação de criar dependências indevidas passa a ser grande. Se não houver um constante cuidado com o código, será inevitável o acúmulo de dívida técnica até termos uma Big Ball of Mud.
- a **produtividade do desenvolvedor** é diminuída. As IDEs travam devido à massiva quantidade de código. A aplicação demora a iniciar, dificultando o ciclo de código-compilação-execução-teste de cada desenvolvedor.
- os **testes**, manuais ou automatizados, passam a ser muito **demorados**. Uma estratégia de continuous integration/deployment demora demais, atrasando o feedback e diminuindo a confiança.
- é **difícil de atualizar as tecnologias** ou de usar diferentes tecnologias para diferentes partes dos problemas. O código pode até ser "poliglota" (usar diferentes linguagens), desde que compatíveis com mesma plataforma (p. ex., JVM).

Há também **complicações na operação**:

- a **implantação**, apesar de fácil, é **dificultada**. Um time grande produz várias funcionalidades em cadências diferentes. É preciso haver uma **grande coordenação** para que não haja implantação de funcionalidades ainda em desenvolvimento. O uso de feature branches pode ajudar, mas pode levar a merges complicadíssimos. A ideia de continuous deployment/delivery, de publicar código em produção várias vezes ao dia, parece muito distante.
- há um **ponto único de falha**. Não há isolamento de falhas. Se algo derrubar a aplicação (por exemplo, por vazamento de memória), tudo fica indisponível.
- há **ineficiência na escalabilidade**. Podemos replicar o entregável em várias instâncias, usando um load balancer. Porém, toda a aplicação será replicada e não apenas as partes que sofrem maior carga em termos de CPU ou memória. Isso leva a subutilização de recursos.

## O Monólito Modular

É possível resolver parte dos problemas através de técnicas de modularização, que são abordadas no curso FJ-38 (Práticas de Design e Arquitetura de código para aplicações Java).

A base de código seria "fatiada" em módulos menores. Isso levaria a um melhor gerenciamento das dependências entre os módulos.

Poderíamos ter equipes mais enxutas, focadas em cada módulo.

Cada desenvolvedor passaria a trabalhar com apenas uma parte do código, suavizando a IDE. Subir a aplicação seria algo relativamente demorado, já que seria preciso colocar todos os módulos no ar.

Os testes poderiam ser agilizados, concentrando-se em cada módulo separadamente. Porém, teria que ser feito um teste de sistemas, que avaliaria a integração entre todos os módulos na aplicação.

Os problemas nas operações continuariam.

# Microsserviços

Uma Arquitetura de Microsserviços quebra um monólito usando como estratégia a **modularização em serviços**.

Um serviço é um componente de software que provê alguma funcionalidade e pode ser implantado independentemente. Cada serviço provê uma **API** que pode ser "consumida" por seus clientes. Uma chamada a um serviço é feita por meio de **comunicação interprocessos** que, no fim das contas, é comunicação pela rede. Isso faz com que uma Arquitetura de Microsserviços seja uma Arquitetura de **Sistemas Distribuídos**.

No fim das contas, o objetivo principal é alinhar negócio à TI.

## Tamanho do serviço

Os serviços em uma Arquitetura de Microsserviços devem ser pequenos. Por isso, o "micro" no nome. Mas o que deve ser considerado "micro"? Algo menor que um miliserviço ou maior que um nanoserviço😂? Não! O tamanho não é importante! O termo "micro" é enganoso.

O critério para decomposição deve ser, em geral, algo alinhado com o negócio da organização. Um serviço pequeno é um que embarca uma capacidade de negócio.

Ou, nos termos do DDD, um microsserviço deve representar um subdomínio, uma área de expertise em que há um contexto delimitado (*bounded context*) em que a linguagem (*ubiquitous language*) do especialista de domínio será representada no código (*domain model*).

## Tamanho do time

O objetivo de uma arquitetura de microsserviços é que um serviço seja pequeno o bastante para ser desenvolvido por uma equipe autônoma e multidisciplinar de no máximo 9 pessoas, bem no espírito do Manifesto Ágil.

Como dizem na Amazon, devemos ter um *two-pizza team*, um time que pode ser alimentado por duas pizzas no máximo.

O time deve ser responsável por manter o microsserviço da concepção à operação.

### Encapsulamento do microsserviço

Um microsserviço é um nível macro de encapsulamento. Os detalhes de implementação devem estar escondidos por meio de sua API.

Qualquer interação deve ser feita por meio de APIs. Os consumidores devem comunicar com o microsserviço apenas através de sua API. Um microsserviço só deve consumir outros microsserviços por suas APIs. Isso encaminha os microsserviços na direção de um menor acoplamento e uma maior coesão.

Um detalhe importante é que cada **microsserviço deve ter o seu próprio mecanismo de persistência**. Compartilhar o BD, por exemplo, leva a uma intimidade inapropriada entre os microsserviços. Esconder a estratégia de persistência permite com que diferentes mecanismos sejam utilizados: BD relacionais, orientados a documentos, orientados a grafos, chave-valor, e qualquer outro.

# Pontos positivos de uma Arquitetura de Microsserviços

## Entrega contínua de aplicações complexas

Com microsserviços, só um pedaço do sistema fica fora do ar ao implantarmos novas versões. Isso minimiza o risco e impacto na disponibilidade. A consequência é que podemos passar a fazer mais deploys em produção, talvez várias vezes por dia.

## Times pequenos

Um microsserviço permite times menores, com uma possibilidade de melhor comunicação e mais focados em uma área de negócio.

## Escalabilidade independente

Necessidades diferentes em termos computacionais, como processamentos intensivos em termos de memória e/ou CPU, podem ter recursos específicos. Isso minimiza o impacto em outras partes da aplicação, otimiza recursos e diminui custos de operação.

## Resiliência e isolamento de falhas

Quando há uma falha ou indisponibilidade em um serviço, os outros serviços continuam no ar e, portanto, parte da aplicação ainda permanece disponível e utilizável.

## “Trocabilidade”

Com microsserviços mais focados, independentes e pequenos, é menos provável acabar com uma parte do sistema que ninguém toca. Caso surja uma nova ideia de implementação melhor e mais eficiente, será mais fácil de trocar a antiga. Caso não haja mais necessidade de um determinado microsserviço, podemos removê-lo.

## Heterogeneidade das tecnologias e experimentação

Partes do sistema podem ser implementadas em tecnologias que estejam mais de acordo com o problema.

## Reuso e composibilidade

Diferentes microsserviços podem ser compostos em novos microsserviços, atendendo com agilidade às demandas do negócio.

Referências
Building Microservices - Cap 1 - Key Benefits
Microservice Patterns - Cap 1 - Benefits of the microservice architecture

# Pontos negativos de uma Arquitetura de Microsserviços

## Complexidade de encontrar os microsserviços certos

Definir de quais microsserviços devem ser consistido um sistema é algo complexo e subjetivo. Temos que tomar cuidado com os monólitos distribuídos, em que uma série de mudanças sempre acaba afetando um conjunto de microsserviços, que acabam tendo que ser implantados ao mesmo tempo.

## Dificuldades inerentes a um sistema distribuído

> "Scale breaks hardware.
> Speed breaks software.
> Speed at scale breaks everything."
> Adrian Cockcroft, na palestra da Flowcon 2013 Velocity and Volume (or Speed Wins)

> “Law of conservation of complexity in software: when we break up big things into small pieces we invariably push the complexity to their interaction."
> Michael Feathers, no post Microservices Until Macro Complexity

> “We are shifting the accidental complexity *[...]* from inside our application *[...]* out into our infrastructure *[...]*
> NOW is a good time for this because we’ve got many more ways to manage that complexity *[...]*
> Programmable infrastructure, infrastructure automation, the movement to the cloud *[...]*
> We’ve got better tools to address these things NOW.”
> James Lewis, no podcast SE Radio

Por ser um sistema distribuído, uma chamada de um microsserviço a outro envolve a rede. A performance é afetada negativamente. É preciso tomar cuidado com latência, banda disponível, falhas na rede, entre outros problemas. Além disso, transações distribuídas são um problema muito complexo.

## Coordenação de funcionalidades que envolvem muitos microsserviços

Mesmo com microsserviços bem projetados, às vezes uma mudança envolve vários deles e, consequentemente, vários times. Nesses casos, é preciso uma coordenação e governança cuidadosa, um controle dos contratos entre os serviços.

## Saber o momento correto de adoção é difícil

Fatiar o sistema, criando barreiras arquiteturais e separando os microsserviços independentes pode ser obscuro no começo de um projeto, quando não se conhece claramente o negócio ou as possíveis alterações. Isso é especialmente difícil para startups, que ainda estão validando o modelo de negócios e fazem mudanças drásticas com frequência.

## Monitoramento complicado

Monitorar um monólito é fácil: o sistema está ou não fora do ar, os logs ficam apenas em uma máquina, sabemos claramente por onde uma requisição passou. Com uma arquitetura de microsserviços, precisamos saber da “saúde” de cada um deles, além de agregar logs e rastrear por onde passa uma requisição.

Referências

Microservice Patterns - Cap 1 - Drawbacks of the microservice architecture

# O contexto que levou aos microsserviços

> “Domain-driven design. Continuous delivery. On-demand virtualization. Infrastructure automation. Small autonomous teams. Systems at scale. Microservices have emerged from this world.”
> Sam Newman, no livro Building Microservices, Fev. 2015, O’Reilly

Na década de 90, aplicações Desktop deram lugar à Web. A arquitetura Web, do estilo Cliente/Servidor com “telas” geradas pelo Servidor (*thin clients*), influenciou na implantação. As publicações de novas versões da aplicação foram facilitadas.

Em 2001, vários metodologistas publicaram o **Manifesto Ágil** em que definem os valores e princípios de metodologias leves, que serviram como uma resposta às maneiras burocráticas que levaram vários projetos ao fracasso durante a década de 90. Uma maneira mais adequada seria a entrega frequente de software funcionando através ciclos curtos de colaboração com os clientes, permitindo resposta às mudanças do negócio. Tudo feito por times autônomos e pequenos, de 9 pessoas, no máximo.

Em 2003, Eric Evans documentou sua abordagem de design no livro **Domain-Driven Design** (DDD), em que divide um problema complexo em sub-domínios alinhados com áreas de expertise do negócio. Cada sub-domínio define um contexto delimitado (*bounded context*) em que há uma linguagem (*ubiquitous language*). Um modelo dessa linguagem é representado no código: o modelo do domínio (*domain model*).

Entre 2005 e 2006, a Intel e a AMD criaram extensões em seus processadores para permitir a criação eficiente de **virtual machines** (máquinas virtuais). Já havia tecnologia semelhante em mainframes desde a década de 1960. Porém, com essas novas capacidades em hardwares mais baratos, surgiram uma profusão de soluções como VMWare, VirtualBox, Hyper-V, entre outras.

As tecnologias de criação de máquinas virtuais permitiram o provisionamento (configuração) de máquinas virtuais por meio de scripts, o que ficou conhecido como **infrastructure as code**. A partir de 2005, surgiram várias soluções do tipo como Puppet, Chef, Vagrant, Salt e Ansible.

Em 2006, foi inaugurada a Amazon Web Services (AWS) que, por meio do Elastic Compute Cloud (EC2), cunhou o termo **Cloud Computing**. A ideia é que o código de uma aplicação seria executado na “nuvem”, sem a necessidade de compra, manutenção e configuração de máquina físicas. O poder computacional poderia ser consumido sob-demanda, como luz ou água, permitindo que a infraestrutura de TI seja ajustada às reais necessidades, minimizando máquinas ociosas.

Em 2009, foi organizada a primeira conferência devopsdays, que unia tópicos de desenvolvimento de software e operações de TI, cunhando o termo **DevOps**.

Em 2010, Jez Humble e David Farley publicaram o livro **Continuous Delivery**, em que descrevem como algumas grandes empresas conseguem publicar software várias vezes ao dia, com poucos defeitos e alta disponibilidade (*zero downtime*). Partindo de técnicas ágeis como *continuous integration*, há um grande foco em automação, inclusive de testes.

# Microsserviços vs SOA

> “One ESB
> One bus to rule them all,
> One bus to find them,
> One bus to bring them all
> And in the darkness bind them”
> Rafael Benevides, na palestra do WorkJava 2018 Desenvolva aplicativos Java de forma confiável utilizando arquitetura Microservices

SOA (Service-Oriented Architecture) é uma abordagem arquitetural documentada pela Gartner em um artigo de 1996 que, no começo da década de 2000, passou a ser adotada por várias grandes corporações. A oportunidade de vender soluções de software e hardware foi aproveitada por empresas de TI como IBM, Oracle, HP, SAP e Sun durante essa década.

Martin Fowler diz em seu artigo sobre Microsserviços, que há uma grande ambiguidade sobre o que SOA realmente é. Dependendo da definição, uma Arquitetura de Microsserviços é SOA. Mas pode não ser.

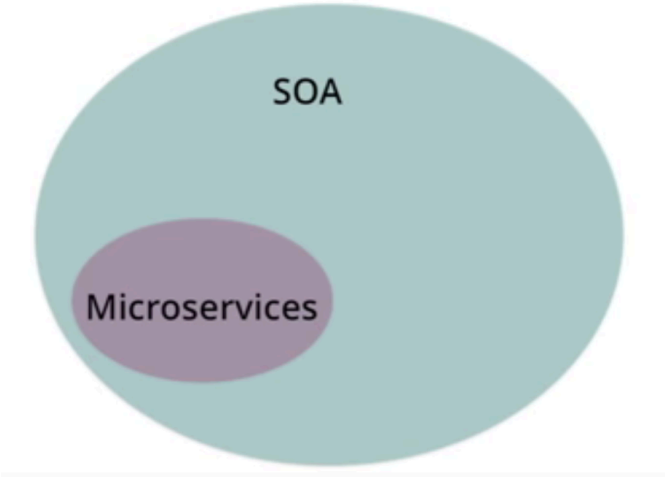
Chris Richardson descreve SOA em seu livro como sendo uma arquitetura que usa *smart pipes* como ESB, protocolos pesados como SOAP e WS-*, Persistência centralizada em BDs corporativos e serviços grandes. Talvez seja a versão mais comum de SOA que vemos implementada nas organizações.

Sam Newman, em seu livro, reconhece que SOA trouxe boas ideias, mas que houve uma falta de consenso em como fazer SOA bem e dificuldade em ter uma narrativa alternativa à dos vendedores. SOA passou a ser visto como uma coleção de ferramentas e não como uma abordagem arquitetural.

Henrique Lobo mostra em um artigo que SOA como descrito pelo consórcio de padrões abertos OASIS é muito parecido com o espírito dos microsserviços.

Microsserviços seriam, então, uma abordagem para SOA. Talvez “SOA do jeito certo”, como diz Martin Fowler em seu artigo. A Netflix, empresa referência na implementação de microsserviços, inicialmente chamava sua abordagem de *fine-grained SOA*. Sam Newman

fala que microsserviços está para SOA assim como XP e Scrum estão para Agile, uma abordagem específica que veio de projetos reais.




# Microservice Patterns

No livro Microservice Patterns, Chris Richardson descreve uma *Pattern Language*, que dá nomes a soluções relacionadas a uma Arquitetura de Microsserviços.

Há *patterns* relacionados à modelagem (ou decomposição funcional), a implementação de relatórios, segurança, estilos de integração, publicação de APIs, UI, testes, implantação, monitoramento, entre outros. Alguns desses *patterns*:

## Service Registry

Com dezenas ou centenas de microsserviços e considerando a natureza dinâmica das implantações, deixar o endereço de cada um deles no código ou em arquivos de configuração torna-se inviável.

Um Service Registry funciona com um banco de dados de instâncias de microsserviços. Cada uma das instâncias deve se registrar com o Registry, que deve ser consultado sempre que houver necessidade de saber quais os endereços de instâncias dos microsserviços estão no ar.

## Distributed Tracing

Em uma Arquitetura de Microsserviços, uma requisição passa por diferentes instâncias. Quando acontece um erro, como saber por onde a requisição passou?

É necessário deixar um “rastro” de cada uma das requisições que passaram por cada um dos microsserviços. Um  servidor como o Zipkin fica responsável por agregar esses rastros. Para isso, é necessário um *correlation ID*, que permanece o mesmo durante toda a requisição.

## Circuit Breaker

O que acontece quando uma chamada síncrona dá errado? Se o microsserviço que está sendo chamado está fora do ar, o erro acontecerá em poucos milisegundos. Mas e se houver uma demora? O cliente ficará esperando indefinidamente? E se chegarem milhares de outras requisições? Todas elas ficarão esperando?

Para evitar problemas quando há demora em chamadas síncronas, podemos usar um *Circuit Breaker* (ou disjuntor), descrito por Michael Nygard no livro Release It!. Depois de um período de *timeout*, o "disjuntor" se desarma e as chamadas dão erro imediatamente. Depois de um tempo, algumas chamadas são testadas até que o microsserviço passe a responder normalmente e o "disjuntor" se feche novamente.

## API Gateway

A(s) UI(s) e outros clientes precisam de informações de diferentes microsserviços. Será que o clientes devem chamar cada um deles, conhecendo seus endereços e realizando várias RTT (*round trip time*)?

É melhor implementar um *API Gateway*, um ponto único de entrada para os microsserviços. Esse *API Gateway* fica na borda do sistema, podendo ser chamado externamente. Por isso, é chamado de *Edge Service*. Uma chamada ocasiona vários outras a cada microsserviço. Como são chamadas na rede interna, são bem mais performáticas.

## Health Check

Como saber se um microsserviço está respondendo em tempo hábil?

Cada microsserviço pode prover uma *Health Check* API que indica informações sobre sua "saúde": memória e/ou disco disponível, status das conexões, métricas de negócio. Essa API pode ser usada por um *Service Registry*, *Circuit Breaker* e *API Gateway* para invocar os serviços.

## Externalized Configuration

Um microsserviço precisa de diferentes configurações como relacionadas a outros microsserviços, a bancos de dados, middlewares e APIs externas. A mudança de uma dessas configurações precisa ter efeito sem uma mudança de código. Idealmente, se alguma configuração for modificada, um microsserviço não precisaria ser reiniciado.

Toda configuração deve ser recebida externamente como por variáveis de ambiente, arquivos ou argumentos de linha de comando. Idealmente, um servidor centraliza as configurações e cada microsserviço recebe atualizações.

## Contract Tests

Como implementar um teste automatizado de um microsserviço? Será que o microsserviço que será chamado precisa estar no ar? E os dados?

Os desenvolvedores de um microsserviço devem prover um contrato com exemplos de uso da API. Esse contrato deve ser usado pra implementar testes automatizados tanto no cliente como no servidor, os *contract tests*. Como cada cliente pode ter interesses diferentes na chamada de um microsserviço, pode ser uma boa ideia que os clientes determinem o que é esperado do microsserviço, os *consumer-driven contract tests*.

# Para que serve o Spring Cloud?

O Spring Cloud oferece bibliotecas que facilitam a criação de sistemas distribuídos no estilo de Microsserviços, ajudando a implementar vários dos *patterns* mencionados anteriormente. Por meio de simples anotações, conseguimos aplicar esses *patterns* de maneira muito fácil.

Entre os principais projetos:

## Spring Cloud Netflix Eureka

O Eureka, da Netflix, provê um servidor que serve como um *Service Registry*. O Spring Cloud facilita a criação desse servidor e a conexão dos clientes por meio de anotações.

## Spring Cloud Sleuth

O Spring Cloud Sleuth instrumentaliza pontos de entrada e saída de aplicações Spring, inserindo *correlation IDs* no *trace* e nos *spans*, que são cada etapa de uma chamada distribuída. Além disso, conecta com servidores de *distributed tracing* como Zipkin, Dapper e HTrace.

## Spring Cloud Netflix Hystrix

O Hystrix, da Netflix, implementa o *Circuit Breaker pattern*. O Spring Cloud facilita o uso por meio de anotações.

## Spring Cloud Netflix Zuul e Spring Cloud Gateway

O Zuul, da Netflix, implementa um edge service proxy que pode ser usado para implementar uma API Gateway de maneira muito simples. O Spring Cloud facilita ainda mais, minimizando as configurações.

O Spring Cloud Gateway provê uma implementação própria, baseada no Spring Reactor.

## Spring Actuator

O Spring Actuator, parte do Spring Boot, provê funcionalidades para implementar uma *Health Check API*.

## Spring Cloud Config

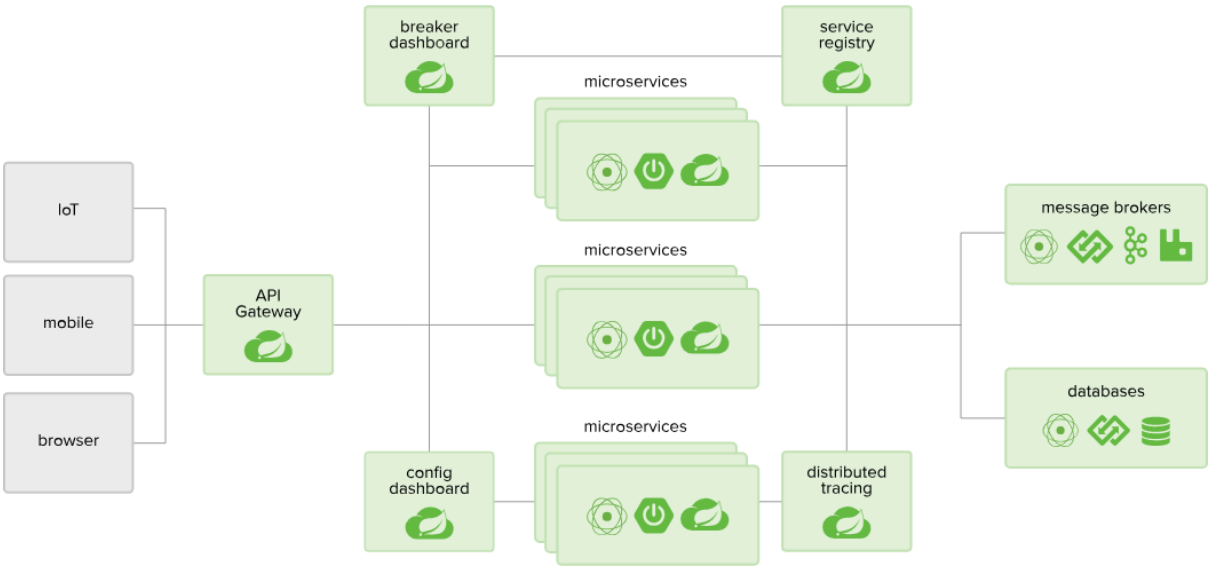
O Spring Cloud Config provê suporte do lado do cliente e do servidor para configurações externas e distribuídas. É possível fazer atualização automática das configurações, sem reiniciar os serviços.

## Spring Cloud Contract

O Spring Cloud Contract provê uma série de soluções para implementar *Contract Tests*, tanto para APIs REST como para baseadas em mensageria.

O Spring Cloud possui ainda:

- *Client-side load balancing* com Spring Cloud Netflix Ribbon
- Mensageria facilitada com Spring Cloud Bus e Spring Cloud Stream
- Clientes REST declarativos com Spring Cloud OpenFeign
- Agregação de métricas e informações de monitoramento com Netflix Turbine, Hystrix Dashboard e Spring Boot Admin



# Tópicos e tecnologias a serem abordados no curso

1. **Monólito**
2. **Modelagem**: planejando decomposição
    - Monólitos Distribuídos (anti-pattern)
    - Capacidades de Negócio
    - DDD: Bounded Contexts, Ubiquitous Language, Domain Model
    - Conway's Law e Reverse Conway's Maneuver
3. **Refatoração**: quebrando o monólito
    - Strangler Application
    - Quebrando os schemas do BD (estilo Building Microservices)
    - Modularização do código e o Monólito Modular
    - Aggregates
    - UI e API Gateway
        - Tecnologias: Spring Cloud Netflix Zuul e Spring Cloud Gateway (só mencionar)
        - Extra (só mencionar): Backend for Frontend
    - Feature Toggles
        - ff4j
4. **Integração**
    - Síncrona: REST
        - Tecnologias: @RestController, RestTemplate, Feign, Spring HATEOAS
    - Tolerância a falhas
        - Circuit Breaker, Fallback e Retry
        - Tecnologias: Spring Cloud Netflix Hystrix
    - Performance e Cache
    - Assíncrona: Mensageria
        - Event Sourcing, Domain Events
        - Tecnologias: AMQP com RabbitMQ, Spring Cloud Stream
    - Extra (só mencionar?): gRPC e Google Protocol Buffers, Apache Thrift, Apache Avro
5. **Testes**
    - Testes de integração (será que precisa? Já é abordado no FJ-27)
        - Tecnologias: Spring MockMvc, WireMock, Mockito
    - Contract Tests
        - Tecnologias: Spring Cloud Contract
6. **Implantação**
    - Automação de Infraestrutura, VMs e Containers
        - Tecnologias: Docker, Docker Compose
    - Configurabilidade
        - Spring Cloud Config,  Spring Cloud Bus
    - Service Discovery, Service Registry

        - Tecnologias: Spring Cloud Netflix Eureka
    - Client Side Load Balancing
        - Tecnologias: Spring Cloud Netflix Ribbon
    - Extra (só mencionar): Docker Swarm, Kubernetes, Side Car/Service Mesh
7. **Monitoramento e Observabilidade**
    - Health Check API
        - Tecnologias: Spring Boot Actuator
    - Log Aggregation
    - Distributed Tracing
        - Tecnologias: Netflix Turbine, Hystrix Dashboard,  Zipkin, Spring Cloud Sleuth e Spring Boot Admin
8. **Segurança**
    - OAuth
    - Single Sign On (SSO)
    - JWT
9. **Relatórios e Transações (só mencionar?)**
    - Data Pumps, API Composition, CQRS
    - Saga pattern